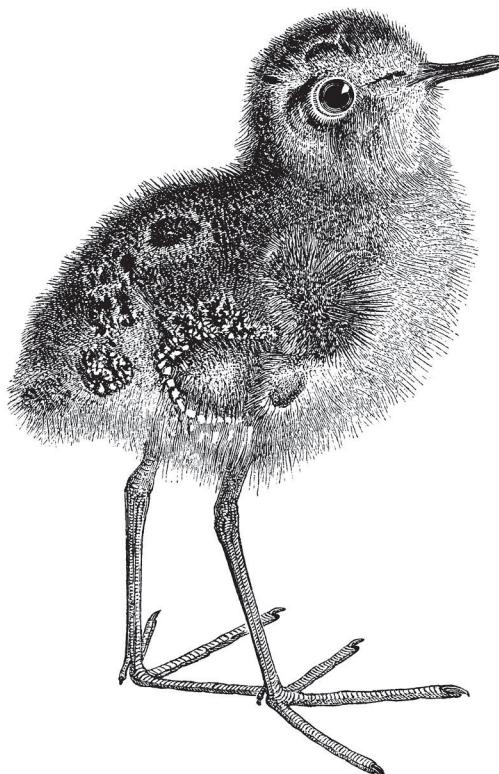


O'REILLY®

3rd Edition

Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow

Concepts, Tools, and Techniques
to Build Intelligent Systems



Early
Release
RAW &
UNEDITED

Aurélien Géron

THIRD EDITION

Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

*Concepts, Tools, and Techniques to
Build Intelligent Systems*

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Aurélien Géron

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

by Aurélien Géron

Copyright © 2022 Aurélien Géron. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Acquisitions Editor: Nicole Butterfield

Development Editor: Michele Cronin

Production Editor: Beth Kelly

Copyeditor: Kim Cofer

Proofreader:

Indexer:

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

October 2022: Third Edition

Revision History for the Early Release

2022-02-08: First Release

2022-03-14: Second Release

2022-04-18: Third Release

2022-06-010: Fourth Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781098125974> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author, and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-12597-4

[LSI]

Table of Contents

Preface.....	xv
--------------	----

Part I. The Fundamentals of Machine Learning

1. The Machine Learning Landscape.....	3
What Is Machine Learning?	4
Why Use Machine Learning?	5
Examples of Applications	8
Types of Machine Learning Systems	9
Training Supervision	10
Batch versus Online Learning	17
Instance-Based Versus Model-Based Learning	20
Main Challenges of Machine Learning	27
Insufficient Quantity of Training Data	27
Nonrepresentative Training Data	28
Poor-Quality Data	30
Irrelevant Features	30
Overfitting the Training Data	31
Underfitting the Training Data	33
Stepping Back	33
Testing and Validating	34
Hyperparameter Tuning and Model Selection	35
Data Mismatch	36
Exercises	37
2. End-to-End Machine Learning Project.....	39
Working with Real Data	40

Look at the Big Picture	41
Frame the Problem	41
Select a Performance Measure	44
Check the Assumptions	46
Get the Data	46
Running the Code Examples Using Google Colab	47
Saving Your Code Changes and Your Data	49
The Power and Danger of Interactivity	50
Book Code vs Notebook Code	51
Download the Data	51
Take a Quick Look at the Data Structure	52
Create a Test Set	56
Discover and Visualize the Data to Gain Insights	61
Visualizing Geographical Data	61
Looking for Correlations	64
Experimenting with Attribute Combinations	67
Prepare the Data for Machine Learning Algorithms	68
Data Cleaning	69
Handling Text and Categorical Attributes	72
Feature Scaling and Transformation	75
Custom Transformers	80
Transformation Pipelines	83
Select and Train a Model	88
Training and Evaluating on the Training Set	88
Better Evaluation Using Cross-Validation	90
Fine-Tune Your Model	91
Grid Search	92
Randomized Search	94
Ensemble Methods	95
Analyze the Best Models and Their Errors	95
Evaluate Your System on the Test Set	96
Launch, Monitor, and Maintain Your System	97
Try It Out!	101
Exercises	101
3. Classification.....	103
MNIST	103
Training a Binary Classifier	107
Performance Measures	107
Measuring Accuracy Using Cross-Validation	107
Confusion Matrix	109
Precision and Recall	111

Precision/Recall Trade-off	112
The ROC Curve	116
Multiclass Classification	120
Error Analysis	122
Multilabel Classification	126
Multioutput Classification	128
Exercises	130
4. Training Models.....	133
Linear Regression	134
The Normal Equation	137
Computational Complexity	140
Gradient Descent	140
Batch Gradient Descent	144
Stochastic Gradient Descent	147
Mini-batch Gradient Descent	151
Polynomial Regression	152
Learning Curves	154
Regularized Linear Models	158
Ridge Regression	158
Lasso Regression	161
Elastic Net	164
Early Stopping	165
Logistic Regression	166
Estimating Probabilities	167
Training and Cost Function	168
Decision Boundaries	170
Softmax Regression	173
Exercises	176
5. Support Vector Machines.....	179
Linear SVM Classification	179
Soft Margin Classification	180
Nonlinear SVM Classification	182
Polynomial Kernel	184
Similarity Features	185
Gaussian RBF Kernel	185
SVM Classes and Computational Complexity	187
SVM Regression	188
Under The Hood of Linear SVM Classifiers	190
The Dual Problem	193
Kernelized SVMs	194

Exercises	198
6. Decision Trees.....	201
Training and Visualizing a Decision Tree	202
Making Predictions	203
Estimating Class Probabilities	205
The CART Training Algorithm	206
Computational Complexity	207
Gini Impurity or Entropy?	207
Regularization Hyperparameters	208
Regression	210
Sensitivity to axis orientation	212
Decision Trees have a high variance	214
Exercises	215
7. Ensemble Learning and Random Forests.....	217
Voting Classifiers	218
Bagging and Pasting	222
Bagging and Pasting in Scikit-Learn	223
Out-of-Bag Evaluation	224
Random Patches and Random Subspaces	225
Random Forests	226
Extra-Trees	227
Feature Importance	227
Boosting	228
AdaBoost	229
Gradient Boosting	232
Histogram-Based Gradient Boosting	236
Stacking	237
Exercises	241
8. Dimensionality Reduction.....	243
The Curse of Dimensionality	244
Main Approaches for Dimensionality Reduction	246
Projection	246
Manifold Learning	247
PCA	250
Preserving the Variance	250
Principal Components	251
Projecting Down to d Dimensions	252
Using Scikit-Learn	252
Explained Variance Ratio	253

Choosing the Right Number of Dimensions	253
PCA for Compression	255
Randomized PCA	256
Incremental PCA	257
Random Projection	258
LLE	260
Other Dimensionality Reduction Techniques	262
Exercises	264
9. Unsupervised Learning Techniques.....	265
Clustering Algorithms: K-Means and DBSCAN	266
K-Means	269
Limits of K-Means	278
Using Clustering for Image Segmentation	279
Using Clustering for Semi-Supervised Learning	281
DBSCAN	284
Other Clustering Algorithms	287
Gaussian Mixtures	289
Using Gaussian Mixtures for Anomaly Detection	293
Selecting the Number of Clusters	295
Bayesian Gaussian Mixture Models	297
Other Algorithms for Anomaly and Novelty Detection	298
Exercises	299

Part II. Neural Networks and Deep Learning

10. Introduction to Artificial Neural Networks with Keras.....	305
From Biological to Artificial Neurons	306
Biological Neurons	307
Logical Computations with Neurons	309
The Perceptron	310
The Multilayer Perceptron and Backpropagation	315
Regression MLPs	319
Classification MLPs	321
Implementing MLPs with Keras	322
Building an Image Classifier Using the Sequential API	323
Building a Regression MLP Using the Sequential API	334
Building Complex Models Using the Functional API	335
Using the Subclassing API to Build Dynamic Models	341
Saving and Restoring a Model	343
Using Callbacks	344

Using TensorBoard for Visualization	345
Fine-Tuning Neural Network Hyperparameters	349
Number of Hidden Layers	355
Number of Neurons per Hidden Layer	356
Learning Rate, Batch Size, and Other Hyperparameters	356
Exercises	358
11. Training Deep Neural Networks.....	363
The Vanishing/Exploding Gradients Problems	364
Glorot and He Initialization	365
Better Activation Functions	367
Batch Normalization	373
Gradient Clipping	378
Reusing Pretrained Layers	379
Transfer Learning with Keras	381
Unsupervised Pretraining	383
Pretraining on an Auxiliary Task	384
Faster Optimizers	385
Momentum Optimization	385
Nesterov Accelerated Gradient	387
AdaGrad	388
RMSProp	389
Adam Optimization	390
AdaMax	392
Nadam	392
AdamW	392
Learning Rate Scheduling	394
Avoiding Overfitting Through Regularization	399
ℓ_1 and ℓ_2 Regularization	399
Dropout	400
Monte Carlo (MC) Dropout	403
Max-Norm Regularization	406
Summary and Practical Guidelines	406
Exercises	408
12. Custom Models and Training with TensorFlow.....	411
A Quick Tour of TensorFlow	412
Using TensorFlow like NumPy	415
Tensors and Operations	415
Tensors and NumPy	417
Type Conversions	417
Variables	418

Other Data Structures	419
Customizing Models and Training Algorithms	420
Custom Loss Functions	420
Saving and Loading Models That Contain Custom Components	421
Custom Activation Functions, Initializers, Regularizers, and Constraints	423
Custom Metrics	424
Custom Layers	427
Custom Models	430
Losses and Metrics Based on Model Internals	432
Computing Gradients Using Autodiff	434
Custom Training Loops	438
TensorFlow Functions and Graphs	441
AutoGraph and Tracing	443
TF Function Rules	444
Exercises	446
13. Loading and Preprocessing Data with TensorFlow.....	449
The tf.data API	450
Chaining Transformations	451
Shuffling the Data	453
Preprocessing the Data	456
Putting Everything Together	457
Prefetching	458
Using the Dataset with Keras	460
The TFRecord Format	462
Compressed TFRecord Files	462
A Brief Introduction to Protocol Buffers	463
TensorFlow Protobufs	464
Loading and Parsing Examples	466
Handling Lists of Lists Using the SequenceExample Protobuf	467
Keras Preprocessing Layers	468
The Normalization Layer	468
The Discretization Layer	471
The CategoryEncoding Layer	471
The StringLookup Layer	473
The Hashing Layer	474
Encoding Categorical Features Using Embeddings	474
Text Preprocessing	479
Using Pretrained Language Model Components	481
Image Preprocessing Layers	482
The TensorFlow Datasets (TFDS) Project	483
Exercises	485

14. Deep Computer Vision Using Convolutional Neural Networks.....	487
The Architecture of the Visual Cortex	488
Convolutional Layers	490
Filters	492
Stacking Multiple Feature Maps	493
Implementing Convolutional Layers With Keras	495
Memory Requirements	499
Pooling Layers	500
Implementing Pooling Layers With Keras	502
CNN Architectures	504
LeNet-5	507
AlexNet	508
GoogLeNet	510
VGGNet	514
ResNet	514
Xception	518
SENet	520
Other Noteworthy Architectures	522
Choosing The Right CNN Architecture	523
Implementing a ResNet-34 CNN Using Keras	524
Using Pretrained Models from Keras	526
Pretrained Models for Transfer Learning	528
Classification and Localization	530
Object Detection	532
Fully Convolutional Networks	534
You Only Look Once (YOLO)	537
Object Tracking	539
Semantic Segmentation	540
Exercises	544
15. Processing Sequences Using RNNs and CNNs.....	547
Recurrent Neurons and Layers	548
Memory Cells	551
Input and Output Sequences	551
Training RNNs	553
Forecasting a Time Series	554
The ARMA Model Family	559
Preparing The Data For Machine Learning Models	562
Forecasting Using a Linear Model	565
Forecasting Using a Simple RNN	566
Forecasting Using a Deep RNN	568
Forecasting Multivariate Time Series	569

Forecasting Several Time Steps Ahead	570
Forecasting Using a Sequence-To-Sequence Model	572
Handling Long Sequences	575
Fighting the Unstable Gradients Problem	575
Tackling the Short-Term Memory Problem	578
Exercises	586
16. Natural Language Processing with RNNs and Attention.....	587
Generating Shakespearean Text Using a Character RNN	588
Creating the Training Dataset	589
Building and Training the Char-RNN Model	591
Generating Fake Shakespearean Text	593
Stateful RNN	594
Sentiment Analysis	597
Masking	600
Reusing Pretrained Embeddings and Language Models	603
An Encoder–Decoder Network for Neural Machine Translation	605
Bidirectional RNNs	611
Beam Search	613
Attention Mechanisms	615
Attention Is All You Need: The Original Transformer Architecture	619
An Avalanche of Transformer Models	630
Vision Transformers	635
Hugging Face’s Transformers Library	639
Exercises	643
17. Autoencoders, GANs, and Diffusion Models.....	645
Efficient Data Representations	647
Performing PCA with an Undercomplete Linear Autoencoder	648
Stacked Autoencoders	650
Implementing a Stacked Autoencoder Using Keras	651
Visualizing the Reconstructions	652
Visualizing the Fashion MNIST Dataset	653
Unsupervised Pretraining Using Stacked Autoencoders	654
Tying Weights	655
Training One Autoencoder at a Time	657
Convolutional Autoencoders	658
Denoising Autoencoders	659
Sparse Autoencoders	661
Variational Autoencoders	664
Generating Fashion MNIST Images	668
Generative Adversarial Networks	669

The Difficulties of Training GANs	673
Deep Convolutional GANs	675
Progressive Growing of GANs	678
StyleGANs	681
Diffusion Models	683
Exercises	690
18. Reinforcement Learning.....	693
Learning to Optimize Rewards	694
Policy Search	696
Introduction to OpenAI Gym	698
Neural Network Policies	702
Evaluating Actions: The Credit Assignment Problem	703
Policy Gradients	705
Markov Decision Processes	709
Temporal Difference Learning	714
Q-Learning	715
Exploration Policies	717
Approximate Q-Learning and Deep Q-Learning	718
Implementing Deep Q-Learning	719
Deep Q-Learning Variants	724
Fixed Q-Value Targets	724
Double DQN	725
Prioritized Experience Replay	725
Dueling DQN	726
Overview of Some Popular RL Algorithms	727
Exercises	730
19. Training and Deploying TensorFlow Models at Scale.....	733
Serving a TensorFlow Model	734
Using TensorFlow Serving	735
Creating a Prediction Service on Vertex AI	743
Running Batch Prediction Jobs on Vertex AI	751
Deploying a Model to a Mobile or Embedded Device	753
Running a Model in a Web Page	756
Using GPUs to Speed Up Computations	758
Getting Your Own GPU	759
Managing the GPU RAM	760
Placing Operations and Variables on Devices	763
Parallel Execution Across Multiple Devices	764
Training Models Across Multiple Devices	767
Model Parallelism	767

Data Parallelism	769
Training at Scale Using the Distribution Strategies API	776
Training a Model on a TensorFlow Cluster	777
Running Large Training Jobs on Vertex AI	781
Hyperparameter Tuning on Vertex AI	783
Exercises	786
Thank You!	787
A. Machine Learning Project Checklist	789
B. Autodiff	795
C. Special Data Structures	803
D. TensorFlow Graphs	811
Index	821

Preface

The Machine Learning Tsunami

In 2006, Geoffrey Hinton et al. published [a paper¹](#) showing how to train a deep neural network capable of recognizing handwritten digits with state-of-the-art precision (>98%). They branded this technique “Deep Learning.” A deep neural network is a (very) simplified model of our cerebral cortex, composed of a stack of layers of artificial neurons. Training a deep neural net was widely considered impossible at the time,² and most researchers had abandoned the idea in the late 1990s. This paper revived the interest of the scientific community, and before long many new papers demonstrated that Deep Learning was not only possible, but capable of mind-blowing achievements that no other Machine Learning (ML) technique could hope to match (with the help of tremendous computing power and great amounts of data). This enthusiasm soon extended to many other areas of Machine Learning.

A decade later, Machine Learning had conquered the industry, and today it is at the heart of much of the magic in high-tech products, ranking your web search results, powering your smartphone’s speech recognition, recommending videos, and perhaps even driving your car.

Machine Learning in Your Projects

So, naturally you are excited about Machine Learning and would love to join the party!

¹ Geoffrey E. Hinton et al., “A Fast Learning Algorithm for Deep Belief Nets,” *Neural Computation* 18 (2006): 1527–1554.

² Despite the fact that Yann LeCun’s deep convolutional neural networks had worked well for image recognition since the 1990s, although they were not as general-purpose.

Perhaps you would like to give your homemade robot a brain of its own? Make it recognize faces? Or learn to walk around?

Or maybe your company has tons of data (user logs, financial data, production data, machine sensor data, hotline stats, HR reports, etc.), and more than likely you could unearth some hidden gems if you just knew where to look. With Machine Learning, you could accomplish the following **and much more**:

- Segment customers and find the best marketing strategy for each group.
- Recommend products for each client based on what similar clients bought.
- Detect which transactions are likely to be fraudulent.
- Forecast next year's revenue.

Whatever the reason, you have decided to learn Machine Learning and implement it in your projects. Great idea!

Objective and Approach

This book assumes that you know close to nothing about Machine Learning. Its goal is to give you the concepts, tools, and intuition you need to implement programs capable of *learning from data*.

We will cover a large number of techniques, from the simplest and most commonly used (such as Linear Regression) to some of the Deep Learning techniques that regularly win competitions. For this, we will be using production-ready Python frameworks:

- **Scikit-Learn** is very easy to use, yet it implements many Machine Learning algorithms efficiently, so it makes for a great entry point to learning Machine Learning. It was created by David Cournapeau in 2007, and is now led by a team of researchers at the French Institute for Research in Computer Science and Automation (Inria).
- **TensorFlow** is a more complex library for distributed numerical computation. It makes it possible to train and run very large neural networks efficiently by distributing the computations across potentially hundreds of multi-GPU (graphics processing unit) servers. TensorFlow (TF) was created at Google and supports many of its large-scale Machine Learning applications. It was open sourced in November 2015, and version 2.0 was released in September 2019.
- **Keras** is a high-level Deep Learning API that makes it very simple to train and run neural networks. Keras comes bundled with TensorFlow, and it relies on TensorFlow for all the intensive computations.

The book favors a hands-on approach, growing an intuitive understanding of Machine Learning through concrete working examples and just a little bit of theory.



While you can read this book without picking up your laptop, I highly recommend you experiment with the code examples.

Code Examples

All the code examples in this book are open source and available online at <https://github.com/ageron/handson-ml3>, as Jupyter notebooks. These are interactive documents containing text, images, and executable code snippets (Python in our case). The easiest and quickest way to get started is to run these notebooks using Google Colab: this is a free service that allows you to run any Jupyter notebook directly online, without having to install anything on your machine. All you need is a web browser and a Google account.



In this book, I will assume that you are using Google Colab, but I have also tested the notebooks on other online platforms such as Kaggle or Binder, so you can use those if you prefer. Alternatively, you can install the required libraries and tools (or the Docker image for this book) and run the notebooks directly on your own machine. See the instructions at <https://homl.info/install>.

Prerequisites

This book assumes that you have some Python programming experience. If you don't know Python yet, <https://learnpython.org/> is a great place to start. The official tutorial on [Python.org](https://python.org) is also quite good.

This book also assumes that you are familiar with Python's main scientific libraries—in particular, **NumPy**, **Pandas**, and **Matplotlib**. If you have never used these libraries, don't worry, they're easy to learn, and I created a tutorial for each of them. You can access them online at <https://homl.info/tutorials>.

Moreover, if you want to fully understand how the Machine Learning algorithms work (not just how to use them), then you should have at least a basic understanding of a few math concepts, especially linear algebra. Specifically, you should know what vectors and matrices are, and how to perform some simple operations like adding vectors, or transposing and multiplying matrices. If you need a quick introduction to linear algebra (it's really not rocket science!), I created a tutorial available at <https://homl.info/tutorials>. You will also find a tutorial on differential calculus, which may be

helpful to understand how neural networks are trained, but it's not entirely essential to grasp the important concepts. This book also uses other mathematical concepts occasionally, such as exponentials and logarithms, a bit of probability theory and some basic statistics concepts, but nothing too advanced. If you need help on any of these, please check out <https://khanacademy.org/>, which offers many excellent and free math courses online.

Roadmap

This book is organized in two parts. **Part I, “The Fundamentals of Machine Learning”**, covers the following topics:

- What Machine Learning is, what problems it tries to solve, and the main categories and fundamental concepts of its systems
- The steps in a typical Machine Learning project
- Learning by fitting a model to data
- Optimizing a cost function
- Handling, cleaning, and preparing data
- Selecting and engineering features
- Selecting a model and tuning hyperparameters using cross-validation
- The challenges of Machine Learning, in particular underfitting and overfitting (the bias/variance trade-off)
- The most common learning algorithms: Linear and Polynomial Regression, Logistic Regression, k-Nearest Neighbors, Support Vector Machines, Decision Trees, Random Forests, and Ensemble methods
- Reducing the dimensionality of the training data to fight the “curse of dimensionality”
- Other unsupervised learning techniques, including clustering, density estimation, and anomaly detection

Part II, “Neural Networks and Deep Learning”, covers the following topics:

- What neural nets are and what they're good for
- Building and training neural nets using TensorFlow and Keras
- The most important neural net architectures: feedforward neural nets for tabular data, convolutional nets for computer vision, recurrent nets and long short-term memory (LSTM) nets for sequence processing, encoder/decoders and Transformers for natural language processing (and more!), autoencoders, generative adversarial networks (GANs), and diffusion models for generative learning

- Techniques for training deep neural nets
- How to build an agent (e.g., a bot in a game) that can learn good strategies through trial and error, using Reinforcement Learning
- Loading and preprocessing large amounts of data efficiently
- Training and deploying TensorFlow models at scale

The first part is based mostly on Scikit-Learn, while the second part uses TensorFlow and Keras.



Don't jump into deep waters too hastily: while Deep Learning is no doubt one of the most exciting areas in Machine Learning, you should master the fundamentals first. Moreover, most problems can be solved quite well using simpler techniques such as Random Forests and Ensemble methods (discussed in [Part I](#)). Deep Learning is best suited for complex problems such as image recognition, speech recognition, or natural language processing, and it requires a lot of data, computing power, and patience (unless you can leverage a pretrained neural network, as we will see).

Changes Between the First and the Second Edition

If you have already read the first edition, here are the main changes between the first and the second edition:

- All the code was migrated from TensorFlow 1.x to TensorFlow 2.x, and replaced most of the low-level TensorFlow code (graphs, sessions, feature columns, Estimators, and so on), with much simpler Keras code.
- The second edition also introduced the Data API for loading and preprocessing large datasets, the Distribution Strategies API to train and deploy TF models at scale, TF Serving and Google Cloud AI Platform to productionize models, and it briefly introduced TF Transform, TFLite, TF Addons/Seq2Seq, TensorFlow.js, and TF Agents.
- It also introduced many additional ML topics, including a new chapter on unsupervised learning, computer vision techniques for object detection and semantic segmentation, handling sequences using convolutional neural networks (CNNs), natural language processing (NLP) using recurrent neural networks (RNNs), CNNs and Transformers, generative adversarial networks (GANs), and more.

See <https://homl.info/changes2> for more details.

Changes Between the Second and the Third Edition

If you read the second edition, here are the main changes between the second and the third edition:

- All the code was updated to the latest library versions. In particular, this third edition introduces many new additions to Scikit-Learn (e.g., feature name tracking, Histogram-based Gradient Boosting, label propagation, and more). It also introduces the *Keras Tuner* library for hyperparameter tuning, Hugging Face's *transformers* library for natural language processing, and Keras' new preprocessing and data augmentation layers.
- Several vision models were added (ResNeXt, DenseNet, MobileNet, CSPNet, and EfficientNet), as well as guidelines for choosing the right one.
- **Chapter 15** now analyzes the Chicago bus and rail ridership data instead of generated time series, and it introduces the ARMA model and its variants.
- **Chapter 16** on natural language processing now builds an English-to-Spanish translation model, first using an encoder-decoder RNN, then using a Transformer model. The chapter also includes additional language models, such as Switch Transformers, DistilBERT, T5, and PaLM (with Chain-of-Thought Prompting). It also introduces visual Transformers (ViTs), and gives an overview of a few Transformer-based visual models, such as the DEtection TRansformer (DETR), Perceiver, DINO, as well as a brief overview of some large multimodal models, including CLIP, DALL-E, Flamingo, and GATO.
- **Chapter 17** on generative learning now introduces diffusion models, and implements a Denoising Diffusion Probabilistic Model (DDPM) from scratch.
- **Chapter 19** migrated from Google AI Platform to Google Vertex AI, and uses distributed Keras Tuner for large scale hyperparameter search. The chapter now includes TensorFlow.js code that you can experiment with online. It also introduces additional distributed training techniques, including Pipedream and Pathways.
- To allow for all the new content, some sections were moved online, including installation instructions, Kernel PCA, mathematical details of Bayesian Gaussian Mixtures, TF Agents, and former appendix A (exercise solutions), C (SVM maths), and E (extra neural net architectures).

See <https://homl.info/changes3> for more details.

Other Resources

Many excellent resources are available to learn about Machine Learning. For example, Andrew Ng's [ML course on Coursera](#) is amazing, although it requires a significant time investment.

There are also many interesting websites about Machine Learning, including Scikit-Learn's exceptional [User Guide](#). You may also enjoy [Dataquest](#), which provides very nice interactive tutorials, and ML blogs such as those listed on [Quora](#).

There are many other introductory books about Machine Learning. In particular:

- Joel Grus's [Data Science from Scratch](#), 2nd edition (O'Reilly) presents the fundamentals of Machine Learning and implements some of the main algorithms in pure Python (from scratch, as the name suggests).
- Stephen Marsland's [Machine Learning: An Algorithmic Perspective](#), 2nd edition (Chapman & Hall) is a great introduction to Machine Learning, covering a wide range of topics in depth with code examples in Python (also from scratch, but using NumPy).
- Sebastian Raschka's [Python Machine Learning](#), 3rd edition (Packt Publishing) is also a great introduction to Machine Learning and leverages Python open source libraries (Pylearn 2 and Theano).
- François Chollet's [Deep Learning with Python](#), 2nd edition (Manning) is a very practical book that covers a large range of topics in a clear and concise way, as you might expect from the author of the excellent Keras library. It favors code examples over mathematical theory.
- Andriy Burkov's [The Hundred-Page Machine Learning Book](#) is very short and covers an impressive range of topics, introducing them in approachable terms without shying away from the math equations.
- Yaser S. Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin's [Learning from Data](#) (MLBook) is a rather theoretical approach to ML that provides deep insights, in particular on the bias/variance trade-off (see [Chapter 4](#)).
- Stuart Russell and Peter Norvig's [Artificial Intelligence: A Modern Approach](#), 4th Edition (Pearson), is a great (and huge) book covering an incredible amount of topics, including Machine Learning. It helps put ML into perspective.
- Jeremy Howard and Sylvain Gugger's [Deep Learning for Coders with fastai and PyTorch](#) (O'Reilly) provides a wonderfully clear and practical introduction to Deep Learning using the fastai and PyTorch libraries.

Finally, joining ML competition websites such as [Kaggle.com](#) will allow you to practice your skills on real-world problems, with help and insights from some of the best ML professionals out there.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.



This element signifies a tip or suggestion.



This element signifies a general note.



This element indicates a warning or caution.

O'Reilly Online Learning



For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning

platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit <https://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <https://homl.info/oreilly2>.

Email bookquestions@oreilly.com to comment or ask technical questions about this book.

For news and information about our books and courses, visit <https://oreilly.com>.

Find us on LinkedIn: <https://linkedin.com/company/oreilly-media>

Follow us on Twitter: <https://twitter.com/oreillymedia>

Watch us on YouTube: <https://www.youtube.com/oreillymedia>

Acknowledgments

Never in my wildest dreams did I imagine that the first and second editions of this book would get such a large audience. I received so many messages from readers, many asking questions, some kindly pointing out errata, and most sending me encouraging words. I cannot express how grateful I am to all these readers for their tremendous support. Thank you all so very much! Please do not hesitate to [file issues on GitHub](#) if you find errors in the code examples (or just to ask questions), or to submit [errata](#) if you find errors in the text. Some readers also shared how this book helped them get their first job, or how it helped them solve a concrete problem they were working on. I find such feedback incredibly motivating. If you find this book helpful, I would love it if you could share your story with me, either privately (e.g., via [LinkedIn](#)) or publicly (e.g., tweet me at @aureliengeron or write an [Amazon review](#)).

Huge thanks as well to all the wonderful people who offered their time and expertise to review this third edition, correcting errors and making countless suggestions. This edition is so much better thanks to them: Olzhas Akpambetov, George Bonner,

François Chollet, Siddha Gangju, Sam Goodman, Matt Harrison, Sasha Sobran, Lewis Tunstall, Leandro von Werra, and my dear brother Sylvain. You are all amazing!

I am also very grateful to the many people who supported me along the way, by answering my questions, suggesting improvements, and contributing to the code on Github, in particular Yannick Assogba, Ian Beauregard, Ulf Bissbort, Rick Chao, Peretz Cohen, Kyle Gallatin, Hannes Hapke, Victor Khaustov, Soonson Kwon, Eric Lebigot, Jason Mayes, Laurence Moroney, Sara Robinson, Joaquín Ruales, and Yue-feng Zhou.

This book wouldn't exist without O'Reilly's fantastic staff, in particular Nicole Taché, who gave me insightful feedback and was always cheerful, encouraging, and helpful: I could not dream of a better editor. Big thanks to Michele Cronin as well, who cheered me on through the final chapters and managed to get me past the finish line. Thanks to Kristen Brown, the production editor for the second and third editions, who saw it through all the steps (she also coordinated fixes and updates for each reprint of the first and second editions). Thanks as well to Rachel Monaghan and Amanda Kersey for their thorough copyediting, and to Johnny O'Toole who managed the relationship with Amazon and answered many of my questions. Thanks to Marie Beaugureau, Ben Lorica, Mike Loukides, and Laurel Ruma for believing in this project and helping me define its scope. Thanks to Matt Hacker and all of the Atlas team for answering all my technical questions regarding formatting, AsciiDoc, MathML, and LaTeX, and thanks to Nick Adams, Rebecca Demarest, Rachel Head, Judith McConville, Helen Monroe, Karen Montgomery, Rachel Roumeliotis, and everyone else at O'Reilly who contributed to this book.

I'll never forget all the wonderful people who helped me with the first and second editions of this book: friends, colleagues, experts, including many members of the TensorFlow team, the list is long: Olzhass Akpambetov, Karmel Allison, Martin Andrews, David Andrzejewski, Paige Bailey, Lukas Biewald, Eugene Brevdo, William Chargin, François Chollet, Clément Courbet, Robert Crowe, Mark Daoust, Daniel "Wolff" Dobson, Julien Dubois, Mathias Kende, Daniel Kitachewsky, Nick Felt, Bruce Fontaine, Justin Francis, Goldie Gadde, Irene Giannoumis, Ingrid von Glehn, Vincent Guilbeau, Sandeep Gupta, Priya Gupta, Kevin Haas, Eddy Hung, Konstantinos Katsiapis, Viacheslav Kovalevskyi, Jon Krohn, Allen Lavoie, Karim Matrah, Grégoire Mesnil, Clemens Mewald, Dan Moldovan, Dominic Monn, Sean Morgan, Tom O'Malley, James Pack, Alexander Pak, Haesun Park, Alexandre Passos, Ankur Patel, Josh Patterson, André Susano Pinto, Anthony Platanios, Anosh Raj, Oscar Ramirez, Anna Revinskaya, Saurabh Saxena, Salim Sémaoune, Ryan Sepassi, Vitor Sessak, Jiri Simsa, Iain Smears, Xiaodan Song, Christina Sorokin, Michel Tessier, Wiktor Tomczak, Dustin Tran, Todd Wang, Pete Warden, Rich Washington, Martin Wicke, Edd Wilder-James, Sam Witteveen, Jason Zaman, Yuefeng Zhou, and my brother Sylvain.

Last but not least, I am infinitely grateful to my beloved wife, Emmanuelle, and to our three wonderful children, Alexandre, Rémi, and Gabrielle, for encouraging me to work hard on this book. Their insatiable curiosity was priceless: explaining some of the most difficult concepts in this book to my wife and children helped me clarify my thoughts and directly improved many parts of it. Plus, they keep bringing me cookies and coffee, who could ask for more?

PART I

The Fundamentals of Machine Learning

The Machine Learning Landscape

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. The GitHub repo is [https://git
hub.com/ageron/handsonml3](https://github.com/ageron/handsonml3).

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

Not so long ago, if you had picked up your phone and asked it the way home, it would have ignored you—and people would have questioned your sanity. But Machine Learning is no longer Science Fiction: billions of people use it every day. And the truth is it has actually been around for decades in some specialized applications, such as Optical Character Recognition (OCR). The first ML application that really became mainstream, improving the lives of hundreds of millions of people, took over the world back in the 1990s: the *spam filter*. It’s not exactly a self-aware robot, but it does technically qualify as Machine Learning: it has actually learned so well that you seldom need to flag an email as spam anymore. It was followed by hundreds of ML applications that now quietly power hundreds of products and features that you use regularly: voice prompts, automatic translation, image search, product recommendations, and many more.

Where does Machine Learning start and where does it end? What exactly does it mean for a machine to *learn* something? If I download a copy of all Wikipedia

articles, has my computer really learned something? Is it suddenly smarter? In this chapter I will start by clarifying what Machine Learning is and why you may want to use it.

Then, before we set out to explore the Machine Learning continent, we will take a look at the map and learn about the main regions and the most notable landmarks: supervised versus unsupervised learning and their variants, online versus batch learning, instance-based versus model-based learning. Then we will look at the workflow of a typical ML project, discuss the main challenges you may face, and cover how to evaluate and fine-tune a Machine Learning system.

This chapter introduces a lot of fundamental concepts (and jargon) that every data scientist should know by heart. It will be a high-level overview (it's the only chapter without much code), all rather simple, but my goal is to ensure everything is crystal clear to you before we continue on to the rest of the book. So grab a coffee and let's get started!



If you are already familiar with Machine Learning basics, you may want to skip directly to [Chapter 2](#). If you are not sure, try to answer all the questions listed at the end of the chapter before moving on.

What Is Machine Learning?

Machine Learning is the science (and art) of programming computers so they can *learn from data*.

Here is a slightly more general definition:

[Machine Learning is the] field of study that gives computers the ability to learn without being explicitly programmed.

—Arthur Samuel, 1959

And a more engineering-oriented one:

A computer program is said to learn from experience E with respect to some task T and some performance measure P , if its performance on T , as measured by P , improves with experience E .

—Tom Mitchell, 1997

Your spam filter is a Machine Learning program that, given examples of spam emails (flagged by users) and examples of regular emails (nonspam, also called “ham”), can learn to flag spam. The examples that the system uses to learn are called the *training set*. Each training example is called a *training instance* (or *sample*). The part of a

Machine Learning system that learns and makes predictions is called a *model*. Neural networks and random forests are examples of models.

In this case, the task T is to flag spam for new emails, the experience E is the *training data*, and the performance measure P needs to be defined; for example, you can use the ratio of correctly classified emails. This particular performance measure is called *accuracy*, and it is often used in classification tasks.

If you just download a copy of all Wikipedia articles, your computer has a lot more data, but it is not suddenly better at any task. It is not Machine Learning.

Why Use Machine Learning?

Consider how you would write a spam filter using traditional programming techniques (Figure 1-1):

1. First you would examine what spam typically looks like. You might notice that some words or phrases (such as “4U,” “credit card,” “free,” and “amazing”) tend to come up a lot in the subject line. Perhaps you would also notice a few other patterns in the sender’s name, the email’s body, and other parts of the email.
2. You would write a detection algorithm for each of the patterns that you noticed, and your program would flag emails as spam if a number of these patterns were detected.
3. You would test your program and repeat steps 1 and 2 until it was good enough to launch.

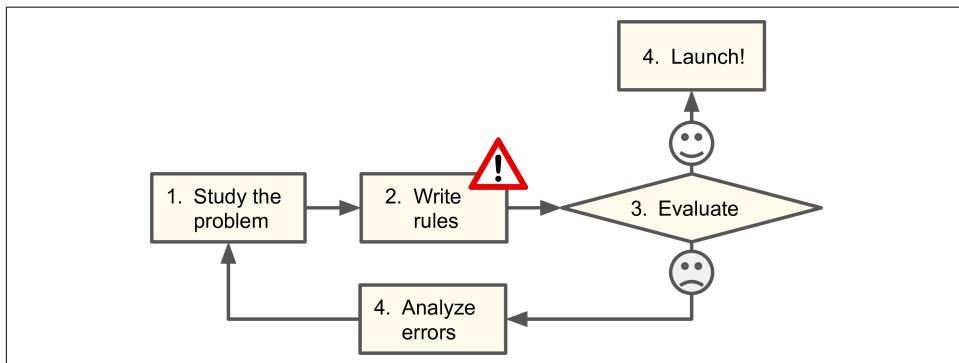


Figure 1-1. The traditional approach

Since the problem is difficult, your program will likely become a long list of complex rules—pretty hard to maintain.

In contrast, a spam filter based on Machine Learning techniques automatically learns which words and phrases are good predictors of spam by detecting unusually

frequent patterns of words in the spam examples compared to the ham examples (Figure 1-2). The program is much shorter, easier to maintain, and most likely more accurate.

What if spammers notice that all their emails containing “4U” are blocked? They might start writing “For U” instead. A spam filter using traditional programming techniques would need to be updated to flag “For U” emails. If spammers keep working around your spam filter, you will need to keep writing new rules forever.

In contrast, a spam filter based on Machine Learning techniques automatically notices that “For U” has become unusually frequent in spam flagged by users, and it starts flagging them without your intervention (Figure 1-3).

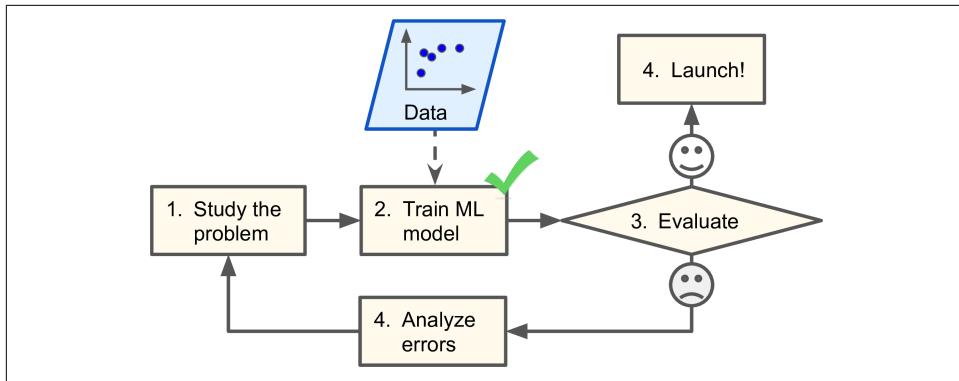


Figure 1-2. The Machine Learning approach

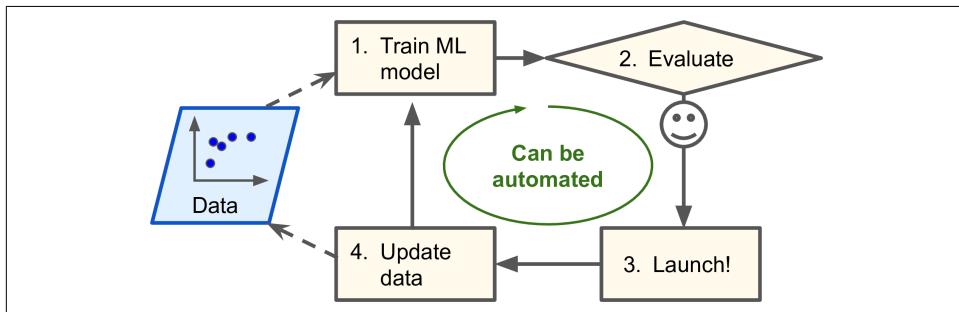


Figure 1-3. Automatically adapting to change

Another area where Machine Learning shines is for problems that either are too complex for traditional approaches or have no known algorithm. For example, consider speech recognition. Say you want to start simple and write a program capable of distinguishing the words “one” and “two.” You might notice that the word “two” starts with a high-pitch sound (“T”), so you could hardcode an algorithm that measures high-pitch sound intensity and use that to distinguish ones and twos—but obviously

this technique will not scale to thousands of words spoken by millions of very different people in noisy environments and in dozens of languages. The best solution (at least today) is to write an algorithm that learns by itself, given many example recordings for each word.

Finally, Machine Learning can help humans learn (Figure 1-4). ML models can be inspected to see what they have learned (although for some models this can be tricky). For instance, once a spam filter has been trained on enough spam, it can easily be inspected to reveal the list of words and combinations of words that it believes are the best predictors of spam. Sometimes this will reveal unsuspected correlations or new trends, and thereby lead to a better understanding of the problem. Digging into large amounts of data to discover hidden patterns is called *data mining*, and Machine Learning excels at it.

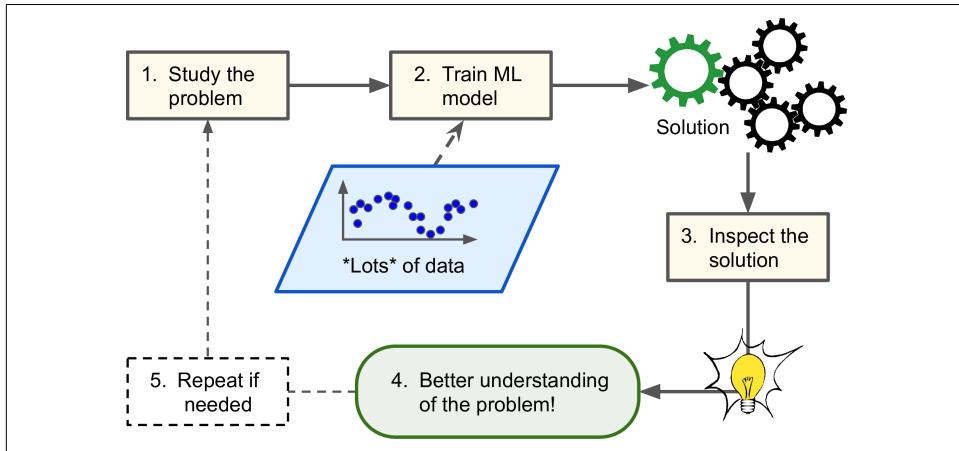


Figure 1-4. Machine Learning can help humans learn

To summarize, Machine Learning is great for:

- Problems for which existing solutions require a lot of fine-tuning or long lists of rules: one Machine Learning model can often simplify code and perform better than the traditional approach.
- Complex problems for which using a traditional approach yields no good solution: the best Machine Learning techniques can perhaps find a solution.
- Fluctuating environments: a Machine Learning system can easily be retrained on new data, always keeping it up to date.
- Getting insights about complex problems and large amounts of data.

Examples of Applications

Let's look at some concrete examples of Machine Learning tasks, along with the techniques that can tackle them:

Analyzing images of products on a production line to automatically classify them

This is image classification, typically performed using convolutional neural networks (CNNs; see [Chapter 14](#)) or sometimes Transformers (see [Chapter 16](#)).

Detecting tumors in brain scans

This is semantic image segmentation, where each pixel in the image is classified (as we want to determine the exact location and shape of tumors), typically using CNNs or Transformers.

Automatically classifying news articles

This is natural language processing (NLP), and more specifically text classification, which can be tackled using recurrent neural networks (RNNs) and CNNs, but Transformers work even better (see [Chapter 16](#)).

Automatically flagging offensive comments on discussion forums

This is also text classification, using the same NLP tools.

Summarizing long documents automatically

This is a branch of NLP called text summarization, again using the same tools.

Creating a chatbot or a personal assistant

This involves many NLP components, including natural language understanding (NLU) and question-answering modules.

Forecasting your company's revenue next year, based on many performance metrics

This is a regression task (i.e., predicting values) that may be tackled using any regression model, such as a Linear Regression or Polynomial Regression model (see [Chapter 4](#)), a regression SVM (see [Chapter 5](#)), a regression Random Forest (see [Chapter 7](#)), or an artificial neural network (see [Chapter 10](#)). If you want to take into account sequences of past performance metrics, you may want to use RNNs, CNNs, or Transformers (see Chapters [15](#) and [16](#)).

Making your app react to voice commands

This is speech recognition, which requires processing audio samples: since they are long and complex sequences, they are typically processed using RNNs, CNNs, or Transformers (see Chapters [15](#) and [16](#)).

Detecting credit card fraud

This is anomaly detection, which can be tackled using isolation forests, Gaussian mixture models (see [Chapter 9](#)) or autoencoders (see [Chapter 17](#)).

Segmenting clients based on their purchases so that you can design a different marketing strategy for each segment

This is clustering, which can be achieved using K-Means, DBSCAN and more (see [Chapter 9](#)).

Representing a complex, high-dimensional dataset in a clear and insightful diagram

This is data visualization, often involving dimensionality reduction techniques (see [Chapter 8](#)).

Recommending a product that a client may be interested in, based on past purchases

This is a recommender system. One approach is to feed past purchases (and other information about the client) to an artificial neural network (see [Chapter 10](#)), and get it to output the most likely next purchase. This neural net would typically be trained on past sequences of purchases across all clients.

Building an intelligent bot for a game

This is often tackled using Reinforcement Learning (RL; see [Chapter 18](#)), which is a branch of Machine Learning that trains agents (such as bots) to pick the actions that will maximize their rewards over time (e.g., a bot may get a reward every time the player loses some life points), within a given environment (such as the game). The famous AlphaGo program that beat the world champion at the game of Go was built using RL.

This list could go on and on, but hopefully it gives you a sense of the incredible breadth and complexity of the tasks that Machine Learning can tackle, and the types of techniques that you would use for each task.

Types of Machine Learning Systems

There are so many different types of Machine Learning systems that it is useful to classify them in broad categories, based on the following criteria:

- How they are supervised during training (supervised, unsupervised, semi-supervised, self-supervised, and others)
- Whether or not they can learn incrementally on the fly (online versus batch learning)
- Whether they work by simply comparing new data points to known data points, or instead by detecting patterns in the training data and building a predictive model, much like scientists do (instance-based versus model-based learning)

These criteria are not exclusive; you can combine them in any way you like. For example, a state-of-the-art spam filter may learn on the fly using a deep neural network model trained using human-provided examples of spam and ham; this makes it an online, model-based, supervised learning system.

Let's look at each of these criteria a bit more closely.

Training Supervision

ML systems can be classified according to the amount and type of supervision they get during training. There are many categories, but we'll discuss the main ones: supervised learning, unsupervised learning, self-supervised learning, semi-supervised learning, and Reinforcement Learning.

Supervised learning

In *supervised learning*, the training set you feed to the algorithm includes the desired solutions, called *labels* (Figure 1-5).

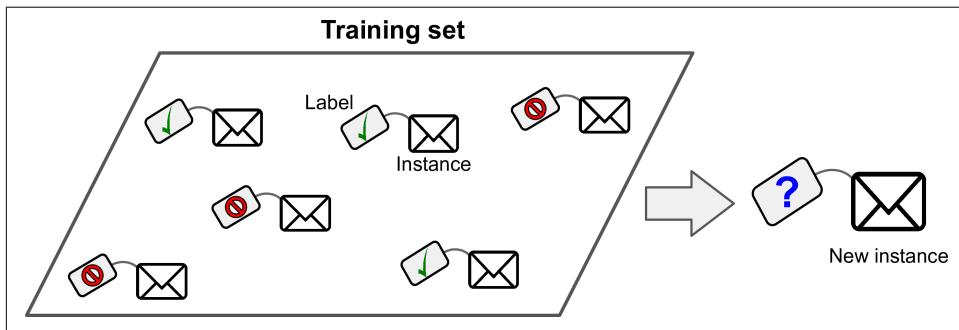


Figure 1-5. A labeled training set for spam classification (an example of supervised learning)

A typical supervised learning task is *classification*. The spam filter is a good example of this: it is trained with many example emails along with their *class* (spam or ham), and it must learn how to classify new emails.

Another typical task is to predict a *target* numeric value, such as the price of a car, given a set of *features* (mileage, age, brand, etc.). This sort of task is called *regression* (Figure 1-6).¹ To train the system, you need to give it many examples of cars, including both their features and their targets (i.e., their prices).

¹ Fun fact: this odd-sounding name is a statistics term introduced by Francis Galton while he was studying the fact that the children of tall people tend to be shorter than their parents. Since the children were shorter, he called this *regression to the mean*. This name was then applied to the methods he used to analyze correlations between variables.



The words *target* and *label* are generally treated as synonyms in supervised learning, but *target* is more common in regression tasks, and *label* is more common in classification tasks. Moreover, *features* are sometimes called *predictors* or *attributes*. These terms may refer to individual samples (e.g., “this car’s mileage feature is equal to 15,000”) or to all samples (e.g., “the mileage feature is strongly correlated with price”).

Note that some regression models can be used for classification as well, and vice versa. For example, *Logistic Regression* is commonly used for classification, as it can output a value that corresponds to the probability of belonging to a given class (e.g., 20% chance of being spam).



Figure 1-6. A regression problem: predict a value, given an input feature (there are usually multiple input features, and sometimes multiple output values)

Unsupervised learning

In *unsupervised learning*, as you might guess, the training data is unlabeled (Figure 1-7). The system tries to learn without a teacher.

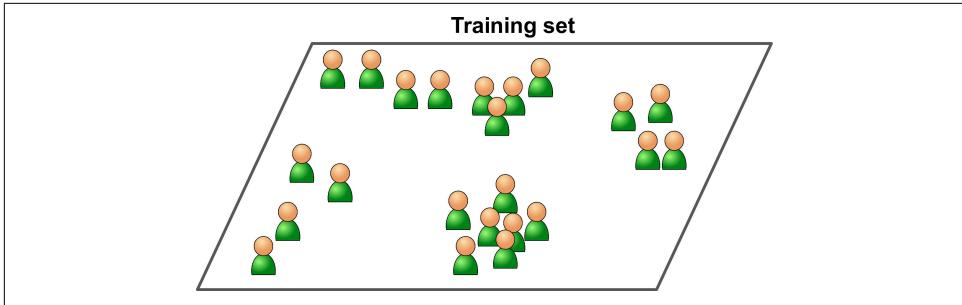


Figure 1-7. An unlabeled training set for unsupervised learning

For example, say you have a lot of data about your blog's visitors. You may want to run a *clustering* algorithm to try to detect groups of similar visitors (Figure 1-8). At no point do you tell the algorithm which group a visitor belongs to: it finds those connections without your help. For example, it might notice that 40% of your visitors are teenagers who love comic books and generally read your blog after school, while 20% are adults who enjoy sci-fi and who visit during the weekends. If you use a *hierarchical clustering* algorithm, it may also subdivide each group into smaller groups. This may help you target your posts for each group.

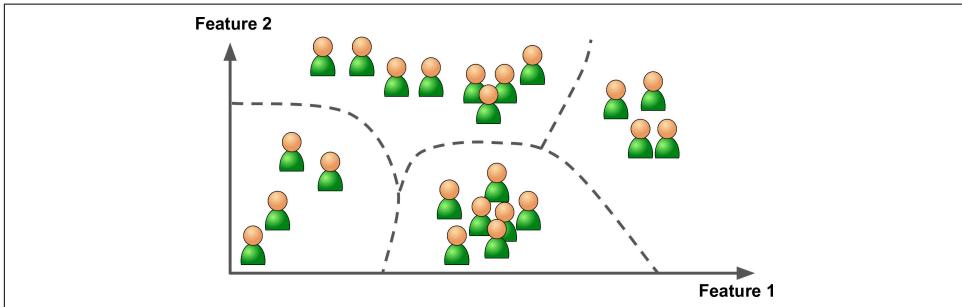


Figure 1-8. Clustering

Visualization algorithms are also good examples of unsupervised learning: you feed them a lot of complex and unlabeled data, and they output a 2D or 3D representation of your data that can easily be plotted (Figure 1-9). These algorithms try to preserve as much structure as they can (e.g., trying to keep separate clusters in the input space from overlapping in the visualization) so that you can understand how the data is organized and perhaps identify unsuspected patterns.

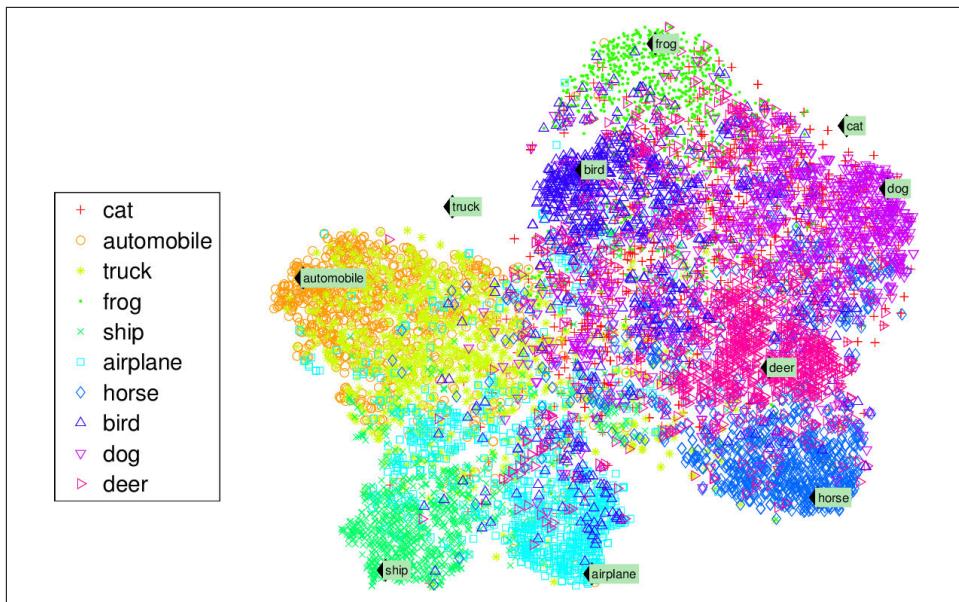


Figure 1-9. Example of a t-SNE visualization highlighting semantic clusters²

A related task is *dimensionality reduction*, in which the goal is to simplify the data without losing too much information. One way to do this is to merge several correlated features into one. For example, a car's mileage may be strongly correlated with its age, so the dimensionality reduction algorithm will merge them into one feature that represents the car's wear and tear. This is called *feature extraction*.



It is often a good idea to try to reduce the dimension of your training data using a dimensionality reduction algorithm before you feed it to another Machine Learning algorithm (such as a supervised learning algorithm). It will run much faster, the data will take up less disk and memory space, and in some cases it may also perform better.

Yet another important unsupervised task is *anomaly detection*—for example, detecting unusual credit card transactions to prevent fraud, catching manufacturing defects, or automatically removing outliers from a dataset before feeding it to another learning algorithm. The system is shown mostly normal instances during training, so it

² Notice how animals are rather well separated from vehicles and how horses are close to deer but far from birds. Figure reproduced with permission from Richard Socher et al., “Zero-Shot Learning Through Cross-Modal Transfer,” *Proceedings of the 26th International Conference on Neural Information Processing Systems 1* (2013): 935–943.

learns to recognize them; then, when it sees a new instance, it can tell whether it looks like a normal one or whether it is likely an anomaly (see [Figure 1-10](#)). A very similar task is *novelty detection*: it aims to detect new instances that look different from all instances in the training set. This requires having a very “clean” training set, devoid of any instance that you would like the algorithm to detect. For example, if you have thousands of pictures of dogs, and 1% of these pictures represent Chihuahuas, then a novelty detection algorithm should not treat new pictures of Chihuahuas as novelties. On the other hand, anomaly detection algorithms may consider these dogs as so rare and so different from other dogs that they would likely classify them as anomalies (no offense to Chihuahuas).

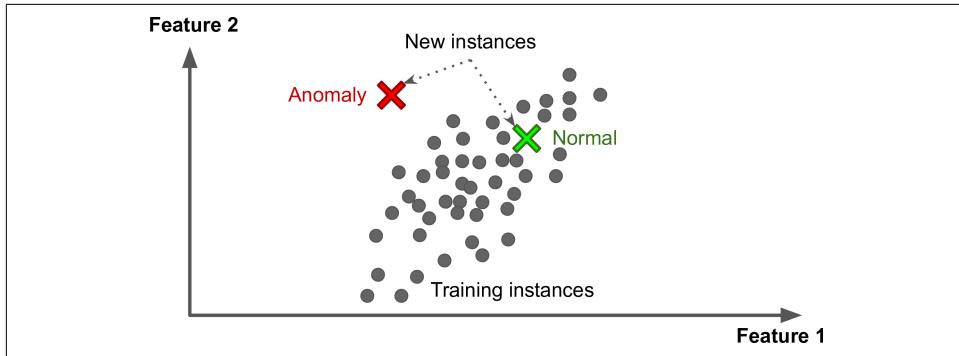


Figure 1-10. Anomaly detection

Finally, another common unsupervised task is *association rule learning*, in which the goal is to dig into large amounts of data and discover interesting relations between attributes. For example, suppose you own a supermarket. Running an association rule on your sales logs may reveal that people who purchase barbecue sauce and potato chips also tend to buy steak. Thus, you may want to place these items close to one another.

Semi-supervised learning

Since labeling data is usually time-consuming and costly, you will often have plenty of unlabeled instances, and few labeled instances. Some algorithms can deal with data that's partially labeled. This is called *semi-supervised learning* ([Figure 1-11](#)).

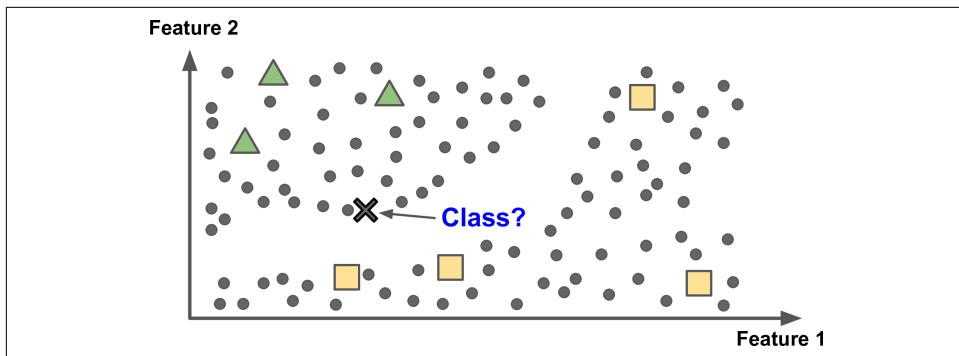


Figure 1-11. Semi-supervised learning with two classes (triangles and squares): the unlabeled examples (circles) help classify a new instance (the cross) into the triangle class rather than the square class, even though it is closer to the labeled squares

Some photo-hosting services, such as Google Photos, are good examples of this. Once you upload all your family photos to the service, it automatically recognizes that the same person A shows up in photos 1, 5, and 11, while another person B shows up in photos 2, 5, and 7. This is the unsupervised part of the algorithm (clustering). Now all the system needs is for you to tell it who these people are. Just add one label per person³ and it is able to name everyone in every photo, which is useful for searching photos.

Most semi-supervised learning algorithms are combinations of unsupervised and supervised algorithms. For example a clustering algorithm may be used to group similar instances together, and then every unlabeled instance can be labeled with the most common label in their cluster. Once the whole dataset is labeled, it is possible to use any supervised learning algorithm.

Self-supervised learning

Another approach to Machine Learning involves actually generating a fully labeled dataset from a fully unlabeled one. Again, once the whole dataset is labeled, any supervised learning algorithm can be used. This approach is called *self-supervised learning*.

For example, if you have a large dataset of unlabeled images, you can randomly mask a small part of each image and then train a model to recover the original image (Figure 1-12). During training, the masked images are used as the inputs to the model, and the original images are used as the labels.

³ That's when the system works perfectly. In practice it often creates a few clusters per person, and sometimes mixes up two people who look alike, so you may need to provide a few labels per person and manually clean up some clusters.

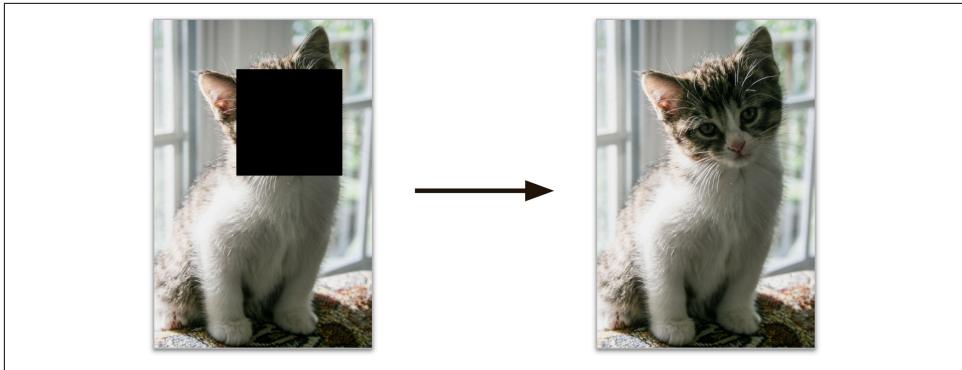


Figure 1-12. Self-supervised learning example: input (left) and target (right)

The resulting model may be quite useful in itself, for example to repair damaged images or to erase unwanted objects from pictures. But more often than not, a model trained using self-supervised learning is not the final goal. You usually want to tweak and fine-tune the model for a slightly different task. One that you actually care about.

For example, suppose that what you really want is to have a pet classification model: given a picture of any pet, it will tell you what species it belongs to. If you have a large dataset of unlabeled photos of pets, you can start by training an image-repairing model using self-supervised learning. Once it performs well, it must be able to distinguish different pet species: indeed, when it repairs an image of a cat whose face is masked, it knows it must not add a dog's face. Assuming your model's architecture allows it (and most neural network architectures do), it is then possible to tweak the model so that it predicts pet species instead of repairing images. The final step consists of fine-tuning the model on a labeled dataset: the model already knows what cats, dogs and other pet species look like, so this step is only needed so the model can learn the mapping between the species it already knows and the labels we expect from it.



Transferring knowledge from one task to another is called *transfer learning*, and it's one of the most important techniques in Machine Learning today, especially when using *deep neural networks* (i.e., neural networks composed of many layers of neurons). We will discuss this in detail in [Part II](#).

Some people consider self-supervised learning to be a part of unsupervised learning, since it deals with fully unlabeled datasets. But self-supervised learning uses (generated) labels during training, so in that regard it's closer to supervised learning. And the term “unsupervised learning” is generally used when dealing with tasks like clustering, dimensionality reduction or anomaly detection, whereas self-supervised

learning focuses on the same tasks as supervised learning: mainly classification and regression. In short, it's best to treat self-supervised learning as its own category.

Reinforcement Learning

Reinforcement Learning is a very different beast. The learning system, called an *agent* in this context, can observe the environment, select and perform actions, and get *rewards* in return (or *penalties* in the form of negative rewards, as shown in Figure 1-13). It must then learn by itself what is the best strategy, called a *policy*, to get the most reward over time. A policy defines what action the agent should choose when it is in a given situation.

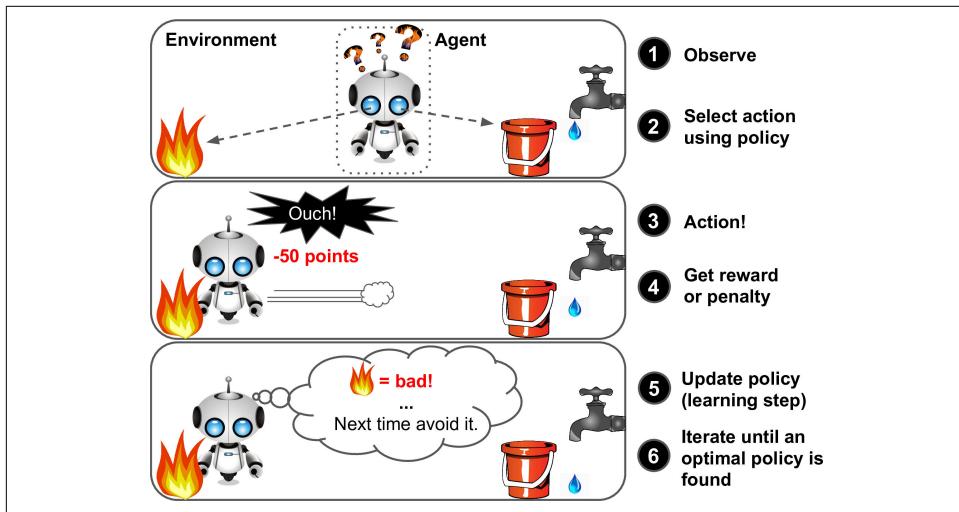


Figure 1-13. Reinforcement Learning

For example, many robots implement Reinforcement Learning algorithms to learn how to walk. DeepMind's AlphaGo program is also a good example of Reinforcement Learning: it made the headlines in May 2017 when it beat Ke Jie, the number one ranked player in the world at the time, at the game of Go. It learned its winning policy by analyzing millions of games, and then playing many games against itself. Note that learning was turned off during the games against the champion; AlphaGo was just applying the policy it had learned. As we will see in the next section, this is called *offline learning*.

Batch versus Online Learning

Another criterion used to classify Machine Learning systems is whether or not the system can learn incrementally from a stream of incoming data.

Batch learning

In *batch learning*, the system is incapable of learning incrementally: it must be trained using all the available data. This will generally take a lot of time and computing resources, so it is typically done offline. First the system is trained, and then it is launched into production and runs without learning anymore; it just applies what it has learned. This is called *offline learning*.

Unfortunately, a model's performance tends to decay slowly over time, simply because the world continues to evolve while the model remains unchanged. This phenomenon is often called *model rot* or *data drift*. The solution is to regularly retrain the model on up-to-date data. How often you need to do that depends on the use case: if the model classifies pictures of cats and dogs, its performance will decay very slowly, but if the model deals with fast-evolving systems, for example making predictions on the financial market, then it is likely to decay quite fast.



Even a model trained to classify pictures of cats and dogs may need to be retrained regularly, not because cats and dogs will mutate overnight, but because cameras keep changing, along with image formats, sharpness, brightness, and size ratios. Moreover, people may love different breeds next year, or they may decide to dress their pets with tiny hats—who knows?

If you want a batch learning system to know about new data (such as a new type of spam), you need to train a new version of the system from scratch on the full dataset (not just the new data, but also the old data), then replace the old model with the new one. Fortunately, the whole process of training, evaluating, and launching a Machine Learning system can be automated fairly easily (as we saw in [Figure 1-3](#)), so even a batch learning system can adapt to change. Simply update the data and train a new version of the system from scratch as often as needed.

This solution is simple and often works fine, but training using the full set of data can take many hours, so you would typically train a new system only every 24 hours or even just weekly. If your system needs to adapt to rapidly changing data (e.g., to predict stock prices), then you need a more reactive solution.

Also, training on the full set of data requires a lot of computing resources (CPU, memory space, disk space, disk I/O, network I/O, etc.). If you have a lot of data and you automate your system to train from scratch every day, it will end up costing you a lot of money. If the amount of data is huge, it may even be impossible to use a batch learning algorithm.

Finally, if your system needs to be able to learn autonomously and it has limited resources (e.g., a smartphone application or a rover on Mars), then carrying around

large amounts of training data and taking up a lot of resources to train for hours every day is a showstopper.

Fortunately, a better option in all these cases is to use algorithms that are capable of learning incrementally.

Online learning

In *online learning*, you train the system incrementally by feeding it data instances sequentially, either individually or in small groups called *mini-batches*. Each learning step is fast and cheap, so the system can learn about new data on the fly, as it arrives (see [Figure 1-14](#)).

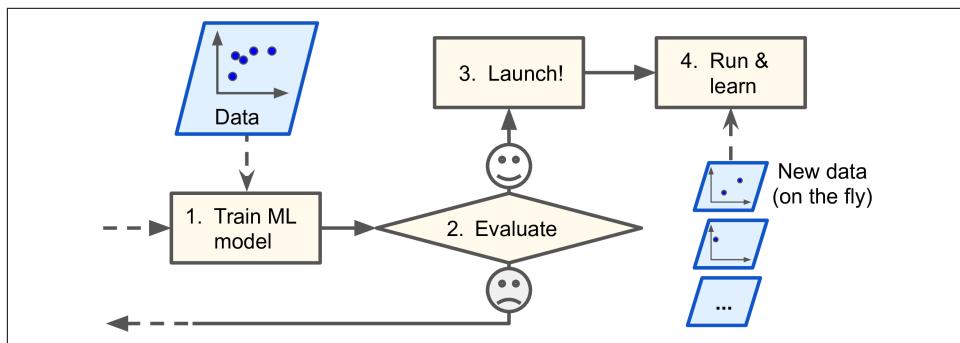


Figure 1-14. In online learning, a model is trained and launched into production, and then it keeps learning as new data comes in

Online learning is useful for systems that need to adapt to change extremely rapidly (e.g., to detect new patterns in the stock market). It is also a good option if you have limited computing resources, for example if the model is trained on a mobile device.

Online learning algorithms can also be used to train models on huge datasets that cannot fit in one machine's main memory (this is called *out-of-core* learning). The algorithm loads part of the data, runs a training step on that data, and repeats the process until it has run on all of the data (see [Figure 1-15](#)).



Out-of-core learning is usually done offline (i.e., not on the live system), so *online learning* can be a confusing name. Think of it as *incremental learning*.

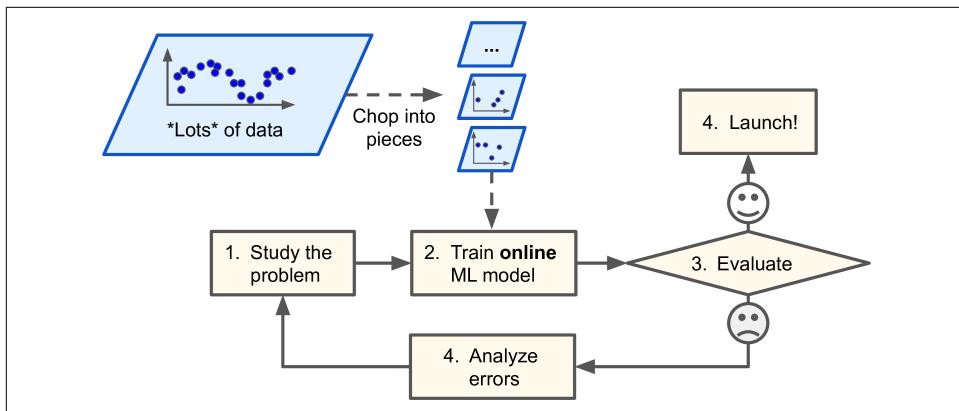


Figure 1-15. Using online learning to handle huge datasets

One important parameter of online learning systems is how fast they should adapt to changing data: this is called the *learning rate*. If you set a high learning rate, then your system will rapidly adapt to new data, but it will also tend to quickly forget the old data (you don't want a spam filter to flag only the latest kinds of spam it was shown). Conversely, if you set a low learning rate, the system will have more inertia; that is, it will learn more slowly, but it will also be less sensitive to noise in the new data or to sequences of nonrepresentative data points (outliers).

A big challenge with online learning is that if bad data is fed to the system, the system's performance will decline, possibly quickly (depending on the data quality and learning rate). If it's a live system, your clients will notice. For example, bad data could come from a bug (e.g., a malfunctioning sensor on a robot), or it could come from someone trying to game the system (e.g., spamming a search engine to try to rank high in search results). To reduce this risk, you need to monitor your system closely and promptly switch learning off (and possibly revert to a previously working state) if you detect a drop in performance. You may also want to monitor the input data and react to abnormal data, for example using an anomaly detection algorithm (see [Chapter 9](#)).

Instance-Based Versus Model-Based Learning

One more way to categorize Machine Learning systems is by how they *generalize*. Most Machine Learning tasks are about making predictions. This means that given a number of training examples, the system needs to be able to make good predictions for (generalize to) examples it has never seen before. Having a good performance measure on the training data is good, but insufficient; the true goal is to perform well on new instances.

There are two main approaches to generalization: instance-based learning and model-based learning.

Instance-based learning

Possibly the most trivial form of learning is simply to learn by heart. If you were to create a spam filter this way, it would just flag all emails that are identical to emails that have already been flagged by users—not the worst solution, but certainly not the best.

Instead of just flagging emails that are identical to known spam emails, your spam filter could be programmed to also flag emails that are very similar to known spam emails. This requires a *measure of similarity* between two emails. A (very basic) similarity measure between two emails could be to count the number of words they have in common. The system would flag an email as spam if it has many words in common with a known spam email.

This is called *instance-based learning*: the system learns the examples by heart, then generalizes to new cases by using a similarity measure to compare them to the learned examples (or a subset of them). For example, in [Figure 1-16](#) the new instance would be classified as a triangle because the majority of the most similar instances belong to that class.



Figure 1-16. Instance-based learning

Model-based learning, and a typical Machine Learning workflow

Another way to generalize from a set of examples is to build a model of these examples and then use that model to make *predictions*. This is called *model-based learning* ([Figure 1-17](#)).

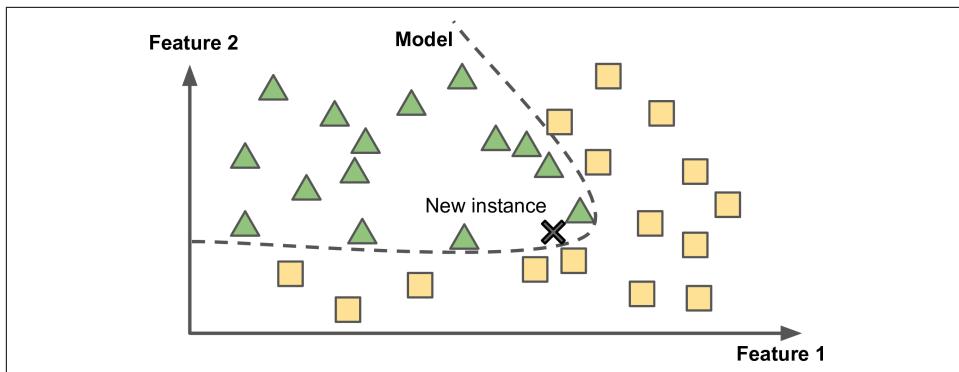


Figure 1-17. Model-based learning

For example, suppose you want to know if money makes people happy, so you download the Better Life Index data from the OECD's website and World Bank stats about gross domestic product (GDP) per capita from OurWorldInData.org. Then you join the tables and sort by GDP per capita. [Table 1-1](#) shows an excerpt of what you get.

Table 1-1. Does money make people happier?

Country	GDP per capita (USD)	Life satisfaction
Turkey	28,384	5.5
Hungary	31,008	5.6
France	42,026	6.5
United States	60,236	6.9
New Zealand	42,404	7.3
Australia	48,698	7.3
Denmark	55,938	7.6

Let's plot the data for these countries ([Figure 1-18](#)).

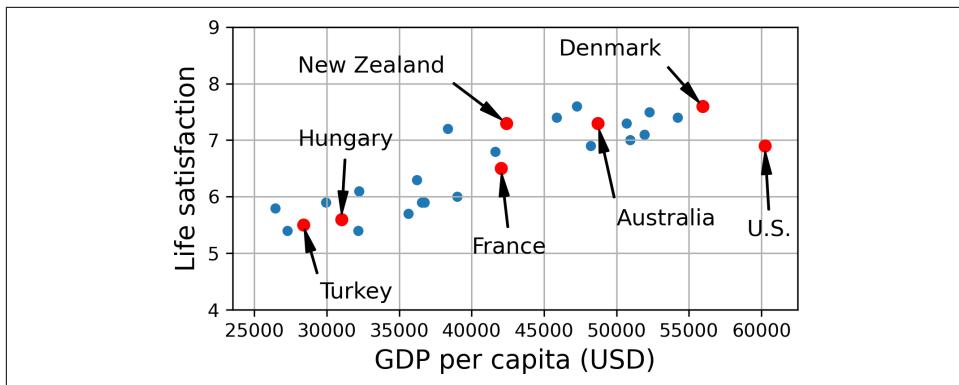


Figure 1-18. Do you see a trend here?

There does seem to be a trend here! Although the data is *noisy* (i.e., partly random), it looks like life satisfaction goes up more or less linearly as the country's GDP per capita increases. So you decide to model life satisfaction as a linear function of GDP per capita. This step is called *model selection*: you selected a *linear model* of life satisfaction with just one attribute, GDP per capita (Equation 1-1).

Equation 1-1. A simple linear model

$$\text{life_satisfaction} = \theta_0 + \theta_1 \times \text{GDP_per_capita}$$

This model has two *model parameters*, θ_0 and θ_1 .⁴ By tweaking these parameters, you can make your model represent any linear function, as shown in Figure 1-19.

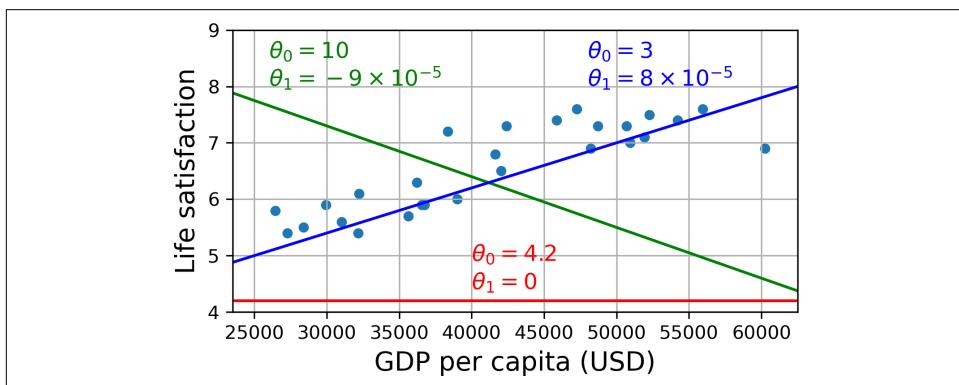


Figure 1-19. A few possible linear models

⁴ By convention, the Greek letter θ (theta) is frequently used to represent model parameters.

Before you can use your model, you need to define the parameter values θ_0 and θ_1 . How can you know which values will make your model perform best? To answer this question, you need to specify a performance measure. You can either define a *utility function* (or *fitness function*) that measures how *good* your model is, or you can define a *cost function* that measures how *bad* it is. For Linear Regression problems, people typically use a cost function that measures the distance between the linear model's predictions and the training examples; the objective is to minimize this distance.

This is where the Linear Regression algorithm comes in: you feed it your training examples, and it finds the parameters that make the linear model fit best to your data. This is called *training* the model. In our case, the algorithm finds that the optimal parameter values are $\theta_0 = 3.75$ and $\theta_1 = 6.78 \times 10^{-5}$.



Confusingly, the same word “model” can refer to a *type of model* (e.g., Linear Regression), to a *fully specified model architecture* (e.g., Linear Regression with one input and one output), or to the *final trained model* ready to be used for predictions (e.g., Linear Regression with one input and one output, using $\theta_0 = 3.75$ and $\theta_1 = 6.78 \times 10^{-5}$). Model selection consists in choosing the type of model and fully specifying its architecture. Training a model means running an algorithm to find the model parameters that will make it best fit the training data, and hopefully make good predictions on new data.

Now the model fits the training data as closely as possible (for a linear model), as you can see in [Figure 1-20](#).

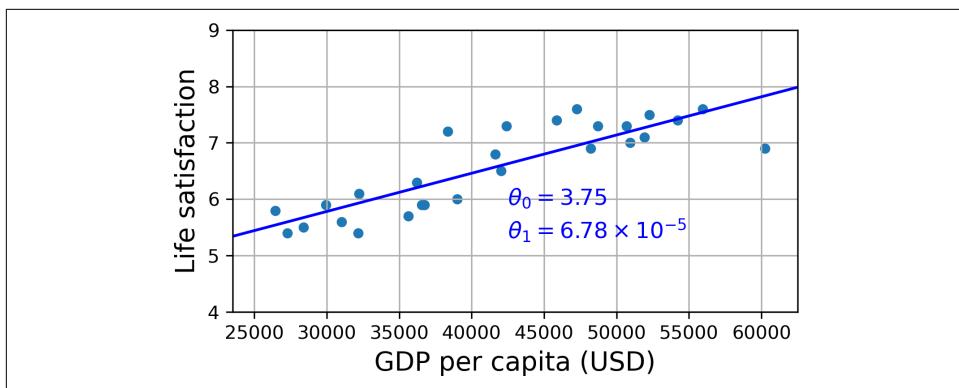


Figure 1-20. The linear model that fits the training data best

You are finally ready to run the model to make predictions. For example, say you want to know how happy Cypriots are, and the OECD data does not have the answer. Fortunately, you can use your model to make a good prediction: you look up

Cyprus's GDP per capita, find \$37,655, and then apply your model and find that life satisfaction is likely to be somewhere around $3.75 + 37,655 \times 6.78 \times 10^{-5} = 6.30$.

To whet your appetite, [Example 1-1](#) shows the Python code that loads the data, separates the inputs X from the labels y, creates a scatterplot for visualization, and then trains a linear model and makes a prediction.⁵

Example 1-1. Training and running a linear model using Scikit-Learn

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression

# Download and prepare the data
data_root = "https://github.com/ageron/data/raw/main/"
lifesat = pd.read_csv(data_root + "lifesat/lifesat.csv")
X = lifesat[["GDP per capita (USD)"]].values
y = lifesat[["Life satisfaction"]].values

# Visualize the data
lifesat.plot(kind='scatter', grid=True,
             x="GDP per capita (USD)", y="Life satisfaction")
plt.axis([23_500, 62_500, 4, 9])
plt.show()

# Select a linear model
model = LinearRegression()

# Train the model
model.fit(X, y)

# Make a prediction for Cyprus
X_new = [[37_655.2]] # Cyprus' GDP per capita in 2020
print(model.predict(X_new)) # outputs [[6.30165767]]
```

⁵ It's OK if you don't understand all the code yet; I will present Scikit-Learn in the following chapters.



If you had used an instance-based learning algorithm instead, you would have found that Israel has the closest GDP per capita to that of Cyprus (\$38,341), and since the OECD data tells us that Israelis' life satisfaction is 7.2, you would have predicted a life satisfaction of 7.2 for Cyprus. If you zoom out a bit and look at the two next-closest countries, you will find Lithuania and Slovenia, both with a life satisfaction of 5.9. Averaging these three values, you get 6.33, which is pretty close to your model-based prediction. This simple algorithm is called *k-Nearest Neighbors* regression (in this example, $k = 3$).

Replacing the Linear Regression model with k-Nearest Neighbors regression in the previous code is as simple as replacing these two lines:

```
from sklearn.linear_model import LinearRegression  
model = LinearRegression()
```

with these two:

```
from sklearn.neighbors import KNeighborsRegressor  
model = KNeighborsRegressor(n_neighbors=3)
```

If all went well, your model will make good predictions. If not, you may need to use more attributes (employment rate, health, air pollution, etc.), get more or better-quality training data, or perhaps select a more powerful model (e.g., a Polynomial Regression model).

In summary:

- You studied the data.
- You selected a model.
- You trained it on the training data (i.e., the learning algorithm searched for the model parameter values that minimize a cost function).
- Finally, you applied the model to make predictions on new cases (this is called *inference*), hoping that this model will generalize well.

This is what a typical Machine Learning project looks like. In [Chapter 2](#) you will experience this firsthand by going through a project end to end.

We have covered a lot of ground so far: you now know what Machine Learning is really about, why it is useful, what some of the most common categories of ML systems are, and what a typical project workflow looks like. Now let's look at what can go wrong in learning and prevent you from making accurate predictions.

Main Challenges of Machine Learning

In short, since your main task is to select a model and train it on some data, the two things that can go wrong are “bad model” and “bad data.” Let’s start with examples of bad data.

Insufficient Quantity of Training Data

For a toddler to learn what an apple is, all it takes is for you to point to an apple and say “apple” (possibly repeating this procedure a few times). Now the child is able to recognize apples in all sorts of colors and shapes. Genius.

Machine Learning is not quite there yet; it takes a lot of data for most Machine Learning algorithms to work properly. Even for very simple problems you typically need thousands of examples, and for complex problems such as image or speech recognition you may need millions of examples (unless you can reuse parts of an existing model).

The Unreasonable Effectiveness of Data

In a [famous paper](#) published in 2001, Microsoft researchers Michele Banko and Eric Brill showed that very different Machine Learning algorithms, including fairly simple ones, performed almost identically well on a complex problem of natural language disambiguation⁶ once they were given enough data (as you can see in [Figure 1-21](#)).

⁶ For example, knowing whether to write “to,” “two,” or “too,” depending on the context.

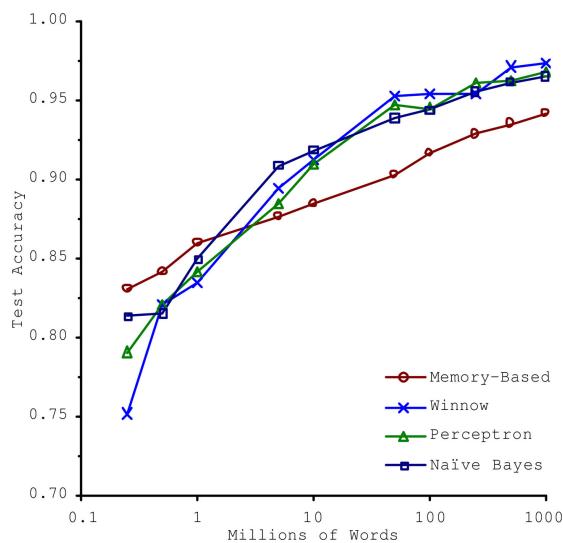


Figure 1-21. The importance of data versus algorithms⁷

As the authors put it, “these results suggest that we may want to reconsider the trade-off between spending time and money on algorithm development versus spending it on corpus development.”

The idea that data matters more than algorithms for complex problems was further popularized by Peter Norvig et al. in a paper titled “[The Unreasonable Effectiveness of Data](#)”, published in 2009.⁸ It should be noted, however, that small- and medium-sized datasets are still very common, and it is not always easy or cheap to get extra training data—so don’t abandon algorithms just yet.

Nonrepresentative Training Data

In order to generalize well, it is crucial that your training data be representative of the new cases you want to generalize to. This is true whether you use instance-based learning or model-based learning.

For example, the set of countries we used earlier for training the linear model was not perfectly representative; it did not contain any country with a GDP per capita lower

⁷ Figure reproduced with permission from Michele Banko and Eric Brill, “Scaling to Very Very Large Corpora for Natural Language Disambiguation,” *Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics* (2001): 26–33.

⁸ Peter Norvig et al., “The Unreasonable Effectiveness of Data,” *IEEE Intelligent Systems* 24, no. 2 (2009): 8–12.

than \$23,500 or higher than \$62,500. Figure 1-22 shows what the data looks like when you add such countries.

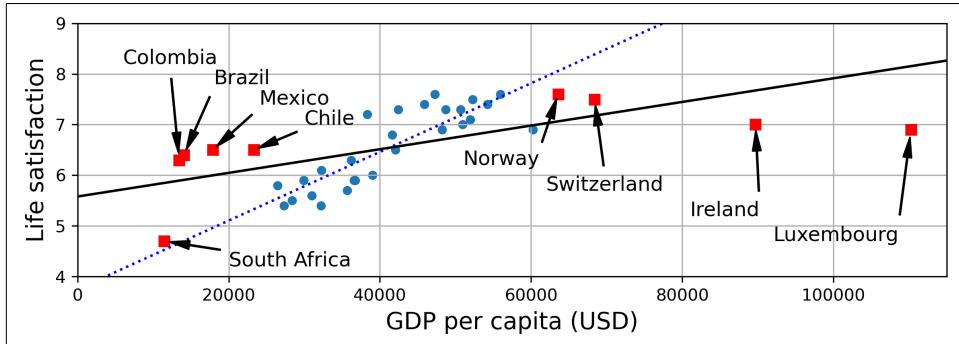


Figure 1-22. A more representative training sample

If you train a linear model on this data, you get the solid line, while the old model is represented by the dotted line. As you can see, not only does adding a few missing countries significantly alter the model, but it makes it clear that such a simple linear model is probably never going to work well. It seems that very rich countries are not happier than moderately rich countries (in fact, they seem slightly unhappier!), and conversely some poor countries seem happier than many rich countries.

By using a nonrepresentative training set, we trained a model that is unlikely to make accurate predictions, especially for very poor and very rich countries.

It is crucial to use a training set that is representative of the cases you want to generalize to. This is often harder than it sounds: if the sample is too small, you will have *sampling noise* (i.e., nonrepresentative data as a result of chance), but even very large samples can be nonrepresentative if the sampling method is flawed. This is called *sampling bias*.

Examples of Sampling Bias

Perhaps the most famous example of sampling bias happened during the US presidential election in 1936, which pitted Landon against Roosevelt: the *Literary Digest* conducted a very large poll, sending mail to about 10 million people. It got 2.4 million answers, and predicted with high confidence that Landon would get 57% of the votes. Instead, Roosevelt won with 62% of the votes. The flaw was in the *Literary Digest's* sampling method:

- First, to obtain the addresses to send the polls to, the *Literary Digest* used telephone directories, lists of magazine subscribers, club membership lists, and the like. All of these lists tended to favor wealthier people, who were more likely to vote Republican (hence Landon).

- Second, less than 25% of the people who were polled answered. Again this introduced a sampling bias, by potentially ruling out people who didn't care much about politics, people who didn't like the *Literary Digest*, and other key groups. This is a special type of sampling bias called *nonresponse bias*.

Here is another example: say you want to build a system to recognize funk music videos. One way to build your training set is to search for “funk music” on YouTube and use the resulting videos. But this assumes that YouTube’s search engine returns a set of videos that are representative of all the funk music videos on YouTube. In reality, the search results are likely to be biased toward popular artists (and if you live in Brazil you will get a lot of “funk carioca” videos, which sound nothing like James Brown). On the other hand, how else can you get a large training set?

Poor-Quality Data

Obviously, if your training data is full of errors, outliers, and noise (e.g., due to poor-quality measurements), it will make it harder for the system to detect the underlying patterns, so your system is less likely to perform well. It is often well worth the effort to spend time cleaning up your training data. The truth is, most data scientists spend a significant part of their time doing just that. The following are a couple examples of when you’d want to clean up training data:

- If some instances are clearly outliers, it may help to simply discard them or try to fix the errors manually.
- If some instances are missing a few features (e.g., 5% of your customers did not specify their age), you must decide whether you want to ignore this attribute altogether, ignore these instances, fill in the missing values (e.g., with the median age), or train one model with the feature and one model without it.

Irrelevant Features

As the saying goes: garbage in, garbage out. Your system will only be capable of learning if the training data contains enough relevant features and not too many irrelevant ones. A critical part of the success of a Machine Learning project is coming up with a good set of features to train on. This process, called *feature engineering*, involves the following steps:

- *Feature selection* (selecting the most useful features to train on among existing features)
- *Feature extraction* (combining existing features to produce a more useful one—as we saw earlier, dimensionality reduction algorithms can help)
- Creating new features by gathering new data

Now that we have looked at many examples of bad data, let's look at a couple examples of bad algorithms.

Overfitting the Training Data

Say you are visiting a foreign country and the taxi driver rips you off. You might be tempted to say that *all* taxi drivers in that country are thieves. Overgeneralizing is something that we humans do all too often, and unfortunately machines can fall into the same trap if we are not careful. In Machine Learning this is called *overfitting*: it means that the model performs well on the training data, but it does not generalize well.

Figure 1-23 shows an example of a high-degree polynomial life satisfaction model that strongly overfits the training data. Even though it performs much better on the training data than the simple linear model, would you really trust its predictions?

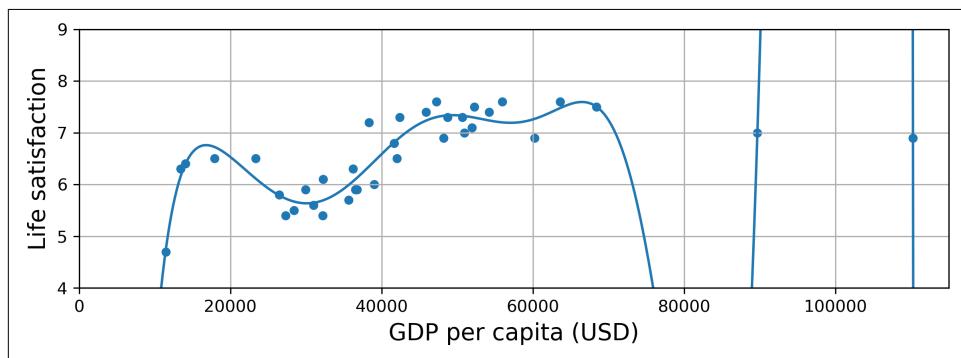


Figure 1-23. Overfitting the training data

Complex models such as deep neural networks can detect subtle patterns in the data, but if the training set is noisy, or if it is too small (as in the taxi driver example) which introduces sampling noise, then the model is likely to detect patterns in the noise itself. Obviously these patterns will not generalize to new instances. For example, say you feed your life satisfaction model many more attributes, including uninformative ones such as the country's name. In that case, a complex model may detect patterns like the fact that all countries in the training data with a *w* in their name have a life satisfaction greater than 7: New Zealand (7.3), Norway (7.6), Sweden (7.3), and Switzerland (7.5). How confident are you that the *w*-satisfaction rule generalizes to Rwanda or Zimbabwe? Obviously this pattern occurred in the training data by pure chance, but the model has no way to tell whether a pattern is real or simply the result of noise in the data.



Overfitting happens when the model is too complex relative to the amount and noisiness of the training data. Here are possible solutions:

- Simplify the model by selecting one with fewer parameters (e.g., a linear model rather than a high-degree polynomial model), by reducing the number of attributes in the training data, or by constraining the model.
- Gather more training data.
- Reduce the noise in the training data (e.g., fix data errors and remove outliers).

Constraining a model to make it simpler and reduce the risk of overfitting is called *regularization*. For example, the linear model we defined earlier has two parameters, θ_0 and θ_1 . This gives the learning algorithm two *degrees of freedom* to adapt the model to the training data: it can tweak both the height (θ_0) and the slope (θ_1) of the line. If we forced $\theta_1 = 0$, the algorithm would have only one degree of freedom and would have a much harder time fitting the data properly: all it could do is move the line up or down to get as close as possible to the training instances, so it would end up around the mean. A very simple model indeed! If we allow the algorithm to modify θ_1 but we force it to keep it small, then the learning algorithm will effectively have somewhere in between one and two degrees of freedom. It will produce a model that's simpler than one with two degrees of freedom, but more complex than one with just one. You want to find the right balance between fitting the training data perfectly and keeping the model simple enough to ensure that it will generalize well.

Figure 1-24 shows three models. The dotted line represents the original model that was trained on the countries represented as circles (without the countries represented as squares), the solid line is our second model trained with all countries (circles and squares), and the dashed line is a model trained with the same data as the first model but with a regularization constraint. You can see that regularization forced the model to have a smaller slope: this model does not fit the training data (circles) as well as the first model, but it actually generalizes better to new examples that it did not see during training (squares).

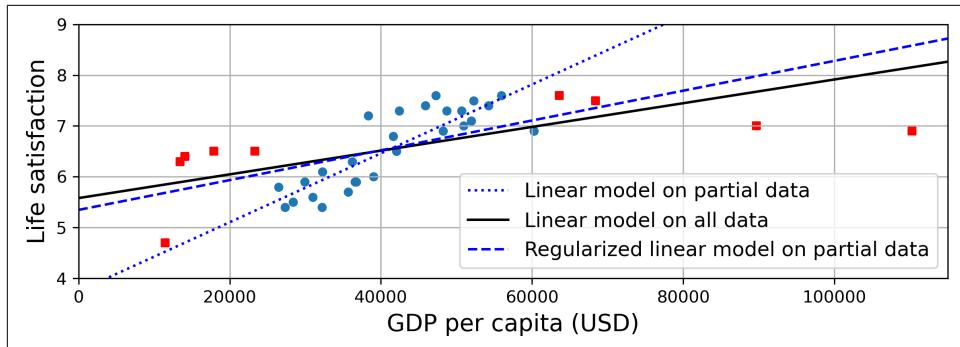


Figure 1-24. Regularization reduces the risk of overfitting

The amount of regularization to apply during learning can be controlled by a *hyperparameter*. A hyperparameter is a parameter of a learning algorithm (not of the model). As such, it is not affected by the learning algorithm itself; it must be set prior to training and remains constant during training. If you set the regularization hyperparameter to a very large value, you will get an almost flat model (a slope close to zero); the learning algorithm will almost certainly not overfit the training data, but it will be less likely to find a good solution. Tuning hyperparameters is an important part of building a Machine Learning system (you will see a detailed example in the next chapter).

Underfitting the Training Data

As you might guess, *underfitting* is the opposite of overfitting; it occurs when your model is too simple to learn the underlying structure of the data. For example, a linear model of life satisfaction is prone to underfit; reality is just more complex than the model, so its predictions are bound to be inaccurate, even on the training examples.

Here are the main options for fixing this problem:

- Select a more powerful model, with more parameters.
- Feed better features to the learning algorithm (feature engineering).
- Reduce the constraints on the model (e.g., reduce the regularization hyperparameter).

Stepping Back

By now you know a lot about Machine Learning. However, we went through so many concepts that you may be feeling a little lost, so let's step back and look at the big picture:

- Machine Learning is about making machines get better at some task by learning from data, instead of having to explicitly code rules.
- There are many different types of ML systems: supervised or not, batch or online, instance-based or model-based.
- In an ML project you gather data in a training set, and you feed the training set to a learning algorithm. If the algorithm is model-based, it tunes some parameters to fit the model to the training set (i.e., to make good predictions on the training set itself), and then hopefully it will be able to make good predictions on new cases as well. If the algorithm is instance-based, it just learns the examples by heart and generalizes to new instances by using a similarity measure to compare them to the learned instances.
- The system will not perform well if your training set is too small, or if the data is not representative, is noisy, or is polluted with irrelevant features (garbage in, garbage out). Lastly, your model needs to be neither too simple (in which case it will underfit) nor too complex (in which case it will overfit).

There's just one last important topic to cover: once you have trained a model, you don't want to just "hope" it generalizes to new cases. You want to evaluate it and fine-tune it if necessary. Let's see how to do that.

Testing and Validating

The only way to know how well a model will generalize to new cases is to actually try it out on new cases. One way to do that is to put your model in production and monitor how well it performs. This works well, but if your model is horribly bad, your users will complain—not the best idea.

A better option is to split your data into two sets: the *training set* and the *test set*. As these names imply, you train your model using the training set, and you test it using the test set. The error rate on new cases is called the *generalization error* (or *out-of-sample error*), and by evaluating your model on the test set, you get an estimate of this error. This value tells you how well your model will perform on instances it has never seen before.

If the training error is low (i.e., your model makes few mistakes on the training set) but the generalization error is high, it means that your model is overfitting the training data.



It is common to use 80% of the data for training and *hold out* 20% for testing. However, this depends on the size of the dataset: if it contains 10 million instances, then holding out 1% means your test set will contain 100,000 instances, probably more than enough to get a good estimate of the generalization error.

Hyperparameter Tuning and Model Selection

Evaluating a model is simple enough: just use a test set. But suppose you are hesitating between two types of models (say, a linear model and a polynomial model): how can you decide between them? One option is to train both and compare how well they generalize using the test set.

Now suppose that the linear model generalizes better, but you want to apply some regularization to avoid overfitting. The question is, how do you choose the value of the regularization hyperparameter? One option is to train 100 different models using 100 different values for this hyperparameter. Suppose you find the best hyperparameter value that produces a model with the lowest generalization error—say, just 5% error. You launch this model into production, but unfortunately it does not perform as well as expected and produces 15% errors. What just happened?

The problem is that you measured the generalization error multiple times on the test set, and you adapted the model and hyperparameters to produce the best model *for that particular set*. This means that the model is unlikely to perform as well on new data.

A common solution to this problem is called *holdout validation* (Figure 1-25): you simply hold out part of the training set to evaluate several candidate models and select the best one. The new held-out set is called the *validation set* (or the *development set*, or *dev set*). More specifically, you train multiple models with various hyperparameters on the reduced training set (i.e., the full training set minus the validation set), and you select the model that performs best on the validation set. After this holdout validation process, you train the best model on the full training set (including the validation set), and this gives you the final model. Lastly, you evaluate this final model on the test set to get an estimate of the generalization error.

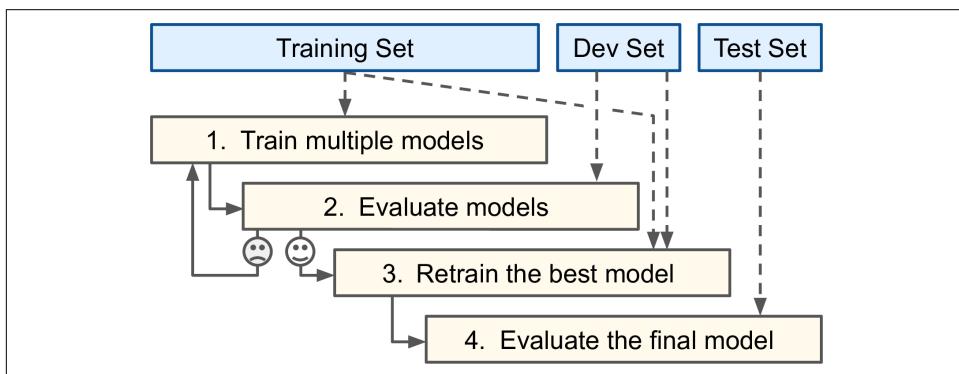


Figure 1-25. Model selection using holdout validation

This solution usually works quite well. However, if the validation set is too small, then model evaluations will be imprecise: you may end up selecting a suboptimal model by mistake. Conversely, if the validation set is too large, then the remaining training set will be much smaller than the full training set. Why is this bad? Well, since the final model will be trained on the full training set, it is not ideal to compare candidate models trained on a much smaller training set. It would be like selecting the fastest sprinter to participate in a marathon. One way to solve this problem is to perform repeated *cross-validation*, using many small validation sets. Each model is evaluated once per validation set after it is trained on the rest of the data. By averaging out all the evaluations of a model, you get a much more accurate measure of its performance. There is a drawback, however: the training time is multiplied by the number of validation sets.

Data Mismatch

In some cases, it's easy to get a large amount of data for training, but this data probably won't be perfectly representative of the data that will be used in production. For example, suppose you want to create a mobile app to take pictures of flowers and automatically determine their species. You can easily download millions of pictures of flowers on the web, but they won't be perfectly representative of the pictures that will actually be taken using the app on a mobile device. Perhaps you only have 1,000 representative pictures (i.e., actually taken with the app).

In this case, the most important rule to remember is that both the validation set and the test set must be as representative as possible of the data you expect to use in production, so they should be composed exclusively of representative pictures: you can shuffle them and put half in the validation set and half in the test set (making sure that no duplicates or near-duplicates end up in both sets). After training your model on the web pictures, if you observe that the performance of the model on the validation set is disappointing, you will not know whether this is because your model has overfit the training set, or whether this is just due to the mismatch between the web pictures and the mobile app pictures.

One solution is to hold out some of the training pictures (from the web) in yet another set that Andrew Ng dubbed the *train-dev set* (Figure 1-26). After the model is trained (on the training set, *not* on the train-dev set), you can evaluate it on the train-dev set. If the model performs poorly, then it must have overfit the training set, so you should try to simplify or regularize the model, get more training data, and clean up the training data. But if it performs well on the train-dev set, then you can evaluate the model on the dev set. If it performs poorly, then the problem must be coming from the data mismatch. You can try to tackle this problem by preprocessing the web images to make them look more like the pictures that will be taken by the mobile app, and then retraining the model. Once you have a model that performs

well on both the train-dev set and the dev set, you can evaluate it one last time on the test set to know how well it is likely to perform in production.

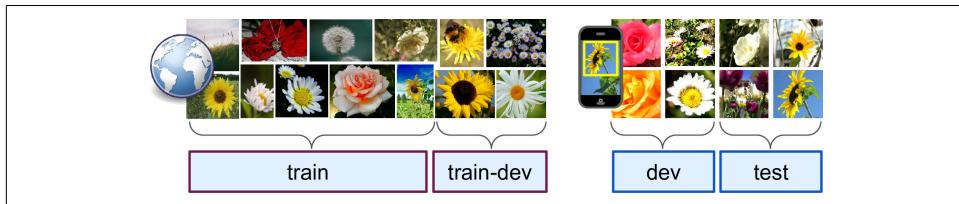


Figure 1-26. When real data is scarce (right), you may use similar abundant data (left) for training and hold out some of it in a train-dev set to evaluate overfitting. The real data is then used to evaluate data-mismatch (dev set), and to evaluate the final model's performance (test set).

No Free Lunch Theorem

A model is a simplified representation of the data. The simplifications are meant to discard the superfluous details that are unlikely to generalize to new instances. When you select a particular type of model, you are implicitly making *assumptions* about the data. For example, if you choose a linear model, you are implicitly assuming that the data is fundamentally linear and that the distance between the instances and the straight line is just noise, which can safely be ignored.

In a [famous 1996 paper](#),⁹ David Wolpert demonstrated that if you make absolutely no assumption about the data, then there is no reason to prefer one model over any other. This is called the *No Free Lunch* (NFL) theorem. For some datasets the best model is a linear model, while for other datasets it is a neural network. There is no model that is *a priori* guaranteed to work better (hence the name of the theorem). The only way to know for sure which model is best is to evaluate them all. Since this is not possible, in practice you make some reasonable assumptions about the data and evaluate only a few reasonable models. For example, for simple tasks you may evaluate linear models with various levels of regularization, and for a complex problem you may evaluate various neural networks.

Exercises

In this chapter we have covered some of the most important concepts in Machine Learning. In the next chapters we will dive deeper and write more code, but before we do, make sure you can answer the following questions:

⁹ David Wolpert, “The Lack of A Priori Distinctions Between Learning Algorithms,” *Neural Computation* 8, no. 7 (1996): 1341–1390.

1. How would you define Machine Learning?
2. Can you name four types of applications where it shines?
3. What is a labeled training set?
4. What are the two most common supervised tasks?
5. Can you name four common unsupervised tasks?
6. What type of algorithm would you use to allow a robot to walk in various unknown terrains?
7. What type of algorithm would you use to segment your customers into multiple groups?
8. Would you frame the problem of spam detection as a supervised learning problem or an unsupervised learning problem?
9. What is an online learning system?
10. What is out-of-core learning?
11. What type of algorithm relies on a similarity measure to make predictions?
12. What is the difference between a model parameter and a model hyperparameter?
13. What do model-based algorithms search for? What is the most common strategy they use to succeed? How do they make predictions?
14. Can you name four of the main challenges in Machine Learning?
15. If your model performs great on the training data but generalizes poorly to new instances, what is happening? Can you name three possible solutions?
16. What is a test set, and why would you want to use it?
17. What is the purpose of a validation set?
18. What is the train-dev set, when do you need it, and how do you use it?
19. What can go wrong if you tune hyperparameters using the test set?

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab3>.

End-to-End Machine Learning Project

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. The GitHub repo is [https://git
hub.com/ageron/handsonml3](https://github.com/ageron/handsonml3).

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

In this chapter you will work through an example project end to end, pretending to be a recently hired data scientist at a real estate company. The example project is fictitious; the goal is to illustrate the main steps of a Machine Learning project, not to learn anything about the real estate business. Here are the main steps we will walk through:

1. Look at the big picture.
2. Get the data.
3. Explore and visualize the data to gain insights.
4. Prepare the data for Machine Learning algorithms.
5. Select a model and train it.
6. Fine-tune your model.
7. Present your solution.

8. Launch, monitor, and maintain your system.

Working with Real Data

When you are learning about Machine Learning, it is best to experiment with real-world data, not artificial datasets. Fortunately, there are thousands of open datasets to choose from, ranging across all sorts of domains. Here are a few places you can look to get data:

- Popular open data repositories
 - [OpenML.org](#)
 - [Kaggle.com](#)
 - [PapersWithCode.com](#)
 - [UC Irvine Machine Learning Repository](#)
 - [Amazon's AWS datasets](#)
 - [TensorFlow Datasets](#)
- Meta portals (they list open data repositories)
 - [DataPortals.org](#)
 - [OpenDataMonitor.eu](#)
- Other pages listing many popular open data repositories
 - [Wikipedia's list of Machine Learning datasets](#)
 - [Quora.com](#)
 - [The datasets subreddit](#)

In this chapter we'll use the California Housing Prices dataset from the StatLib repository¹ (see [Figure 2-1](#)). This dataset is based on data from the 1990 California census. It is not exactly recent (a nice house in the Bay Area was still affordable at the time), but it has many qualities for learning, so we will pretend it is recent data. For teaching purposes I've added a categorical attribute and removed a few features.

¹ The original dataset appeared in R. Kelley Pace and Ronald Barry, "Sparse Spatial Autoregressions," *Statistics & Probability Letters* 33, no. 3 (1997): 291–297.

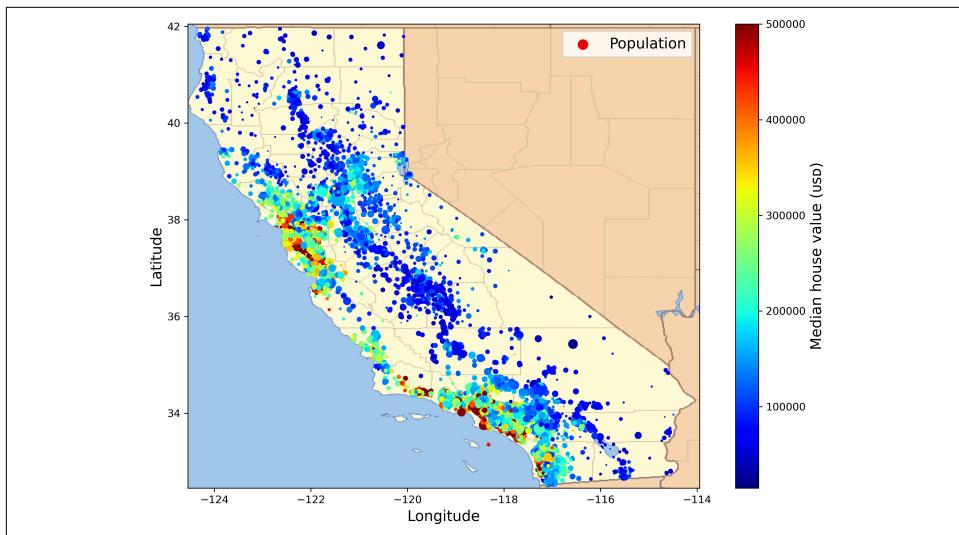


Figure 2-1. California housing prices

Look at the Big Picture

Welcome to the Machine Learning Housing Corporation! Your first task is to use California census data to build a model of housing prices in the state. This data includes metrics such as the population, median income, and median housing price for each block group in California. Block groups are the smallest geographical unit for which the US Census Bureau publishes sample data (a block group typically has a population of 600 to 3,000 people). I will call them “districts” for short.

Your model should learn from this data and be able to predict the median housing price in any district, given all the other metrics.



Since you are a well-organized data scientist, the first thing you should do is pull out your Machine Learning project checklist. You can start with the one in [Appendix A](#); it should work reasonably well for most Machine Learning projects, but make sure to adapt it to your needs. In this chapter we will go through many checklist items, but we will also skip a few, either because they are self-explanatory or because they will be discussed in later chapters.

Frame the Problem

The first question to ask your boss is what exactly the business objective is. Building a model is probably not the end goal. How does the company expect to use and benefit from this model? Knowing the objective is important because it will determine

how you frame the problem, which algorithms you will select, which performance measure you will use to evaluate your model, and how much effort you will spend tweaking it.

Your boss answers that your model's output (a prediction of a district's median housing price) will be fed to another Machine Learning system (see [Figure 2-2](#)), along with many other signals.² This downstream system will determine whether it is worth investing in a given area or not. Getting this right is critical, as it directly affects revenue.

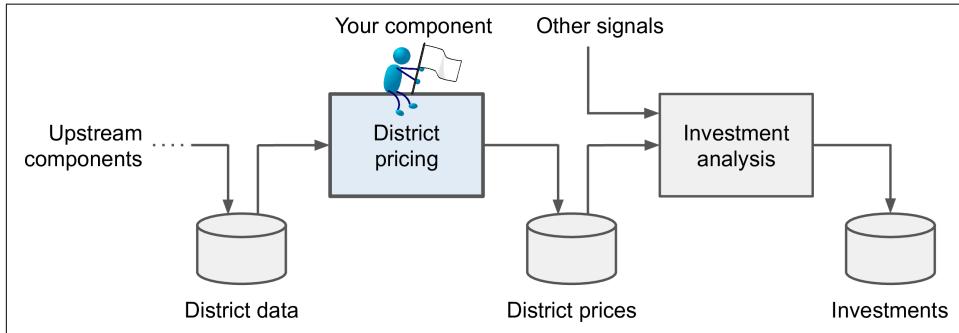


Figure 2-2. A Machine Learning pipeline for real estate investments

Pipelines

A sequence of data processing components is called a data *pipeline*. Pipelines are very common in Machine Learning systems, since there is a lot of data to manipulate and many data transformations to apply.

Components typically run asynchronously. Each component pulls in a large amount of data, processes it, and spits out the result in another data store. Then, some time later, the next component in the pipeline pulls this data and spits out its own output. Each component is fairly self-contained: the interface between components is simply the data store. This makes the system simple to grasp (with the help of a data flow graph), and different teams can focus on different components. Moreover, if a component breaks down, the downstream components can often continue to run normally (at least for a while) by just using the last output from the broken component. This makes the architecture quite robust.

² A piece of information fed to a Machine Learning system is often called a *signal*, in reference to Claude Shannon's information theory, which he developed at Bell Labs to improve telecommunications. His theory: you want a high signal-to-noise ratio.

On the other hand, a broken component can go unnoticed for some time if proper monitoring is not implemented. The data gets stale and the overall system's performance drops.

The next question to ask your boss is what the current solution looks like (if any). The current situation will often give you a reference for performance, as well as insights on how to solve the problem. Your boss answers that the district housing prices are currently estimated manually by experts: a team gathers up-to-date information about a district, and when they cannot get the median housing price, they estimate it using complex rules.

This is costly and time-consuming, and their estimates are not great; in cases where they manage to find out the actual median housing price, they often realize that their estimates were off by more than 30%. This is why the company thinks that it would be useful to train a model to predict a district's median housing price, given other data about that district. The census data looks like a great dataset to exploit for this purpose, since it includes the median housing prices of thousands of districts, as well as other data.

With all this information, you are now ready to start designing your system. First, determine what kind of training supervision the model will need: is it supervised, unsupervised, semi-supervised, self-supervised, or Reinforcement Learning? And is it a classification task, a regression task, or something else? Should you use batch learning or online learning techniques? Before you read on, pause and try to answer these questions for yourself.

Have you found the answers? Let's see: it is clearly a typical supervised learning task, since the model can be trained with *labeled* examples (each instance comes with the expected output, i.e., the district's median housing price). It is also a typical regression task, since the model will be asked to predict a value. More specifically, this is a *multiple regression* problem, since the system will use multiple features to make a prediction (it will use the district's population, the median income, etc.). It is also a *univariate regression* problem, since we are only trying to predict a single value for each district. If we were trying to predict multiple values per district, it would be a *multivariate regression* problem. Finally, there is no continuous flow of data coming into the system, there is no particular need to adjust to changing data rapidly, and the data is small enough to fit in memory, so plain batch learning should do just fine.



If the data were huge, you could either split your batch learning work across multiple servers (using the MapReduce technique) or use an online learning technique.

Select a Performance Measure

Your next step is to select a performance measure. A typical performance measure for regression problems is the Root Mean Square Error (RMSE). It gives an idea of how much error the system typically makes in its predictions, with a higher weight given to large errors. [Equation 2-1](#) shows the mathematical formula to compute the RMSE.

Equation 2-1. Root Mean Square Error (RMSE)

$$\text{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m} \sum_{i=1}^m (h(\mathbf{x}^{(i)}) - y^{(i)})^2}$$

Notations

This equation introduces several very common Machine Learning notations that I will use throughout this book:

- m is the number of instances in the dataset you are measuring the RMSE on.
 - For example, if you are evaluating the RMSE on a validation set of 2,000 districts, then $m = 2,000$.
- $\mathbf{x}^{(i)}$ is a vector of all the feature values (excluding the label) of the i^{th} instance in the dataset, and $y^{(i)}$ is its label (the desired output value for that instance).
 - For example, if the first district in the dataset is located at longitude -118.29° , latitude 33.91° , and it has 1,416 inhabitants with a median income of \$38,372, and the median house value is \$156,400 (ignoring the other features for now), then:

$$\mathbf{x}^{(1)} = \begin{pmatrix} -118.29 \\ 33.91 \\ 1,416 \\ 38,372 \end{pmatrix}$$

and:

$$y^{(1)} = 156,400$$

- \mathbf{X} is a matrix containing all the feature values (excluding labels) of all instances in the dataset. There is one row per instance, and the i^{th} row is equal to the transpose of $\mathbf{x}^{(i)}$, noted $(\mathbf{x}^{(i)})^T$.³

— For example, if the first district is as just described, then the matrix \mathbf{X} looks like this:

$$\mathbf{X} = \begin{pmatrix} (\mathbf{x}^{(1)})^T \\ (\mathbf{x}^{(2)})^T \\ \vdots \\ (\mathbf{x}^{(1999)})^T \\ (\mathbf{x}^{(2000)})^T \end{pmatrix} = \begin{pmatrix} -118.29 & 33.91 & 1,416 & 38,372 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

- h is your system's prediction function, also called a *hypothesis*. When your system is given an instance's feature vector $\mathbf{x}^{(i)}$, it outputs a predicted value $\hat{y}^{(i)} = h(\mathbf{x}^{(i)})$ for that instance (\hat{y} is pronounced "y-hat").
 - For example, if your system predicts that the median housing price in the first district is \$158,400, then $\hat{y}^{(1)} = h(\mathbf{x}^{(1)}) = 158,400$. The prediction error for this district is $\hat{y}^{(1)} - y^{(1)} = 2,000$.
- $\text{RMSE}(\mathbf{X}, h)$ is the cost function measured on the set of examples using your hypothesis h .

We use lowercase italic font for scalar values (such as m or $y^{(i)}$) and function names (such as h), lowercase bold font for vectors (such as $\mathbf{x}^{(i)}$), and uppercase bold font for matrices (such as \mathbf{X}).

Even though the RMSE is generally the preferred performance measure for regression tasks, in some contexts you may prefer to use another function. For example, suppose that there are many outlier districts. In that case, you may consider using the *mean absolute error* (MAE, also called the average absolute deviation; see [Equation 2-2](#)):

Equation 2-2. Mean absolute error (MAE)

$$\text{MAE}(\mathbf{X}, h) = \frac{1}{m} \sum_{i=1}^m |h(\mathbf{x}^{(i)}) - y^{(i)}|$$

³ Recall that the transpose operator flips a column vector into a row vector (and vice versa).

Both the RMSE and the MAE are ways to measure the distance between two vectors: the vector of predictions and the vector of target values. Various distance measures, or *norms*, are possible:

- Computing the root of a sum of squares (RMSE) corresponds to the *Euclidean norm*: this is the notion of distance you are familiar with. It is also called the ℓ_2 norm, noted $\|\cdot\|_2$ (or just $\|\cdot\|$).
- Computing the sum of absolutes (MAE) corresponds to the ℓ_1 norm, noted $\|\cdot\|_1$. This is sometimes called the *Manhattan norm* because it measures the distance between two points in a city if you can only travel along orthogonal city blocks.
- More generally, the ℓ_k norm of a vector \mathbf{v} containing n elements is defined as $\|\mathbf{v}\|_k = (|v_0|^k + |v_1|^k + \dots + |v_n|^k)^{1/k}$. ℓ_0 gives the number of nonzero elements in the vector, and ℓ_∞ gives the maximum absolute value in the vector.
- The higher the norm index, the more it focuses on large values and neglects small ones. This is why the RMSE is more sensitive to outliers than the MAE. But when outliers are exponentially rare (like in a bell-shaped curve), the RMSE performs very well and is generally preferred.

Check the Assumptions

Lastly, it is good practice to list and verify the assumptions that have been made so far (by you or others); this can help you catch serious issues early on. For example, the district prices that your system outputs are going to be fed into a downstream Machine Learning system, and you assume that these prices are going to be used as such. But what if the downstream system converts the prices into categories (e.g., “cheap,” “medium,” or “expensive”) and then uses those categories instead of the prices themselves? In this case, getting the price perfectly right is not important at all; your system just needs to get the category right. If that’s so, then the problem should have been framed as a classification task, not a regression task. You don’t want to find this out after working on a regression system for months.

Fortunately, after talking with the team in charge of the downstream system, you are confident that they do indeed need the actual prices, not just categories. Great! You’re all set, the lights are green, and you can start coding now!

Get the Data

It’s time to get your hands dirty. Don’t hesitate to pick up your laptop and walk through the code examples. As I mentioned in the preface, all the code examples in this book are open source and available online at <https://github.com/ageron/handson-ml3>, as Jupyter notebooks, which are interactive documents containing text, images, and executable code snippets (Python in our case). In this book I will assume you are

running these notebooks on Google Colab, a free service that lets you run any Jupyter notebook directly online, without having to install anything on your machine. But if you want to use another online platform (e.g., Kaggle kernels) or if you want to install everything locally on your own machine, please see the instructions on this project's home page at the GitHub page (above).

Running the Code Examples Using Google Colab

First, open a web browser and visit <https://homl.info/colab3>: this will lead you to Google Colab, and it will display the list of Jupyter notebooks for this book (see [Figure 2-3](#)). You will find one notebook per chapter, plus a few extra notebooks, including an extra chapter on Support Vector Machines, and tutorials for NumPy, Matplotlib, Pandas, Linear Algebra and Differential Calculus.

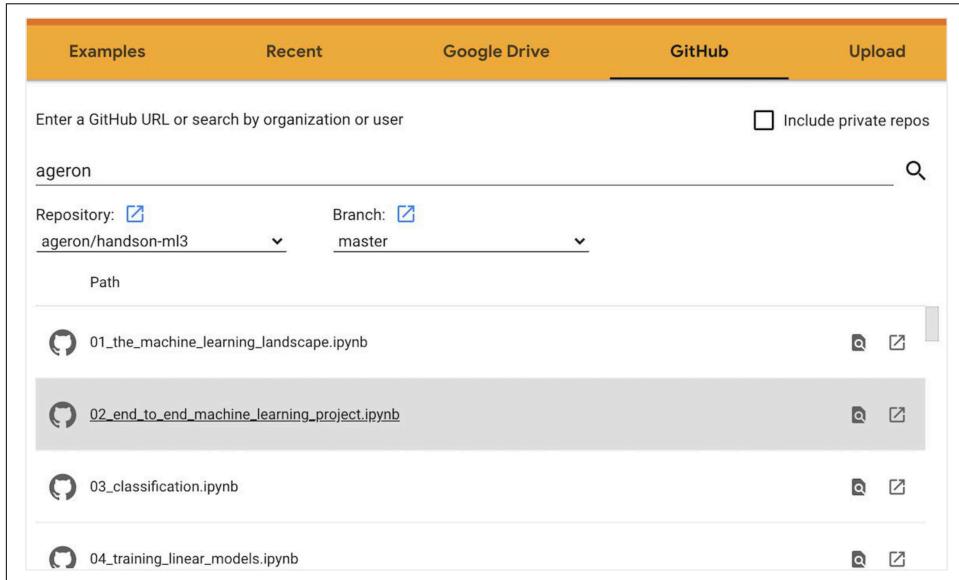


Figure 2-3. List of notebooks in Google Colab

For example, if you click on `02_end_to_end_machine_learning_project.ipynb`, the notebook from [Chapter 2](#) will open up in Google Colab (see [Figure 2-4](#)).

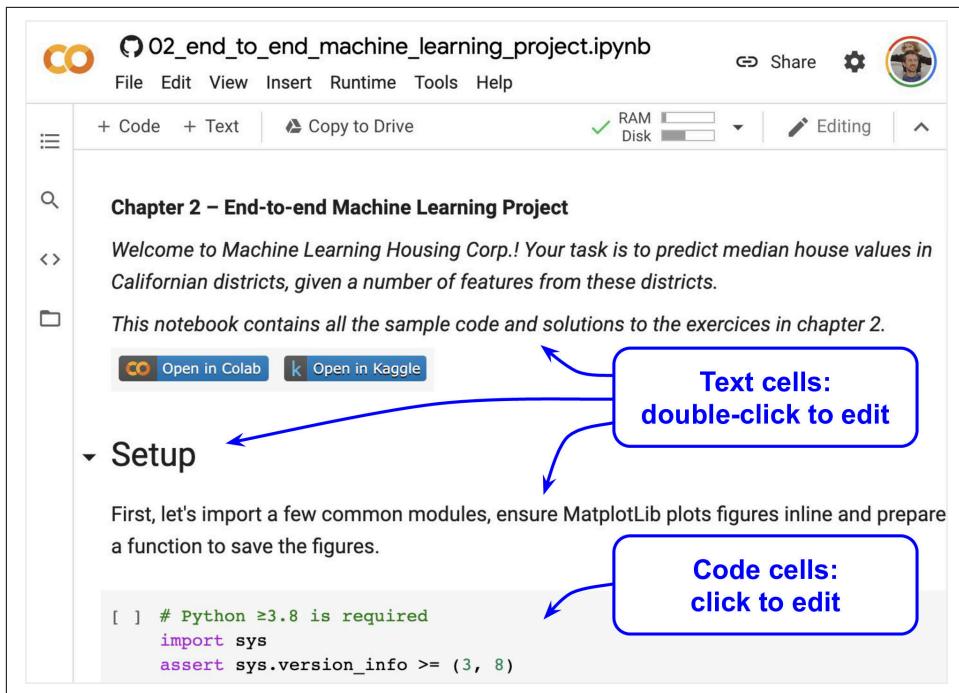


Figure 2-4. Your notebook in Google Colab

A Jupyter notebook is composed of a list of cells. Each cell contains either executable code or text. Try double-clicking on the first text cell (which contains the sentence “Welcome to Machine Learning Housing Corp.”). This will open the cell for editing. Notice that Jupyter notebooks use Markdown syntax for formatting (e.g., **bold**, *italics*, # Title, [url](link text), and so on). Try modifying this text, then press **Shift-Enter** to see the result.

Next, create a new code cell by selecting **Insert > Code cell** from the menu. Alternatively, you can click the + Code button in the toolbar, or hover your mouse over the bottom of a cell until you see + Code and + Text appear, then click on + Code. In the new code cell, type some Python code, such as `print("Hello World")` then press **Shift-Enter** to run this code (or click on the ▶ button on the left side of the cell).

If you are not logged in to your Google Account, you will be asked to login now (if you don’t already have a Google account, you will need to create one). Once you are logged in and you try to run the code, you will get a security warning telling you that this notebook was not authored by Google. Indeed, a malicious person could create a notebook that tries to trick you into entering your Google credentials so they can access your personal data. So before you run a notebook, always make sure you trust its author (or double-check what each code cell will do before running it). Now,

assuming you trust me (or you plan to check every code cell), you can go ahead and click “Run anyway”.

Colab will then allocate a new *Runtime* for you: this is a free virtual machine located on Google’s servers, containing a bunch of tools and Python libraries, including everything we need for most chapters (in some chapters, we will need to run a command to install some additional libraries). This will take a few seconds. Next, Colab will automatically connect to this Runtime and use it to execute your new code cell. Importantly, the code runs on the Runtime, *not* on your machine. The code’s output will be displayed under the cell. Congrats, you’ve run some Python code on Colab!



To insert a new code cell, you can also type **Ctrl-m** (or **Cmd-m** on macOS) followed by **a** (to insert above the active cell) or **b** (to insert below). There are many other keyboard shortcuts available: you can view and edit them by typing **Ctrl-m** (or **Cmd-m**) then **h**. If you choose to run the notebooks on Kaggle or on your own machine using JupyterLab or an IDE such as Visual Studio Code with the Jupyter extension, you will see some minor differences: Runtimes are called *Kernels*, the User Interface and keyboard shortcuts are slightly different, etc. But switching from one Jupyter environment to another is not too hard.

Saving Your Code Changes and Your Data

You can make changes to a Colab notebook, and they will persist for as long as you keep your browser tab open, but once you close it, the changes will be lost. To avoid this, make sure you save a copy of the notebook to your Google Drive by selecting File > Save a copy in Drive. Alternatively, you can download the notebook to your computer by selecting File > Download > Download .ipynb. Either way, you can later visit <https://colab.research.google.com/> and open the notebook again (either from Google Drive or by uploading it from your computer).



Google Colab is meant only for interactive use: you can play around in the notebooks and tweak the code as you like, but you cannot let the notebooks run unattended for a long period of time, or else the Runtime will be shutdown and all of its data will be lost.

If the notebook generates data that you care about, make sure you download this data before the Runtime shuts down. To do this, click on the Files icon (see step 1 in [Figure 2-5](#)), find the file you want to download, click on the vertical dots next to it (step 2), and click Download (step 3).



Figure 2-5. Downloading a file from a Google Colab Runtime (steps 1 to 3), or mounting your Google Drive (circled icon)

Alternatively, you can mount your Google Drive on the Runtime, allowing the notebook to read and write files directly to Google Drive, as if it were a local directory. For this, click on the Files icon (step 1) then click on the Google Drive icon (circled in Figure 2-5) and follow the on-screen instructions. By default, your Google Drive will be mounted at `/content/drive/MyDrive`. If you want to backup a data file, simply copy it to this directory by running `!cp /content/my_great_model /content/drive/MyDrive`. Any command starting with a bang (!) is treated as a shell command, not as Python code: `cp` is the Linux shell command to copy a file from one path to another. Note that Colab Runtimes run on Linux (specifically, Ubuntu).

The Power and Danger of Interactivity

Jupyter notebooks are interactive, and that's a great thing: you can run each cell one by one, stop at any point, insert a cell, play with the code, go back and run the same cell again, etc., and I highly encourage you to do so. If you just run the cells one by one without ever playing with them, you won't learn as fast. However, this flexibility comes at a price: it's very easy to run cells in the wrong order, or to forget to run a cell. If this happens, the subsequent code cells are likely to fail. For example, the very first code cell in each notebook contains setup code (such as imports), so make sure you run it first, or else nothing will work.



If you ever run into a weird error, try restarting the Runtime (by selecting Runtime > Restart runtime from the menu) and then run all the cells again from the beginning of the notebook: this will often solve the problem. If not, then it's likely that one of the changes you made broke the notebook: just revert to the original notebook and try again. If it still fails, then please file an issue on GitHub.

Book Code vs Notebook Code

You may sometimes notice some little differences between the code in this book and the code in the notebooks. This may happen for several reasons:

- A library may have changed slightly by the time you read these lines, or perhaps despite my best efforts I made an error in the book. Sadly, I cannot magically fix the code in your copy of this book (unless you are reading an electronic copy and you can download the latest version), but I *can* fix the notebooks: so if you run into an error after copying code from this book, please look for the fixed code in the notebooks: I will strive to keep them error-free and up-to-date with the latest library versions.
- The notebooks contain some extra code to beautify the figures (adding labels, setting font size, etc.), and to save them in high resolution for this book. You can safely ignore this extra code if you want.

I optimized the code for readability and simplicity: I made it as linear and flat as possible, defining very few functions or classes. The goal is to ensure that the code you are running is generally right in front of you, and not nested within several layers of abstractions that you have to search through. This also makes it easier for you to play with the code. For simplicity, there's limited error handling, and I placed some of the least common imports right where they are needed (instead of placing them at the top of the file, as is recommended by the PEP 8 Python style guide). That said, your production code will not be very different: just a bit more modular, and with additional tests and error handling, that's all.

OK! Once you're comfortable with Colab, you're ready to download the data.

Download the Data

In typical environments your data would be available in a relational database or some other common data store, and spread across multiple tables/documents/files.

To access it, you would first need to get your credentials and access authorizations⁴ and familiarize yourself with the data schema. In this project, however, things are much simpler: you will just download a single compressed file, *housing.tgz*, which contains a comma-separated values (CSV) file called *housing.csv* with all the data.

Rather than manually downloading and decompressing the data, it's usually preferable to write a function that does it for you. This is useful in particular if the data changes regularly: you can write a small script that uses the function to fetch the latest data (or you can set up a scheduled job to do that automatically at regular intervals). Automating the process of fetching the data is also useful if you need to install the dataset on multiple machines.

Here is the function to fetch and load the data:

```
from pathlib import Path
import pandas as pd
import tarfile
import urllib.request

def load_housing_data():
    tarball_path = Path("datasets/housing.tgz")
    if not tarball_path.is_file():
        Path("datasets").mkdir(parents=True, exist_ok=True)
        url = "https://github.com/ageron/data/raw/main/housing.tgz"
        urllib.request.urlretrieve(url, tarball_path)
        with tarfile.open(tarball_path) as housing_tarball:
            housing_tarball.extractall(path="datasets")
    return pd.read_csv(Path("datasets/housing/housing.csv"))

housing = load_housing_data()
```

When `load_housing_data()` is called, it looks for the *datasets/housing.tgz* file, and if it does not find it, it creates the *datasets* directory inside the current directory (which is */content* by default, in Colab), it downloads the *housing.tgz* file from the *ageron/data* GitHub repository, it extracts its content into the *datasets* directory, and this creates the *datasets/housing* directory with the *housing.csv* file inside it. Lastly, the function loads this CSV file into a Pandas DataFrame object containing all the data, and it returns it.

Take a Quick Look at the Data Structure

Let's take a look at the top five rows of data using the DataFrame's `head()` method (see [Figure 2-6](#)).

⁴ You might also need to check legal constraints, such as private fields that should never be copied to unsafe data stores.

```
housing.head()
```

	longitude	latitude	housing_median_age	median_income	ocean_proximity	median_house_value
0	-122.23	37.88	41.0	8.3252	NEAR BAY	452600.0
1	-122.22	37.86	21.0	8.3014	NEAR BAY	358500.0
2	-122.24	37.85	52.0	7.2574	NEAR BAY	352100.0
3	-122.25	37.85	52.0	5.6431	NEAR BAY	341300.0
4	-122.25	37.85	52.0	3.8462	NEAR BAY	342200.0

Figure 2-6. Top five rows in the dataset

Each row represents one district. There are 10 attributes (they are not all shown in the screenshot): `longitude`, `latitude`, `housing_median_age`, `total_rooms`, `total_bedrooms`, `population`, `households`, `median_income`, `median_house_value`, and `ocean_proximity`.

The `info()` method is useful to get a quick description of the data, in particular the total number of rows, each attribute's type, and the number of non-null values:

```
>>> housing.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column            Non-Null Count  Dtype  
--- 
 0   longitude         20640 non-null   float64
 1   latitude          20640 non-null   float64
 2   housing_median_age 20640 non-null   float64
 3   total_rooms        20640 non-null   float64
 4   total_bedrooms     20433 non-null   float64
 5   population         20640 non-null   float64
 6   households         20640 non-null   float64
 7   median_income      20640 non-null   float64
 8   median_house_value 20640 non-null   float64
 9   ocean_proximity    20640 non-null   object  
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```



In this book, when a code example contains a mix of code and outputs, as is the case here, it is formatted like in the Python interpreter, for better readability: the code lines are prefixed with `>>>` (or `...` for indented blocks), and the outputs have no prefix.

There are 20,640 instances in the dataset, which means that it is fairly small by Machine Learning standards, but it's perfect to get started. Notice that the `total_bed`

`rooms` attribute has only 20,433 non-null values, meaning that 207 districts are missing this feature. We will need to take care of this later.

All attributes are numerical, except the `ocean_proximity` field. Its type is `object`, so it could hold any kind of Python object. But since you loaded this data from a CSV file, you know that it must be a text attribute. When you looked at the top five rows, you probably noticed that the values in the `ocean_proximity` column were repetitive, which means that it is probably a categorical attribute. You can find out what categories exist and how many districts belong to each category by using the `value_counts()` method:

```
>>> housing["ocean_proximity"].value_counts()
<1H OCEAN      9136
INLAND        6551
NEAR OCEAN     2658
NEAR BAY       2290
ISLAND          5
Name: ocean_proximity, dtype: int64
```

Let's look at the other fields. The `describe()` method shows a summary of the numerical attributes (Figure 2-7).

housing.describe()						
	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	median_house_value
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	20640.000000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	206855.816909
std	2.003532	2.135952	12.585558	2181.615252	421.385070	115395.615874
min	-124.350000	32.540000	1.000000	2.000000	1.000000	14999.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	119600.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	179700.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	264725.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	500001.000000

Figure 2-7. Summary of each numerical attribute

The `count`, `mean`, `min`, and `max` rows are self-explanatory. Note that the null values are ignored (so, for example, the `count` of `total_bedrooms` is 20,433, not 20,640). The `std` row shows the *standard deviation*, which measures how dispersed the values are.⁵ The 25%, 50%, and 75% rows show the corresponding *percentiles*: a percentile

⁵ The standard deviation is generally denoted σ (the Greek letter sigma), and it is the square root of the *variance*, which is the average of the squared deviation from the mean. When a feature has a bell-shaped *normal distribution* (also called a *Gaussian distribution*), which is very common, the “68-95-99.7” rule applies: about 68% of the values fall within 1σ of the mean, 95% within 2σ , and 99.7% within 3σ .

indicates the value below which a given percentage of observations in a group of observations fall. For example, 25% of the districts have a `housing_median_age` lower than 18, while 50% are lower than 29 and 75% are lower than 37. These are often called the 25th percentile (or first *quartile*), the median, and the 75th percentile (or third quartile).

Another quick way to get a feel of the type of data you are dealing with is to plot a histogram for each numerical attribute. A histogram shows the number of instances (on the vertical axis) that have a given value range (on the horizontal axis). You can either plot this one attribute at a time, or you can call the `hist()` method on the whole dataset (as shown in the following code example), and it will plot a histogram for each numerical attribute (see Figure 2-8):

```
import matplotlib.pyplot as plt

housing.hist(bins=50, figsize=(12, 8))
plt.show()
```

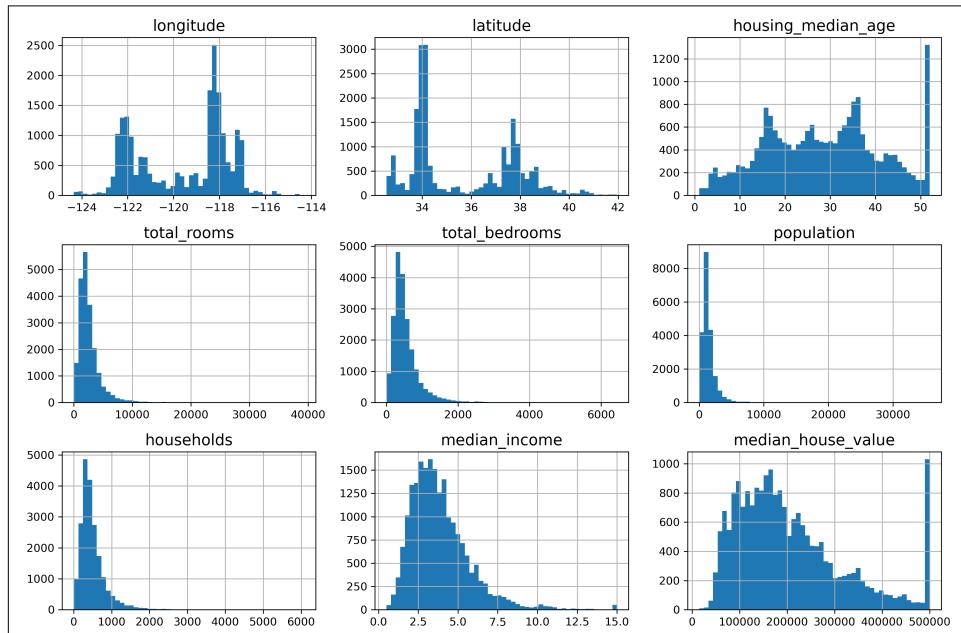


Figure 2-8. A histogram for each numerical attribute

There are a few things you might notice in these histograms:

1. First, the median income attribute does not look like it is expressed in US dollars (USD). After checking with the team that collected the data, you are told that the data has been scaled and capped at 15 (actually, 15.0001) for higher median incomes, and at 0.5 (actually, 0.4999) for lower median incomes. The numbers

represent roughly tens of thousands of dollars (e.g., 3 actually means about \$30,000). Working with preprocessed attributes is common in Machine Learning, and it is not necessarily a problem, but you should try to understand how the data was computed.

2. The housing median age and the median house value were also capped. The latter may be a serious problem since it is your target attribute (your labels). Your Machine Learning algorithms may learn that prices never go beyond that limit. You need to check with your client team (the team that will use your system's output) to see if this is a problem or not. If they tell you that they need precise predictions even beyond \$500,000, then you have two options:
 - a. Collect proper labels for the districts whose labels were capped.
 - b. Remove those districts from the training set (and also from the test set, since your system should not be evaluated poorly if it predicts values beyond \$500,000).
3. These attributes have very different scales. We will discuss this later in this chapter, when we explore feature scaling.
4. Finally, many histograms are *skewed right*: they extend much farther to the right of the median than to the left. This may make it a bit harder for some Machine Learning algorithms to detect patterns. We will try transforming these attributes later on to have more symmetrical and bell-shaped distributions.

Hopefully you now have a better understanding of the kind of data you are dealing with.



Wait! Before you look at the data any further, you need to create a test set, put it aside, and never look at it.

Create a Test Set

It may sound strange to voluntarily set aside part of the data at this stage. After all, you have only taken a quick glance at the data, and surely you should learn a whole lot more about it before you decide what algorithms to use, right? This is true, but your brain is an amazing pattern detection system, which also means that it is highly prone to overfitting: if you look at the test set, you may stumble upon some seemingly interesting pattern in the test data that leads you to select a particular kind of Machine Learning model. When you estimate the generalization error using the test set, your estimate will be too optimistic, and you will launch a system that will not perform as well as expected. This is called *data snooping* bias.

Creating a test set is theoretically simple: pick some instances randomly, typically 20% of the dataset (or less if your dataset is very large), and set them aside:

```
import numpy as np

def shuffle_and_split_data(data, test_ratio):
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
```

You can then use this function like this:

```
>>> train_set, test_set = shuffle_and_split_data(housing, 0.2)
>>> len(train_set)
16512
>>> len(test_set)
4128
```

Well, this works, but it is not perfect: if you run the program again, it will generate a different test set! Over time, you (or your Machine Learning algorithms) will get to see the whole dataset, which is what you want to avoid.

One solution is to save the test set on the first run and then load it in subsequent runs. Another option is to set the random number generator's seed (e.g., with `np.random.seed(42)`)⁶ before calling `np.random.permutation()` so that it always generates the same shuffled indices.

But both these solutions will break the next time you fetch an updated dataset. To have a stable train/test split even after updating the dataset, a common solution is to use each instance's identifier to decide whether or not it should go in the test set (assuming instances have a unique and immutable identifier). For example, you could compute a hash of each instance's identifier and put that instance in the test set if the hash is lower than or equal to 20% of the maximum hash value. This ensures that the test set will remain consistent across multiple runs, even if you refresh the dataset. The new test set will contain 20% of the new instances, but it will not contain any instance that was previously in the training set.

Here is a possible implementation:

```
from zlib import crc32

def is_id_in_test_set(identifier, test_ratio):
    return crc32(np.int64(identifier)) < test_ratio * 2**32
```

⁶ You will often see people set the random seed to 42. This number has no special property, other than to be the Answer to the Ultimate Question of Life, the Universe, and Everything.

```
def split_data_with_id_hash(data, test_ratio, id_column):
    ids = data[id_column]
    in_test_set = ids.apply(lambda id_: is_id_in_test_set(id_, test_ratio))
    return data.loc[~in_test_set], data.loc[in_test_set]
```

Unfortunately, the housing dataset does not have an identifier column. The simplest solution is to use the row index as the ID:

```
housing_with_id = housing.reset_index() # adds an `index` column
train_set, test_set = split_data_with_id_hash(housing_with_id, 0.2, "index")
```

If you use the row index as a unique identifier, you need to make sure that new data gets appended to the end of the dataset and that no row ever gets deleted. If this is not possible, then you can try to use the most stable features to build a unique identifier. For example, a district's latitude and longitude are guaranteed to be stable for a few million years, so you could combine them into an ID like so:⁷

```
housing_with_id["id"] = housing["longitude"] * 1000 + housing["latitude"]
train_set, test_set = split_data_with_id_hash(housing_with_id, 0.2, "id")
```

Scikit-Learn provides a few functions to split datasets into multiple subsets in various ways. The simplest function is `train_test_split()`, which does pretty much the same thing as the function `shuffle_and_split_data()` we defined earlier, with a couple of additional features. First, there is a `random_state` parameter that allows you to set the random generator seed. Second, you can pass it multiple datasets with an identical number of rows, and it will split them on the same indices (this is very useful, for example, if you have a separate DataFrame for labels):

```
from sklearn.model_selection import train_test_split

train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
```

So far we have considered purely random sampling methods. This is generally fine if your dataset is large enough (especially relative to the number of attributes), but if it is not, you run the risk of introducing a significant sampling bias. When a survey company decides to call 1,000 people to ask them a few questions, they don't just pick 1,000 people randomly in a phone book. They try to ensure that these 1,000 people are representative of the whole population, with regards to the questions they want to ask. For example, the US population is 51.1% females and 48.9% males, so a well-conducted survey in the US would try to maintain this ratio in the sample: 511 female and 489 male (at least if we believe that the answers may vary across genders). This is called *stratified sampling*: the population is divided into homogeneous subgroups called *strata*, and the right number of instances are sampled from each stratum to guarantee that the test set is representative of the overall population. If

⁷ The location information is actually quite coarse, and as a result many districts will have the exact same ID, so they will end up in the same set (test or train). This introduces some unfortunate sampling bias.

the people running the survey used purely random sampling, there would be about 10.7% chance of sampling a skewed test set with less than 48.5% female or more than 53.5% female. Either way, the survey results would likely be quite biased.

Suppose you chatted with experts who told you that the median income is a very important attribute to predict median housing prices. You may want to ensure that the test set is representative of the various categories of incomes in the whole dataset. Since the median income is a continuous numerical attribute, you first need to create an income category attribute. Let's look at the median income histogram more closely (back in [Figure 2-8](#)): most median income values are clustered around 1.5 to 6 (i.e., \$15,000–\$60,000), but some median incomes go far beyond 6. It is important to have a sufficient number of instances in your dataset for each stratum, or else the estimate of a stratum's importance may be biased. This means that you should not have too many strata, and each stratum should be large enough. The following code uses the `pd.cut()` function to create an income category attribute with five categories (labeled from 1 to 5): category 1 ranges from 0 to 1.5 (i.e., less than \$15,000), category 2 from 1.5 to 3, and so on:

```
housing["income_cat"] = pd.cut(housing["median_income"],
                                bins=[0., 1.5, 3.0, 4.5, 6., np.inf],
                                labels=[1, 2, 3, 4, 5])
```

These income categories are represented in [Figure 2-9](#):

```
housing["income_cat"].value_counts().sort_index().plot.bar(rot=0, grid=True)
plt.xlabel("Income category")
plt.ylabel("Number of districts")
plt.show()
```

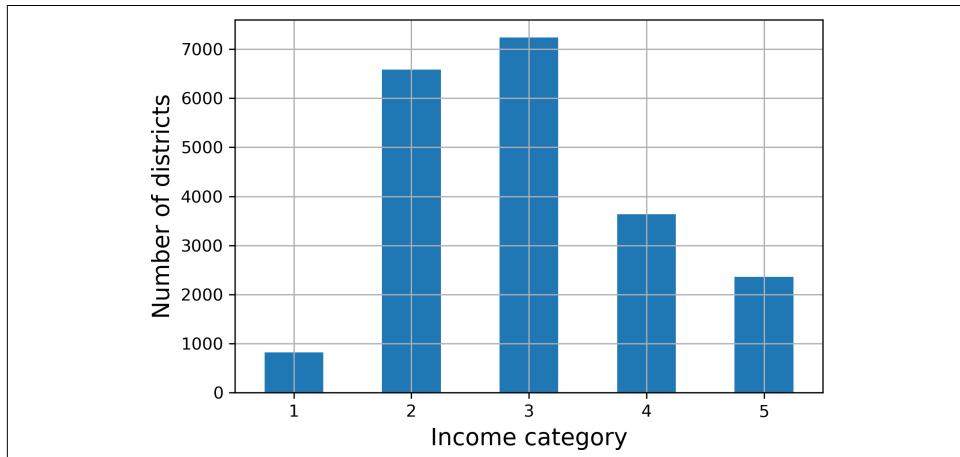


Figure 2-9. Histogram of income categories

Now you are ready to do stratified sampling based on the income category. Scikit-Learn provides a number of splitter classes in the `sklearn.model_selection` package, which implement various strategies to split your dataset into a train set and a test set. Each splitter has a `split()` method which returns an iterator over different train/test splits of the same data. To be precise, the `split()` method yields the train and test *indices*, not the data itself. Having multiple splits can be useful if you want to better estimate the performance of your model, as we will see when we discuss cross-validation later in this chapter. For example, the following code generates 10 different stratified splits of the same dataset:

```
from sklearn.model_selection import StratifiedShuffleSplit

splitter = StratifiedShuffleSplit(n_splits=10, test_size=0.2, random_state=42)
strat_splits = []
for train_index, test_index in splitter.split(housing, housing["income_cat"]):
    strat_train_set_n = housing.loc[train_index]
    strat_test_set_n = housing.loc[test_index]
    strat_splits.append([strat_train_set_n, strat_test_set_n])
```

For now, we can just use the first split:

```
strat_train_set, strat_test_set = strat_splits[0]
```

Since stratified sampling is fairly common, there's a shorter way to get a single split using the `train_test_split()` function with the `stratify` argument:

```
strat_train_set, strat_test_set = train_test_split(
    housing, test_size=0.2, stratify=housing["income_cat"], random_state=42)
```

Let's see if this worked as expected. You can start by looking at the income category proportions in the test set:

```
>>> strat_test_set["income_cat"].value_counts() / len(strat_test_set)
3    0.350533
2    0.318798
4    0.176357
5    0.114341
1    0.039971
Name: income_cat, dtype: float64
```

With similar code you can measure the income category proportions in the full dataset. [Figure 2-10](#) compares the income category proportions in the overall dataset, in the test set generated with stratified sampling, and in a test set generated using purely random sampling. As you can see, the test set generated using stratified sampling has income category proportions almost identical to those in the full dataset, whereas the test set generated using purely random sampling is skewed.

Income Category	Overall %	Stratified %	Random %	Strat. Error %	Rand. Error %
1	3.98	4.00	4.24	0.36	6.45
2	31.88	31.88	30.74	-0.02	-3.59
3	35.06	35.05	34.52	-0.01	-1.53
4	17.63	17.64	18.41	0.03	4.42
5	11.44	11.43	12.09	-0.08	5.63

Figure 2-10. Sampling bias comparison of stratified versus purely random sampling

Now we won't use the `income_cat` column anymore so we might as well drop it, reverting the data back to its original state:

```
for set_ in (strat_train_set, strat_test_set):
    set_.drop("income_cat", axis=1, inplace=True)
```

We spent quite a bit of time on test set generation for a good reason: this is an often neglected but critical part of a Machine Learning project. Moreover, many of these ideas will be useful later when we discuss cross-validation. Now it's time to move on to the next stage: exploring the data.

Discover and Visualize the Data to Gain Insights

So far you have only taken a quick glance at the data to get a general understanding of the kind of data you are manipulating. Now the goal is to go into a little more depth.

First, make sure you have put the test set aside and you are only exploring the training set. Also, if the training set is very large, you may want to sample an exploration set, to make manipulations easy and fast during the exploration phase. In our case, the training set is quite small, so we can just work directly on the full set. Since you're going to experiment with various transformations of the full training set, you should make a copy of the original so you can revert to it afterwards:

```
housing = strat_train_set.copy()
```

Visualizing Geographical Data

Since there is geographical information (latitude and longitude), it is a good idea to create a scatterplot of all districts to visualize the data (Figure 2-11):

```
housing.plot(kind="scatter", x="longitude", y="latitude", grid=True)
plt.show()
```

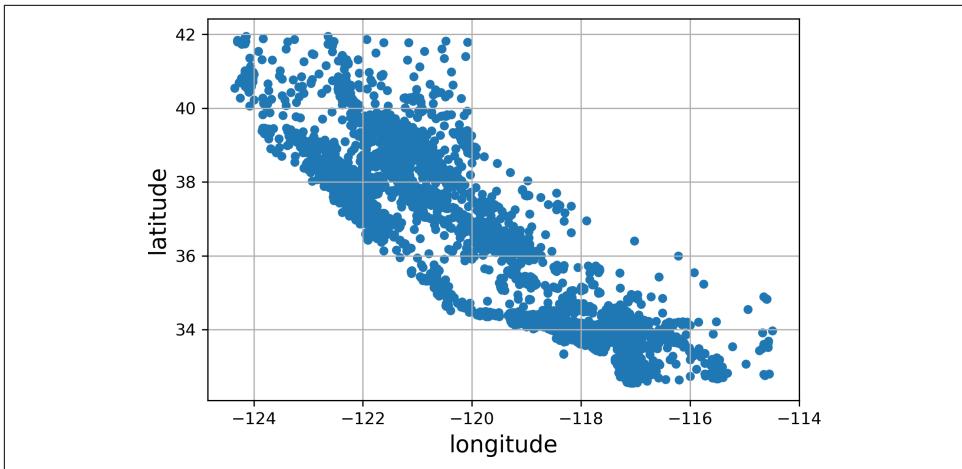


Figure 2-11. A geographical scatterplot of the data

This looks like California all right, but other than that it is hard to see any particular pattern. Setting the `alpha` option to `0.2` makes it much easier to visualize the places where there is a high density of data points (Figure 2-12):

```
housing.plot(kind="scatter", x="longitude", y="latitude", grid=True, alpha=0.2)
plt.show()
```

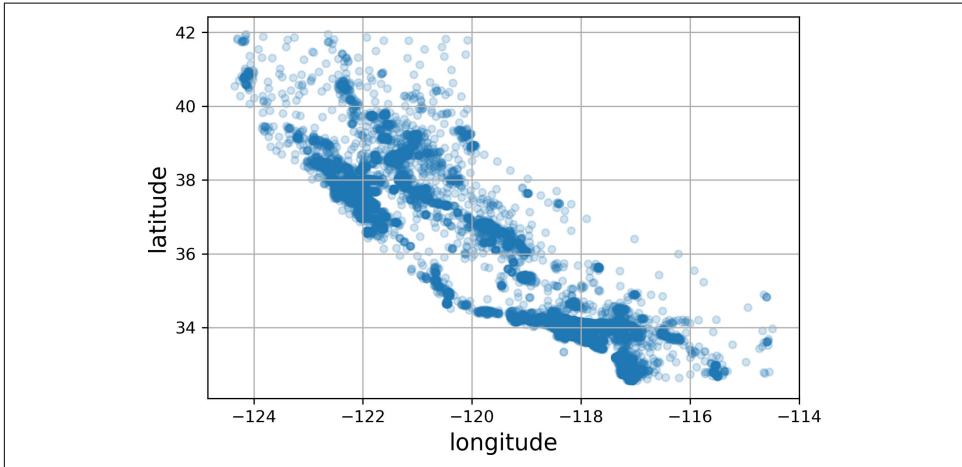


Figure 2-12. A better visualization that highlights high-density areas

Now that's much better: you can clearly see the high-density areas, namely the Bay Area and around Los Angeles and San Diego, plus a long line of fairly high density in the Central Valley, in particular around Sacramento and Fresno.

Our brains are very good at spotting patterns in pictures, but you may need to play around with visualization parameters to make the patterns stand out.

Now let's look at the housing prices (Figure 2-13). The radius of each circle represents the district's population (option `s`), and the color represents the price (option `c`). We will use a predefined color map (option `cmap`) called `jet`, which ranges from blue (low values) to red (high prices):⁸

```
housing.plot(kind="scatter", x="longitude", y="latitude", grid=True,
             s=housing["population"] / 100, label="population",
             c="median_house_value", cmap="jet", colorbar=True,
             legend=True, sharex=False, figsize=(10, 7))
plt.show()
```

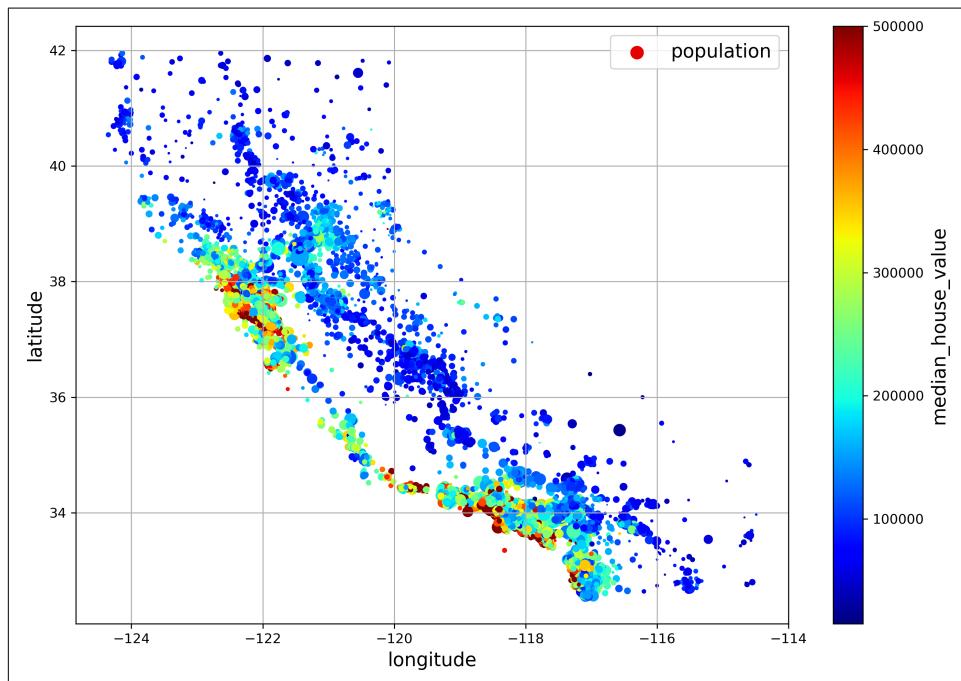


Figure 2-13. California housing prices: red is expensive, blue is cheap, larger circles indicate areas with a larger population

This image tells you that the housing prices are very much related to the location (e.g., close to the ocean) and to the population density, as you probably knew already. A clustering algorithm should be useful for detecting the main cluster and for adding new features that measure the proximity to the cluster centers. The ocean proximity

⁸ If you are reading this in grayscale, grab a red pen and scribble over most of the coastline from the Bay Area down to San Diego (as you might expect). You can add a patch of yellow around Sacramento as well.

attribute may be useful as well, although in Northern California the housing prices in coastal districts are not too high, so it is not a simple rule.

Looking for Correlations

Since the dataset is not too large, you can easily compute the *standard correlation coefficient* (also called *Pearson's r*) between every pair of attributes using the `corr()` method:

```
corr_matrix = housing.corr()
```

Now let's look at how much each attribute correlates with the median house value:

```
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value    1.000000
median_income         0.688380
total_rooms           0.137455
housing_median_age    0.102175
households            0.071426
total_bedrooms        0.054635
population            -0.020153
longitude             -0.050859
latitude              -0.139584
Name: median_house_value, dtype: float64
```

The correlation coefficient ranges from -1 to 1 . When it is close to 1 , it means that there is a strong positive correlation; for example, the median house value tends to go up when the median income goes up. When the coefficient is close to -1 , it means that there is a strong negative correlation; you can see a small negative correlation between the latitude and the median house value (i.e., prices have a slight tendency to go down when you go north). Finally, coefficients close to 0 mean that there is no linear correlation.

Another way to check for correlation between attributes is to use the Pandas `scatter_matrix()` function, which plots every numerical attribute against every other numerical attribute. Since there are now 11 numerical attributes, you would get $11^2 = 121$ plots, which would not fit on a page—so let's just focus on a few promising attributes that seem most correlated with the median housing value (Figure 2-14):

```
from pandas.plotting import scatter_matrix

attributes = ["median_house_value", "median_income", "total_rooms",
              "housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8))
plt.show()
```

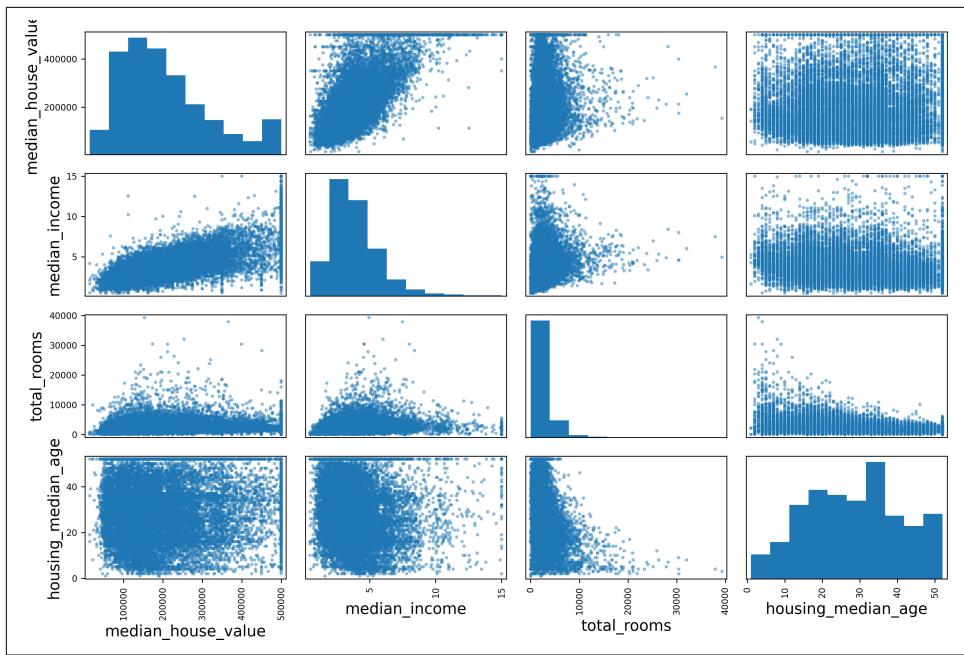


Figure 2-14. This scatter matrix plots every numerical attribute against every other numerical attribute, plus a histogram of each numerical attribute's values on the main diagonal (top left to bottom right)

The main diagonal would be full of straight lines if Pandas plotted each variable against itself, which would not be very useful. So instead Pandas displays a histogram of each attribute (other options are available; see Pandas' documentation for more details).

Looking at the correlation scatterplots, it seems like the most promising attribute to predict the median house value is the median income, so let's zoom in on their scatterplot ([Figure 2-15](#)):

```
housing.plot(kind="scatter", x="median_income", y="median_house_value",
             alpha=0.1, grid=True)
plt.show()
```

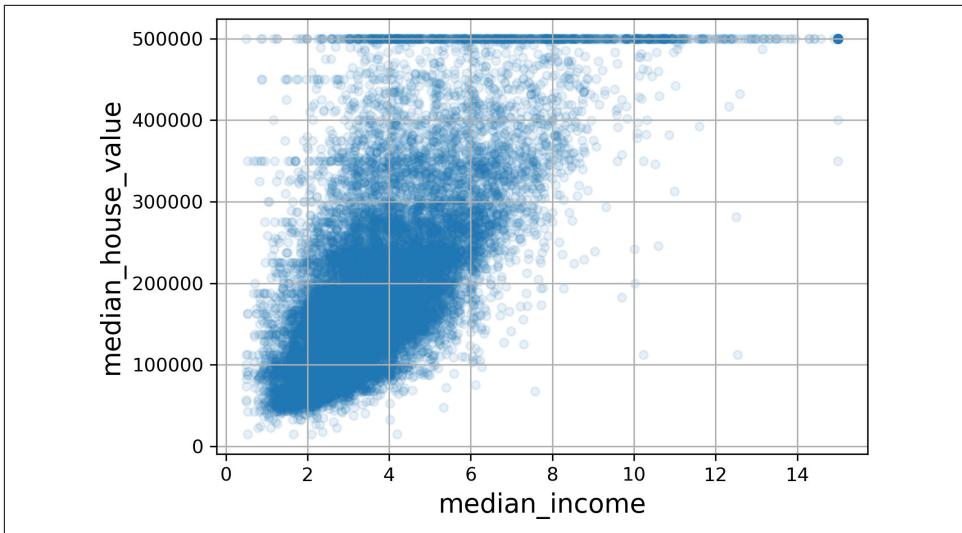


Figure 2-15. Median income versus median house value

This plot reveals a few things. First, the correlation is indeed quite strong; you can clearly see the upward trend, and the points are not too dispersed. Second, the price cap that we noticed earlier is clearly visible as a horizontal line at \$500,000. But this plot reveals other less obvious straight lines: a horizontal line around \$450,000, another around \$350,000, perhaps one around \$280,000, and a few more below that. You may want to try removing the corresponding districts to prevent your algorithms from learning to reproduce these data quirks.



The correlation coefficient only measures linear correlations (“as x goes up, y generally goes up/down”). It may completely miss out on nonlinear relationships (e.g., “as x approaches 0, y generally goes up”). Figure 2-16 shows a variety of datasets along with their correlation coefficient. Note how all the plots of the bottom row have a correlation coefficient equal to 0, despite the fact that their axes are clearly *not* independent: these are examples of nonlinear relationships. Also, the second row shows examples where the correlation coefficient is equal to 1 or -1; notice that this has nothing to do with the slope. For example, your height in inches has a correlation coefficient of 1 with your height in feet or in nanometers.

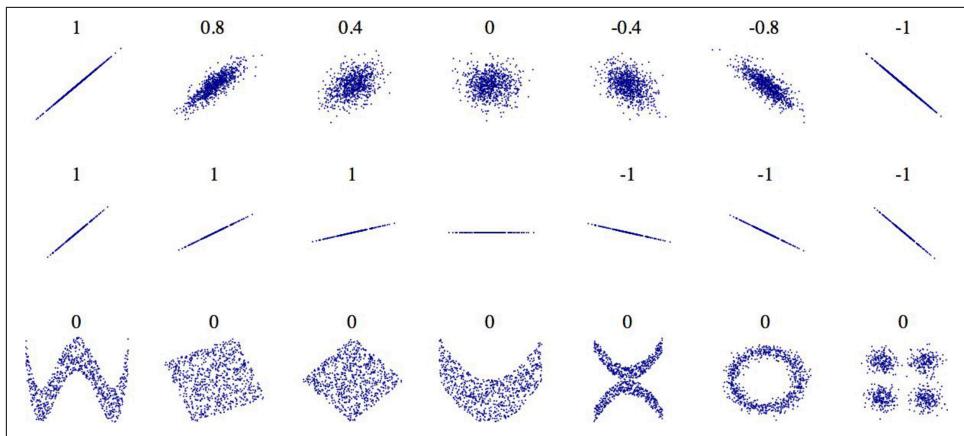


Figure 2-16. Standard correlation coefficient of various datasets (source: Wikipedia; public domain image)

Experimenting with Attribute Combinations

Hopefully the previous sections gave you an idea of a few ways you can explore the data and gain insights. You identified a few data quirks that you may want to clean up before feeding the data to a Machine Learning algorithm, and you found interesting correlations between attributes, in particular with the target attribute. You also noticed that some attributes have a skewed-right distribution, so you may want to transform them (e.g., by computing their logarithm or square root). Of course, your mileage will vary considerably with each project, but the general ideas are similar.

One last thing you may want to do before preparing the data for Machine Learning algorithms is to try out various attribute combinations. For example, the total number of rooms in a district is not very useful if you don't know how many households there are. What you really want is the number of rooms per household. Similarly, the total number of bedrooms by itself is not very useful: you probably want to compare it to the number of rooms. And the population per household also seems like an interesting attribute combination to look at. Let's create these new attributes:

```
housing["rooms_per_house"] = housing["total_rooms"] / housing["households"]
housing["bedrooms_ratio"] = housing["total_bedrooms"] / housing["total_rooms"]
housing["people_per_house"] = housing["population"] / housing["households"]
```

And now let's look at the correlation matrix again:

```
>>> corr_matrix = housing.corr()
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value    1.000000
median_income        0.688380
rooms_per_house     0.143663
```

```
total_rooms      0.137455
housing_median_age 0.102175
households       0.071426
total_bedrooms    0.054635
population        -0.020153
people_per_house  -0.038224
longitude         -0.050859
latitude          -0.139584
bedrooms_ratio    -0.256397
Name: median_house_value, dtype: float64
```

Hey, not bad! The new `bedrooms_ratio` attribute is much more correlated with the median house value than the total number of rooms or bedrooms. Apparently houses with a lower bedroom/room ratio tend to be more expensive. The number of rooms per household is also more informative than the total number of rooms in a district—obviously the larger the houses, the more expensive they are.

This round of exploration does not have to be absolutely thorough; the point is to start off on the right foot and quickly gain insights that will help you get a first reasonably good prototype. But this is an iterative process: once you get a prototype up and running, you can analyze its output to gain more insights and come back to this exploration step.

Prepare the Data for Machine Learning Algorithms

It's time to prepare the data for your Machine Learning algorithms. Instead of doing this manually, you should write functions for this purpose, for several good reasons:

- This will allow you to reproduce these transformations easily on any dataset (e.g., the next time you get a fresh dataset).
- You will gradually build a library of transformation functions that you can reuse in future projects.
- You can use these functions in your live system to transform the new data before feeding it to your algorithms.
- This will make it possible for you to easily try various transformations and see which combination of transformations works best.

But first let's revert to a clean training set (by copying `strat_train_set` once again). Let's also separate the predictors and the labels, since we don't necessarily want to apply the same transformations to the predictors and the target values (note that `drop()` creates a copy of the data and does not affect `strat_train_set`):

```
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()
```

Data Cleaning

Most Machine Learning algorithms cannot work with missing features, so let's create a few functions to take care of them. We saw earlier that the `total_bedrooms` attribute has some missing values, so let's fix this. You have three options:

1. Get rid of the corresponding districts.
2. Get rid of the whole attribute.
3. Set the missing values to some value (zero, the mean, the median, etc.). This is called *imputation*.

You can accomplish these easily using DataFrame's `dropna()`, `drop()`, and `fillna()` methods:

```
housing.dropna(subset=["total_bedrooms"], inplace=True) # option 1

housing.drop("total_bedrooms", axis=1) # option 2

median = housing["total_bedrooms"].median() # option 3
housing["total_bedrooms"].fillna(median, inplace=True)
```

Let's go for option 3 since it is the least destructive, but instead of the code above, we will use a handy Scikit-Learn class : `SimpleImputer`. The benefit is that it will store the median value of each feature: this will make it possible to impute missing values not only on the training set, but also on the validation set, the test set, and any new data fed to the model. Here is how to use it. First, you need to create a `SimpleImputer` instance, specifying that you want to replace each attribute's missing values with the median of that attribute:

```
from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy="median")
```

Since the median can only be computed on numerical attributes, you need to create a copy of the data with only the numerical attributes (this will exclude the text attribute `ocean_proximity`):

```
housing_num = housing.select_dtypes(include=[np.number])
```

Now you can fit the `imputer` instance to the training data using the `fit()` method:

```
imputer.fit(housing_num)
```

The `imputer` has simply computed the median of each attribute and stored the result in its `statistics_` instance variable. Only the `total_bedrooms` attribute had missing values, but we cannot be sure that there won't be any missing values in new data after the system goes live, so it is safer to apply the `imputer` to all the numerical attributes:

```
>>> imputer.statistics_
array([-118.51 , 34.26 , 29. , 2125. , 434. , 1167. , 408. , 3.5385])
>>> housing_num.median().values
array([-118.51 , 34.26 , 29. , 2125. , 434. , 1167. , 408. , 3.5385])
```

Now you can use this “trained” `imputer` to transform the training set by replacing missing values with the learned medians:

```
X = imputer.transform(housing_num)
```

Missing values can also be replaced with the mean value (`strategy="mean"`), or with the most frequent value (`strategy="most_frequent"`), or with a constant value (`strategy="constant"`, `fill_value=...`). The last two strategies support non-numerical data.



There are also more powerful imputers available in the `sklearn.impute` package (both for numerical features only):

- `KNNImputer` replaces each missing value with the mean of the k nearest neighbors’ values for that feature. The distance is based on all the available features.
- `IterativeImputer` trains a regression model per feature to predict the missing values based on all the other available features. It then trains the model again on the updated data, and repeats the process several times, improving the models and the replacement values at each iteration.

Scikit-Learn Design

Scikit-Learn’s API is remarkably well designed. These are the [main design principles](#):⁹

Consistency

All objects share a consistent and simple interface:

Estimators

Any object that can estimate some parameters based on a dataset is called an *estimator* (e.g., a `SimpleImputer` is an estimator). The estimation itself is performed by the `fit()` method, and it takes a dataset as a parameter, or two for supervised learning algorithms—the second dataset contains the labels. Any other parameter needed to guide the estimation process is considered a hyperparameter (such as a `SimpleImputer`’s `strategy`), and it must be set as an instance variable (generally via a constructor parameter).

⁹ For more details on the design principles, see Lars Buitinck et al., “API Design for Machine Learning Software: Experiences from the Scikit-Learn Project,” arXiv preprint arXiv:1309.0238 (2013).

Transformers

Some estimators (such as a `SimpleImputer`) can also transform a dataset; these are called *transformers*. Once again, the API is simple: the transformation is performed by the `transform()` method with the dataset to transform as a parameter. It returns the transformed dataset. This transformation generally relies on the learned parameters, as is the case for a `SimpleImputer`. All transformers also have a convenience method called `fit_transform()` which is equivalent to calling `fit()` and then `transform()` (but sometimes `fit_transform()` is optimized and runs much faster).

Predictors

Finally, some estimators, given a dataset, are capable of making predictions; they are called *predictors*. For example, the `LinearRegression` model in the previous chapter was a predictor: given a country's GDP per capita, it predicted life satisfaction. A predictor has a `predict()` method that takes a dataset of new instances and returns a dataset of corresponding predictions. It also has a `score()` method that measures the quality of the predictions, given a test set (and the corresponding labels, in the case of supervised learning algorithms).{wj}¹⁰

Inspection

All the estimator's hyperparameters are accessible directly via public instance variables (e.g., `imputer.strategy`), and all the estimator's learned parameters are accessible via public instance variables with an underscore suffix (e.g., `imputer.statistics_`).

Nonproliferation of classes

Datasets are represented as NumPy arrays or SciPy sparse matrices, instead of homemade classes. Hyperparameters are just regular Python strings or numbers.

Composition

Existing building blocks are reused as much as possible. For example, it is easy to create a `Pipeline` estimator from an arbitrary sequence of transformers followed by a final estimator, as we will see.

Sensible defaults

Scikit-Learn provides reasonable default values for most parameters, making it easy to quickly create a baseline working system.

Scikit-Learn transformers output NumPy arrays (or sometimes SciPy sparse matrices) even when they are fed Pandas DataFrames as input.¹¹ So the output of `imputer.transform(housing_num)` is a NumPy array: `X` has neither column names

¹⁰ Some predictors also provide methods to measure the confidence of their predictions.

nor index. Luckily, it's not too hard to wrap `X` in a DataFrame and recover the column names and index from `housing_num`:

```
housing_tr = pd.DataFrame(X, columns=housing_num.columns,
                           index=housing_num.index)
```

Handling Text and Categorical Attributes

So far we have only dealt with numerical attributes, but now let's look at text attributes. In this dataset, there is just one: the `ocean_proximity` attribute. Let's look at its value for the first few instances:

```
>>> housing_cat = housing[["ocean_proximity"]]
>>> housing_cat.head(8)
   ocean_proximity
13096      NEAR BAY
14973      <1H OCEAN
3785       INLAND
14689       INLAND
20507      NEAR OCEAN
1286        INLAND
18078      <1H OCEAN
4396      NEAR BAY
```

It's not arbitrary text: there are a limited number of possible values, each of which represents a category. So this attribute is a categorical attribute. Most Machine Learning algorithms prefer to work with numbers, so let's convert these categories from text to numbers. For this, we can use Scikit-Learn's `OrdinalEncoder` class:

```
from sklearn.preprocessing import OrdinalEncoder

ordinal_encoder = OrdinalEncoder()
housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
```

Here's what the first few encoded values in `housing_cat_encoded` look like:

```
>>> housing_cat_encoded[:8]
array([[3.],
       [0.],
       [1.],
       [1.],
       [4.],
       [1.],
       [0.],
       [3.]])
```

¹¹ By the time you read these lines, it may be possible to make all transformers output Pandas DataFrames when they receive a DataFrame as input: Pandas in, Pandas out. There will be a global configuration option for this: `sklearn.set_config(pandas_in_out=True)`.

You can get the list of categories using the `categories_` instance variable. It is a list containing a 1D array of categories for each categorical attribute (in this case, a list containing a single array since there is just one categorical attribute):

```
>>> ordinal_encoder.categories_
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
      dtype=object)]
```

One issue with this representation is that ML algorithms will assume that two nearby values are more similar than two distant values. This may be fine in some cases (e.g., for ordered categories such as “bad,” “average,” “good,” and “excellent”), but it is obviously not the case for the `ocean_proximity` column (for example, categories 0 and 4 are clearly more similar than categories 0 and 1). To fix this issue, a common solution is to create one binary attribute per category: one attribute equal to 1 when the category is “`<1H OCEAN`” (and 0 otherwise), another attribute equal to 1 when the category is “`INLAND`” (and 0 otherwise), and so on. This is called *one-hot encoding*, because only one attribute will be equal to 1 (hot), while the others will be 0 (cold). The new attributes are sometimes called *dummy* attributes. Scikit-Learn provides a `OneHotEncoder` class to convert categorical values into one-hot vectors:

```
from sklearn.preprocessing import OneHotEncoder

cat_encoder = OneHotEncoder()
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
```

By default, the output of a `OneHotEncoder` is a SciPy *sparse matrix*, instead of a NumPy array:

```
>>> housing_cat_1hot
<16512x5 sparse matrix of type '<class 'numpy.float64'>'>
with 16512 stored elements in Compressed Sparse Row format>
```

A sparse matrix is a very efficient representation for matrices that contain mostly zeroes. Indeed, internally it only stores the non-zero values and their positions. When a categorical attribute has hundreds or thousands of categories, then one-hot-encoding it results in a very large matrix full of zeros except for a single 1 per row. In this case, a sparse matrix is exactly what we need: it will save plenty of memory and speed up computations. You can use a sparse matrix mostly like a normal 2D array,¹² but if you want to convert it to a (dense) NumPy array, just call the `toarray()` method:

```
>>> housing_cat_1hot.toarray()
array([[0., 0., 0., 1., 0.],
       [1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       ...,
```

¹² See SciPy’s documentation for more details.

```
[0., 0., 0., 0., 1.],  
[1., 0., 0., 0., 0.],  
[0., 0., 0., 0., 1.])
```

Alternatively, you can set `sparse=False` when creating the `OneHotEncoder`, in which case the `transform()` method will return a regular (dense) NumPy array directly.

As with the `OrdinalEncoder`, you can get the list of categories using the encoder's `categories_` instance variable:

```
>>> cat_encoder.categories_  
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],  
      dtype=object)]
```

Pandas has a function called `get_dummies()` which also converts each categorical feature into a one-hot-representation, with one binary feature per category:

```
>>> df_test = pd.DataFrame({"ocean_proximity": ["INLAND", "NEAR BAY"]})  
>>> pd.get_dummies(df_test)  
ocean_proximity_INLAND  ocean_proximity_NEAR BAY  
0                      1                      0  
1                      0                      1
```

It looks nice and simple, so why not use it instead of `OneHotEncoder`? Well, the advantage of `OneHotEncoder` is that it remembers which categories it was trained on. This is very important because once your model is in production, it should be fed exactly the same features as during training: no more, no less. Watch what our trained `cat_encoder` outputs when we make it transform the same `df_test` (using `transform()`, not `fit_transform()`):

```
>>> cat_encoder.transform(df_test)  
array([[0., 1., 0., 0., 0.],  
      [0., 0., 0., 1., 0.]])
```

See the difference? `get_dummies()` saw only two categories, so it output two columns, whereas `OneHotEncoder` output one column per learned category, in the right order. Moreover, if you feed `get_dummies()` a DataFrame containing an unknown category (e.g., "<2H OCEAN"), it will happily generate a column for it:

```
>>> df_test_unknown = pd.DataFrame({"ocean_proximity": ["<2H OCEAN", "ISLAND"]})  
>>> pd.get_dummies(df_test_unknown)  
ocean_proximity_<2H OCEAN  ocean_proximity_ISLAND  
0                          1                          0  
1                          0                          1
```

But `OneHotEncoder` is smarter: it will detect the unknown category and raise an exception. If you prefer, you can set the `handle_unknown` hyperparameter to "ignore", in which case it will just represent the unknown category with zeros:

```
>>> cat_encoder.handle_unknown = "ignore"  
>>> cat_encoder.transform(df_test_unknown)
```

```
array([[0., 0., 0., 0., 0.],  
       [0., 0., 1., 0., 0.]])
```

When you fit any Scikit-Learn estimator using a DataFrame, the estimator stores the column names in the `feature_names_in_` attribute. Scikit-Learn then ensures that any DataFrame fed to this estimator after that (e.g., to `transform()` or `predict()`) has the same column names. Transformers also provide a `get_feature_names_out()` method which you can use to build a DataFrame around the transformer's output:



If a categorical attribute has a large number of possible categories (e.g., country code, profession, species), then one-hot encoding will result in a large number of input features. This may slow down training and degrade performance. If this happens, you may want to replace the categorical input with useful numerical features related to the categories: for example, you could replace the `ocean_proximity` feature with the distance to the ocean (similarly, a country code could be replaced with the country's population and GDP per capita). Alternatively, you can use one of the encoders provided by the `category_encoders` package on <https://github.com/scikit-learn-contrib>. Or when dealing with neural networks, you can replace each category with a learnable, low-dimensional vector called an *embedding*. This is an example of *representation learning* (see Chapters 13 and 17 for more details).

Feature Scaling and Transformation

One of the most important transformations you need to apply to your data is *feature scaling*. With few exceptions, Machine Learning algorithms don't perform well when the input numerical attributes have very different scales. This is the case for the housing data: the total number of rooms ranges from about 6 to 39,320, while the median incomes only range from 0 to 15. Without any scaling, most models will be biased towards ignoring the median income, and focusing more on the number of rooms.

There are two common ways to get all attributes to have the same scale: *min-max scaling* and *standardization*.

Min-max scaling (many people call this *normalization*) is the simplest: for each attribute, the values are shifted and rescaled so that they end up ranging from 0 to 1. This is performed by subtracting the min value and dividing by the difference between the min and the max. Scikit-Learn provides a transformer called `MinMaxScaler` for this. It has a `feature_range` hyperparameter that lets you change the range if, for some reason, you don't want 0–1 (e.g., neural networks work best with zero-mean inputs, so a range of –1 to 1 is preferable). It's quite easy to use:

```
from sklearn.preprocessing import MinMaxScaler  
  
min_max_scaler = MinMaxScaler(feature_range=(-1, 1))  
housing_num_min_max_scaled = min_max_scaler.fit_transform(housing_num)
```



As with all estimators, it is important to fit the scalers to the training data only: never use `fit()` or `fit_transform()` for anything else than the training set. Once you have a trained scaler, you can then use it to `transform()` any other set, including the validation set, the test set, and new data. Note that while the training set values will always be scaled to the specified range, if new data contains outliers, these may end up scaled outside the range. If you want to avoid this, just set the `clip` hyperparameter to `True`.

Standardization is different: first it subtracts the mean value (so standardized values have a zero mean), then it divides the result by the standard deviation (so standardized values have a standard deviation equal to 1). Unlike min-max scaling, standardization does not bound values to a specific range. However, standardization is much less affected by outliers. For example, suppose a district has a median income equal to 100 (by mistake), instead of the usual 0–15. Min-max scaling to the 0–1 range would map this outlier down to 1 and it would crush all the other values down to 0–0.15, whereas standardization would not be much affected. Scikit-Learn provides a transformer called `StandardScaler` for standardization:

```
from sklearn.preprocessing import StandardScaler  
  
std_scaler = StandardScaler()  
housing_num_std_scaled = std_scaler.fit_transform(housing_num)
```



If you want to scale a sparse matrix without converting it to a dense matrix first, you can use a `StandardScaler` with its `with_mean` hyperparameter set to `False`: it will only divide the data by the standard deviation, without subtracting the mean (as this would break sparsity).

When a feature's distribution has a *heavy tail* (i.e., when values far from the mean are not exponentially rare), both min-max scaling and standardization will squash

most values into a small range. Machine Learning models generally don't like this at all, as we will see in [Chapter 4](#). So *before* you scale the feature, you should first transform it to shrink the heavy tail, and if possible to make the distribution roughly symmetrical. For example, a common way to do this for positive features with a heavy tail to the right is to replace the feature with its square root (or raise the feature to a power between 0 and 1). If the feature has a really long and heavy tail, such as a *power law distribution*, then replacing the feature with its logarithm may help. For example, the population feature roughly follows a power law: districts with 10,000 inhabitants are only 10 times less frequent than districts with 1,000 inhabitants, not exponentially less frequent. [Figure 2-17](#) shows how much better this feature looks when you compute its log: it's very close to a Gaussian distribution (i.e., bell-shaped).

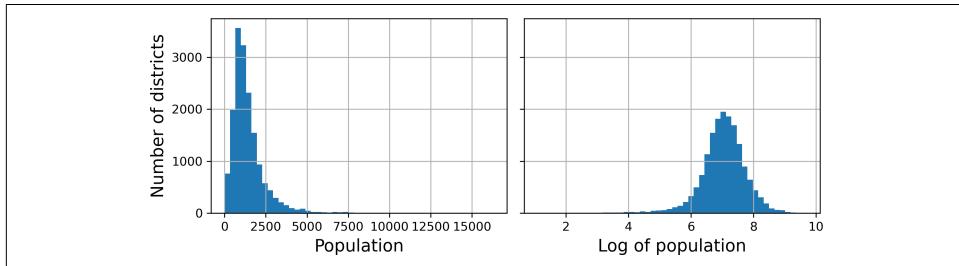


Figure 2-17. Transforming a feature to make it closer to a Gaussian distribution

Another approach to handle heavy tailed features consists in *bucketizing* the feature. This means chopping its distribution into roughly equal size buckets, and replacing each feature value with the index of the bucket it belongs to, much like we did to create the `income_cat` feature (although we only used it for stratified sampling). For example, you could replace each value with its percentile. Bucketizing with equal-sized buckets results in a feature with an almost uniform distribution, so there's no need for further scaling, or you can just divide by the number of buckets to force the values to the 0–1 range.

When a feature has a multimodal distribution (i.e., with two or more clear peaks, called *modes*), such as the `housing_median_age` feature, it can also be helpful to bucketize it, but this time treating the bucket ids as categories, rather than as numerical values. This means that the bucket indices must be encoded, for example using a `OneHotEncoder` (so you usually don't want to use too many buckets). This approach will allow the regression model to more easily learn different rules for different ranges of this feature value. For example, perhaps houses built around 35 years ago have a peculiar style that fell out of fashion, and therefore they're cheaper than their age alone would suggest.

Another approach for transforming multimodal distributions is to add a feature for each of the modes (at least the main ones), representing the similarity between the

housing median age and that particular mode. The similarity measure is typically computed using a *Radial Basis Function* (RBF): this is any function which depends only on the distance between the input value and a fixed point. The most commonly used RBF is the Gaussian RBF whose output value decays exponentially as the input value moves away from the fixed point. For example, the Gaussian RBF similarity between the housing age x and 35 is given by the equation $\exp(-|x - 35|^2)$. The hyperparameter `gamma` determines how quickly the similarity measure decays as x moves away from 35. Using Scikit-Learn's `rbf_kernel()` function, we can create a new Gaussian RBF feature measuring the similarity between the housing median age and 35:

```
from sklearn.metrics.pairwise import rbf_kernel

age_simil_35 = rbf_kernel(housing[["housing_median_age"]], [[35]], gamma=0.1)
```

Figure 2-18 shows this new feature as a function of the housing median age (solid line). It also shows what the feature would look like if we used a smaller `gamma` value. As the chart shows, the new age similarity feature peaks at 35, right around the spike in the housing median age distribution: if this particular age group is well correlated with lower prices, there's a good chance that this new feature will help.

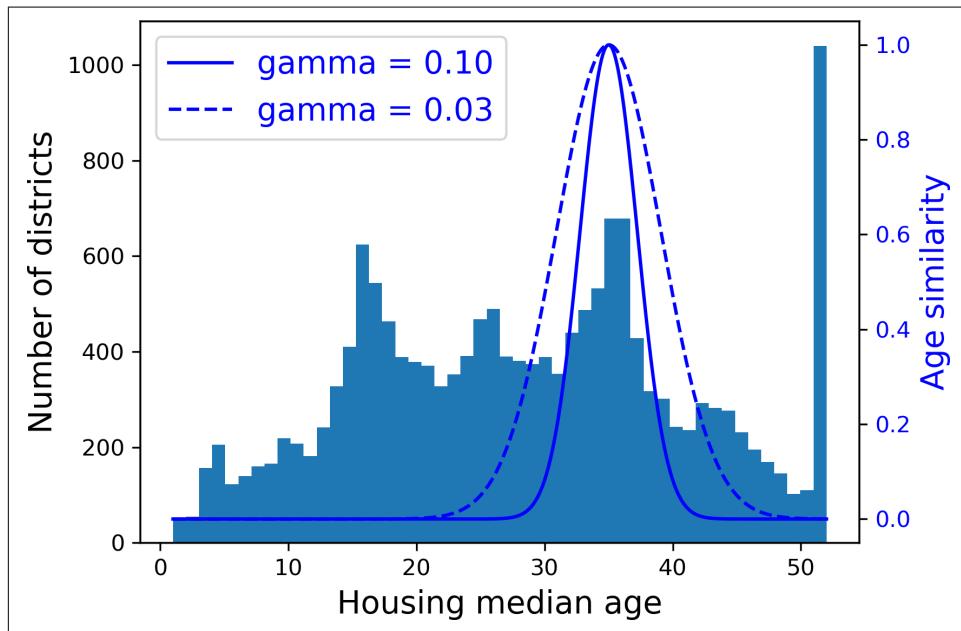


Figure 2-18. Gaussian RBF feature measuring the similarity between the housing median age and 35

So far we've only looked at the input features, but the target values may also need to be transformed. For example, if the target distribution has a heavy tail, you may choose to replace the target with its logarithm. But if you do, the regression model will now predict the *log* of the median house value, not the median house value itself. So you will need to compute the exponential of the model's prediction if you want the predicted median house value.

Luckily, most of Scikit-Learn's transformers have an `inverse_transform()` method, making it easy to compute the inverse of their transformations. For example, the following code example shows how to scale the labels using a `StandardScaler` (just like we did for inputs), then train a simple linear regression model on the resulting scaled labels, and use it to make predictions on some new data, which we transform back to the original scale using the trained scaler's `inverse_transform()` method. Note that we convert the labels from a Pandas Series to a DataFrame, since the `StandardScaler` expects 2D inputs. Also, in this example we just train the model on a single raw input feature (median income), for simplicity.

```
from sklearn.linear_model import LinearRegression

target_scaler = StandardScaler()
scaled_labels = target_scaler.fit_transform(housing_labels.to_frame())

model = LinearRegression()
model.fit(housing[["median_income"]], scaled_labels)
some_new_data = housing[["median_income"]].iloc[:5] # pretend this is new data

scaled_predictions = model.predict(some_new_data)
predictions = target_scaler.inverse_transform(scaled_predictions)
```

This works fine, but there's a simpler way: we can use a `TransformedTargetRegressor`. We just need to construct it, giving it the regression model and the label transformer, then fit it on the training set, using the original unscaled labels. It will automatically use the transformer to scale the labels, and train the regression model on the resulting scaled labels, just like we did above. Then when we make a prediction, it will call the regression model's `predict()` method and use the scaler's `inverse_transform()` method to produce the predictions:

```
from sklearn.compose import TransformedTargetRegressor

model = TransformedTargetRegressor(LinearRegression(),
                                    transformer=StandardScaler())
model.fit(housing[["median_income"]], housing_labels)
predictions = model.predict(some_new_data)
```

Custom Transformers

Although Scikit-Learn provides many useful transformers, you will need to write your own for tasks such as custom transformations, cleanup operations, or combining specific attributes.

For transformations that don't require any training, you can just write a function that takes a NumPy array as input, and outputs the transformed array. For example, as we discussed in the previous section, it's often a good idea to transform features with heavy-tailed distributions by replacing them with their logarithm (assuming the feature is positive and the tail is on the right). Let's create a log-transformer, and apply it to the population feature:

```
from sklearn.preprocessing import FunctionTransformer

log_transformer = FunctionTransformer(np.log, inverse_func=np.exp)
log_pop = log_transformer.transform(housing[["population"]])
```

The `inverse_func` argument is optional. It lets you specify an inverse transform function, for example if you plan to use your transformer in a `TransformedTargetRegressor`.

Your transformation function can take hyperparameters as additional arguments. For example, here's how to create a transformer that computes the same Gaussian RBF similarity measure as earlier:

```
rbf_transformer = FunctionTransformer(rbf_kernel,
                                      kw_args=dict(Y=[[35.]], gamma=0.1))
age_simil_35 = rbf_transformer.transform(housing[["housing_median_age"]])
```

Note that there's no inverse function for the RBF kernel, since there are always two values at a given distance from a fixed point (except at distance 0). Also note that `rbf_kernel()` does not treat the features separately. If you pass it an array with 2 features, it will measure the 2D distance (Euclidean) to measure similarity. For example, here's how to add a feature that will measure the geographic similarity between each district and San Francisco:

```
sf_coords = 37.7749, -122.41
sf_transformer = FunctionTransformer(rbf_kernel,
                                     kw_args=dict(Y=[sf_coords], gamma=0.1))
sf_simil = sf_transformer.transform(housing[["latitude", "longitude"]])
```

Custom transformers are also useful to combine features. For example, here's a `FunctionTransformer` that computes the ratio between the input features 0 and 1:

```
>>> ratio_transformer = FunctionTransformer(lambda X: X[:, [0]] / X[:, [1]])
>>> ratio_transformer.transform(np.array([[1., 2.], [3., 4.]]))
array([[0.5],
       [0.75]])
```

`FunctionTransformer` is very handy, but what if you would like your transformer to be trainable, learning some parameters in the `fit()` method and using them later in the `transform()` method? For this, you need to write a custom class. Scikit-Learn relies on duck typing, so this class does not have to inherit from any particular base class: all it needs is three methods: `fit()` (which must return `self`), `transform()`, and `fit_transform()`.

You can get `fit_transform()` for free by simply adding `TransformerMixin` as a base class: the default implementation will just call `fit()` then `transform()`. If you add `BaseEstimator` as a base class (and avoid using `*args` and `**kwargs` in your constructor), you will also get two extra methods: `get_params()` and `set_params()`. These will be useful for automatic hyperparameter tuning.

For example, here's a custom transformer that acts much like the `StandardScaler`:

```
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.utils.validation import check_array, check_is_fitted

class StandardScalerClone(BaseEstimator, TransformerMixin):
    def __init__(self, with_mean=True): # no *args or **kwargs!
        self.with_mean = with_mean

    def fit(self, X, y=None): # y is required even though we don't use it
        X = check_array(X) # checks that X is an array with finite float values
        self.mean_ = X.mean(axis=0)
        self.scale_ = X.std(axis=0)
        self.n_features_in_ = X.shape[1] # every estimator stores this in fit()
        return self # always return self!

    def transform(self, X):
        check_is_fitted(self) # looks for learned attributes (with trailing _)
        X = check_array(X)
        assert self.n_features_in_ == X.shape[1]
        if self.with_mean:
            X = X - self.mean_
        return X / self.scale_
```

Here are a few things to note:

- The `sklearn.utils.validation` package contains several functions we can use to validate the inputs. For simplicity, we will skip such tests in the rest of this book, but production code should have them.
- Scikit-Learn pipelines require the `fit()` method to have two arguments `X` and `y`, which is why we need the `y=None` argument even though we don't use `y`.
- All Scikit-Learn estimators set `n_features_in_` in the `fit()` method, and they ensure that the data passed to `transform()` or `predict()` has this number of features.

- The `fit()` method must return `self`.
- This implementation is not 100% complete: all estimators should set `feature_names_in_` in the `fit()` method when they are passed a DataFrame. Moreover, all transformers should provide a `get_feature_names_out()` method, as well as an `inverse_transform()` method when their transformation can be reversed. See the last exercise at the end of this chapter for more details.



You can check whether your custom estimator respects Scikit-Learn's API by passing an instance to `check_estimator()` from the `sklearn.utils.estimator_checks` package. For the full API, check out <https://scikit-learn.org/stable/developers/>.

A custom transformer can (and often does) use other estimators in its implementation. For example, below we see a custom transformer that uses a `KMeans` clusterer in the `fit()` method to identify the main clusters in the training data, and then uses `rbf_kernel()` in the `transform()` method to measure how similar each sample is to each cluster center:

```
from sklearn.cluster import KMeans

class ClusterSimilarity(BaseEstimator, TransformerMixin):
    def __init__(self, n_clusters=10, gamma=1.0, random_state=None):
        self.n_clusters = n_clusters
        self.gamma = gamma
        self.random_state = random_state

    def fit(self, X, y=None, sample_weight=None):
        self.kmeans_ = KMeans(self.n_clusters, random_state=self.random_state)
        self.kmeans_.fit(X, sample_weight=sample_weight)
        return self # always return self!

    def transform(self, X):
        return rbf_kernel(X, self.kmeans_.cluster_centers_, gamma=self.gamma)

    def get_feature_names_out(self, names=None):
        return [f"Cluster {i} similarity" for i in range(self.n_clusters)]
```

As we will see in [Chapter 9](#), K-Means is a clustering algorithm that locates clusters in the data. How many it searches for is controlled by the `n_clusters` hyperparameter. After training, the cluster centers are available via the `cluster_centers_` attribute. The `fit()` method of `KMeans` supports an optional argument `sample_weight` which lets the user specify the relative weights of the samples. K-Means is a stochastic algorithm, meaning that it relies on randomness to locate the clusters, so if you want reproducible results, you must set the `random_state` parameter. As you can see,

despite the complexity of the task, the code is fairly straightforward. Now let's use this custom transformer:

```
cluster_simil = ClusterSimilarity(n_clusters=10, gamma=1., random_state=42)
similarities = cluster_simil.fit_transform(housing[["latitude", "longitude"]],
                                         sample_weight=housing_labels)
```

This code creates a `ClusterSimilarity` transformer, setting the number of clusters to 10 clusters. Then it calls `fit_transform()` with the latitude and longitude of every district in the training set, weighting each district by its median house value. The transformer uses K-Means to locate the clusters, then measures the Gaussian RBF similarity between each district and all 10 cluster centers. The result is a matrix with one row per district, and one column per cluster. Let's look at the first 3 rows, rounding to 2 decimals:

```
>>> similarities[:3].round(2)
array([[0. , 0.14, 0. , 0. , 0. , 0.08, 0. , 0.99, 0. , 0.6 ],
       [0.63, 0. , 0.99, 0. , 0. , 0. , 0.04, 0. , 0.11, 0. ],
       [0. , 0.29, 0. , 0. , 0.01, 0.44, 0. , 0.7 , 0. , 0.3 ]])
```

Figure 2-19 shows the 10 cluster centers found by K-Means. The districts are colored according to their geographic similarity to their closest cluster center. As you can see, most clusters are located in highly populated and expensive areas.

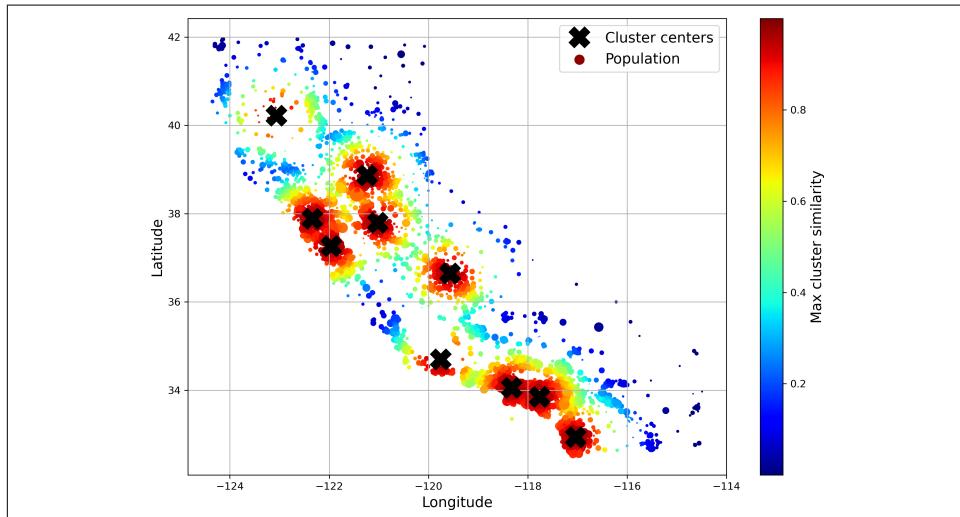


Figure 2-19. Gaussian RBF similarity to the nearest cluster center

Transformation Pipelines

As you can see, there are many data transformation steps that need to be executed in the right order. Fortunately, Scikit-Learn provides the `Pipeline` class to help with

such sequences of transformations. Here is a small pipeline for numerical attributes, which will first impute then scale the input features:

```
from sklearn.pipeline import Pipeline

num_pipeline = Pipeline([
    ("impute", SimpleImputer(strategy="median")),
    ("standardize", StandardScaler()),
])

```

The `Pipeline` constructor takes a list of name/estimator pairs (2-tuples) defining a sequence of steps. The names can be anything you like, as long as they are unique and don't contain double underscores, `__`. They will be useful later when we discuss hyperparameter tuning. The estimators must all be transformers (i.e., they must have a `fit_transform()` method), except for the last one, which can be anything: a transformer, a predictor, or any other type of estimator.



In a Jupyter notebook, if you `import sklearn` and run `sklearn.set_config(display="diagram")`, all Scikit-Learn estimators will be rendered as interactive diagrams. This is particularly useful to visualize pipelines. To visualize `num_pipeline`, run a cell with `num_pipeline` as the last line. Clicking on an estimator will show more details.

If you don't want to bother naming the transformers, you can use the `make_pipeline()` function instead: it just takes transformers as positional arguments, and it creates a `Pipeline` using the names of the transformers' classes, in lower case and without underscores (e.g., "simpleimputer"). In case multiple transformers have the same name, an index is appended to their names (e.g., "foo-1", "foo-2", etc.).

```
from sklearn.pipeline import make_pipeline

num_pipeline = make_pipeline(SimpleImputer(strategy="median"), StandardScaler())
```

When you call the pipeline's `fit()` method, it calls `fit_transform()` sequentially on all transformers, passing the output of each call as the parameter to the next call until it reaches the final estimator, for which it just calls the `fit()` method.

The pipeline exposes the same methods as the final estimator. In this example, the last estimator is a `StandardScaler` which is a transformer so the pipeline also acts like a transformer. If you call the pipeline's `transform()` method, it will sequentially apply all the transformations to the data. If the last estimator were a predictor instead of a transformer, then the pipeline would have a `predict()` method rather than a `transform()` method. Calling it would sequentially apply all the transformations to the data and pass the result to the predictor's `predict()` method.

Let's call the pipeline's `fit_transform()` method and look at the output's first two rows, rounded to 2 decimals:

```
>>> housing_num_prepared = num_pipeline.fit_transform(housing_num)
>>> housing_num_prepared[:2].round(2)
array([[-1.42,  1.01,  1.86,  0.31,  1.37,  0.14,  1.39, -0.94],
       [ 0.6 , -0.7 ,  0.91, -0.31, -0.44, -0.69, -0.37,  1.17]])
```

As we saw earlier, if you want to recover a nice DataFrame, you can use the pipeline's `get_feature_names_out()` method:

```
df_housing_num_prepared = pd.DataFrame(
    housing_num_prepared, columns=num_pipeline.get_feature_names_out(),
    index=housing_num.index)
```

Pipelines support indexing, for example `pipeline[1]` returns the second estimator in the pipeline, and `pipeline[:-1]` returns a Pipeline object containing all but the last estimator. You can also access the estimators via the `steps` attribute, which is a list of name/estimator pairs, or via the `name_steps` dictionary attribute which maps the names to the estimators. For example `num_pipeline["simpleimputer"]` returns the estimator named "simpleimputer".

So far, we have handled the categorical columns and the numerical columns separately. It would be more convenient to have a single transformer capable of handling all columns, applying the appropriate transformations to each column. For this, you can use a `ColumnTransformer`. For example, the following `ColumnTransformer` will apply `num_pipeline` (the one we just defined) to the numerical attributes, and `cat_pipeline` to the categorical attribute:

```
from sklearn.compose import ColumnTransformer

num_attribs = ["longitude", "latitude", "housing_median_age", "total_rooms",
               "total_bedrooms", "population", "households", "median_income"]
cat_attribs = ["ocean_proximity"]

cat_pipeline = make_pipeline(
    SimpleImputer(strategy="most_frequent"),
    OneHotEncoder(handle_unknown="ignore"))

preprocessing = ColumnTransformer([
    ("num", num_pipeline, num_attribs),
    ("cat", cat_pipeline, cat_attribs),
])
```

First we import the `ColumnTransformer` class, next we define the list of numerical and categorical column names, we construct a simple pipeline for categorical attributes, and lastly we construct a `ColumnTransformer`. Its constructor requires a list of triplets (3-tuples), each containing a name (which must be unique and not contain double

underscores), a transformer, and a list of names (or indices) of columns that the transformer should be applied to.



Instead of using a transformer, you can specify the string "drop" if you want the columns to be dropped, or you can specify "pass through" if you want the columns to be left untouched. By default, the remaining columns (i.e., the ones that were not listed) will be dropped, but you can set the `remainder` hyperparameter to any transformer (or to "passthrough") if you want these columns to be handled differently.

Since listing all the column names is not very convenient, Scikit-Learn provides a `make_column_selector()` function that returns a selector function you can use to automatically select all the features of a given type, such as numerical or categorical. This selector function can be passed to the `ColumnTransformer` instead of column names or indices. Moreover, if you don't care about naming the transformers, you can use `make_column_transformer()` which chooses the names for you, just like `make_pipeline()` does. For example, the following code creates the same `ColumnTransformer` as above, except the transformers are automatically named "pipeline-1" and "pipeline-2" instead of "num" and "cat":

```
from sklearn.compose import make_column_selector, make_column_transformer

preprocessing = make_column_transformer(
    (num_pipeline, make_column_selector(dtype_include=np.number)),
    (cat_pipeline, make_column_selector(dtype_include=np.object)),
)
```

Now we're ready to apply this `ColumnTransformer` to the housing data:

```
housing_prepared = preprocessing.fit_transform(housing)
```

Great! We have a preprocessing pipeline that takes the entire training data and applies each transformer to the appropriate columns, then concatenates the transformed columns horizontally (transformers must never change the number of rows). Once again this returns a NumPy array, but you can get the column names using `preprocessing.get_feature_names_out()` and wrap the data in a nice DataFrame as we did before.



The `OneHotEncoder` returns a sparse matrix, while the `num_pipeline` returns a dense matrix. When there is such a mix of sparse and dense matrices, the `ColumnTransformer` estimates the density of the final matrix (i.e., the ratio of nonzero cells), and it returns a sparse matrix if the density is lower than a given threshold (by default, `sparse_threshold=0.3`). In this example, it returns a dense matrix.

Your project's going really well, you're almost ready to train some models! You now want to create a single pipeline that will perform all the transformations you've experimented with up to now. Let's recap what the pipeline will do and why:

- Missing values in numerical features will be imputed by replacing them with the median, as most ML algorithms don't expect missing values. As for the categorical feature, any missing values will be replaced by the most frequent category.
- The categorical feature will be one-hot encoded, as most ML algorithms only accept numerical inputs.
- A few ratio features will be computed and added: `bedrooms_ratio`, `rooms_per_house` and `people_per_house`. Hopefully these will better correlate with the median housing value, and thereby help the ML models.
- A few cluster similarity features will also be added. These will likely be more useful to the model than latitude and longitude.
- Features with a long tail will be replaced by their logarithm, as most models prefer features with roughly uniform or Gaussian distributions.
- All numerical features will be standardized, as most ML algorithms prefer when all features have roughly the same scale.

The code that builds the pipeline to do all of this should look familiar to you by now:

```
def column_ratio(X):
    return X[:, [0]] / X[:, [1]]

def ratio_pipeline(name=None):
    return make_pipeline(
        SimpleImputer(strategy="median"),
        FunctionTransformer(column_ratio,
                            feature_names_out=lambda input_features: [name]),
        StandardScaler())

log_pipeline = make_pipeline(SimpleImputer(strategy="median"),
                             FunctionTransformer(np.log),
                             StandardScaler())

cluster_simil = ClusterSimilarity(n_clusters=10, gamma=1., random_state=42)
default_num_pipeline = make_pipeline(SimpleImputer(strategy="median"),
                                     StandardScaler())

preprocessing = ColumnTransformer([
    ("bedrooms_ratio", ratio_pipeline("bedrooms_ratio"),
     ["total_bedrooms", "total_rooms"]),
    ("rooms_per_house", ratio_pipeline("rooms_per_house"),
     ["total_rooms", "households"]),
    ("people_per_house", ratio_pipeline("people_per_house"),
     ["population", "households"]),
    ("log", log_pipeline, ["total_bedrooms", "total_rooms",
                          "log"]),
])
```

```

        "population", "households", "median_income"]),
        ("geo", cluster_simil, ["latitude", "longitude"]),
        ("cat", cat_pipeline, make_column_selector(dtype_include=np.object)),
    ],
    remainder=default_num_pipeline) # one column remaining: housing_median_age

```

If you run this column transformer, it performs all the transformations and outputs a NumPy array with 24 features:

```

>>> housing_prepared = preprocessing.fit_transform(housing)
>>> housing_prepared.shape
(16512, 24)
>>> preprocessing.get_feature_names_out()
array(['bedrooms_ratio_bedrooms_ratio',
       'rooms_per_house_rooms_per_house',
       'people_per_house_people_per_house', 'log_total_bedrooms',
       'log_total_rooms', 'log_population', 'log_households',
       'log_median_income', 'geo_Cluster 0 similarity', [...],
       'geo_Cluster 9 similarity', 'cat_ocean_proximity_<1H OCEAN',
       'cat_ocean_proximity_INLAND', 'cat_ocean_proximity_ISLAND',
       'cat_ocean_proximity_NEAR BAY', 'cat_ocean_proximity_NEAR OCEAN',
       'remainder_housing_median_age'], dtype=object)

```

Select and Train a Model

At last! You framed the problem, you got the data and explored it, you sampled a training set and a test set, and you wrote a preprocessing pipeline to automatically clean up and prepare your data for Machine Learning algorithms. You are now ready to select and train a Machine Learning model.

Training and Evaluating on the Training Set

The good news is that thanks to all these previous steps, things are now going to be easy! You decide to train a very basic Linear Regression model to get started:

```

from sklearn.linear_model import LinearRegression

lin_reg = make_pipeline(preprocessing, LinearRegression())
lin_reg.fit(housing, housing_labels)

```

Done! You now have a working Linear Regression model. Let's try it out on the training set, looking at the first 5 predictions and comparing them to the labels:

```

>>> housing_predictions = lin_reg.predict(housing)
>>> housing_predictions[:5].round(-2) # -2 = rounded to the nearest hundred
array([243700., 372400., 128800., 94400., 328300.])
>>> housing_labels.iloc[:5].values
array([458300., 483800., 101700., 96100., 361800.])

```

Well, it works, but not always: the first prediction is way off (by over \$200,000!), while the other predictions are better: two are off by about 25%, and two are off

by less than 10%. Remember that you chose to use the RMSE as your performance measure, so you want to measure this regression model's RMSE on the whole training set using Scikit-Learn's `mean_squared_error()` function, with the `squared` argument set to `False`:

```
>>> from sklearn.metrics import mean_squared_error
>>> lin_rmse = mean_squared_error(housing_labels, housing_predictions,
...                                 squared=False)
...
>>> lin_rmse
68687.89176589991
```

This is better than nothing, but clearly not a great score: most districts' `median_housing_values` range between \$120,000 and \$265,000, so a typical prediction error of \$68,628 is really not very satisfying. This is an example of a model underfitting the training data. When this happens it can mean that the features do not provide enough information to make good predictions, or that the model is not powerful enough. As we saw in the previous chapter, the main ways to fix underfitting are to select a more powerful model, to feed the training algorithm with better features, or to reduce the constraints on the model. This model is not regularized, which rules out the last option. You could try to add more features, but first you want to try a more complex model to see how it does.

So you decide to try a `DecisionTreeRegressor`, as this is a fairly powerful model capable of finding complex nonlinear relationships in the data (Decision Trees are presented in more detail in [Chapter 6](#)):

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = make_pipeline(preprocessing, DecisionTreeRegressor(random_state=42))
tree_reg.fit(housing, housing_labels)
```

Now that the model is trained, you evaluate it on the training set:

```
>>> housing_predictions = tree_reg.predict(housing)
>>> tree_rmse = mean_squared_error(housing_labels, housing_predictions,
...                                 squared=False)
...
>>> tree_rmse
0.0
```

Wait, what!? No error at all? Could this model really be absolutely perfect? Of course, it is much more likely that the model has badly overfit the data. How can you be sure? As we saw earlier, you don't want to touch the test set until you are ready to launch a model you are confident about, so you need to use part of the training set for training and part of it for model validation.

Better Evaluation Using Cross-Validation

One way to evaluate the Decision Tree model would be to use the `train_test_split()` function to split the training set into a smaller training set and a validation set, then train your models against the smaller training set and evaluate them against the validation set. It's a bit of work, but nothing too difficult, and it would work fairly well.

A great alternative is to use Scikit-Learn's *K-fold cross-validation* feature. The following code randomly splits the training set into 10 non-overlapping subsets called *folds*, then it trains and evaluates the Decision Tree model 10 times, picking a different fold for evaluation every time and using the other 9 folds for training. The result is an array containing the 10 evaluation scores:

```
from sklearn.model_selection import cross_val_score

tree_rmses = -cross_val_score(tree_reg, housing, housing_labels,
                             scoring="neg_root_mean_squared_error", cv=10)
```



Scikit-Learn's cross-validation features expect a utility function (greater is better) rather than a cost function (lower is better), so the scoring function is actually the opposite of the RMSE. It's a negative value, which is why we need to switch the sign of the output to get the RMSE scores.

Let's look at the results:

```
>>> pd.Series(tree_rmses).describe()
count    10.000000
mean    66868.027288
std     2060.966425
min     63649.536493
25%    65338.078316
50%    66801.953094
75%    68229.934454
max    70094.778246
dtype: float64
```

Now the Decision Tree doesn't look as good as it did earlier. In fact, it seems to perform almost as poorly as the Linear Regression model! Notice that cross-validation allows you to get not only an estimate of the performance of your model, but also a measure of how precise this estimate is (i.e., its standard deviation). The Decision Tree has an RMSE of about 66,868, with a standard deviation of about 2,061. You would not have this information if you just used one validation set. But cross-validation comes at the cost of training the model several times, so it is not always feasible.

If you compute the same metric for the Linear Regression model, you will find that the mean RMSE is 69,858 and the standard deviation is 4,182. So the Decision Tree model seems to perform very slightly better than the linear model, but only marginally better due to severe overfitting. We know there's an overfitting problem because the training error is low (actually zero) while the validation error is high.

Let's try one last model now: the `RandomForestRegressor`. As we will see in [Chapter 7](#), Random Forests work by training many Decision Trees on random subsets of the features, then averaging out their predictions. Such models composed of many other models are called *ensembles*: they are capable of boosting the performance of the underlying model (in this case, Decision Trees). The code is much the same as earlier:

```
from sklearn.ensemble import RandomForestRegressor

forest_reg = make_pipeline(preprocessing,
                           RandomForestRegressor(random_state=42))
forest_rmses = -cross_val_score(forest_reg, housing, housing_labels,
                               scoring="neg_root_mean_squared_error", cv=10)
```

Let's look at these scores:

```
>>> pd.Series(forest_rmses).describe()
count    10.000000
mean     47019.561281
std      1033.957120
min      45458.112527
25%     46464.031184
50%     46967.596354
75%     47325.694987
max     49243.765795
dtype: float64
```

Wow, this is much better: Random Forests really look very promising for this task! However, if you train a `RandomForest` and measure the RMSE on the training set, you will find roughly 17,474: that's much lower, meaning that there's still quite a lot of overfitting going on. Possible solutions are to simplify the model, constrain it (i.e., regularize it), or get a lot more training data. Before you dive much deeper into Random Forests, however, you should try out many other models from various categories of Machine Learning algorithms (e.g., several Support Vector Machines with different kernels, and possibly a neural network), without spending too much time tweaking the hyperparameters. The goal is to shortlist a few (two to five) promising models.

Fine-Tune Your Model

Let's assume that you now have a shortlist of promising models. You now need to fine-tune them. Let's look at a few ways you can do that.

Grid Search

One option would be to fiddle with the hyperparameters manually, until you find a great combination of hyperparameter values. This would be very tedious work, and you may not have time to explore many combinations.

Instead, you can use Scikit-Learn's `GridSearchCV` class to search for you. All you need to do is tell it which hyperparameters you want it to experiment with and what values to try out, and it will use cross-validation to evaluate all the possible combinations of hyperparameter values. For example, the following code searches for the best combination of hyperparameter values for the `RandomForestRegressor`:

```
from sklearn.model_selection import GridSearchCV

full_pipeline = Pipeline([
    ("preprocessing", preprocessing),
    ("random_forest", RandomForestRegressor(random_state=42)),
])
param_grid = [
    {'preprocessing__geo_n_clusters': [5, 8, 10],
     'random_forest__max_features': [4, 6, 8]},
    {'preprocessing__geo_n_clusters': [10, 15],
     'random_forest__max_features': [6, 8, 10]},
]
grid_search = GridSearchCV(full_pipeline, param_grid, cv=3,
                           scoring='neg_root_mean_squared_error')
grid_search.fit(housing, housing_labels)
```

Notice that you can refer to any hyperparameter of any estimator in a pipeline, even if this estimator is nested deep inside several pipelines and column transformers. For example, when Scikit-Learn sees "`preprocessing__geo_n_clusters`", it splits this string at the double underscores, then it looks for an estimator named "`preprocessing`" inside the pipeline and finds the `preprocessing` `ColumnTransformer`. Then it looks for a transformer named "`geo`" inside this `ColumnTransformer` and finds the `ClusterSimilarity` transformer we used on the latitude and longitude attributes. The it finds this transformer's `n_clusters` hyperparameter. Similarly, `random_forest__max_features` refers to the `max_features` hyperparameter of the estimator named "`random_forest`", which is of course the `RandomForest` model (the `max_features` hyperparameter will be explained in [Chapter 7](#)).



Wrapping preprocessing steps in a Scikit-Learn pipeline allows you to tune the preprocessing hyperparameters along with the model hyperparameters. This is a good thing since they often interact. For example, perhaps increasing `n_clusters` requires increasing `max_features` as well. If fitting the pipeline transformers is computationally expensive, you can set the pipeline's `memory` hyperparameter to the path of a caching directory: when you first fit the pipeline, Scikit-Learn will save the fitted transformers to this directory. If you then fit the pipeline again with the same hyperparameters, Scikit-Learn will just load the cached transformers.

There are two dictionaries in this `param_grid`, so `GridSearchCV` will first evaluate all $3 \times 3 = 9$ combinations of `n_clusters` and `max_features` hyperparameter values specified in the first `dict`, then it will try all $2 \times 3 = 6$ combinations of hyperparameter values in the second `dict`. So in total the grid search will explore $9 + 6 = 15$ combinations of hyperparameter values, and it will train the pipeline 3 times per combination, since we are using three-fold cross validation. So there will be a grand total of $15 \times 3 = 45$ rounds of training! It may take a while, but when it is done you can get the best combination of parameters like this:

```
>>> grid_search.best_params_
{'preprocessing__geo_n_clusters': 15, 'random_forest__max_features': 6}
```

In this example, the best model is obtained by setting `n_clusters` to 15 and `max_features` to 8.



Since 15 is the maximum value that was evaluated for `n_clusters`, you should probably try searching again with higher values; the score may continue to improve.

The best estimator can be accessed using `grid_search.best_estimator_`. If `GridSearchCV` is initialized with `refit=True` (which is the default), then once it finds the best estimator using cross-validation, it retrains it on the whole training set. This is usually a good idea, since feeding it more data will likely improve its performance.

The evaluation scores are available using `grid_search.cv_results_`. This is a dictionary, but if you wrap it in a `DataFrame`, you get a nice list of all the test scores for each combination of hyperparameters and for each cross-validation split, as well as the mean test score across all splits.

```
>>> cv_res = pd.DataFrame(grid_search.cv_results_)
>>> cv_res.sort_values(by="mean_test_score", ascending=False, inplace=True)
>>> [...] # change column names to fit on this page, and show rmse = -score
>>> cv_res.head() # note: the 1st column is the row ID
```

	n_clusters	max_features	split0	split1	split2	mean_test_rmse
12	15	6	43460	43919	44748	44042
13	15	8	44132	44075	45010	44406
14	15	10	44374	44286	45316	44659
7	10	6	44683	44655	45657	44999
9	10	6	44683	44655	45657	44999

The mean test RMSE score for the best model is 44,042, which is better than the score you got earlier using the default hyperparameter values (which was 47,019). Congratulations, you have successfully fine-tuned your best model!

Randomized Search

The grid search approach is fine when you are exploring relatively few combinations, like in the previous example, but `RandomizedSearchCV` is often preferable, especially when the hyperparameter search space is large. This class can be used in much the same way as the `GridSearchCV` class, but instead of trying out all possible combinations, it evaluates a fixed number of combinations, selecting a random value for each hyperparameter at every iteration. This may sound surprising, but this approach has several benefits:

- If some of your hyperparameters are continuous (or discrete but with many possible values), and you let randomized search run for, say, 1,000 iterations, then it will explore 1,000 different values for each of these hyperparameters, whereas grid search would only explore the few values you listed for each one.
- Suppose a hyperparameter does not actually make much difference, but you don't know it yet. If it has 10 possible values and you add it to your grid search, then training will take 10 times longer. But if you add it to a random search, it will not make any difference.
- Suppose there are 6 hyperparameters to explore, each with 10 possible values, then grid search offers no other choice than training the model a million times, whereas random search can always run for any number of iterations you choose.

For each hyperparameter, you must provide either a list of possible values, or a probability distribution:

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

param_distributions = {'preprocessing__geo_n_clusters': randint(low=3, high=50),
                      'random_forest__max_features': randint(low=2, high=20)}

rnd_search = RandomizedSearchCV(
    full_pipeline, param_distributions=param_distributions, n_iter=10, cv=3,
    scoring='neg_root_mean_squared_error', random_state=42)

rnd_search.fit(housing, housing_labels)
```

Scikit-Learn also has two other hyperparameter search classes: `HalvingRandomSearchCV` and `HalvingGridSearchCV`. Their goal is to use the computational resources more efficiently, either to train faster or to explore a larger hyperparameter space. Here's how they work: in the first round, many hyperparameter combinations (called "candidates") are generated using either the grid approach or the random approach. These candidates are then used to train models which are then evaluated using cross-validation, as usual. However, training uses limited resources which speeds up this first round considerably. By default, "limited resources" means that the models are trained on a small part of the training set. However, other limitations are possible, such as reducing the number of training iterations if the model has a hyperparameter to set it. Once every candidate has been evaluated, only the best ones go on to the second round, where they are allowed more resources to compete. After several rounds, the final candidates are evaluated using full resources. This may save you some time tuning hyperparameters.

Ensemble Methods

Another way to fine-tune your system is to try to combine the models that perform best. The group (or "ensemble") will often perform better than the best individual model—just like Random Forests perform better than the individual Decision Trees they rely on—especially if the individual models make very different types of errors. For example, you could train and fine-tune a k-Nearest Neighbors model, then create an ensemble model that just predicts the mean of the random forest prediction and the KNN's prediction. We will cover this topic in more detail in [Chapter 7](#).

Analyze the Best Models and Their Errors

You will often gain good insights on the problem by inspecting the best models. For example, the `RandomForestRegressor` can indicate the relative importance of each attribute for making accurate predictions:

```
>>> final_model = rnd_search.best_estimator_ # includes preprocessing
>>> feature_importances = final_model["random_forest"].feature_importances_
>>> feature_importances.round(2)
array([0.07, 0.05, 0.05, 0.01, 0.01, 0.01, 0.01, 0.19, [...], 0.01])
```

Let's sort these importance scores in descending order and display them next to their corresponding attribute names:

```
>>> sorted(zip(feature_importances,
...             final_model["preprocessing"].get_feature_names_out(),
...             reverse=True))
...[(0.18694559869103852, 'log_median_income'),
 (0.0748194905715524, 'cat_ocean_proximity_INLAND'),
 (0.06926417748515576, 'bedrooms_ratio_bedrooms_ratio'),
 (0.05446998753775219, 'rooms_per_house_rooms_per_house'),
```

```
(0.05262301809680712, 'people_per_house__people_per_house'),
(0.03819415873915732, 'geo_Cluster_0_similarity'),
[...]
(0.00015061247730531558, 'cat_ocean_proximity_NEAR BAY'),
(7.301686597099842e-05, 'cat_ocean_proximity_ISLAND'))
```

With this information, you may want to try dropping some of the less useful features (e.g., apparently only one `ocean_proximity` category is really useful, so you could try dropping the others).



The `sklearn.feature_selection.SelectFromModel` transformer can automatically drop the least useful features for you: when you fit it, it trains a model (typically a random forest), it looks at its `feature_importances_` attribute, and it selects the most useful features. Then when you call `transform()`, it just drops the other features.

You should also look at the specific errors that your system makes, then try to understand why it makes them and what could fix the problem: adding extra features or getting rid of uninformative ones, cleaning up outliers, etc.

Now is also a good time to ensure that your model not only works well on average, but also on all categories of districts, whether they're rural or urban, rich or poor, North or South, minority or not, etc. This requires a bit of work creating subsets of your validation set for each category, but it's important: if your model performs poorly on a whole category of districts, then it should probably not be deployed until the issue is solved, or at least it should not be used to make predictions for that category, as it may do more harm than good.

Evaluate Your System on the Test Set

After tweaking your models for a while, you eventually have a system that performs sufficiently well. You are ready to evaluate the final model on the test set. There is nothing special about this process; just get the predictors and the labels from your test set, and run your `final_model` to transform the data and make predictions, then evaluate these predictions:

```
X_test = strat_test_set.drop("median_house_value", axis=1)
y_test = strat_test_set["median_house_value"].copy()

final_predictions = final_model.predict(X_test)

final_rmse = mean_squared_error(y_test, final_predictions, squared=False)
print(final_rmse) # prints 41424.40026462184
```

In some cases, such a point estimate of the generalization error will not be quite enough to convince you to launch: what if it is just 0.1% better than the model cur-

rently in production? You might want to have an idea of how precise this estimate is. For this, you can compute a 95% *confidence interval* for the generalization error using `scipy.stats.t.interval()`. We get a fairly large interval from 39,275 to 43,467, and our previous point estimate of 41,424 is roughly in the middle of it:

```
>>> from scipy import stats
>>> confidence = 0.95
>>> squared_errors = (final_predictions - y_test) ** 2
>>> np.sqrt(stats.t.interval(confidence, len(squared_errors) - 1,
...                           loc=squared_errors.mean(),
...                           scale=stats.sem(squared_errors)))
...
array([39275.40861216, 43467.27680583])
```

If you did a lot of hyperparameter tuning, the performance will usually be slightly worse than what you measured using cross-validation. That's because your system ends up fine-tuned to perform well on the validation data and will likely not perform as well on unknown datasets. It is not the case in this example since the test RMSE is lower than the validation RMSE, but when it happens you must resist the temptation to tweak the hyperparameters to make the numbers look good on the test set; the improvements would be unlikely to generalize to new data.

Now comes the project prelaunch phase: you need to present your solution (highlighting what you have learned, what worked and what did not, what assumptions were made, and what your system's limitations are), document everything, and create nice presentations with clear visualizations and easy-to-remember statements (e.g., "the median income is the number one predictor of housing prices"). In this California housing example, the final performance of the system is not much better than the experts' price estimates which were often off by 30%, but it may still be a good idea to launch it, especially if this frees up some time for the experts so they can work on more interesting and productive tasks.

Launch, Monitor, and Maintain Your System

Perfect, you got approval to launch! You now need to get your solution ready for production (e.g., polish the code, write documentation and tests, and so on). Then you can deploy your model to your production environment. The most basic way to do this is just to save the best model you trained, transfer the file to your production environment, and load it. To save the model, you can use the `joblib` library like this:

```
import joblib

joblib.dump(final_model, "my_california_housing_model.pkl")
```



It's often a good idea to save every model you experiment with so that you can come back easily to any model you want. You may also save the cross-validation scores and perhaps the actual predictions on the validation set. This will allow you to easily compare scores across model types, and compare the types of errors they make.

Once your model is transferred to production, you can load it and use it. For this you must first import any custom classes and functions the model relies on (which means transferring the code to production), then load the model using `joblib` and use it to make predictions:

```
import joblib
[...] # import KMeans, BaseEstimator, TransformerMixin, rbf_kernel, etc.

def column_ratio(X):
    return X[:, [0]] / X[:, [1]]

class ClusterSimilarity(BaseEstimator, TransformerMixin):
    [...]

final_model_reloaded = joblib.load("my_california_housing_model.pkl")

new_data = [...] # some new districts to make predictions for
predictions = final_model_reloaded.predict(new_data)
```

For example, perhaps the model will be used within a website: the user will type in some data about a new district and click the “Estimate Price” button. This will send a query containing the data to the web server, which will forward it to your web application, and finally your code will simply call the model’s `predict()` method (you want to load the model upon server startup, rather than every time the model is used). Alternatively, you can wrap the model within a dedicated web service that your web application can query through a REST API¹³ (see Figure 2-20). This makes it easier to upgrade your model to new versions without interrupting the main application. It also simplifies scaling, since you can start as many web services as needed and load-balance the requests coming from your web application across these web services. Moreover, it allows your web application to use any programming language, not just Python.

¹³ In a nutshell, a REST (or RESTful) API is an HTTP-based API that follows some conventions, such as using standard HTTP verbs to read, update, create, or delete resources (GET, POST, PUT, and DELETE) and using JSON for the inputs and outputs.

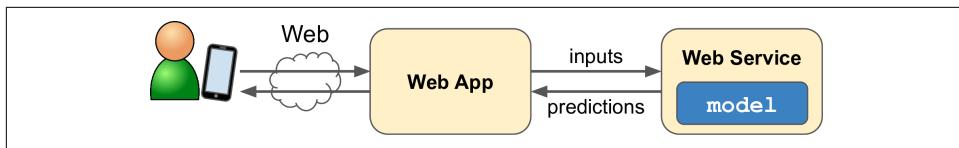


Figure 2-20. A model deployed as a web service and used by a web application

Another popular strategy is to deploy your model to the cloud, for example on Google’s Vertex AI (formerly known as Google Cloud AI Platform and Google Cloud ML Engine): just save your model using `joblib` and upload it to Google Cloud Storage (GCS), then head over to Google Cloud AI Platform and create a new model version, pointing it to the GCS file. That’s it! This gives you a simple web service that takes care of load balancing and scaling for you. It takes JSON requests containing the input data (e.g., of a district) and returns JSON responses containing the predictions. You can then use this web service in your website (or whatever production environment you are using). As we will see in [Chapter 19](#), deploying TensorFlow models on Vertex AI is not much different from deploying Scikit-Learn models.

But deployment is not the end of the story. You also need to write monitoring code to check your system’s live performance at regular intervals and trigger alerts when it drops. It may drop very quickly, for example if a component breaks in your infrastructure, but be aware that it could also decay very slowly, which can easily go unnoticed for a long time. This is quite common because of model rot: if the model was trained with last year’s data, it may not be adapted to today’s data.

So you need to monitor your model’s live performance. But how do you that? Well, it depends. In some cases, the model’s performance can be inferred from downstream metrics. For example, if your model is part of a recommender system and it suggests products that the users may be interested in, then it’s easy to monitor the number of recommended products sold each day. If this number drops (compared to non-recommended products), then the prime suspect is the model. This may be because the data pipeline is broken, or perhaps the model needs to be retrained on fresh data (as we will discuss shortly).

However, you may also need human analysis to determine the model’s performance. For example, suppose you trained an image classification model (which we will see in [Chapter 3](#)) to detect several product defects on a production line. How can you get an alert if the model’s performance drops, before thousands of defective products get shipped to your clients? One solution is to send to human raters a sample of all the pictures that the model classified (especially pictures that the model wasn’t so sure about). Depending on the task, the raters may need to be experts, or they could be nonspecialists, such as workers on a crowdsourcing platform (e.g., Amazon

Mechanical Turk). In some applications they could even be the users themselves, responding for example via surveys or repurposed captchas.¹⁴

Either way, you need to put in place a monitoring system (with or without human raters to evaluate the live model), as well as all the relevant processes to define what to do in case of failures and how to prepare for them. Unfortunately, this can be a lot of work. In fact, it is often much more work than building and training a model.

If the data keeps evolving, you will need to update your datasets and retrain your model regularly. You should probably automate the whole process as much as possible. Here are a few things you can automate:

- Collect fresh data regularly and label it (e.g., using human raters).
- Write a script to train the model and fine-tune the hyperparameters automatically. This script could run automatically, for example every day or every week, depending on your needs.
- Write another script that will evaluate both the new model and the previous model on the updated test set, and deploy the model to production if the performance has not decreased (if it did, make sure you investigate why). The script should probably test the performance of your model on various subsets of the test set, such as poor or rich districts, rural or urban districts, etc.

You should also make sure you evaluate the model's input data quality. Sometimes performance will degrade slightly because of a poor-quality signal (e.g., a malfunctioning sensor sending random values, or another team's output becoming stale), but it may take a while before your system's performance degrades enough to trigger an alert. If you monitor your model's inputs, you may catch this earlier. For example, you could trigger an alert if more and more inputs are missing a feature, or if its mean or standard deviation drifts too far from the training set, or a categorical feature starts containing new categories.

Finally, make sure you keep backups of every model you create and have the process and tools in place to roll back to a previous model quickly, in case the new model starts failing badly for some reason. Having backups also makes it possible to easily compare new models with previous ones. Similarly, you should keep backups of every version of your datasets so that you can roll back to a previous dataset if the new one ever gets corrupted (e.g., if the fresh data that gets added to it turns out to be full of outliers). Having backups of your datasets also allows you to evaluate any model against any previous dataset.

¹⁴ A captcha is a test to ensure a user is not a robot. These tests have often been used as a cheap way to label training data.

As you can see, Machine Learning involves quite a lot of infrastructure. Chapter 19 discusses some aspects of this, but it's a very broad topic called *ML Operations* (ML Ops) which deserves its own book. So don't be surprised if your first ML project takes a lot of effort and time to build and deploy to production. Fortunately, once all the infrastructure is in place, going from idea to production will be much faster.

Try It Out!

Hopefully this chapter gave you a good idea of what a Machine Learning project looks like as well as showing you some of the tools you can use to train a great system. As you can see, much of the work is in the data preparation step: building monitoring tools, setting up human evaluation pipelines, and automating regular model training. The Machine Learning algorithms are important, of course, but it is probably preferable to be comfortable with the overall process and know three or four algorithms well rather than to spend all your time exploring advanced algorithms.

So, if you have not already done so, now is a good time to pick up a laptop, select a dataset that you are interested in, and try to go through the whole process from A to Z. A good place to start is on a competition website such as <https://kaggle.com/>: you will have a dataset to play with, a clear goal, and people to share the experience with. Have fun!

Exercises

The following exercises are based on this chapter's housing dataset:

1. Try a Support Vector Machine regressor (`sklearn.svm.SVR`) with various hyperparameters, such as `kernel="linear"` (with various values for the `C` hyperparameter) or `kernel="rbf"` (with various values for the `C` and `gamma` hyperparameters). Note that SVMs don't scale well to large datasets, so you should probably train your model on just the first 5,000 instances of the training set and use only 3-fold cross-validation, or else it will take hours. Don't worry about what the hyperparameters mean for now, we'll discuss them in Chapter 5. How does the best SVR predictor perform?
2. Try replacing the `GridSearchCV` with a `RandomizedSearchCV`.
3. Try adding a `SelectFromModel` transformer in the preparation pipeline to select only the most important attributes.
4. Try creating a custom transformer that trains a k-Nearest Neighbors regressor (`sklearn.neighbors.KNeighborsRegressor`) in its `fit()` method, and outputs the model's predictions in its `transform()` method. Then add this feature to the preprocessing pipeline, using latitude and longitude as the inputs to this

transformer. This will add a feature in the model that corresponds to the housing median price of the nearest districts.

5. Automatically explore some preparation options using `GridSearchCV`.
6. Try to implement the `StandardScalerClone` class again from scratch, then add support for the `inverse_transform()` method: executing `scaler.inverse_transform(scaler.fit_transform(X))` should return an array very close to `X`. Then add support for feature names: set `feature_names_in_` in the `fit()` method if the input is a `DataFrame`. This attribute should be a NumPy array of column names. Lastly, implement the `get_feature_names_out()` method: it should have one optional `input_features=None` argument. If passed, the method should check that its length matches `n_features_in_`, and it should match `feature_names_in_` if it is defined, then `input_features` should be returned. If `input_features` is `None`, then the method should return `feature_names_in_` if it is defined or `np.array(["x0", "x1", ...])` with length `n_features_in_` otherwise.

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab3>.

CHAPTER 3

Classification

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. The GitHub repo is <https://github.com/ageron/handsonml3>.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

In [Chapter 1](#) I mentioned that the most common supervised learning tasks are regression (predicting values) and classification (predicting classes). In [Chapter 2](#) we explored a regression task, predicting housing values, using various algorithms such as Linear Regression, Decision Trees, and Random Forests (which will be explained in further detail in later chapters). Now we will turn our attention to classification systems.

MNIST

In this chapter we will be using the MNIST dataset, which is a set of 70,000 small images of digits handwritten by high school students and employees of the US Census Bureau. Each image is labeled with the digit it represents. This set has been studied so much that it is often called the “hello world” of Machine Learning: whenever people come up with a new classification algorithm they are curious to see how it will perform on MNIST, and anyone who learns Machine Learning tackles this dataset sooner or later.

Scikit-Learn provides many helper functions to download popular datasets. MNIST is one of them. The following code fetches the MNIST dataset from OpenML.org.¹

```
from sklearn.datasets import fetch_openml

mnist = fetch_openml('mnist_784', as_frame=False)
```

The `sklearn.datasets` package contains mostly three types of functions: `fetch_*` functions such as `fetch_openml()` to download real-life datasets, `load_*` functions to load small toy datasets bundled with Scikit-Learn (so they don't need to be downloaded over the Internet), and `make_*` functions to generate fake datasets, useful for tests. Generated datasets are usually returned as an (X, y) tuple containing the input data and the targets, both as NumPy arrays. Other Datasets are returned as `sklearn.utils.Bunch` objects, which are dictionaries whose entries can also be accessed as attributes. They generally contain the following entries:

- "DESCR": a description of the dataset
- "data": the input data, usually as a 2D NumPy array
- "target": the labels, usually as a 1D NumPy array

The `fetch_openml()` function is a bit unusual since by default it returns the inputs as a Pandas DataFrame and the labels as a Pandas Series (unless the dataset is sparse). But the MNIST dataset contains images, and DataFrames aren't ideal for that, so it's preferable to set `as_frame=False` to get the data as NumPy arrays instead. Let's look at these arrays:

```
>>> X, y = mnist.data, mnist.target
>>> X
array([[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.]])
>>> X.shape
(70000, 784)
>>> y
array(['5', '0', '4', ..., '4', '5', '6'], dtype=object)
>>> y.shape
(70000,)
```

There are 70,000 images, and each image has 784 features. This is because each image is 28×28 pixels, and each feature simply represents one pixel's intensity, from 0

¹ By default Scikit-Learn caches downloaded datasets in a directory called `scikit_learn_data` in your home directory.

(white) to 255 (black). Let's take a peek at one digit from the dataset. All you need to do is grab an instance's feature vector, reshape it to a 28×28 array, and display it using Matplotlib's `imshow()` function (Figure 3-1). We use `cmap="binary"` to get a grayscale color map where 0 is white and 255 is black:

```
import matplotlib.pyplot as plt

def plot_digit(image_data):
    image = image_data.reshape(28, 28)
    plt.imshow(image, cmap="binary")
    plt.axis("off")

some_digit = X[0]
plot_digit(some_digit)
plt.show()
```

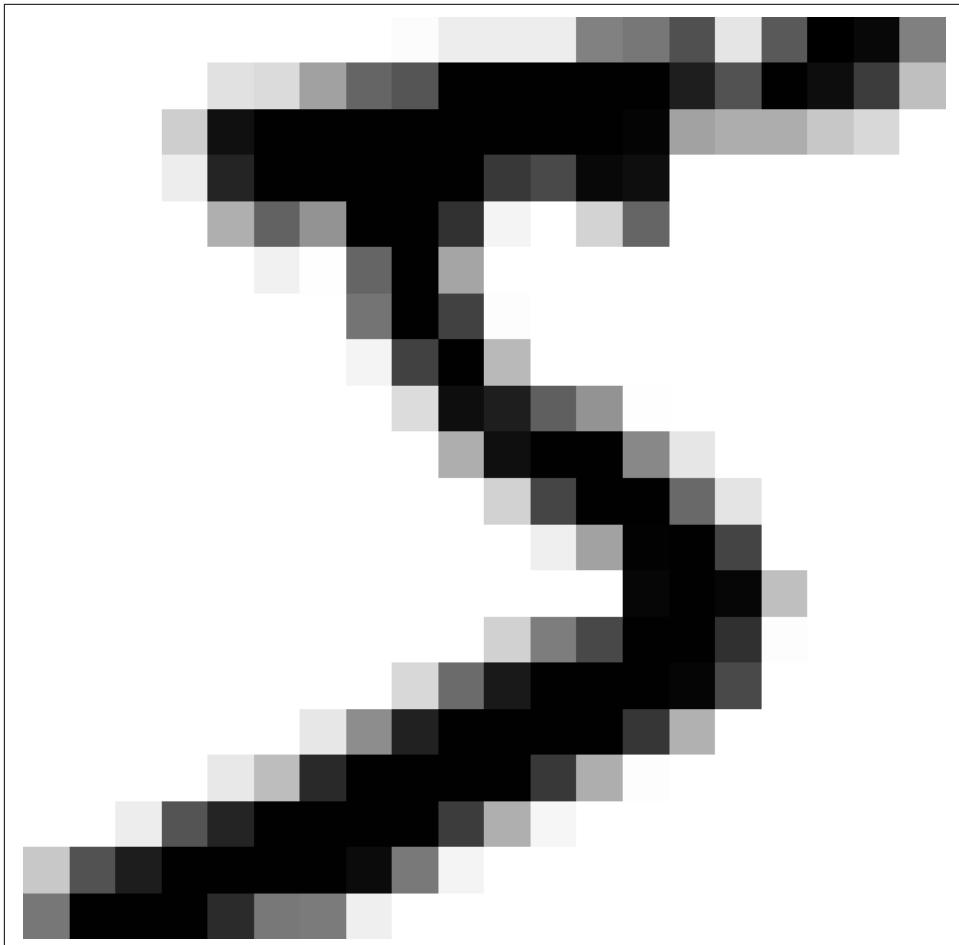


Figure 3-1. Example of an MNIST image

This looks like a 5, and indeed that's what the label tells us:

```
>>> y[0]  
'5'
```

To give you a feel for the complexity of the classification task, Figure 3-2 shows a few more images from the MNIST dataset.



Figure 3-2. Digits from the MNIST dataset

But wait! You should always create a test set and set it aside before inspecting the data closely. The MNIST dataset returned by `fetch_openml()` is actually already split into a training set (the first 60,000 images) and a test set (the last 10,000 images).²

```
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

The training set is already shuffled for us, which is good because this guarantees that all cross-validation folds will be similar (you don't want one fold to be missing some digits). Moreover, some learning algorithms are sensitive to the order of the training instances, and they perform poorly if they get many similar instances in a row. Shuffling the dataset ensures that this won't happen.³

² Datasets returned by `fetch_openml()` are not always shuffled or split.

Training a Binary Classifier

Let's simplify the problem for now and only try to identify one digit—for example, the number 5. This “5-detector” will be an example of a *binary classifier*, capable of distinguishing between just two classes, 5 and not-5. Let's create the target vectors for this classification task:

```
y_train_5 = (y_train == '5') # True for all 5s, False for all other digits
y_test_5 = (y_test == '5')
```

Now let's pick a classifier and train it. A good place to start is with a *Stochastic Gradient Descent* (SGD) classifier, using Scikit-Learn's `SGDClassifier` class. This classifier is capable of handling very large datasets efficiently. This is in part because SGD deals with training instances independently, one at a time, which also makes SGD well suited for online learning, as we will see later. Let's create an `SGDClassifier` and train it on the whole training set:

```
from sklearn.linear_model import SGDClassifier

sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train_5)
```

Now we can use it to detect images of the number 5:

```
>>> sgd_clf.predict([some_digit])
array([ True])
```

The classifier guesses that this image represents a 5 (`True`). Looks like it guessed right in this particular case! Now, let's evaluate this model's performance.

Performance Measures

Evaluating a classifier is often significantly trickier than evaluating a regressor, so we will spend a large part of this chapter on this topic. There are many performance measures available, so grab another coffee and get ready to learn many new concepts and acronyms!

Measuring Accuracy Using Cross-Validation

A good way to evaluate a model is to use cross-validation, just as you did in [Chapter 2](#). Let's use the `cross_val_score()` function to evaluate our `SGDClassifier` model, using K-fold cross-validation with three folds. Remember that K-fold cross-validation means splitting the training set into K folds (in this case, three), then

³ Shuffling may be a bad idea in some contexts—for example, if you are working on time series data (such as stock market prices or weather conditions). We will explore this in [Chapter 15](#).

training the model K times, holding out a different fold each time for evaluation (see Chapter 2):

```
>>> from sklearn.model_selection import cross_val_score
>>> cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([0.95035, 0.96035, 0.9604])
```

Wow! Above 95% accuracy (ratio of correct predictions) on all cross-validation folds? This looks amazing, doesn't it? Well, before you get too excited, let's look at a dummy classifier that just classifies every single image in the most frequent class, in this case the negative class (i.e., *not* a 5):

```
from sklearn.dummy import DummyClassifier

dummy_clf = DummyClassifier()
dummy_clf.fit(X_train, y_train_5)
print(any(dummy_clf.predict(X_train))) # prints False: no 5s detected
```

Can you guess this model's accuracy? Let's find out:

```
>>> cross_val_score(dummy_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([0.90965, 0.90965, 0.90965])
```

That's right, it has over 90% accuracy! This is simply because only about 10% of the images are 5s, so if you always guess that an image is *not* a 5, you will be right about 90% of the time. Beats Nostradamus.

This demonstrates why accuracy is generally not the preferred performance measure for classifiers, especially when you are dealing with *skewed datasets* (i.e., when some classes are much more frequent than others). A much better way to evaluate the performance of a classifier is to look at the *confusion matrix*.

Implementing Cross-Validation

Occasionally you will need more control over the cross-validation process than what Scikit-Learn provides off the shelf. In these cases, you can implement cross-validation yourself. The following code does roughly the same thing as Scikit-Learn's `cross_val_score()` function, and it prints the same result:

```
from sklearn.model_selection import StratifiedKFold
from sklearn.base import clone

skfolds = StratifiedKFold(n_splits=3) # add shuffle=True if the dataset is not
# already shuffled
for train_index, test_index in skfolds.split(X_train, y_train_5):
    clone_clf = clone(sgd_clf)
    X_train_folds = X_train[train_index]
    y_train_folds = y_train_5[train_index]
    X_test_fold = X_train[test_index]
    y_test_fold = y_train_5[test_index]
```

```

clone_clf.fit(X_train_folds, y_train_folds)
y_pred = clone_clf.predict(X_test_fold)
n_correct = sum(y_pred == y_test_fold)
print(n_correct / len(y_pred)) # prints 0.95035, 0.96035 and 0.9604

```

The `StratifiedKFold` class performs stratified sampling (as explained in [Chapter 2](#)) to produce folds that contain a representative ratio of each class. At each iteration the code creates a clone of the classifier, trains that clone on the training folds, and makes predictions on the test fold. Then it counts the number of correct predictions and outputs the ratio of correct predictions.

Confusion Matrix

The general idea of a confusion matrix is to count the number of times instances of class A are classified as class B, for all A/B pairs. For example, to know the number of times the classifier confused images of 8s with 0s, you would look at row #8, column #0 of the confusion matrix.

To compute the confusion matrix, you first need to have a set of predictions so that they can be compared to the actual targets. You could make predictions on the test set, but let's keep it untouched for now (remember that you want to use the test set only at the very end of your project, once you have a classifier that you are ready to launch). Instead, you can use the `cross_val_predict()` function:

```

from sklearn.model_selection import cross_val_predict

y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)

```

Just like the `cross_val_score()` function, `cross_val_predict()` performs K-fold cross-validation, but instead of returning the evaluation scores, it returns the predictions made on each test fold. This means that you get a clean prediction for each instance in the training set (by “clean” I mean “out-of-sample”: the model makes predictions on data that it never saw during training).

Now you are ready to get the confusion matrix using the `confusion_matrix()` function. Just pass it the target classes (`y_train_5`) and the predicted classes (`y_train_pred`):

```

>>> from sklearn.metrics import confusion_matrix
>>> cm = confusion_matrix(y_train_5, y_train_pred)
>>> cm
array([[53892,    687],
       [1891,   3530]])

```

Each row in a confusion matrix represents an *actual class*, while each column represents a *predicted class*. The first row of this matrix considers non-5 images (the *negative class*): 53,892 of them were correctly classified as non-5s (they are called *true negatives*), while the remaining 687 were wrongly classified as 5s (*false positives*, also

called *type I errors*). The second row considers the images of 5s (the *positive class*): 1,891 were wrongly classified as non-5s (*false negatives*, also called *type II errors*), while the remaining 3,530 were correctly classified as 5s (*true positives*). A perfect classifier would only have true positives and true negatives, so its confusion matrix would have nonzero values only on its main diagonal (top left to bottom right):

```
>>> y_train_perfect_predictions = y_train_5 # pretend we reached perfection
>>> confusion_matrix(y_train_5, y_train_perfect_predictions)
array([[54579,      0],
       [      0, 5421]])
```

The confusion matrix gives you a lot of information, but sometimes you may prefer a more concise metric. An interesting one to look at is the accuracy of the positive predictions; this is called the *precision* of the classifier (Equation 3-1).

Equation 3-1. Precision

$$\text{precision} = \frac{TP}{TP + FP}$$

TP is the number of true positives, and *FP* is the number of false positives.

A trivial way to have perfect precision is to create a classifier that always makes negative predictions, except for one single positive prediction on the instance it's most confident about. If this one prediction is correct, then the classifier has 100% precision ($\text{precision} = 1/1 = 100\%$). Obviously, such a classifier would not be very useful, since it would ignore all but one positive instance. So precision is typically used along with another metric named *recall*, also called *sensitivity* or the *true positive rate* (TPR): this is the ratio of positive instances that are correctly detected by the classifier (Equation 3-2).

Equation 3-2. Recall

$$\text{recall} = \frac{TP}{TP + FN}$$

FN is, of course, the number of false negatives.

If you are confused about the confusion matrix, Figure 3-3 may help.

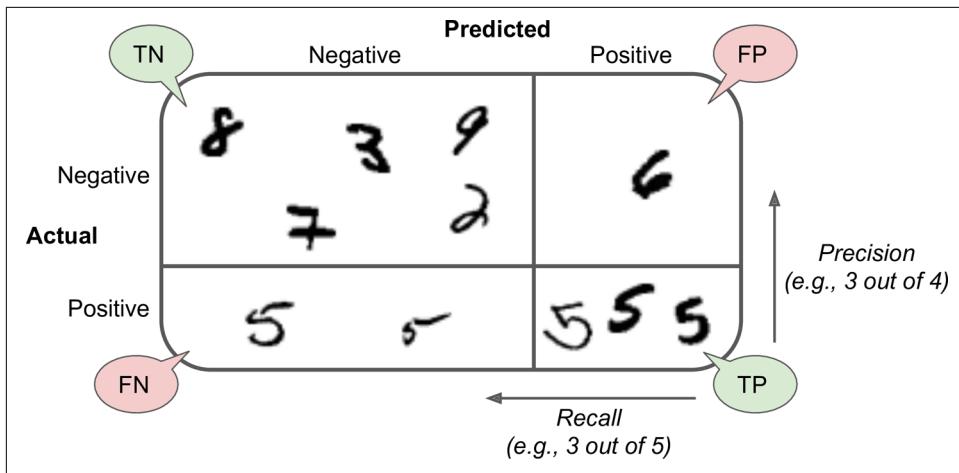


Figure 3-3. An illustrated confusion matrix shows examples of true negatives (top left), false positives (top right), false negatives (lower left), and true positives (lower right)

Precision and Recall

Scikit-Learn provides several functions to compute classifier metrics, including precision and recall:

```
>>> from sklearn.metrics import precision_score, recall_score
>>> precision_score(y_train_5, y_train_pred) # == 3530 / (687 + 3530)
0.8370879772350012
>>> recall_score(y_train_5, y_train_pred) # == 3530 / (1891 + 3530)
0.6511713705958311
```

Now your 5-detector does not look as shiny as it did when you looked at its accuracy. When it claims an image represents a 5, it is correct only 83.7% of the time. Moreover, it only detects 65.1% of the 5s.

It is often convenient to combine precision and recall into a single metric called the F_1 score, especially when you need a single metric to compare two classifiers. The F_1 score is the *harmonic mean* of precision and recall (Equation 3-3). Whereas the regular mean treats all values equally, the harmonic mean gives much more weight to low values. As a result, the classifier will only get a high F_1 score if both recall and precision are high.

Equation 3-3. F_1 score

$$F_1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = \frac{TP}{TP + \frac{FN + FP}{2}}$$

To compute the F_1 score, simply call the `f1_score()` function:

```
>>> from sklearn.metrics import f1_score  
>>> f1_score(y_train_5, y_train_pred)  
0.7325171197343846
```

The F_1 score favors classifiers that have similar precision and recall. This is not always what you want: in some contexts you mostly care about precision, and in other contexts you really care about recall. For example, if you trained a classifier to detect videos that are safe for kids, you would probably prefer a classifier that rejects many good videos (low recall) but keeps only safe ones (high precision), rather than a classifier that has a much higher recall but lets a few really bad videos show up in your product (in such cases, you may even want to add a human pipeline to check the classifier's video selection). On the other hand, suppose you train a classifier to detect shoplifters in surveillance images: it is probably fine if your classifier only has 30% precision as long as it has 99% recall (sure, the security guards will get a few false alerts, but almost all shoplifters will get caught).

Unfortunately, you can't have it both ways: increasing precision reduces recall, and vice versa. This is called the *precision/recall trade-off*.

Precision/Recall Trade-off

To understand this trade-off, let's look at how the `SGDClassifier` makes its classification decisions. For each instance, it computes a score based on a *decision function*. If that score is greater than a threshold, it assigns the instance to the positive class; otherwise it assigns it to the negative class. [Figure 3-4](#) shows a few digits positioned from the lowest score on the left to the highest score on the right. Suppose the *decision threshold* is positioned at the central arrow (between the two 5s): you will find 4 true positives (actual 5s) on the right of that threshold, and 1 false positive (actually a 6). Therefore, with that threshold, the precision is 80% (4 out of 5). But out of 6 actual 5s, the classifier only detects 4, so the recall is 67% (4 out of 6). If you raise the threshold (move it to the arrow on the right), the false positive (the 6) becomes a true negative, thereby increasing the precision (up to 100% in this case), but one true positive becomes a false negative, decreasing recall down to 50%. Conversely, lowering the threshold increases recall and reduces precision.

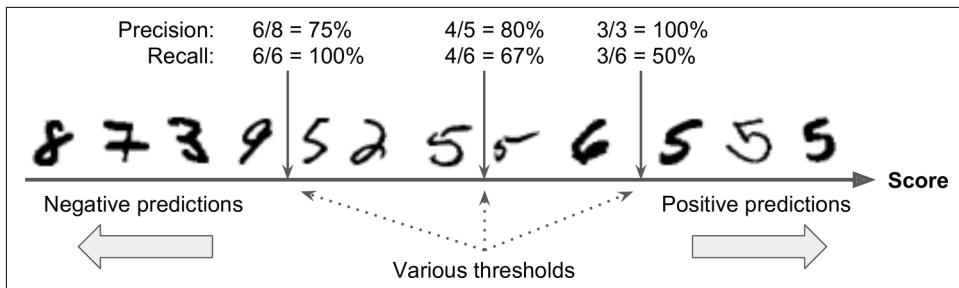


Figure 3-4. In this precision/recall trade-off, images are ranked by their classifier score, and those above the chosen decision threshold are considered positive; the higher the threshold, the lower the recall, but (in general) the higher the precision

Scikit-Learn does not let you set the threshold directly, but it does give you access to the decision scores that it uses to make predictions. Instead of calling the classifier's `predict()` method, you can call its `decision_function()` method, which returns a score for each instance, and then use any threshold you want to make predictions based on those scores:

```
>>> y_scores = sgd_clf.decision_function([some_digit])
>>> y_scores
array([2164.22030239])
>>> threshold = 0
>>> y_some_digit_pred = (y_scores > threshold)
array([ True])
```

The `SGDClassifier` uses a threshold equal to 0, so the previous code returns the same result as the `predict()` method (i.e., `True`). Let's raise the threshold:

```
>>> threshold = 3000
>>> y_some_digit_pred = (y_scores > threshold)
>>> y_some_digit_pred
array([False])
```

This confirms that raising the threshold decreases recall. The image actually represents a 5, and the classifier detects it when the threshold is 0, but it misses it when the threshold is increased to 3,000.

How do you decide which threshold to use? First, use the `cross_val_predict()` function to get the scores of all instances in the training set, but this time specify that you want to return decision scores instead of predictions:

```
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                             method="decision_function")
```

With these scores, use the `precision_recall_curve()` function to compute precision and recall for all possible thresholds (the function adds a last precision of 0 and a last recall of 1, corresponding to an infinite threshold):

```

from sklearn.metrics import precision_recall_curve

precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)

```

Finally, use Matplotlib to plot precision and recall as functions of the threshold value (Figure 3-5), and let's show the threshold of 3,000 we selected:

```

plt.plot(thresholds, precisions[:-1], "b--", label="Precision", linewidth=2)
plt.plot(thresholds, recalls[:-1], "g-", label="Recall", linewidth=2)
plt.vlines(threshold, 0, 1.0, "k", "dotted", label="threshold")
[...] # beautify the figure: add grid, legend, axis, labels and circles
plt.show()

```

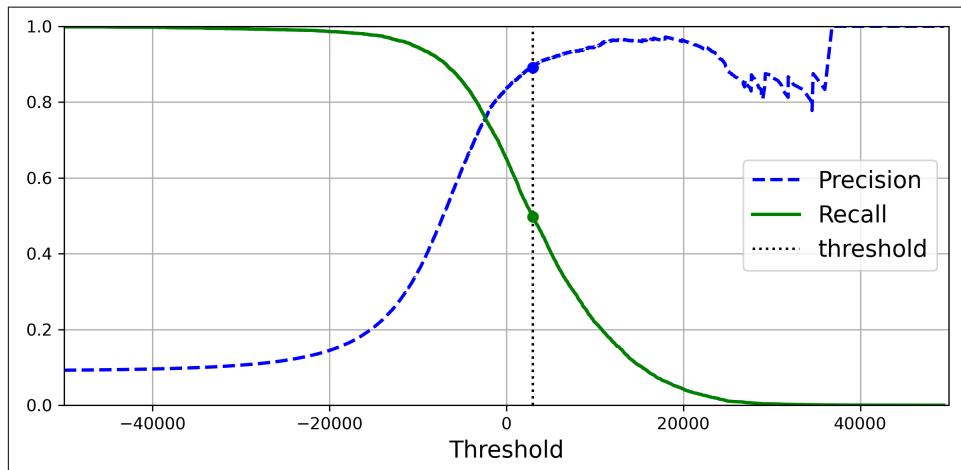


Figure 3-5. Precision and recall versus the decision threshold



You may wonder why the precision curve is bumpier than the recall curve in Figure 3-5. The reason is that precision may sometimes go down when you raise the threshold (although in general it will go up). To understand why, look back at Figure 3-4 and notice what happens when you start from the central threshold and move it just one digit to the right: precision goes from 4/5 (80%) down to 3/4 (75%). On the other hand, recall can only go down when the threshold is increased, which explains why its curve looks smooth.

At this threshold value, precision is near 90% and recall is around 50%. Another way to select a good precision/recall trade-off is to plot precision directly against recall, as shown in Figure 3-6 (the same threshold is shown).

```

plt.plot(recalls, precisions, linewidth=2, label="Precision/Recall curve")
[...] # beautify the figure: add labels, grid, legend, arrow and text
plt.show()

```

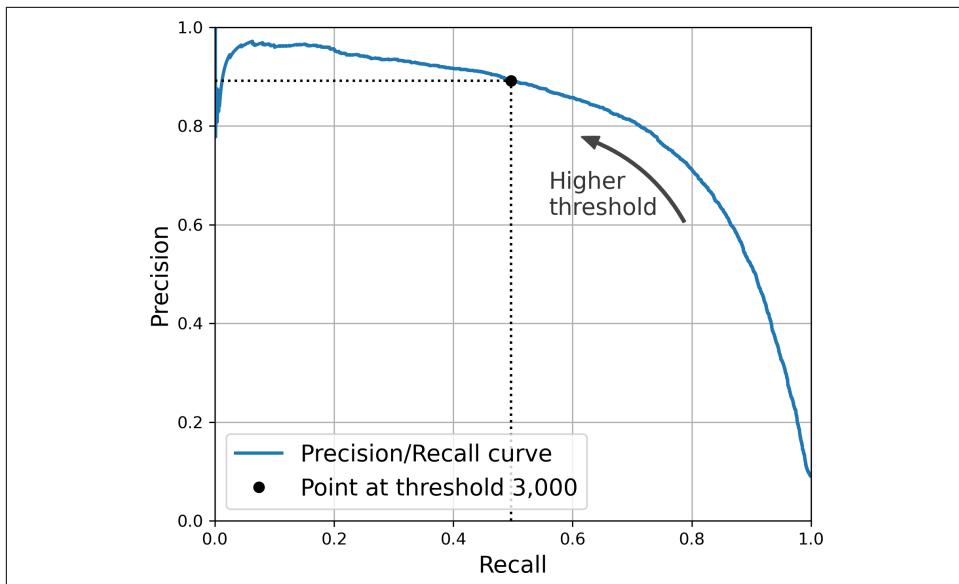


Figure 3-6. Precision versus recall

You can see that precision really starts to fall sharply around 80% recall. You will probably want to select a precision/recall trade-off just before that drop—for example, at around 60% recall. But of course, the choice depends on your project.

Suppose you decide to aim for 90% precision. You could use the first plot to find the threshold you need to use, but that's not very precise. Alternatively, you can search for the lowest threshold that gives you at least 90% precision. For this, we can use NumPy array's `argmax()` method, which returns the first index of the maximum value, which in this case means the first True value:

```
>>> idx_for_90_precision = (precisions >= 0.90).argmax()
>>> threshold_for_90_precision = thresholds[idx_for_90_precision]
>>> threshold_for_90_precision
3370.0194991439557
```

To make predictions (on the training set for now), instead of calling the classifier's `predict()` method, you can run this code:

```
y_train_pred_90 = (y_scores >= threshold_for_90_precision)
```

Let's check these predictions' precision and recall:

```
>>> precision_score(y_train_5, y_train_pred_90)
0.9000345901072293
>>> recall_at_90_precision = recall_score(y_train_5, y_train_pred_90)
>>> recall_at_90_precision
0.4799852425751706
```

Great, you have a 90% precision classifier! As you can see, it is fairly easy to create a classifier with virtually any precision you want: just set a high enough threshold, and you're done. But wait, not so fast: a high-precision classifier is not very useful if its recall is too low! For many applications, 48% recall wouldn't be great at all.



If someone says, “Let’s reach 99% precision,” you should ask, “At what recall?”

The ROC Curve

The *receiver operating characteristic* (ROC) curve is another common tool used with binary classifiers. It is very similar to the precision/recall curve, but instead of plotting precision versus recall, the ROC curve plots the *true positive rate* (another name for recall) against the *false positive rate* (FPR). The FPR (also called the *fall-out*) is the ratio of negative instances that are incorrectly classified as positive. It is equal to 1 – the *true negative rate* (TNR), which is the ratio of negative instances that are correctly classified as negative. The TNR is also called *specificity*. Hence, the ROC curve plots *sensitivity* (recall) versus 1 – *specificity*.

To plot the ROC curve, you first use the `roc_curve()` function to compute the TPR and FPR for various threshold values:

```
from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)
```

Then you can plot the FPR against the TPR using Matplotlib. The following code produces the plot in [Figure 3-7](#). To find the point that corresponds to 90% precision, we need to look for the index of the desired threshold. Since thresholds are listed in decreasing order in this case, we use `<=` instead of `>=` on the first line:

```
idx_for_threshold_at_90 = (thresholds <= threshold_for_90_precision).argmax()
tpr_90, fpr_90 = tpr[idx_for_threshold_at_90], fpr[idx_for_threshold_at_90]

plt.plot(fpr, tpr, linewidth=2, label="ROC curve")
plt.plot([0, 1], [0, 1], 'k:', label="Random classifier's ROC curve")
plt.plot([fpr_90], [tpr_90], "ko", label="Threshold for 90% precision")
[...] # beautify the figure: add labels, grid, legend, arrow and text
plt.show()
```

Once again there is a trade-off: the higher the recall (TPR), the more false positives (FPR) the classifier produces. The dotted line represents the ROC curve of a purely random classifier; a good classifier stays as far away from that line as possible (toward the top-left corner).

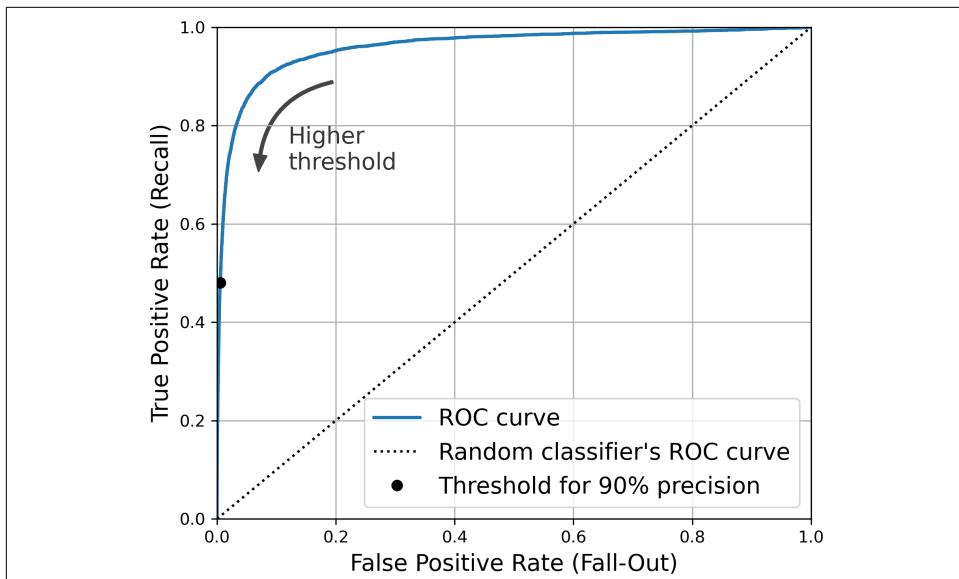


Figure 3-7. This ROC curve plots the false positive rate against the true positive rate for all possible thresholds; the black circle highlights the chosen ratio (at 90% precision and 48% recall)

One way to compare classifiers is to measure the *area under the curve* (AUC). A perfect classifier will have a ROC AUC equal to 1, whereas a purely random classifier will have a ROC AUC equal to 0.5. Scikit-Learn provides a function to estimate the ROC AUC:

```
>>> from sklearn.metrics import roc_auc_score
>>> roc_auc_score(y_train_5, y_scores)
0.9604938554008616
```



Since the ROC curve is so similar to the precision/recall (PR) curve, you may wonder how to decide which one to use. As a rule of thumb, you should prefer the PR curve whenever the positive class is rare or when you care more about the false positives than the false negatives. Otherwise, use the ROC curve. For example, looking at the previous ROC curve (and the ROC AUC score), you may think that the classifier is really good. But this is mostly because there are few positives (5s) compared to the negatives (non-5s). In contrast, the PR curve makes it clear that the classifier has room for improvement: the curve could really be closer to the top-right corner (see Figure 3-6 again).

Let's now create a `RandomForestClassifier`, and we will compare its PR curve and F_1 score to those of the `SGDClassifier`:

```
from sklearn.ensemble import RandomForestClassifier  
  
forest_clf = RandomForestClassifier(random_state=42)
```

The `precision_recall_curve()` function expects labels and scores for each instance, so we need to train the random forest and make it assign a score to each instance. But the `RandomForestClassifier` class does not have a `decision_function()` method, due to the way it works (we will cover this in [Chapter 7](#)). Luckily, it has a `predict_proba()` method which returns class probabilities for each instance, and we can just use the probability of the positive class as a score, it will work fine.⁴ So let's call the `cross_val_predict()` function to train the `RandomForestClassifier` using cross-validation and make it predict class probabilities for every image:

```
y_probas_forest = cross_val_predict(forest_clf, X_train_5, y_train_5, cv=3,  
method="predict_proba")
```

Let's look at the class probabilities for the first two images in the training set:

```
>>> y_probas_forest[:2]  
array([[0.11, 0.89],  
[0.99, 0.01]])
```

The model predicts that the first image is positive with 89% probability, and it predicts that the second image is negative with 99% probability. Since each image is either positive or negative, the probabilities in each row add up to 100%.



These are *estimated* probabilities, not actual probabilities. For example, if you look at all the images that the model classified as positive with an estimated probability between 50% and 60%, roughly 94% of them are actually positive. So the model's estimated probabilities were much too low in this case—but models can be overconfident as well. The `sklearn.calibration` package contains tools to calibrate the estimated probabilities and make them much closer to actual probabilities. See the extra material section in the notebook for more details.

The second column contains the estimated probabilities for the positive class, so let's pass them to the `precision_recall_curve()` function:

⁴ Scikit-Learn classifiers always have either a `decision_function()` method or a `predict_proba()` method, or sometimes both.

```

y_scores_forest = y_probas_forest[:, 1]
precisions_forest, recalls_forest, thresholds_forest = precision_recall_curve(
    y_train_5, y_scores_forest)

```

Now you are ready to plot the PR curve. It is useful to plot the first PR curve as well to see how they compare ([Figure 3-8](#)):

```

plt.plot(recalls_forest, precisions_forest, "b-", linewidth=2,
         label="Random Forest")
plt.plot(recalls, precisions, "--", linewidth=2, label="SGD")
[...] # beautify the figure: add labels, grid, and legend
plt.show()

```

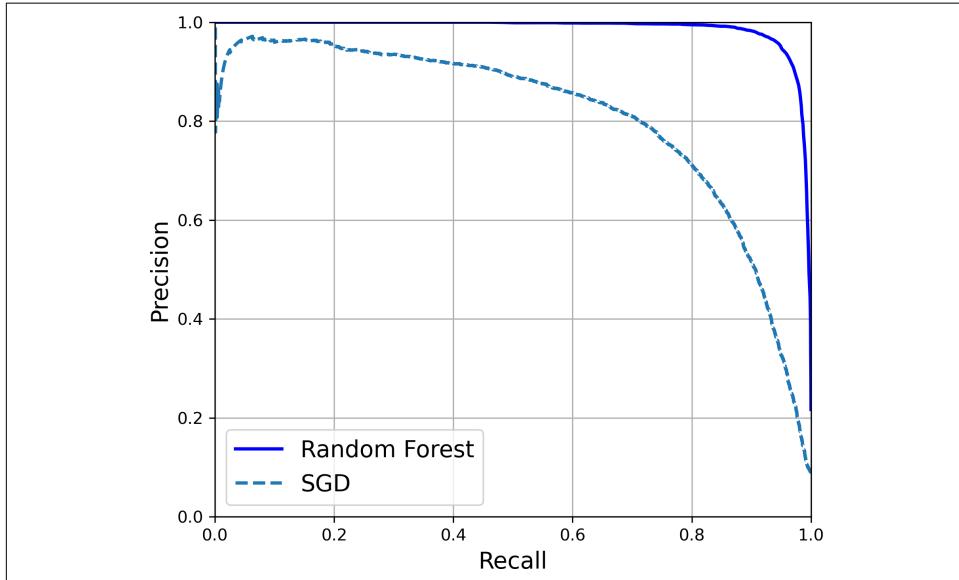


Figure 3-8. Comparing PR curves: the Random Forest classifier is superior to the SGD classifier because its PR curve is much closer to the top-right corner, and it has a greater AUC

As you can see in [Figure 3-8](#), the `RandomForestClassifier`'s PR curve looks much better than the `SGDClassifier`'s: it comes much closer to the top-right corner. Its F_1 score and ROC AUC score are also significantly better:

```

>>> y_train_pred_forest = y_probas_forest[:, 1] >= 0.5 # positive proba ≥ 50%
>>> f1_score(y_train_5, y_pred_forest)
0.9242275142688446
>>> roc_auc_score(y_train_5, y_scores_forest)
0.9983436731328145

```

Try measuring the precision and recall scores: you should find about 99.1% precision and 86.6% recall. Not too bad!

You now know how to train binary classifiers, choose the appropriate metric for your task, evaluate your classifiers using cross-validation, select the precision/recall trade-off that fits your needs, and use several metrics and curves to compare various models. Now let's try to detect more than just the 5s.

Multiclass Classification

Whereas binary classifiers distinguish between two classes, *multiclass classifiers* (also called *multinomial classifiers*) can distinguish between more than two classes.

Some Scikit-Learn classifiers, such as `LogisticRegression`, `RandomForestClassifier`, and `GaussianNB`, are capable of handling multiple classes natively. Others, such as `SGDClassifier` or `SVC`, are strictly binary classifiers. However, there are various strategies that you can use to perform multiclass classification with multiple binary classifiers.

One way to create a system that can classify the digit images into 10 classes (from 0 to 9) is to train 10 binary classifiers, one for each digit (a 0-detector, a 1-detector, a 2-detector, and so on). Then when you want to classify an image, you get the decision score from each classifier for that image and you select the class whose classifier outputs the highest score. This is called the *one-versus-the-rest* (OvR) strategy (also called *one-versus-all*).

Another strategy is to train a binary classifier for every pair of digits: one to distinguish 0s and 1s, another to distinguish 0s and 2s, another for 1s and 2s, and so on. This is called the *one-versus-one* (OvO) strategy. If there are N classes, you need to train $N \times (N - 1) / 2$ classifiers. For the MNIST problem, this means training 45 binary classifiers! When you want to classify an image, you have to run the image through all 45 classifiers and see which class wins the most duels. The main advantage of OvO is that each classifier only needs to be trained on the part of the training set for the two classes that it must distinguish.

Some algorithms (such as Support Vector Machine classifiers) scale poorly with the size of the training set. For these algorithms OvO is preferred because it is faster to train many classifiers on small training sets than to train few classifiers on large training sets. For most binary classification algorithms, however, OvR is preferred.

Scikit-Learn detects when you try to use a binary classification algorithm for a multiclass classification task, and it automatically runs OvR or OvO, depending on the algorithm. Let's try this with a Support Vector Machine classifier using the `sklearn.svm.SVC` class (see [Chapter 5](#)). We'll only train on the first 2,000 images or else it will take a very long time:

```
from sklearn.svm import SVC
```

```
svm_clf = SVC(random_state=42)
svm_clf.fit(X_train[:2000], y_train[:2000]) # y_train, not y_train_5
```

That was easy! We trained the SVC using the original target classes from 0 to 9 (`y_train`), instead of the 5-versus-the-rest target classes (`y_train_5`). Since there are 10 classes, more than 2, Scikit-Learn used the OvO strategy: it trained 45 binary classifiers. Now let's make a prediction on an image:

```
>>> svm_clf.predict([some_digit])
array(['5'], dtype=object)
```

That's correct! This code actually made 45 predictions—one per pair of classes—and it selected the class that won the most duels. If you call the `decision_function()` method, you will see that it returns 10 scores per instance: one per class. Each class gets a score equal to the number of won duels plus or minus a small tweak (max ± 0.33) to break ties, based on the classifier scores:

```
>>> some_digit_scores = svm_clf.decision_function([some_digit])
>>> some_digit_scores.round(2)
array([[ 3.79,  0.73,  6.06,  8.3 , -0.29,  9.3 ,  1.75,  2.77,  7.21,
       4.82]])
```

The highest score is 9.3, and it's indeed the one corresponding to class 5:

```
>>> class_id = some_digit_scores.argmax()
>>> class_id
5
```

When a classifier is trained, it stores the list of target classes in its `classes_` attribute, ordered by value. In the case of MNIST, the index of each class in the `classes_` array conveniently matches the class itself (e.g., the class at index 5 happens to be class '5'), but in general you won't be so lucky: you will need to look up the class label like this:

```
>>> svm_clf.classes_
array(['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'], dtype=object)
>>> svm_clf.classes_[class_id]
'5'
```

If you want to force Scikit-Learn to use one-versus-one or one-versus-the-rest, you can use the `OneVsOneClassifier` or `OneVsRestClassifier` classes. Simply create an instance and pass a classifier to its constructor (it does not even have to be a binary classifier). For example, this code creates a multiclass classifier using the OvR strategy, based on an SVC:

```
from sklearn.multiclass import OneVsRestClassifier

ovr_clf = OneVsRestClassifier(SVC(random_state=42))
ovr_clf.fit(X_train[:2000], y_train[:2000])
```

Let's make a prediction, and check the number of trained classifiers:

```
>>> ovr_clf.predict([some_digit])
array(['5'], dtype='<U1')
>>> len(ovr_clf.estimators_)
10
```

Training an `SGDClassifier` on a multiclass dataset and using it to make predictions is just as easy:

```
>>> sgd_clf = SGDClassifier(random_state=42)
>>> sgd_clf.fit(X_train, y_train)
>>> sgd_clf.predict([some_digit])
array(['3'], dtype='<U1')
```

Oops, that's incorrect. Prediction errors do happen! This time Scikit-Learn used the OvR strategy under the hood: since there are 10 classes, it trained 10 binary classifiers. The `decision_function()` method now returns one value per class. Let's look at the score that the SGD classifier assigned to each class:

```
>>> sgd_clf.decision_function([some_digit]).round()
array([[ -31893., -34420., -9531., 1824., -22320., -1386., -26189.,
       -16148., -4604., -12051.]])
```

You can see that the classifier is not very confident about its prediction: almost all scores are very negative, while class 3 has a score of +1,824, and class 5 is not too far behind at -1,386. Now of course you want to evaluate this classifier on more than one image. Since there are roughly as many images of each class, the accuracy metric is fine. As usual, you can use the `cross_val_score()` function to evaluate the model:

```
>>> cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")
array([0.87365, 0.85835, 0.8689])
```

It gets over 85.8% on all test folds. If you used a random classifier, you would get 10% accuracy, so this is not such a bad score, but you can still do much better. Simply scaling the inputs (as discussed in [Chapter 2](#)) increases accuracy above 89.1%:

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler()
>>> X_train_scaled = scaler.fit_transform(X_train.astype("float64"))
>>> cross_val_score(sgd_clf, X_train_scaled, y_train, cv=3, scoring="accuracy")
array([0.8983, 0.891, 0.9018])
```

Error Analysis

If this were a real project, you would now follow the steps in your Machine Learning project checklist (see [Appendix A](#)). You'd explore data preparation options, try out multiple models, shortlist the best ones, fine-tune their hyperparameters using `GridSearchCV`, and automate as much as possible. Here, we will assume that you have found a promising model and you want to find ways to improve it. One way to do this is to analyze the types of errors it makes.

First, look at the confusion matrix. For this, you first need to make predictions using the `cross_val_predict()` function, then you can pass the labels and predictions to the `confusion_matrix()` function, just like you did earlier. However, since there are now 10 classes instead of 2, the confusion matrix will contain quite a lot of numbers, and it may be hard to read. A colored diagram of the confusion matrix is much easier to analyze. To plot such a diagram, you can use the `ConfusionMatrixDisplay.from_predictions()` function like this:

```
from sklearn.metrics import ConfusionMatrixDisplay

y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train, cv=3)
ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred)
plt.show()
```

This produces the left diagram in [Figure 3-9](#). This confusion matrix looks pretty good: most images are on the main diagonal, which means that they were classified correctly. Notice that the cell on the diagonal on row #5 and column #5 looks slightly darker than the other digits. This could be because the model made more errors on 5s, or because there are simply less 5s in the dataset than the other digits. That's why it's important to normalize the confusion matrix by dividing each value by the total number of images in the corresponding (true) class (i.e., divide by the row's sum). This can be done simply by setting `normalize="true"`. We can also specify `values_format=".0%"` argument to show percentages with no decimals. The following code produces the diagram on the right of [Figure 3-9](#):

```
ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred,
                                       normalize="true", values_format=".0%")
plt.show()
```

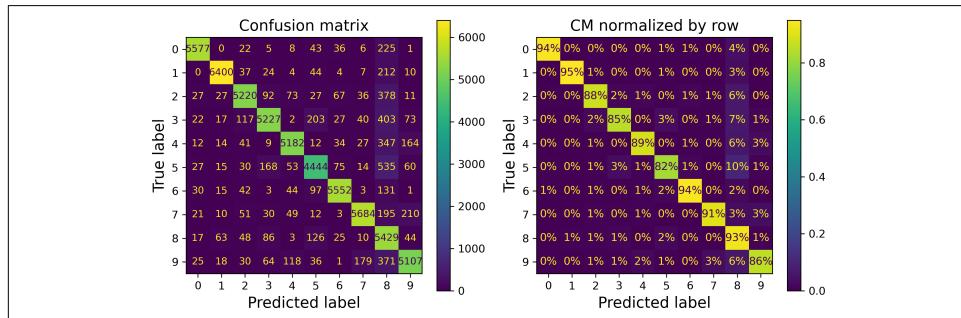


Figure 3-9. Confusion matrix (left) and the same CM normalized by row (right)

Now we can easily see that only 82% of the images of 5s were classified correctly. The most common error the model made with images of 5s was to misclassify them as 8s: this happened for 10% of all 5s. But only 2% of 8s got misclassified as 5s: confusion matrices are generally not symmetrical! If you look carefully, you will notice that many digits have been misclassified as 8s, but this is not immediately obvious on

this diagram. If you want to make the errors stand out much more, you can try putting zero weight on the correct predictions. The following code does just that and produces the diagram on the left of [Figure 3-10](#):

```
sample_weight = (y_train_pred != y_train)
ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred,
                                         sample_weight=sample_weight,
                                         normalize="true", values_format=".0%")
plt.show()
```

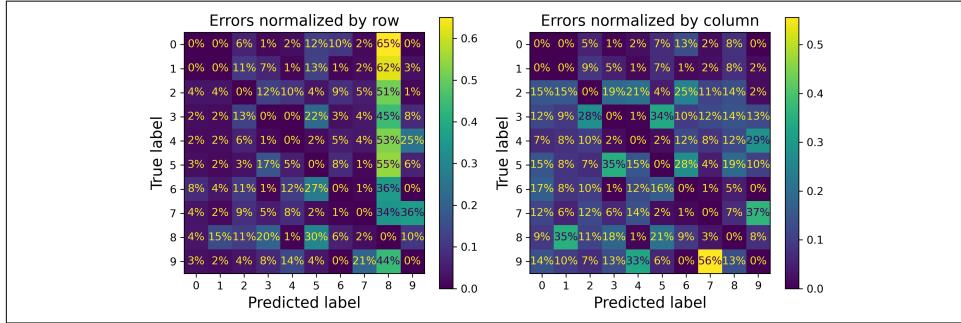


Figure 3-10. Confusion matrix with errors only, normalized by row (left) and by column (right)

Now you can see much more clearly the kinds of errors the classifier makes. The column for class 8 is now really bright, which confirms that many images got misclassified as 8s. In fact this is the most common misclassification for almost all classes. But be careful how you interpret the percentages on this diagram: remember that we've excluded the correct predictions. For example, the 36% in row #7, column #9 does *not* mean that 36% of all images of 7s were misclassified as 9s. It means that 36% of the *errors* the model made on images of 7s were misclassifications as 9s. In reality, only 3% of images of 7s were misclassified as 9s, as you can see in the diagram on the right of [Figure 3-9](#).

It is also possible to normalize the confusion matrix by column rather than by row: if you set `normalize="pred"`, you get the diagram on the right of [Figure 3-10](#). For example, you can see that 56% of misclassified 7s are actually 9s.

Analyzing the confusion matrix often gives you insights into ways to improve your classifier. Looking at these plots, it seems that your efforts should be spent on reducing the false 8s. For example, you could try to gather more training data for digits that look like 8s (but are not) so that the classifier can learn to distinguish them from real 8s. Or you could engineer new features that would help the classifier—for example, writing an algorithm to count the number of closed loops (e.g., 8 has two, 6 has one, 5 has none). Or you could preprocess the images (e.g., using Scikit-Image, Pillow, or OpenCV) to make some patterns, such as closed loops, stand out more.

Analyzing individual errors can also be a good way to gain insights on what your classifier is doing and why it is failing. For example, let's plot examples of 3s and 5s in a confusion matrix style (Figure 3-11):

```
cl_a, cl_b = '3', '5'
X_aa = X_train[(y_train == cl_a) & (y_train_pred == cl_a)]
X_ab = X_train[(y_train == cl_a) & (y_train_pred == cl_b)]
X_ba = X_train[(y_train == cl_b) & (y_train_pred == cl_a)]
X_bb = X_train[(y_train == cl_b) & (y_train_pred == cl_b)]
[...] # plot all images in X_aa, X_ab, X_ba, X_bb in a confusion matrix style
```

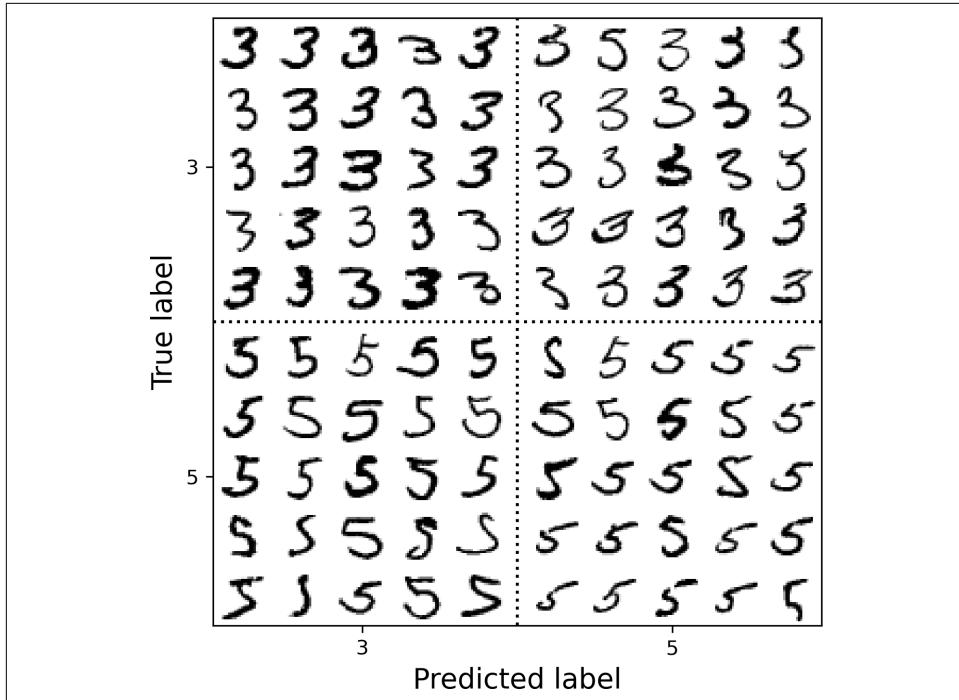


Figure 3-11. Some images of 3s and 5s organized like a confusion matrix

As you can see, some of the digits that the classifier gets wrong (i.e., in the bottom-left and top-right blocks) are so badly written that even a human would have trouble classifying them. However, most misclassified images seem like obvious errors to us, and it's hard to understand why the classifier made the mistakes it did. But remember that our brain is a fantastic pattern recognition system, and our visual system does a lot of complex preprocessing before any information even reaches our consciousness, so the fact that it feels simple does not mean that it is. Remember that we used a simple `SGDClassifier`, which is just a linear model: all it does is assign a weight per class to each pixel, and when it sees a new image it just sums up the weighted pixel

intensities to get a score for each class. Since 3s and 5s differ only by a few pixels, this model will easily confuse them.

The main difference between 3s and 5s is the position of the small line that joins the top line to the bottom arc. If you draw a 3 with the junction slightly shifted to the left, the classifier might classify it as a 5, and vice versa. In other words, this classifier is quite sensitive to image shifting and rotation. So one way to reduce the 3/5 confusion would be to preprocess the images to ensure that they are well centered and not too rotated. However, this may not be easy at all since it requires predicting the correct rotation of each image. A much simpler approach consists in augmenting the training set with slightly shifted and rotated variants of the training images. This will force the model to learn to be more tolerant to such variations. This is called *data augmentation* (we'll cover this in [Chapter 14](#) and also see exercise 2 at the end of this chapter).

Multilabel Classification

Until now each instance has always been assigned to just one class. In some cases you may want your classifier to output multiple classes for each instance. Consider a face-recognition classifier: what should it do if it recognizes several people in the same picture? It should attach one tag per person it recognizes. Say the classifier has been trained to recognize three faces, Alice, Bob, and Charlie. Then when the classifier is shown a picture of Alice and Charlie, it should output [True, False, True] (meaning “Alice yes, Bob no, Charlie yes”). Such a classification system that outputs multiple binary tags is called a *multilabel classification* system.

We won't go into face recognition just yet, but let's look at a simpler example, just for illustration purposes:

```
import numpy as np
from sklearn.neighbors import KNeighborsClassifier

y_train_large = (y_train >= '7')
y_train_odd = (y_train.astype('int8') % 2 == 1)
y_multilabel = np.c_[y_train_large, y_train_odd]

knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train, y_multilabel)
```

This code creates a `y_multilabel` array containing two target labels for each digit image: the first indicates whether or not the digit is large (7, 8, or 9), and the second indicates whether or not it is odd. Then the code creates a `KNeighborsClassifier` instance, which supports multilabel classification (not all classifiers do). Then the code trains this model using the multiple targets array. Now you can make a prediction, and notice that it outputs two labels:

```
>>> knn_clf.predict([some_digit])
array([[False,  True]])
```

And it gets it right! The digit 5 is indeed not large (`False`) and odd (`True`).

There are many ways to evaluate a multilabel classifier, and selecting the right metric really depends on your project. One approach is to measure the F_1 score for each individual label (or any other binary classifier metric discussed earlier), then simply compute the average score. The following code computes the average F_1 score across all labels:

```
>>> y_train_knn_pred = cross_val_predict(knn_clf, X_train, y_multilabel, cv=3)
>>> f1_score(y_multilabel, y_train_knn_pred, average="macro")
0.976410265560605
```

This approach assumes that all labels are equally important, which may not be the case. In particular, if you have many more pictures of Alice than of Bob or Charlie, you may want to give more weight to the classifier's score on pictures of Alice. One simple option is to give each label a weight equal to its *support* (i.e., the number of instances with that target label). To do this, simply set `average="weighted"` when calling the `f1_score()` function.⁵

If you wish to use a classifier that does not natively support multilabel classification, such as `SVC`, one possible strategy is to train one model per label. However, this strategy may have a hard time capturing the dependencies between the labels. For example, a large digit (7, 8, or 9) is twice more likely to be odd than even, but the classifier for the “odd” label does not know what the classifier for the “large” label predicted. To solve this issue, the models can be organized in a chain: when a model makes a prediction, it uses the input features plus all the predictions of the models that come before it in the chain.

Now the good news is that Scikit-Learn has a class called `ChainClassifier` that does just that! By default it will use the true labels for training, feeding each model the appropriate labels depending on their position in the chain. But if you set the `cv` hyperparameter, it will use cross-validation to get “clean” (out-of-sample) predictions from each trained model, for every instance in the training set, and these predictions will then be used to train all the models later in the chain. Here's an example showing how to create and train a `ChainClassifier` using the cross-validation strategy. As earlier, we'll just use the first 2,000 images in the training set to speed things up:

```
from sklearn.multioutput import ClassifierChain

chain_clf = ClassifierChain(SVC(), cv=3, random_state=42)
chain_clf.fit(X_train[:2000], y_multilabel[:2000])
```

⁵ Scikit-Learn offers a few other averaging options and multilabel classifier metrics; see the documentation for more details.

Now we can use this `ChainClassifier` to make predictions:

```
>>> chain_clf.predict([some_digit])
array([[0., 1.]])
```

Multiooutput Classification

The last type of classification task we are going to discuss here is called *multiooutput-multiclass classification* (or just *multiooutput classification*). It is a generalization of multilabel classification where each label can be multiclass (i.e., it can have more than two possible values).

To illustrate this, let's build a system that removes noise from images. It will take as input a noisy digit image, and it will (hopefully) output a clean digit image, represented as an array of pixel intensities, just like the MNIST images. Notice that the classifier's output is multilabel (one label per pixel) and each label can have multiple values (pixel intensity ranges from 0 to 255). It is thus an example of a multiooutput classification system.



The line between classification and regression is sometimes blurry, such as in this example. Arguably, predicting pixel intensity is more akin to regression than to classification. Moreover, multiooutput systems are not limited to classification tasks; you could even have a system that outputs multiple labels per instance, including both class labels and value labels.

Let's start by creating the training and test sets by taking the MNIST images and adding noise to their pixel intensities with NumPy's `randint()` function. The target images will be the original images:

```
np.random.seed(42) # to make this code example reproducible
noise = np.random.randint(0, 100, (len(X_train), 784))
X_train_mod = X_train + noise
noise = np.random.randint(0, 100, (len(X_test), 784))
X_test_mod = X_test + noise
y_train_mod = y_train
y_test_mod = y_test
```

Let's take a peek at the first image from the test set (Figure 3-12). Yes, we're snooping on the test data, so you should be frowning right now.

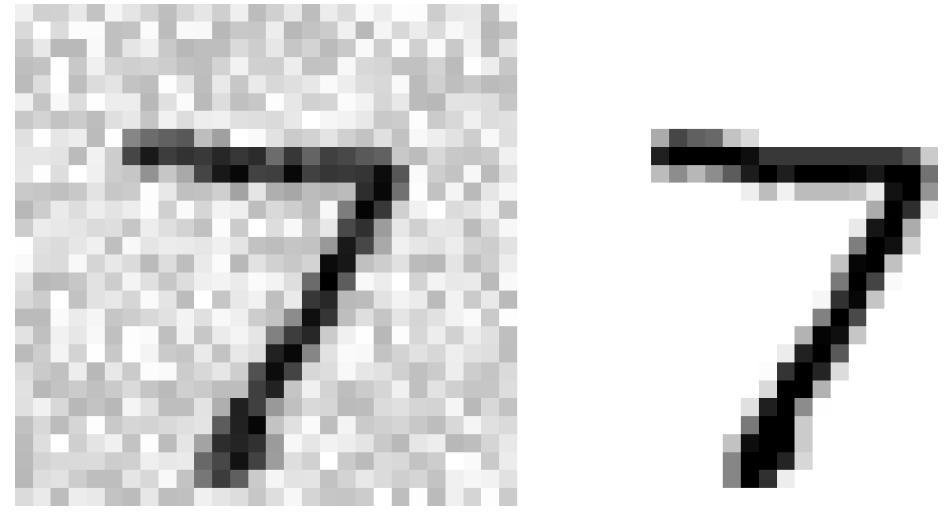


Figure 3-12. A noisy image (left) and the target clean image (right)

On the left is the noisy input image, and on the right is the clean target image. Now let's train the classifier and make it clean up this image (Figure 3-13):

```
knn_clf = KNeighborsClassifier()  
knn_clf.fit(X_train_mod, y_train_mod)  
clean_digit = knn_clf.predict([X_test_mod[0]])  
plot_digit(clean_digit)  
plt.show()
```

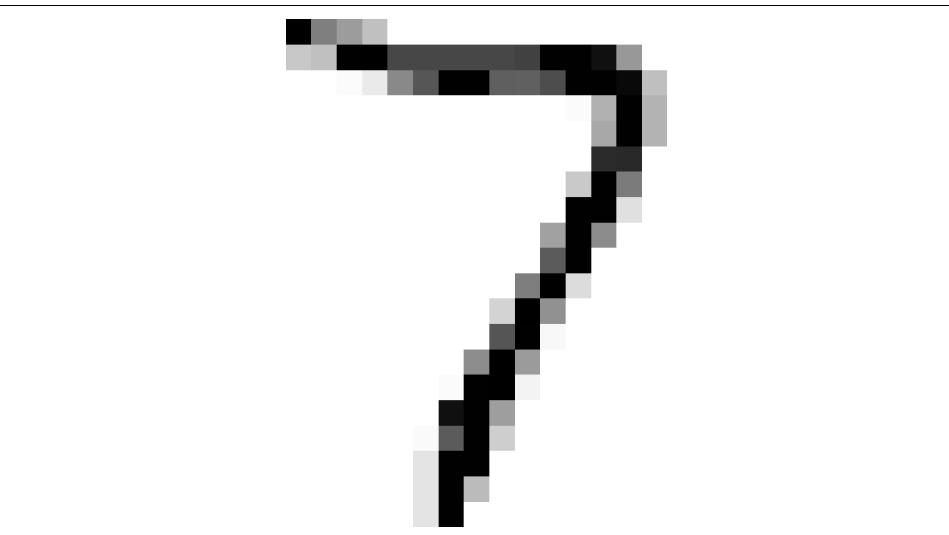


Figure 3-13. The cleaned up image

Looks close enough to the target! Well, this concludes our tour of classification. You now know how to select good metrics for classification tasks, pick the appropriate precision/recall trade-off, compare classifiers, and more generally build good classification systems for a variety of tasks. In the next chapters, we look at how all these Machine Learning models we've been using actually work.

Exercises

1. Try to build a classifier for the MNIST dataset that achieves over 97% accuracy on the test set. Hint: the `KNeighborsClassifier` works quite well for this task; you just need to find good hyperparameter values (try a grid search on the `weights` and `n_neighbors` hyperparameters).
2. Write a function that can shift an MNIST image in any direction (left, right, up, or down) by one pixel.⁶ Then, for each image in the training set, create four shifted copies (one per direction) and add them to the training set. Finally, train your best model on this expanded training set and measure its accuracy on the test set. You should observe that your model performs even better now! This technique of artificially growing the training set is called *data augmentation* or *training set expansion*.
3. Tackle the Titanic dataset. A great place to start is on [Kaggle](#). Alternatively, you can download the data from <https://homl.info/titanic.tgz> and unzip this tarball like you did for the housing data in [Chapter 2](#). This will give you two CSV files: `train.csv` and `test.csv` which you can load using `pandas.read_csv()`. The goal is to train a classifier that can predict the `Survived` column based on the other columns.
4. Build a spam classifier (a more challenging exercise):
 - Download examples of spam and ham from [Apache SpamAssassin's public datasets](#).
 - Unzip the datasets and familiarize yourself with the data format.
 - Split the datasets into a training set and a test set.
 - Write a data preparation pipeline to convert each email into a feature vector. Your preparation pipeline should transform an email into a (sparse) vector that indicates the presence or absence of each possible word. For example, if all emails only ever contain four words, “Hello”, “how”, “are”, “you”, then the email “Hello you Hello Hello you” would be converted into a vector [1, 0, 0, 1] (meaning “[Hello]” is present, “how” is absent, “are” is absent, “you” is

⁶ You can use the `shift()` function from the `scipy.ndimage.interpolation` module. For example, `shift(image, [2, 1], cval=0)` shifts the image two pixels down and one pixel to the right.

present]), or [3, 0, 0, 2] if you prefer to count the number of occurrences of each word.

You may want to add hyperparameters to your preparation pipeline to control whether or not to strip off email headers, convert each email to lowercase, remove punctuation, replace all URLs with “URL”, replace all numbers with “NUMBER”, or even perform *stemming* (i.e., trim off word endings; there are Python libraries available to do this).

Finally, try out several classifiers and see if you can build a great spam classifier, with both high recall and high precision.

Solutions to these exercises are available at the end of this chapter’s notebook, at <https://homl.info/colab3>.

CHAPTER 4

Training Models

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. The GitHub repo is [https://git
hub.com/ageron/handsonml3](https://github.com/ageron/handsonml3).

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

So far we have treated Machine Learning models and their training algorithms mostly like black boxes. If you went through some of the exercises in the previous chapters, you may have been surprised by how much you can get done without knowing anything about what’s under the hood: you optimized a regression system, you improved a digit image classifier, and you even built a spam classifier from scratch, all this without knowing how they actually work. Indeed, in many situations you don’t really need to know the implementation details.

However, having a good understanding of how things work can help you quickly home in on the appropriate model, the right training algorithm to use, and a good set of hyperparameters for your task. Understanding what’s under the hood will also help you debug issues and perform error analysis more efficiently. Lastly, most of the topics discussed in this chapter will be essential in understanding, building, and training neural networks (discussed in [Part II](#) of this book).

In this chapter we will start by looking at the Linear Regression model, one of the simplest models there is. We will discuss two very different ways to train it:

- Using a “closed-form” equation¹ that directly computes the model parameters that best fit the model to the training set (i.e., the model parameters that minimize the cost function over the training set).
- Using an iterative optimization approach called Gradient Descent (GD) that gradually tweaks the model parameters to minimize the cost function over the training set, eventually converging to the same set of parameters as the first method. We will look at a few variants of Gradient Descent that we will use again and again when we study neural networks in [Part II](#): Batch GD, Mini-batch GD, and Stochastic GD.

Next we will look at Polynomial Regression, a more complex model that can fit nonlinear datasets. Since this model has more parameters than Linear Regression, it is more prone to overfitting the training data, so we will look at how to detect whether or not this is the case using learning curves, and then we will look at several regularization techniques that can reduce the risk of overfitting the training set.

Finally, we will look at two more models that are commonly used for classification tasks: Logistic Regression and Softmax Regression.



There will be quite a few math equations in this chapter, using basic notions of linear algebra and calculus. To understand these equations, you will need to know what vectors and matrices are; how to transpose them, multiply them, and inverse them; and what partial derivatives are. If you are unfamiliar with these concepts, please go through the linear algebra and calculus introductory tutorials available as Jupyter notebooks in the [online supplemental material](#). For those who are truly allergic to mathematics, you should still go through this chapter and simply skip the equations; hopefully, the text will be sufficient to help you understand most of the concepts.

Linear Regression

In [Chapter 1](#) we looked at a simple regression model of life satisfaction: $\text{life_satisfaction} = \theta_0 + \theta_1 \times \text{GDP_per_capita}$.

This model is just a linear function of the input feature `GDP_per_capita`. θ_0 and θ_1 are the model’s parameters.

¹ A closed-form equation is only composed of a finite number of constants, variables, and standard operations, for example: $a = \sin(b - c)$. No infinite sums, no limits, no integrals, etc.

More generally, a linear model makes a prediction by simply computing a weighted sum of the input features, plus a constant called the *bias term* (also called the *intercept term*), as shown in [Equation 4-1](#).

Equation 4-1. Linear Regression model prediction

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

In this equation:

- \hat{y} is the predicted value.
- n is the number of features.
- x_i is the i^{th} feature value.
- θ_j is the j^{th} model parameter, including the bias term θ_0 and the feature weights $\theta_1, \theta_2, \dots, \theta_n$.

This can be written much more concisely using a vectorized form, as shown in [Equation 4-2](#).

Equation 4-2. Linear Regression model prediction (vectorized form)

$$\hat{y} = h_{\theta}(\mathbf{x}) = \boldsymbol{\theta} \cdot \mathbf{x}$$

In this equation:

- h_{θ} is the hypothesis function, using the model parameters $\boldsymbol{\theta}$.
- $\boldsymbol{\theta}$ is the model's *parameter vector*, containing the bias term θ_0 and the feature weights θ_1 to θ_n .
- \mathbf{x} is the instance's *feature vector*, containing x_0 to x_n , with x_0 always equal to 1.
- $\boldsymbol{\theta} \cdot \mathbf{x}$ is the dot product of the vectors $\boldsymbol{\theta}$ and \mathbf{x} , which is equal to $\theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$.



In Machine Learning, vectors are often represented as *column vectors*, which are 2D arrays with a single column. If $\boldsymbol{\theta}$ and \mathbf{x} are column vectors, then the prediction is $\hat{y} = \boldsymbol{\theta}^T \mathbf{x}$, where $\boldsymbol{\theta}^T$ is the *transpose* of $\boldsymbol{\theta}$ (a row vector instead of a column vector) and $\boldsymbol{\theta}^T \mathbf{x}$ is the matrix multiplication of $\boldsymbol{\theta}^T$ and \mathbf{x} . It is of course the same prediction, except that it is now represented as a single-cell matrix rather than a scalar value. In this book I will use this notation to avoid switching between dot products and matrix multiplications.

OK, that's the Linear Regression model—but how do we train it? Well, recall that training a model means setting its parameters so that the model best fits the training set. For this purpose, we first need a measure of how well (or poorly) the model fits the training data. In [Chapter 2](#) we saw that the most common performance measure of a regression model is the Root Mean Square Error (RMSE) ([Equation 2-1](#)). Therefore, to train a Linear Regression model, we need to find the value of θ that minimizes the RMSE. In practice, it is simpler to minimize the mean squared error (MSE) than the RMSE, and it leads to the same result (because the value that minimizes a positive function also minimizes its square root).



Learning algorithms will often optimize a different loss function during training than the performance measure used to evaluate the final model. This is generally because the function is easier to optimize and/or because it has extra terms needed during training only (e.g., for regularization). A good performance metric is as close as possible to the final business objective. A good training loss is easy to optimize and strongly correlated with the metric. For example, classifiers are often trained using a cost function such as the log loss (as we will see later in this chapter) but evaluated using precision/recall. The log loss is easy to minimize, and doing so will usually improve precision/recall.

The MSE of a Linear Regression hypothesis h_{θ} on a training set \mathbf{X} is calculated using [Equation 4-3](#).

Equation 4-3. MSE cost function for a Linear Regression model

$$\text{MSE}(\mathbf{X}, h_{\theta}) = \frac{1}{m} \sum_{i=1}^m (\theta^T \mathbf{x}^{(i)} - y^{(i)})^2$$

Most of these notations were presented in [Chapter 2](#) (see “[Notations](#)” on page 44). The only difference is that we write h_{θ} instead of just h to make it clear that the model is parametrized by the vector θ . To simplify notations, we will just write $\text{MSE}(\theta)$ instead of $\text{MSE}(\mathbf{X}, h_{\theta})$.

The Normal Equation

To find the value of θ that minimizes the MSE, there exists a *closed-form solution*—in other words, a mathematical equation that gives the result directly. This is called the *Normal Equation* (Equation 4-4).

Equation 4-4. Normal Equation

$$\hat{\theta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

In this equation:

- $\hat{\theta}$ is the value of θ that minimizes the cost function.
- \mathbf{y} is the vector of target values containing $y^{(1)}$ to $y^{(m)}$.

Let's generate some linear-looking data to test this equation on (Figure 4-1):

```
import numpy as np

np.random.seed(42) # to make this code example reproducible
m = 100 # number of instances
X = 2 * np.random.rand(m, 1) # column vector
y = 4 + 3 * X + np.random.randn(m, 1) # column vector
```

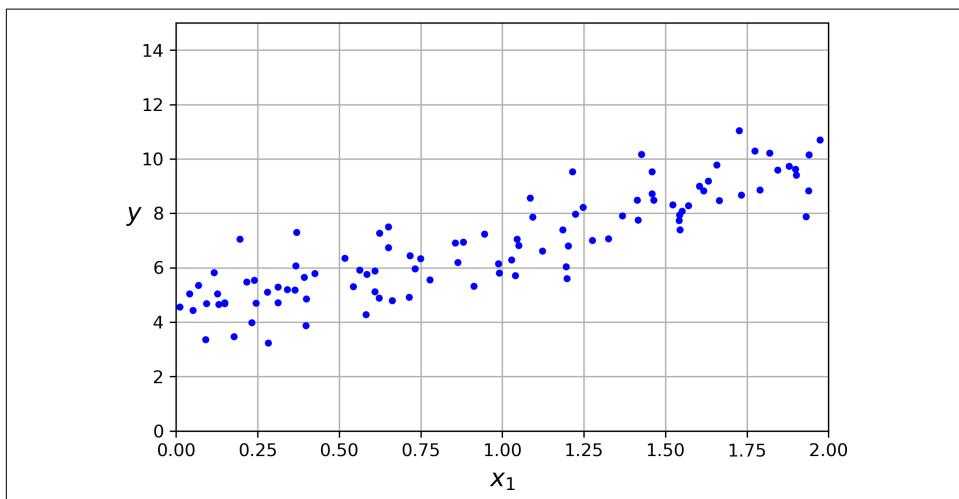


Figure 4-1. Randomly generated linear dataset

Now let's compute $\hat{\theta}$ using the Normal Equation. We will use the `inv()` function from NumPy's linear algebra module (`np.linalg`) to compute the inverse of a matrix, and the `dot()` method for matrix multiplication:

```
from sklearn.preprocessing import add_dummy_feature  
  
X_b = add_dummy_feature(X) # add  $x_0 = 1$  to each instance  
theta_best = np.linalg.inv(X_b.T @ X_b) @ X_b.T @ y
```



The `@` operator performs matrix multiplication. If `A` and `B` are NumPy arrays, then `A @ B` is equivalent to `np.matmul(A, B)`. Many other libraries like TensorFlow, PyTorch, or JAX, support the `@` operator as well. However, you cannot use `@` on pure Python arrays (i.e., lists of lists).

The function that we used to generate the data is $y = 4 + 3x_1 + \text{Gaussian noise}$. Let's see what the equation found:

```
>>> theta_best  
array([[4.21509616],  
       [2.77011339]])
```

We would have hoped for $\theta_0 = 4$ and $\theta_1 = 3$ instead of $\theta_0 = 4.215$ and $\theta_1 = 2.770$. Close enough, but the noise made it impossible to recover the exact parameters of the original function. The smaller and noisier the dataset, the harder it gets.

Now we can make predictions using $\hat{\theta}$:

```
>>> X_new = np.array([[0], [2]])  
>>> X_new_b = add_dummy_feature(X_new) # add  $x_0 = 1$  to each instance  
>>> y_predict = X_new_b @ theta_best  
>>> y_predict  
array([[4.21509616],  
       [9.75532293]])
```

Let's plot this model's predictions (Figure 4-2):

```
import matplotlib.pyplot as plt  
  
plt.plot(X_new, y_predict, "r-", label="Predictions")  
plt.plot(X, y, "b.")  
[...] # beautify the figure: add labels, axis, grid and legend  
plt.show()
```

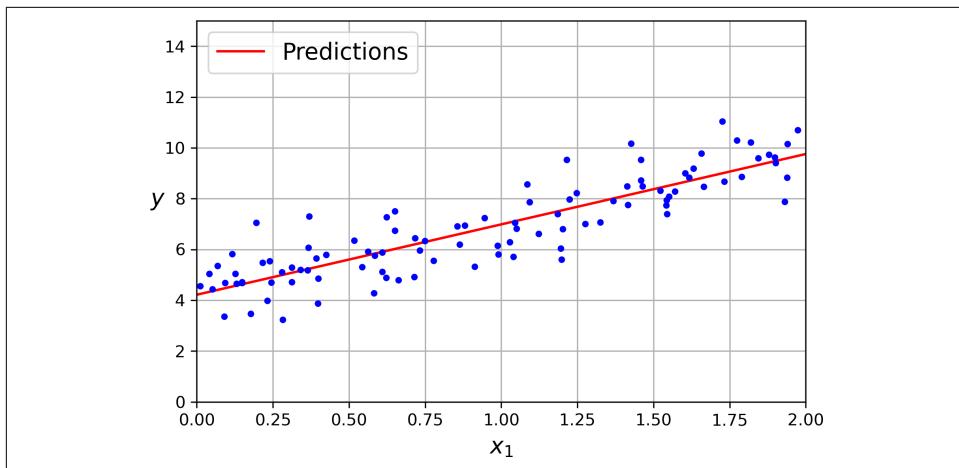


Figure 4-2. Linear Regression model predictions

Performing Linear Regression using Scikit-Learn is relatively straightforward:

```
>>> from sklearn.linear_model import LinearRegression
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([4.21509616]), array([[2.77011339]]))
>>> lin_reg.predict(X_new)
array([[4.21509616],
       [9.75532293]])
```

Notice that Scikit-Learn separates the bias term (`intercept_`) from the feature weights (`coef_`). The `LinearRegression` class is based on the `scipy.linalg.lstsq()` function (the name stands for “least squares”), which you could call directly:

```
>>> theta_best_svd, residuals, rank, s = np.linalg.lstsq(X_b, y, rcond=1e-6)
>>> theta_best_svd
array([[4.21509616],
       [2.77011339]])
```

This function computes $\hat{\theta} = \mathbf{X}^+ \mathbf{y}$, where \mathbf{X}^+ is the *pseudoinverse* of \mathbf{X} (specifically, the Moore-Penrose inverse). You can use `np.linalg.pinv()` to compute the pseudoinverse directly:

```
>>> np.linalg.pinv(X_b) @ y
array([[4.21509616],
       [2.77011339]])
```

The pseudoinverse itself is computed using a standard matrix factorization technique called *Singular Value Decomposition* (SVD) that can decompose the training set matrix \mathbf{X} into the matrix multiplication of three matrices $\mathbf{U} \Sigma \mathbf{V}^\top$ (see `numpy.linalg.svd()`). The pseudoinverse is computed as $\mathbf{X}^+ = \mathbf{V} \Sigma^+ \mathbf{U}^\top$. To compute

the matrix Σ^+ , the algorithm takes Σ and sets to zero all values smaller than a tiny threshold value, then it replaces all the nonzero values with their inverse, and finally it transposes the resulting matrix. This approach is more efficient than computing the Normal Equation, plus it handles edge cases nicely: indeed, the Normal Equation may not work if the matrix $\mathbf{X}^\top \mathbf{X}$ is not invertible (i.e., singular), such as if $m < n$ or if some features are redundant, but the pseudoinverse is always defined.

Computational Complexity

The Normal Equation computes the inverse of $\mathbf{X}^\top \mathbf{X}$, which is an $(n + 1) \times (n + 1)$ matrix (where n is the number of features). The *computational complexity* of inverting such a matrix is typically about $O(n^{2.4})$ to $O(n^3)$, depending on the implementation. In other words, if you double the number of features, you multiply the computation time by roughly $2^{2.4} = 5.3$ to $2^3 = 8$.

The SVD approach used by Scikit-Learn's `LinearRegression` class is about $O(n^2)$. If you double the number of features, you multiply the computation time by roughly 4.



Both the Normal Equation and the SVD approach get very slow when the number of features grows large (e.g., 100,000). On the positive side, both are linear with regard to the number of instances in the training set (they are $O(m)$), so they handle large training sets efficiently, provided they can fit in memory.

Also, once you have trained your Linear Regression model (using the Normal Equation or any other algorithm), predictions are very fast: the computational complexity is linear with regard to both the number of instances you want to make predictions on and the number of features. In other words, making predictions on twice as many instances (or twice as many features) will take roughly twice as much time.

Now we will look at a very different way to train a Linear Regression model, which is better suited for cases where there are a large number of features or too many training instances to fit in memory.

Gradient Descent

Gradient Descent is a generic optimization algorithm capable of finding optimal solutions to a wide range of problems. The general idea of Gradient Descent is to tweak parameters iteratively in order to minimize a cost function.

Suppose you are lost in the mountains in a dense fog, and you can only feel the slope of the ground below your feet. A good strategy to get to the bottom of the valley quickly is to go downhill in the direction of the steepest slope. This is exactly what Gradient Descent does: it measures the local gradient of the error function with

regard to the parameter vector θ , and it goes in the direction of descending gradient. Once the gradient is zero, you have reached a minimum!

In practice, you start by filling θ with random values (this is called *random initialization*). Then you improve it gradually, taking one baby step at a time, each step attempting to decrease the cost function (e.g., the MSE), until the algorithm *converges* to a minimum (see [Figure 4-3](#)).

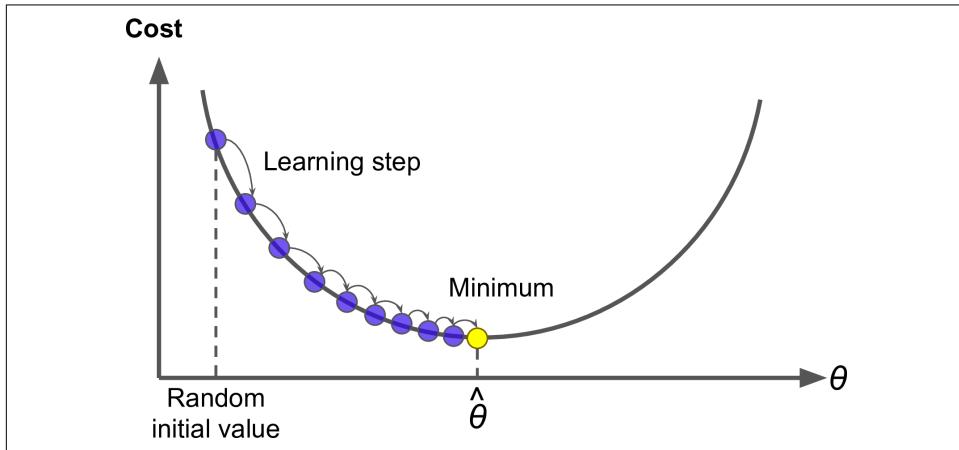


Figure 4-3. In this depiction of Gradient Descent, the model parameters are initialized randomly and get tweaked repeatedly to minimize the cost function; the learning step size is proportional to the slope of the cost function, so the steps gradually get smaller as the cost approaches the minimum

An important parameter in Gradient Descent is the size of the steps, determined by the *learning rate* hyperparameter. If the learning rate is too small, then the algorithm will have to go through many iterations to converge, which will take a long time (see [Figure 4-4](#)).

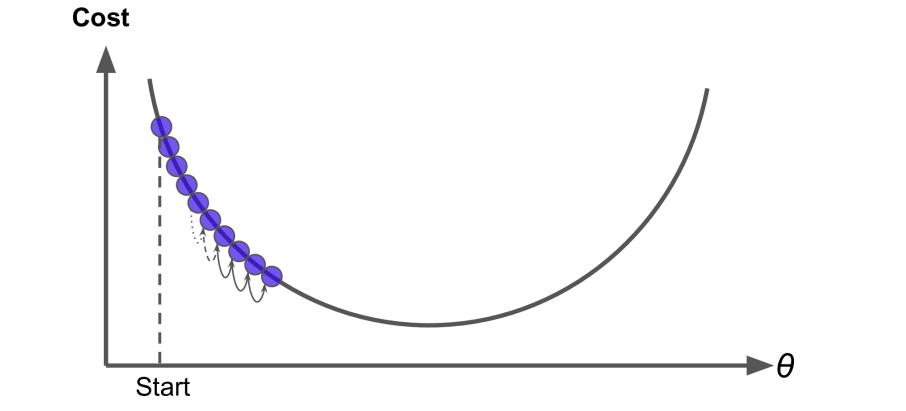


Figure 4-4. The learning rate is too small

On the other hand, if the learning rate is too high, you might jump across the valley and end up on the other side, possibly even higher up than you were before. This might make the algorithm diverge, with larger and larger values, failing to find a good solution (see Figure 4-5).

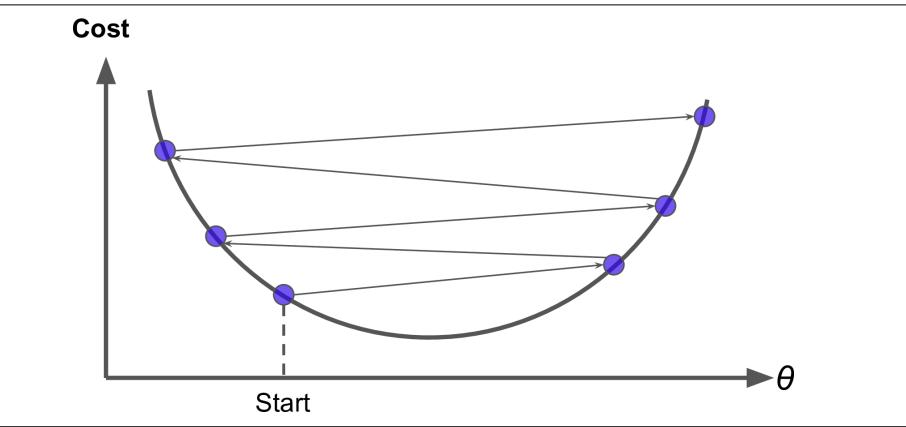


Figure 4-5. The learning rate is too large

Finally, not all cost functions look like nice, regular bowls. There may be holes, ridges, plateaus, and all sorts of irregular terrains, making convergence to the minimum difficult. Figure 4-6 shows the two main challenges with Gradient Descent. If the random initialization starts the algorithm on the left, then it will converge to a *local minimum*, which is not as good as the *global minimum*. If it starts on the right, then it will take a very long time to cross the plateau. And if you stop too early, you will never reach the global minimum.

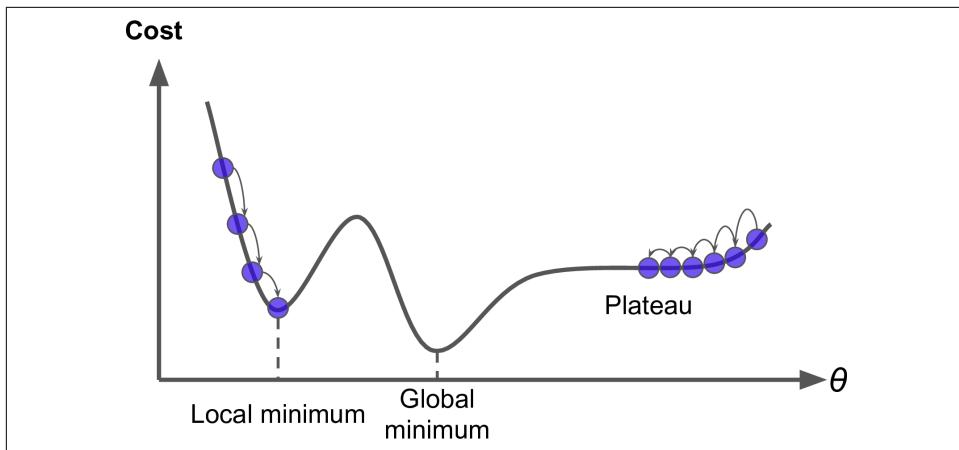


Figure 4-6. Gradient Descent pitfalls

Fortunately, the MSE cost function for a Linear Regression model happens to be a *convex function*, which means that if you pick any two points on the curve, the line segment joining them is never below the curve. This implies that there are no local minima, just one global minimum. It is also a continuous function with a slope that never changes abruptly.² These two facts have a great consequence: Gradient Descent is guaranteed to approach arbitrarily close the global minimum (if you wait long enough and if the learning rate is not too high).

In fact, the cost function has the shape of a bowl, but it can be an elongated bowl if the features have very different scales. Figure 4-7 shows Gradient Descent on a training set where features 1 and 2 have the same scale (on the left), and on a training set where feature 1 has much smaller values than feature 2 (on the right).³

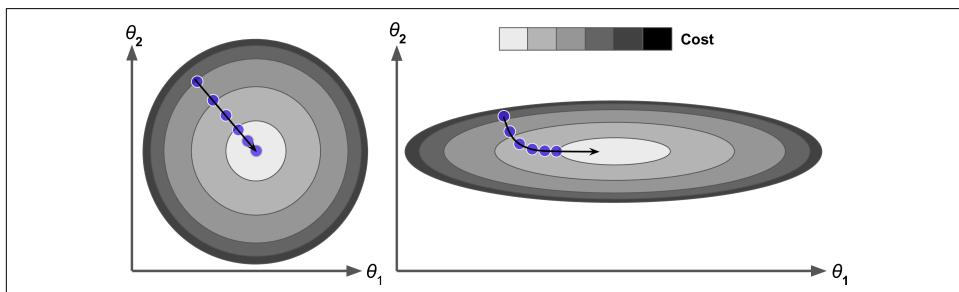


Figure 4-7. Gradient Descent with (left) and without (right) feature scaling

² Technically speaking, its derivative is *Lipschitz continuous*.

³ Since feature 1 is smaller, it takes a larger change in θ_1 to affect the cost function, which is why the bowl is elongated along the θ_1 axis.

As you can see, on the left the Gradient Descent algorithm goes straight toward the minimum, thereby reaching it quickly, whereas on the right it first goes in a direction almost orthogonal to the direction of the global minimum, and it ends with a long march down an almost flat valley. It will eventually reach the minimum, but it will take a long time.



When using Gradient Descent, you should ensure that all features have a similar scale (e.g., using Scikit-Learn's `StandardScaler` class), or else it will take much longer to converge.

This diagram also illustrates the fact that training a model means searching for a combination of model parameters that minimizes a cost function (over the training set). It is a search in the model's *parameter space*: the more parameters a model has, the more dimensions this space has, and the harder the search is: searching for a needle in a 300-dimensional haystack is much trickier than in 3 dimensions. Fortunately, since the cost function is convex in the case of Linear Regression, the needle is simply at the bottom of the bowl.

Batch Gradient Descent

To implement Gradient Descent, you need to compute the gradient of the cost function with regard to each model parameter θ_j . In other words, you need to calculate how much the cost function will change if you change θ_j just a little bit. This is called a *partial derivative*. It is like asking “What is the slope of the mountain under my feet if I face east?” and then asking the same question facing north (and so on for all other dimensions, if you can imagine a universe with more than three dimensions). **Equation 4-5** computes the partial derivative of the MSE with regard to parameter θ_j , noted $\frac{\partial \text{MSE}(\boldsymbol{\theta})}{\partial \theta_j}$.

Equation 4-5. Partial derivatives of the cost function

$$\frac{\partial}{\partial \theta_j} \text{MSE}(\boldsymbol{\theta}) = \frac{2}{m} \sum_{i=1}^m (\boldsymbol{\theta}^\top \mathbf{x}^{(i)} - y^{(i)}) x_j^{(i)}$$

Instead of computing these partial derivatives individually, you can use [Equation 4-6](#) to compute them all in one go. The gradient vector, noted $\nabla_{\theta}\text{MSE}(\theta)$, contains all the partial derivatives of the cost function (one for each model parameter).

Equation 4-6. Gradient vector of the cost function

$$\nabla_{\theta}\text{MSE}(\theta) = \begin{pmatrix} \frac{\partial}{\partial\theta_0}\text{MSE}(\theta) \\ \frac{\partial}{\partial\theta_1}\text{MSE}(\theta) \\ \vdots \\ \frac{\partial}{\partial\theta_n}\text{MSE}(\theta) \end{pmatrix} = \frac{2}{m}\mathbf{X}^T(\mathbf{X}\theta - \mathbf{y})$$



Notice that this formula involves calculations over the full training set \mathbf{X} , at each Gradient Descent step! This is why the algorithm is called *Batch Gradient Descent*: it uses the whole batch of training data at every step (actually, *Full Gradient Descent* would probably be a better name). As a result it is terribly slow on very large training sets (but we will see much faster Gradient Descent algorithms shortly). However, Gradient Descent scales well with the number of features; training a Linear Regression model when there are hundreds of thousands of features is much faster using Gradient Descent than using the Normal Equation or SVD decomposition.

Once you have the gradient vector, which points uphill, just go in the opposite direction to go downhill. This means subtracting $\nabla_{\theta}\text{MSE}(\theta)$ from θ . This is where the learning rate η comes into play:⁴ multiply the gradient vector by η to determine the size of the downhill step ([Equation 4-7](#)).

Equation 4-7. Gradient Descent step

$$\theta^{(\text{next step})} = \theta - \eta \nabla_{\theta}\text{MSE}(\theta)$$

Let's look at a quick implementation of this algorithm:

```
eta = 0.1 # learning rate
n_epochs = 1000
m = len(X_b) # number of instances

np.random.seed(42)
theta = np.random.randn(2, 1) # randomly initialized model parameters
```

⁴ Eta (η) is the seventh letter of the Greek alphabet.

```

for epoch in range(n_epochs):
    gradients = 2 / m * X_b.T @ (X_b @ theta - y)
    theta = theta - eta * gradients

```

That wasn't too hard! Each iteration over the training set is called an *epoch*. Let's look at the resulting `theta`:

```

>>> theta
array([[4.21509616],
       [2.77011339]])

```

Hey, that's exactly what the Normal Equation found! Gradient Descent worked perfectly. But what if you had used a different learning rate `eta`? Figure 4-8 shows the first 20 steps of Gradient Descent using three different learning rates. The line at the bottom of each plot represents the random starting point, then each epoch is represented by a darker and darker line.

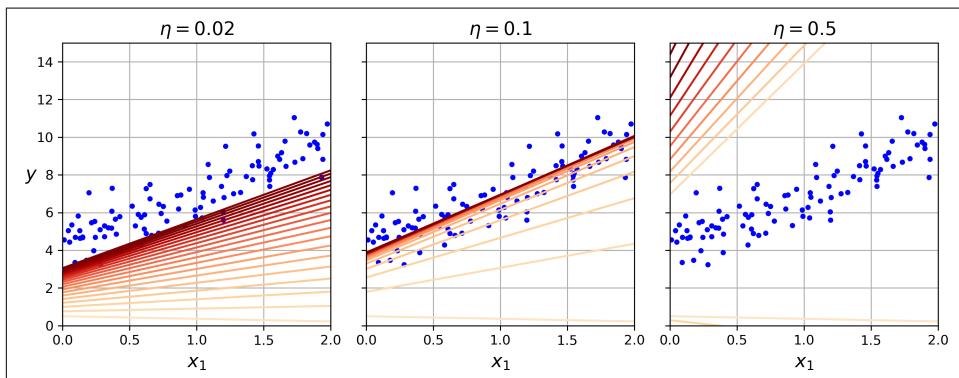


Figure 4-8. Gradient Descent with various learning rates

On the left, the learning rate is too low: the algorithm will eventually reach the solution, but it will take a long time. In the middle, the learning rate looks pretty good: in just a few epochs, it has already converged to the solution. On the right, the learning rate is too high: the algorithm diverges, jumping all over the place and actually getting further and further away from the solution at every step.

To find a good learning rate, you can use grid search (see Chapter 2). However, you may want to limit the number of epochs so that grid search can eliminate models that take too long to converge.

You may wonder how to set the number of epochs. If it is too low, you will still be far away from the optimal solution when the algorithm stops; but if it is too high, you will waste time while the model parameters do not change anymore. A simple solution is to set a very large number of epochs but to interrupt the algorithm when the gradient vector becomes tiny—that is, when its norm becomes smaller than a tiny

number ϵ (called the *tolerance*)—because this happens when Gradient Descent has (almost) reached the minimum.

Convergence Rate

When the cost function is convex and its slope does not change abruptly (as is the case for the MSE cost function), Batch Gradient Descent with a fixed learning rate will eventually converge to the optimal solution, but you may have to wait a while: it can take $O(1/\epsilon)$ iterations to reach the optimum within a range of ϵ , depending on the shape of the cost function. If you divide the tolerance by 10 to have a more precise solution, then the algorithm may have to run about 10 times longer.

Stochastic Gradient Descent

The main problem with Batch Gradient Descent is the fact that it uses the whole training set to compute the gradients at every step, which makes it very slow when the training set is large. At the opposite extreme, *Stochastic Gradient Descent* picks a random instance in the training set at every step and computes the gradients based only on that single instance. Obviously, working on a single instance at a time makes the algorithm much faster because it has very little data to manipulate at every iteration. It also makes it possible to train on huge training sets, since only one instance needs to be in memory at each iteration (Stochastic GD can be implemented as an out-of-core algorithm; see [Chapter 1](#)).

On the other hand, due to its stochastic (i.e., random) nature, this algorithm is much less regular than Batch Gradient Descent: instead of gently decreasing until it reaches the minimum, the cost function will bounce up and down, decreasing only on average. Over time it will end up very close to the minimum, but once it gets there it will continue to bounce around, never settling down (see [Figure 4-9](#)). So once the algorithm stops, the final parameter values are good, but not optimal.

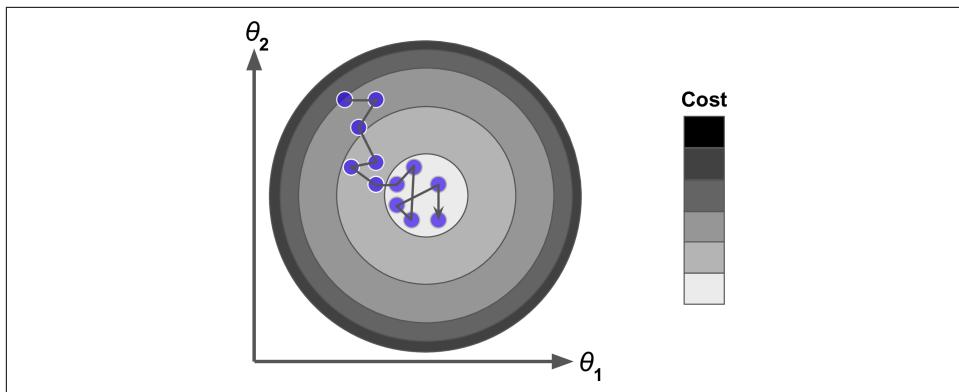


Figure 4-9. With Stochastic Gradient Descent, each training step is much faster but also much more stochastic than when using Batch Gradient Descent

When the cost function is very irregular (as in Figure 4-6), this can actually help the algorithm jump out of local minima, so Stochastic Gradient Descent has a better chance of finding the global minimum than Batch Gradient Descent does.

Therefore, randomness is good to escape from local optima, but bad because it means that the algorithm can never settle at the minimum. One solution to this dilemma is to gradually reduce the learning rate. The steps start out large (which helps make quick progress and escape local minima), then get smaller and smaller, allowing the algorithm to settle at the global minimum. This process is akin to *simulated annealing*, an algorithm inspired from the process in metallurgy of annealing, where molten metal is slowly cooled down. The function that determines the learning rate at each iteration is called the *learning schedule*. If the learning rate is reduced too quickly, you may get stuck in a local minimum, or even end up frozen halfway to the minimum. If the learning rate is reduced too slowly, you may jump around the minimum for a long time and end up with a suboptimal solution if you halt training too early.

This code implements Stochastic Gradient Descent using a simple learning schedule:

```

n_epochs = 50
t0, t1 = 5, 50 # learning schedule hyperparameters

def learning_schedule(t):
    return t0 / (t + t1)

np.random.seed(42)
theta = np.random.randn(2, 1) # random initialization

for epoch in range(n_epochs):
    for iteration in range(m):
        random_index = np.random.randint(m)

```

```

xi = X_b[random_index : random_index + 1]
yi = y[random_index : random_index + 1]
gradients = 2 * xi.T @ (xi @ theta - yi) # for SGD, do not divide by m
eta = learning_schedule(epoch * m + iteration)
theta = theta - eta * gradients

```

By convention we iterate by rounds of m iterations; each round is called an *epoch*, as earlier. While the Batch Gradient Descent code iterated 1,000 times through the whole training set, this code goes through the training set only 50 times and reaches a pretty good solution:

```

>>> theta
array([[4.21076011],
       [2.74856079]])

```

Figure 4-10 shows the first 20 steps of training (notice how irregular the steps are).

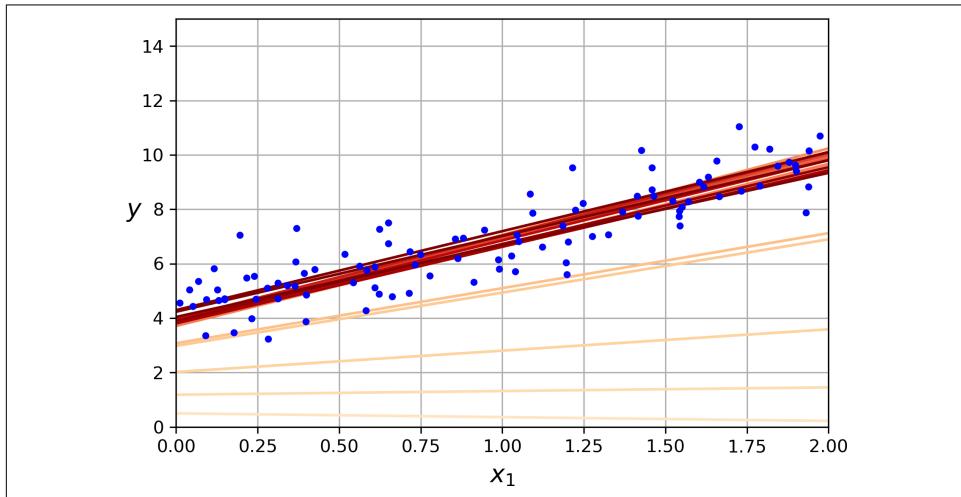


Figure 4-10. The first 20 steps of Stochastic Gradient Descent

Note that since instances are picked randomly, some instances may be picked several times per epoch, while others may not be picked at all. If you want to be sure that the algorithm goes through every instance at each epoch, another approach is to shuffle the training set (making sure to shuffle the input features and the labels jointly), then go through it instance by instance, then shuffle it again, and so on. However, this approach is more complex and it generally does not improve the result.



When using Stochastic Gradient Descent, the training instances must be independent and identically distributed (IID) to ensure that the parameters get pulled toward the global optimum, on average. A simple way to ensure this is to shuffle the instances during training (e.g., pick each instance randomly, or shuffle the training set at the beginning of each epoch). If you do not shuffle the instances—for example, if the instances are sorted by label—then SGD will start by optimizing for one label, then the next, and so on, and it will not settle close to the global minimum.

To perform Linear Regression using Stochastic GD with Scikit-Learn, you can use the `SGDRegressor` class, which defaults to optimizing the MSE cost function. The following code runs for maximum 1,000 epochs (`max_iter`) or until the loss drops by less than 10^{-5} (`tol`) during 100 epochs (`n_iter_no_change`). It starts with a learning rate of 0.01 (`eta0`), using the default learning schedule (different from the one we used). Lastly, it does not use any regularization (`penalty=None`; more details on this shortly):

```
from sklearn.linear_model import SGDRegressor

sgd_reg = SGDRegressor(max_iter=1000, tol=1e-5, penalty=None, eta0=0.01,
                      n_iter_no_change=100, random_state=42)
sgd_reg.fit(X, y.ravel()) # y.ravel() because fit() expects 1D targets
```

Once again, you find a solution quite close to the one returned by the Normal Equation:

```
>>> sgd_reg.intercept_, sgd_reg.coef_
(array([4.21278812]), array([2.77270267]))
```



All Scikit-Learn estimators can be trained using the `fit()` method, but some estimators also have a `partial_fit()` method that you can call to run a single round of training on one or more instances (it ignores hyperparameters like `max_iter` or `tol`). Repeatedly calling `partial_fit()` will gradually train the model. This is useful when you need more control over the training process. Other models have a `warm_start` hyperparameter instead (and some have both): if you set `warm_start=True`, calling the `fit()` method on a trained model will not reset the model, it will just continue training where it left off, respecting hyperparameters like `max_iter` and `tol`. Note that `fit()` resets the iteration counter used by the learning schedule, while `partial_fit()` does not.

Mini-batch Gradient Descent

The last Gradient Descent algorithm we will look at is called *Mini-batch Gradient Descent*. It is straightforward once you know Batch and Stochastic Gradient Descent: at each step, instead of computing the gradients based on the full training set (as in Batch GD) or based on just one instance (as in Stochastic GD), Mini-batch GD computes the gradients on small random sets of instances called *mini-batches*. The main advantage of Mini-batch GD over Stochastic GD is that you can get a performance boost from hardware optimization of matrix operations, especially when using *Graphical Processing Units* (GPUs).

The algorithm's progress in parameter space is less erratic than with Stochastic GD, especially with fairly large mini-batches. As a result, Mini-batch GD will end up walking around a bit closer to the minimum than Stochastic GD—but it may be harder for it to escape from local minima (in the case of problems that suffer from local minima, unlike Linear Regression with the MSE cost function). [Figure 4-11](#) shows the paths taken by the three Gradient Descent algorithms in parameter space during training. They all end up near the minimum, but Batch GD's path actually stops at the minimum, while both Stochastic GD and Mini-batch GD continue to walk around. However, don't forget that Batch GD takes a lot of time to take each step, and Stochastic GD and Mini-batch GD would also reach the minimum if you used a good learning schedule.

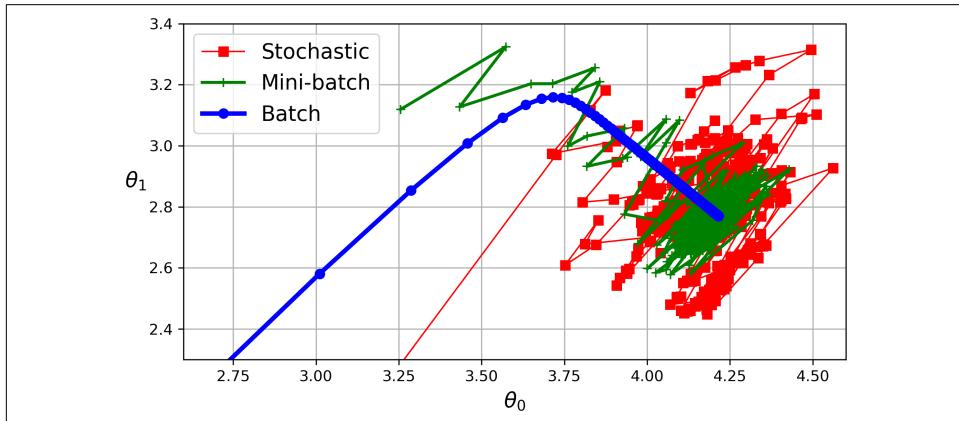


Figure 4-11. Gradient Descent paths in parameter space

Let's compare the algorithms we've discussed so far for Linear Regression⁵ (recall that m is the number of training instances and n is the number of features); see [Table 4-1](#).

⁵ While the Normal Equation can only perform Linear Regression, the Gradient Descent algorithms can be used to train many other models, as we will see.

Table 4-1. Comparison of algorithms for Linear Regression

Algorithm	Large m	Out-of-core support	Large n	Hyperparams	Scaling required	Scikit-Learn
Normal Equation	Fast	No	Slow	0	No	N/A
SVD	Fast	No	Slow	0	No	LinearRegression
Batch GD	Slow	No	Fast	2	Yes	N/A
Stochastic GD	Fast	Yes	Fast	≥ 2	Yes	SGDRegressor
Mini-batch GD	Fast	Yes	Fast	≥ 2	Yes	N/A

There is almost no difference after training: all these algorithms end up with very similar models and make predictions in exactly the same way.

Polynomial Regression

What if your data is more complex than a straight line? Surprisingly, you can use a linear model to fit nonlinear data. A simple way to do this is to add powers of each feature as new features, then train a linear model on this extended set of features. This technique is called *Polynomial Regression*.

Let's look at an example. First, let's generate some nonlinear data (see Figure 4-12), based on a simple **quadratic equation**—that's an equation of the form $y = ax^2 + bx + c$ —plus some noise:

```
np.random.seed(42)
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X ** 2 + X + 2 + np.random.randn(m, 1)
```

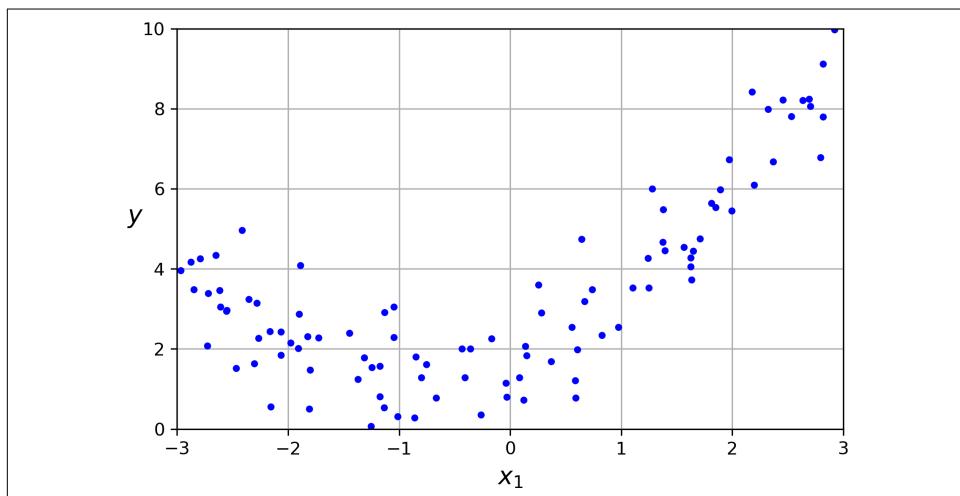


Figure 4-12. Generated nonlinear and noisy dataset

Clearly, a straight line will never fit this data properly. So let's use Scikit-Learn's `PolynomialFeatures` class to transform our training data, adding the square (second-degree polynomial) of each feature in the training set as a new feature (in this case there is just one feature):

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly_features = PolynomialFeatures(degree=2, include_bias=False)
>>> X_poly = poly_features.fit_transform(X)
>>> X[0]
array([-0.75275929])
>>> X_poly[0]
array([-0.75275929,  0.56664654])
```

`X_poly` now contains the original feature of `X` plus the square of this feature. Now you can fit a `LinearRegression` model to this extended training data ([Figure 4-13](#)):

```
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X_poly, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([1.78134581]), array([[0.93366893, 0.56456263]]))
```

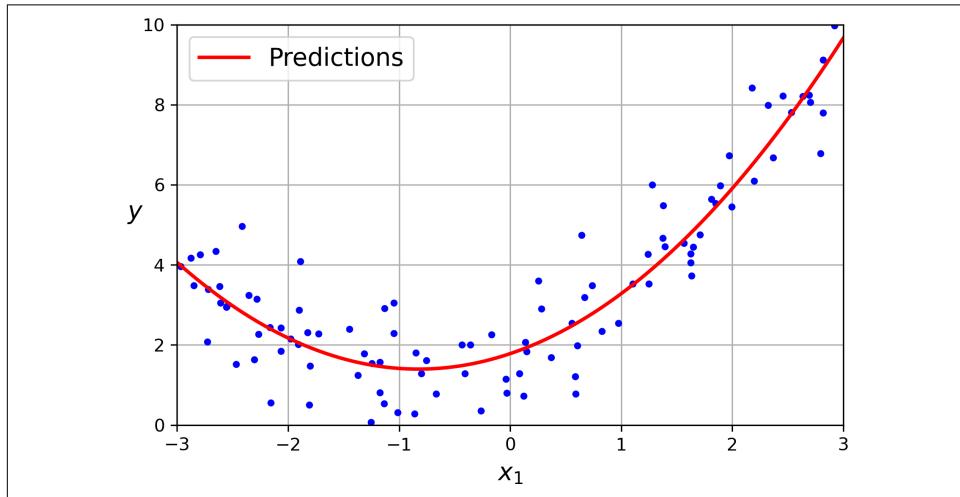


Figure 4-13. Polynomial Regression model predictions

Not bad: the model estimates $\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78$ when in fact the original function was $y = 0.5x_1^2 + 1.0x_1 + 2.0 + \text{Gaussian noise}$.

Note that when there are multiple features, Polynomial Regression is capable of finding relationships between features, which is something a plain Linear Regression model cannot do. This is made possible by the fact that `PolynomialFeatures` also adds all combinations of features up to the given degree. For example, if there were

two features a and b , `PolynomialFeatures` with `degree=3` would not only add the features a^2 , a^3 , b^2 , and b^3 , but also the combinations ab , a^2b , and ab^2 .



`PolynomialFeatures(degree=d)` transforms an array containing n features into an array containing $(n + d)! / d!n!$ features, where $n!$ is the factorial of n , equal to $1 \times 2 \times 3 \times \dots \times n$. Beware of the combinatorial explosion of the number of features!

Learning Curves

If you perform high-degree Polynomial Regression, you will likely fit the training data much better than with plain Linear Regression. For example, Figure 4-14 applies a 300-degree polynomial model to the preceding training data, and compares the result with a pure linear model and a quadratic model (second-degree polynomial). Notice how the 300-degree polynomial model wiggles around to get as close as possible to the training instances.

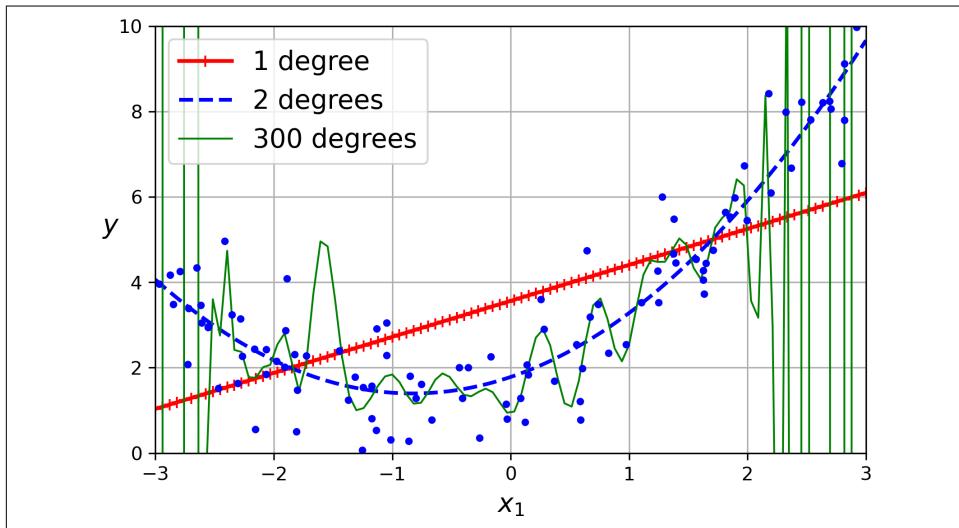


Figure 4-14. High-degree Polynomial Regression

This high-degree Polynomial Regression model is severely overfitting the training data, while the linear model is underfitting it. The model that will generalize best in this case is the quadratic model, which makes sense because the data was generated using a quadratic model. But in general you won't know what function generated the data, so how can you decide how complex your model should be? How can you tell that your model is overfitting or underfitting the data?

In Chapter 2 you used cross-validation to get an estimate of a model's generalization performance. If a model performs well on the training data but generalizes poorly according to the cross-validation metrics, then your model is overfitting. If it performs poorly on both, then it is underfitting. This is one way to tell when a model is too simple or too complex.

Another way to tell is to look at the *learning curves*: these are plots of the model's training error and validation error as a function of the training iteration: just evaluate the model at regular intervals during training on both the training set and the validation set, and plot the results. If the model cannot be trained incrementally (i.e., if it does not support `partial_fit()` or `warm_start`), then you must train the model several times on gradually larger subsets of the training set.

Scikit-Learn has a useful `learning_curve()` function to help with this: it trains and evaluates the model using cross-validation. By default it retrains the model on growing subsets of the training set, but if the model supports incremental learning you can set `exploit_incremental_learning=True` when calling `learning_curve()` and it will train the model incrementally instead. The function returns the training set sizes at which it evaluated the model, and the training and validation scores it measured for each size and for each cross-validation fold. Let's use this function to look at the learning curves of the plain Linear Regression model (see Figure 4-15):

```
from sklearn.model_selection import learning_curve

train_sizes, train_scores, valid_scores = learning_curve(
    LinearRegression(), X, y, train_sizes=np.linspace(0.01, 1.0, 40), cv=5,
    scoring="neg_mean_squared_error")
train_errors = -train_scores.mean(axis=1)
valid_errors = -valid_scores.mean(axis=1)

plt.plot(train_sizes, train_errors, "r--", linewidth=2, label="train")
plt.plot(train_sizes, valid_errors, "b-", linewidth=3, label="valid")
[...] # beautify the figure: add labels, axis, grid and legend.
plt.show()
```

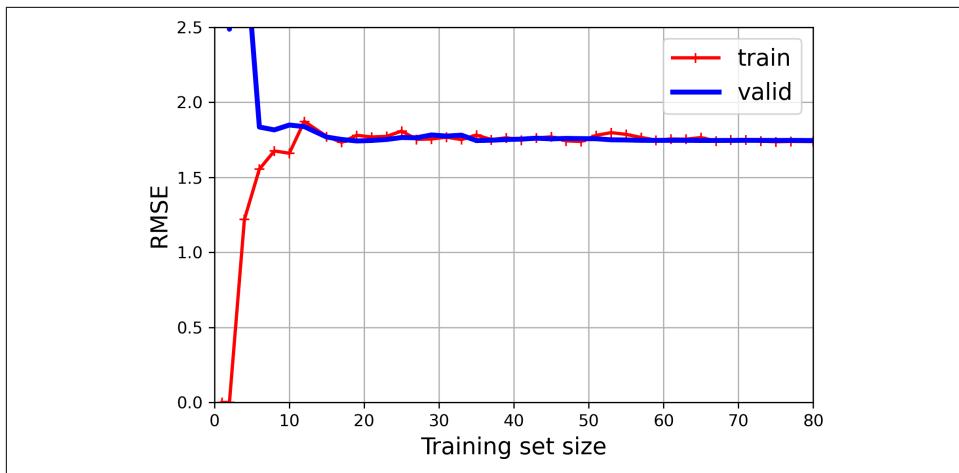


Figure 4-15. Learning curves

This model is underfitting, let's see why. First, let's look at the training error: when there are just one or two instances in the training set, the model can fit them perfectly, which is why the curve starts at zero. But as new instances are added to the training set, it becomes impossible for the model to fit the training data perfectly, both because the data is noisy and because it is not linear at all. So the error on the training data goes up until it reaches a plateau, at which point adding new instances to the training set doesn't make the average error much better or worse. Now let's look at the validation error. When the model is trained on very few training instances, it is incapable of generalizing properly, which is why the validation error is initially quite large. Then, as the model is shown more training examples, it learns, and thus the validation error slowly goes down. However, once again a straight line cannot do a good job modeling the data, so the error ends up at a plateau, very close to the other curve.

These learning curves are typical of a model that's underfitting. Both curves have reached a plateau; they are close and fairly high.



If your model is underfitting the training data, adding more training examples will not help. You need to use a better model or come up with better features.

Now let's look at the learning curves of a 10th-degree polynomial model on the same data ([Figure 4-16](#)):

```
from sklearn.pipeline import make_pipeline
```

```

polynomial_regression = make_pipeline(
    PolynomialFeatures(degree=10, include_bias=False),
    LinearRegression())

train_sizes, train_scores, valid_scores = learning_curve(
    polynomial_regression, X, y, train_sizes=np.linspace(0.01, 1.0, 40), cv=5,
    scoring="neg_root_mean_squared_error")
[...] # same as earlier

```

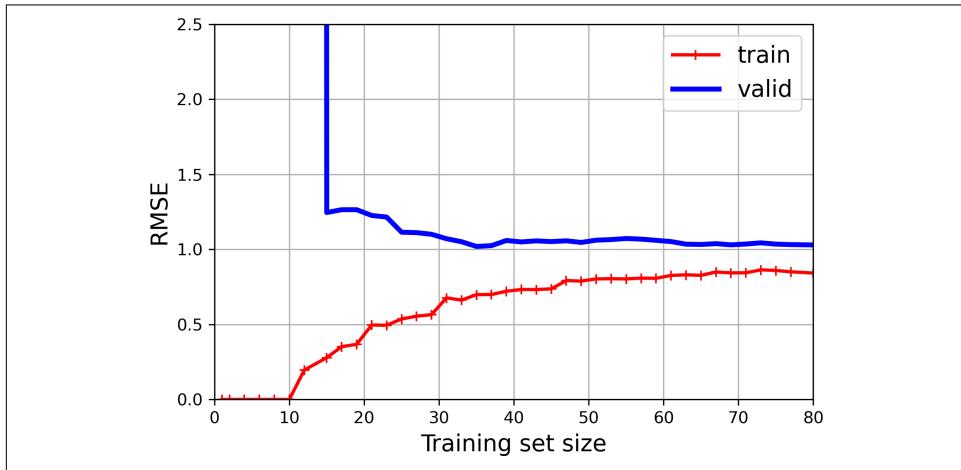


Figure 4-16. Learning curves for the 10th-degree polynomial model

These learning curves look a bit like the previous ones, but there are two very important differences:

- The error on the training data is much lower than before.
- There is a gap between the curves. This means that the model performs significantly better on the training data than on the validation data, which is the hallmark of an overfitting model. If you used a much larger training set, however, the two curves would continue to get closer.



One way to improve an overfitting model is to feed it more training data until the validation error reaches the training error.

The Bias/Variance Trade-off

An important theoretical result of statistics and Machine Learning is the fact that a model's generalization error can be expressed as the sum of three very different errors:

Bias

This part of the generalization error is due to wrong assumptions, such as assuming that the data is linear when it is actually quadratic. A high-bias model is most likely to underfit the training data.⁶

Variance

This part is due to the model's excessive sensitivity to small variations in the training data. A model with many degrees of freedom (such as a high-degree polynomial model) is likely to have high variance and thus overfit the training data.

Irreducible error

This part is due to the noisiness of the data itself. The only way to reduce this part of the error is to clean up the data (e.g., fix the data sources, such as broken sensors, or detect and remove outliers).

Increasing a model's complexity will typically increase its variance and reduce its bias. Conversely, reducing a model's complexity increases its bias and reduces its variance. This is why it is called a trade-off.

Regularized Linear Models

As we saw in Chapters 1 and 2, a good way to reduce overfitting is to regularize the model (i.e., to constrain it): the fewer degrees of freedom it has, the harder it will be for it to overfit the data. A simple way to regularize a polynomial model is to reduce the number of polynomial degrees.

For a linear model, regularization is typically achieved by constraining the weights of the model. We will now look at Ridge Regression, Lasso Regression, and Elastic Net, which implement three different ways to constrain the weights.

Ridge Regression

Ridge Regression (also called *Tikhonov regularization*) is a regularized version of Linear Regression: a *regularization term* equal to $\frac{\alpha}{m} \sum_{i=1}^n \theta_i^2$ is added to the MSE. This forces the learning algorithm to not only fit the data but also keep the model

⁶ This notion of bias is not to be confused with the bias term of linear models.

weights as small as possible. Note that the regularization term should only be added to the cost function during training. Once the model is trained, you want to use the unregularized MSE (or the RMSE) to evaluate the model's performance.

The hyperparameter α controls how much you want to regularize the model. If $\alpha = 0$, then Ridge Regression is just Linear Regression. If α is very large, then all weights end up very close to zero and the result is a flat line going through the data's mean. [Equation 4-8](#) presents the Ridge Regression cost function.⁷

Equation 4-8. Ridge Regression cost function

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + \frac{\alpha}{m} \sum_{i=1}^n \theta_i^2$$

Note that the bias term θ_0 is not regularized (the sum starts at $i = 1$, not 0). If we define \mathbf{w} as the vector of feature weights (θ_1 to θ_n), then the regularization term is equal to $\alpha(\|\mathbf{w}\|_2)^2 / m$, where $\|\mathbf{w}\|_2$ represents the ℓ_2 norm of the weight vector.⁸ For Batch Gradient Descent, just add $2\alpha\mathbf{w} / m$ to the part of the MSE gradient vector that corresponds to the feature weights, without adding anything to the gradient of the bias term (see [Equation 4-6](#)).



It is important to scale the data (e.g., using a `StandardScaler`) before performing Ridge Regression, as it is sensitive to the scale of the input features. This is true of most regularized models.

[Figure 4-17](#) shows several Ridge models trained on some very noisy linear data using different α values. On the left, plain Ridge models are used, leading to linear predictions. On the right, the data is first expanded using `PolynomialFeatures(degree=10)`, then it is scaled using a `StandardScaler`, and finally the Ridge models are applied to the resulting features: this is Polynomial Regression with Ridge regularization. Note how increasing α leads to flatter (i.e., less extreme, more reasonable) predictions, thus reducing the model's variance but increasing its bias.

⁷ It is common to use the notation $J(\boldsymbol{\theta})$ for cost functions that don't have a short name; we will often use this notation throughout the rest of this book. The context will make it clear which cost function is being discussed.

⁸ Norms are discussed in [Chapter 2](#).

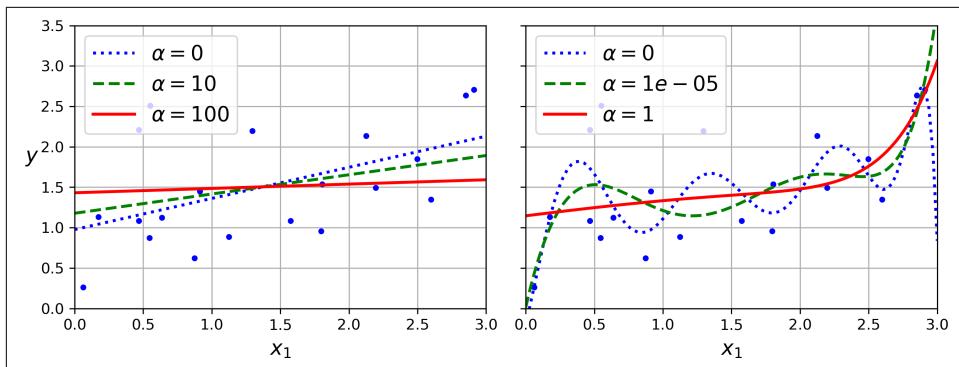


Figure 4-17. A linear model (left) and a polynomial model (right), both with various levels of Ridge regularization

As with Linear Regression, we can perform Ridge Regression either by computing a closed-form equation or by performing Gradient Descent. The pros and cons are the same. [Equation 4-9](#) shows the closed-form solution, where \mathbf{A} is the $(n + 1) \times (n + 1)$ identity matrix,⁹ except with a 0 in the top-left cell, corresponding to the bias term.

Equation 4-9. Ridge Regression closed-form solution

$$\hat{\boldsymbol{\theta}} = (\mathbf{X}^T \mathbf{X} + \alpha \mathbf{A})^{-1} \mathbf{X}^T \mathbf{y}$$

Here is how to perform Ridge Regression with Scikit-Learn using a closed-form solution (a variant of [Equation 4-9](#) that uses a matrix factorization technique by André-Louis Cholesky):

```
>>> from sklearn.linear_model import Ridge
>>> ridge_reg = Ridge(alpha=0.1, solver="cholesky")
>>> ridge_reg.fit(X, y)
>>> ridge_reg.predict([[1.5]])
array([1.55325833])
```

And using Stochastic Gradient Descent:¹⁰

```
>>> sgd_reg = SGDRegressor(penalty="l2", alpha=0.1 / m, tol=None,
...                         max_iter=1000, eta0=0.01, random_state=42)
...
>>> sgd_reg.fit(X, y.ravel()) # y.ravel() because fit() expects 1D targets
```

⁹ A square matrix full of 0s except for 1s on the main diagonal (top left to bottom right).

¹⁰ Alternatively you can use the `Ridge` class with the "sag" solver. Stochastic Average GD is a variant of Stochastic GD. For more details, see the presentation "[Minimizing Finite Sums with the Stochastic Average Gradient Algorithm](#)" by Mark Schmidt et al. from the University of British Columbia.

```
>>> sgd_reg.predict([[1.5]])
array([1.55302613])
```

The `penalty` hyperparameter sets the type of regularization term to use. Specifying "l2" indicates that you want SGD to add a regularization term to the MSE cost function equal to `alpha` times the square of the ℓ_2 norm of the weight vector: this is just like Ridge Regression, except there's no division by m in this case, which is why we passed `alpha=0.1 / m`, to get the same result as `Ridge(alpha=0.1)`.



The `RidgeCV` class also performs Ridge Regression, but it automatically tunes hyperparameters using cross-validation. It's roughly equivalent to using `GridSearchCV`, but it's optimized for Ridge Regression and runs *much* faster. Several other estimators (mostly linear) also have efficient CV variants, such as `LassoCV` or `ElasticNetCV`.

Lasso Regression

Least Absolute Shrinkage and Selection Operator Regression (usually simply called *Lasso Regression*) is another regularized version of Linear Regression: just like Ridge Regression, it adds a regularization term to the cost function, but it uses the ℓ_1 norm of the weight vector instead of the square of the ℓ_2 norm (see [Equation 4-10](#)). Notice that the ℓ_1 norm is multiplied by 2α , whereas the ℓ_2 norm was multiplied by α / m in Ridge regression. These factors were chosen to ensure that the optimal α value is independent from the training set size: different norms lead to different factors (see Scikit-Learn issue #15657 for more details).

Equation 4-10. Lasso Regression cost function

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + 2\alpha \sum_{i=1}^n |\theta_i|$$

[Figure 4-18](#) shows the same thing as [Figure 4-17](#) but replaces Ridge models with Lasso models and uses different α values.

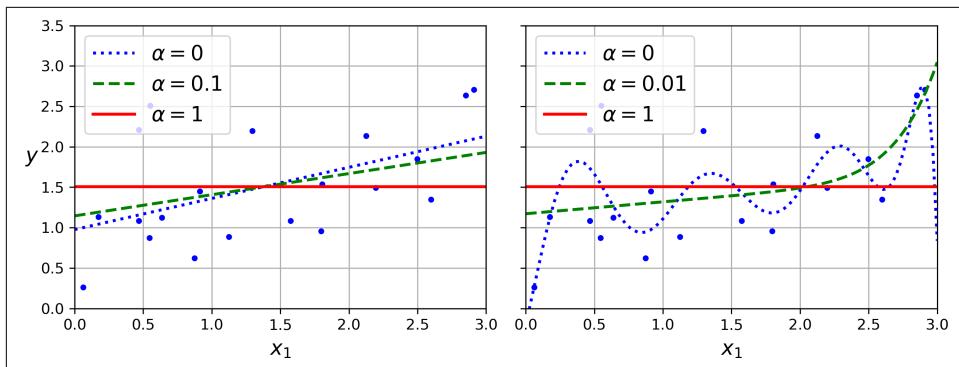


Figure 4-18. A linear model (left) and a polynomial model (right), both using various levels of Lasso regularization

An important characteristic of Lasso Regression is that it tends to eliminate the weights of the least important features (i.e., set them to zero). For example, the dashed line in the righthand plot in Figure 4-18 (with $\alpha = 0.01$) looks roughly cubic: all the weights for the high-degree polynomial features are equal to zero. In other words, Lasso Regression automatically performs feature selection and outputs a *sparse model* (i.e., with few nonzero feature weights).

You can get a sense of why this is the case by looking at Figure 4-19: the axes represent two model parameters, and the background contours represent different loss functions. In the top-left plot, the contours represent the ℓ_1 loss ($|\theta_1| + |\theta_2|$), which drops linearly as you get closer to any axis. For example, if you initialize the model parameters to $\theta_1 = 2$ and $\theta_2 = 0.5$, running Gradient Descent will decrement both parameters equally (as represented by the dashed yellow line); therefore θ_2 will reach 0 first (since it was closer to 0 to begin with). After that, Gradient Descent will roll down the gutter until it reaches $\theta_1 = 0$ (with a bit of bouncing around, since the gradients of ℓ_1 never get close to 0: they are either -1 or 1 for each parameter). In the top-right plot, the contours represent Lasso's cost function (i.e., an MSE cost function plus an ℓ_1 loss). The small white circles show the path that Gradient Descent takes to optimize some model parameters that were initialized around $\theta_1 = 0.25$ and $\theta_2 = -1$: notice once again how the path quickly reaches $\theta_2 = 0$, then rolls down the gutter and ends up bouncing around the global optimum (represented by the red square). If we increased α , the global optimum would move left along the dashed yellow line, while if we decreased α , the global optimum would move right (in this example, the optimal parameters for the unregularized MSE are $\theta_1 = 2$ and $\theta_2 = 0.5$).

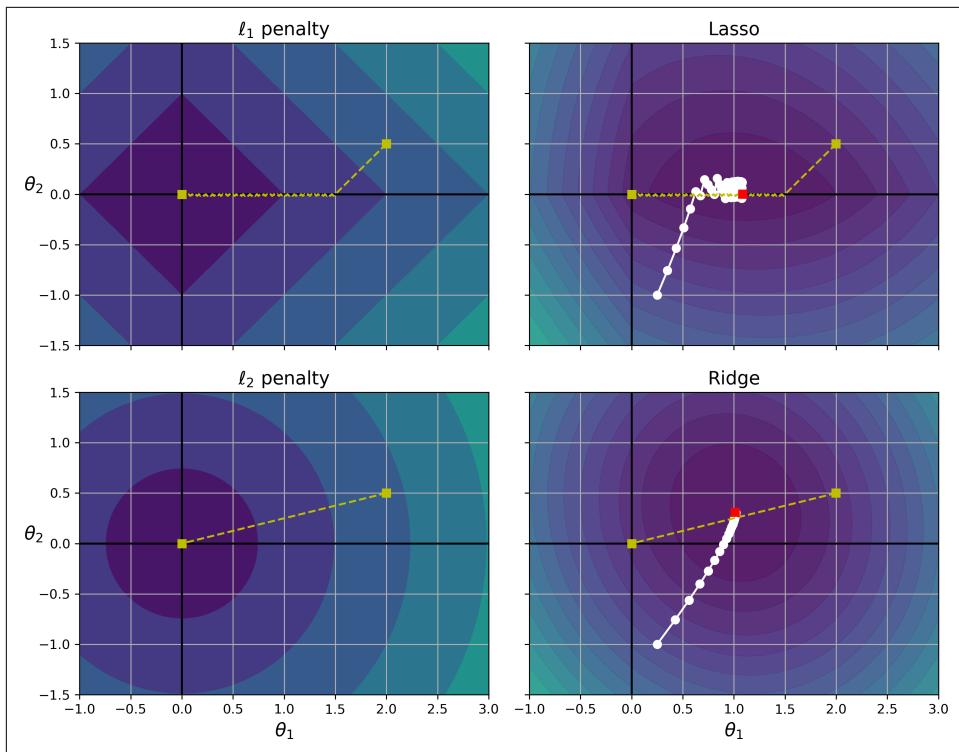


Figure 4-19. Lasso versus Ridge regularization

The two bottom plots show the same thing but with an ℓ_2 penalty instead. In the bottom-left plot, you can see that the ℓ_2 loss decreases as we get closer to the origin, so Gradient Descent just takes a straight path toward that point. In the bottom-right plot, the contours represent Ridge Regression's cost function (i.e., an MSE cost function plus an ℓ_2 loss). As you can see, the gradients get smaller as the parameters approach the global optimum, so Gradient Descent naturally slows down, which helps convergence (as there is no bouncing around). This helps Ridge converge faster than Lasso. Also note that the optimal parameters (represented by the red square) get closer and closer to the origin when you increase α , but they never get eliminated entirely.



To avoid Gradient Descent from bouncing around the optimum at the end when using Lasso, you need to gradually reduce the learning rate during training (it will still bounce around the optimum, but the steps will get smaller and smaller, so it will converge).

The Lasso cost function is not differentiable at $\theta_i = 0$ (for $i = 1, 2, \dots, n$), but Gradient Descent still works if you use a *subgradient vector* \mathbf{g} ¹¹ instead when any $\theta_i = 0$. [Equation 4-11](#) shows a subgradient vector equation you can use for Gradient Descent with the Lasso cost function.

Equation 4-11. Lasso Regression subgradient vector

$$g(\boldsymbol{\theta}, J) = \nabla_{\boldsymbol{\theta}} \text{MSE}(\boldsymbol{\theta}) + 2\alpha \begin{pmatrix} \text{sign}(\theta_1) \\ \text{sign}(\theta_2) \\ \vdots \\ \text{sign}(\theta_n) \end{pmatrix} \quad \text{where } \text{sign}(\theta_i) = \begin{cases} -1 & \text{if } \theta_i < 0 \\ 0 & \text{if } \theta_i = 0 \\ +1 & \text{if } \theta_i > 0 \end{cases}$$

Here is a small Scikit-Learn example using the `Lasso` class:

```
>>> from sklearn.linear_model import Lasso
>>> lasso_reg = Lasso(alpha=0.1)
>>> lasso_reg.fit(X, y)
>>> lasso_reg.predict([[1.5]])
array([1.53788174])
```

Note that you could instead use `SGDRegressor(penalty="l1", alpha=0.1)`.

Elastic Net

Elastic Net is a middle ground between Ridge Regression and Lasso Regression. The regularization term is a weighted sum of both Ridge and Lasso's regularization terms, and you can control the mix ratio r . When $r = 0$, Elastic Net is equivalent to Ridge Regression, and when $r = 1$, it is equivalent to Lasso Regression (see [Equation 4-12](#)).

Equation 4-12. Elastic Net cost function

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + r(2\alpha \sum_{i=1}^n |\theta_i|) + (1-r)\left(\frac{\alpha}{m} \sum_{i=1}^n \theta_i^2\right)$$

So when should you use plain Linear Regression (i.e., without any regularization), Ridge, Lasso, or Elastic Net? It is almost always preferable to have at least a little bit of regularization, so generally you should avoid plain Linear Regression. Ridge is a good default, but if you suspect that only a few features are useful, you should prefer Lasso or Elastic Net because they tend to reduce the useless features' weights down to zero, as we have discussed. In general, Elastic Net is preferred over Lasso because

¹¹ You can think of a subgradient vector at a nondifferentiable point as an intermediate vector between the gradient vectors around that point.

Lasso may behave erratically when the number of features is greater than the number of training instances or when several features are strongly correlated.

Here is a short example that uses Scikit-Learn's `ElasticNet` (`l1_ratio` corresponds to the mix ratio r):

```
>>> from sklearn.linear_model import ElasticNet
>>> elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
>>> elastic_net.fit(X, y)
>>> elastic_net.predict([[1.5]])
array([1.54333232])
```

Early Stopping

A very different way to regularize iterative learning algorithms such as Gradient Descent is to stop training as soon as the validation error reaches a minimum. This is called *early stopping*. Figure 4-20 shows a complex model (in this case, a high-degree Polynomial Regression model) being trained with Batch Gradient Descent on the quadratic dataset we used earlier. As the epochs go by, the algorithm learns, and its prediction error (RMSE) on the training set goes down, along with its prediction error on the validation set. After a while though, the validation error stops decreasing and starts to go back up. This indicates that the model has started to overfit the training data. With early stopping you just stop training as soon as the validation error reaches the minimum. It is such a simple and efficient regularization technique that Geoffrey Hinton called it a “beautiful free lunch.”

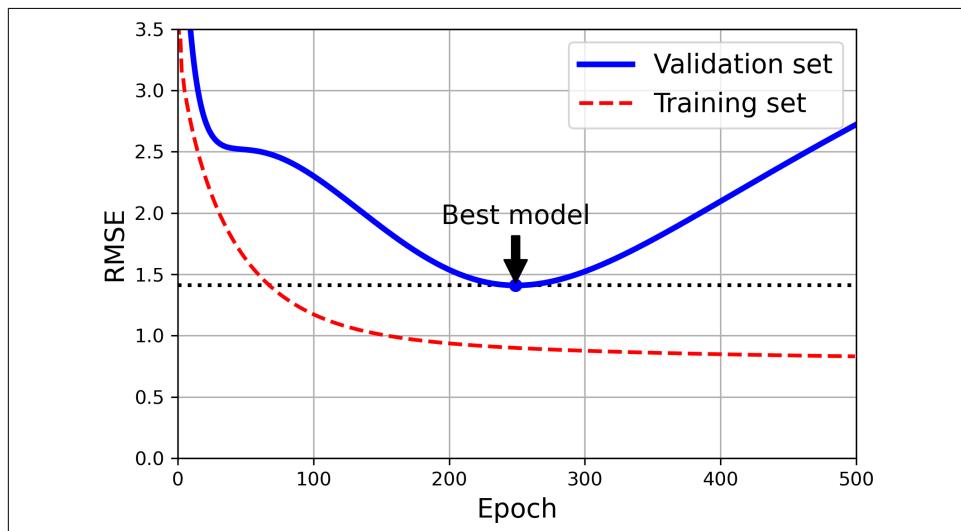


Figure 4-20. Early stopping regularization



With Stochastic and Mini-batch Gradient Descent, the curves are not so smooth, and it may be hard to know whether you have reached the minimum or not. One solution is to stop only after the validation error has been above the minimum for some time (when you are confident that the model will not do any better), then roll back the model parameters to the point where the validation error was at a minimum.

Here is a basic implementation of early stopping:

```
from copy import deepcopy
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import StandardScaler

X_train, y_train, X_valid, y_valid = [...] # split the quadratic dataset

preprocessing = make_pipeline(PolynomialFeatures(degree=90, include_bias=False),
                             StandardScaler())
X_train_prep = preprocessing.fit_transform(X_train)
X_valid_prep = preprocessing.transform(X_valid)
sgd_reg = SGDRegressor(penalty=None, eta0=0.002, random_state=42)
n_epochs = 500
best_valid_rmse = float('inf')

for epoch in range(n_epochs):
    sgd_reg.partial_fit(X_train_prep, y_train)
    y_valid_predict = sgd_reg.predict(X_valid_prep)
    val_error = mean_squared_error(y_valid, y_valid_predict, squared=False)
    if val_error < best_valid_rmse:
        best_valid_rmse = val_error
        best_model = deepcopy(sgd_reg)
```

Here's what this code does: it first adds the polynomial features and scales all the input features, both for the training set and the validation set (this code assumes that you have split the training set into a smaller training set and a validation set). Then it creates an `SGDRegressor` model with no regularization and a small learning rate. In the training loop, it calls `partial_fit()` instead of `fit()`, to perform incremental learning. At each epoch, it measures the RMSE on the validation set. If it is lower than the lowest RMSE seen so far, it saves a copy of the model in the `best_model` variable. This implementation does not actually stop training, but it lets you revert to the best model after training. Note that the model is copied using `copy.deepcopy()`, because it copies both the model's hyperparameters *and* the learned parameters. In contrast, `sklearn.base.clone()` only copies the model's hyperparameters.

Logistic Regression

As we discussed in [Chapter 1](#), some regression algorithms can be used for classification (and vice versa). *Logistic Regression* (also called *Logit Regression*) is commonly

used to estimate the probability that an instance belongs to a particular class (e.g., what is the probability that this email is spam?). If the estimated probability is greater than a given threshold (typically 50%), then the model predicts that the instance belongs to that class (called the *positive class*, labeled “1”), and otherwise it predicts that it does not (i.e., it belongs to the *negative class*, labeled “0”). This makes it a binary classifier.

Estimating Probabilities

So how does Logistic Regression work? Just like a Linear Regression model, a Logistic Regression model computes a weighted sum of the input features (plus a bias term), but instead of outputting the result directly like the Linear Regression model does, it outputs the *logistic* of this result (see [Equation 4-13](#)).

Equation 4-13. Logistic Regression model estimated probability (vectorized form)

$$\hat{p} = h_{\theta}(\mathbf{x}) = \sigma(\boldsymbol{\theta}^T \mathbf{x})$$

The logistic—noted $\sigma(\cdot)$ —is a *sigmoid function* (i.e., S-shaped) that outputs a number between 0 and 1. It is defined as shown in [Equation 4-14](#) and [Figure 4-21](#).

Equation 4-14. Logistic function

$$\sigma(t) = \frac{1}{1 + \exp(-t)}$$

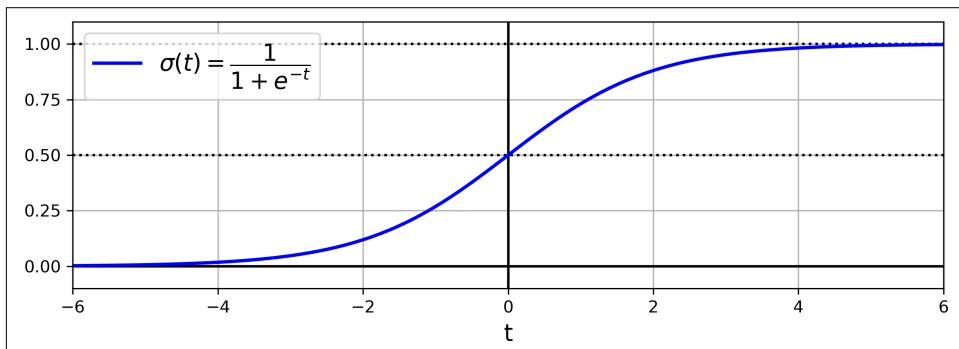


Figure 4-21. Logistic function

Once the Logistic Regression model has estimated the probability $\hat{p} = h_{\theta}(\mathbf{x})$ that an instance \mathbf{x} belongs to the positive class, it can make its prediction \hat{y} easily (see [Equation 4-15](#)).

Equation 4-15. Logistic Regression model prediction using a threshold probability of 50%

$$\hat{y} = \begin{cases} 0 & \text{if } \hat{p} < 0.5 \\ 1 & \text{if } \hat{p} \geq 0.5 \end{cases}$$

Notice that $\sigma(t) < 0.5$ when $t < 0$, and $\sigma(t) \geq 0.5$ when $t \geq 0$, so a Logistic Regression model using the default threshold of 50% probability predicts 1 if $\theta^T \mathbf{x}$ is positive and 0 if it is negative.



The score t is often called the *logit*. The name comes from the fact that the logit function, defined as $\text{logit}(p) = \log(p / (1 - p))$, is the inverse of the logistic function. Indeed, if you compute the logit of the estimated probability p , you will find that the result is t . The logit is also called the *log-odds*, since it is the log of the ratio between the estimated probability for the positive class and the estimated probability for the negative class.

Training and Cost Function

Now you know how a Logistic Regression model estimates probabilities and makes predictions. But how is it trained? The objective of training is to set the parameter vector θ so that the model estimates high probabilities for positive instances ($y = 1$) and low probabilities for negative instances ($y = 0$). This idea is captured by the cost function shown in [Equation 4-16](#) for a single training instance \mathbf{x} .

Equation 4-16. Cost function of a single training instance

$$c(\theta) = \begin{cases} -\log(\hat{p}) & \text{if } y = 1 \\ -\log(1 - \hat{p}) & \text{if } y = 0 \end{cases}$$

This cost function makes sense because $-\log(t)$ grows very large when t approaches 0, so the cost will be large if the model estimates a probability close to 0 for a positive instance, and it will also be very large if the model estimates a probability close to 1 for a negative instance. On the other hand, $-\log(t)$ is close to 0 when t is close to 1, so the cost will be close to 0 if the estimated probability is close to 0 for a negative instance or close to 1 for a positive instance, which is precisely what we want.

The cost function over the whole training set is the average cost over all training instances. It can be written in a single expression called the *log loss*, shown in [Equation 4-17](#).

Equation 4-17. Logistic Regression cost function (log loss)

$$J(\boldsymbol{\theta}) = -\frac{1}{m} \sum_{i=1}^m \left[y^{(i)} \log(\hat{p}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{p}^{(i)}) \right]$$



The log loss was not just pulled out of a hat. It can be shown mathematically (using Bayesian inference) that minimizing this loss will result in the model with the *maximum likelihood* of being optimal, assuming that the instances follow a Gaussian distribution around the mean of their class. When you use the log loss, this is the implicit assumption you are making. The more wrong this assumption is, the more biased the model will be. Similarly, when we used the MSE to train Linear Regression models, we were implicitly assuming that the data was purely linear, plus some Gaussian noise. So if the data is not linear (e.g., quadratic) or if the noise is not Gaussian (e.g., if outliers are not exponentially rare), then the model will be biased.

The bad news is that there is no known closed-form equation to compute the value of $\boldsymbol{\theta}$ that minimizes this cost function (there is no equivalent of the Normal Equation). But the good news is that this cost function is convex, so Gradient Descent (or any other optimization algorithm) is guaranteed to find the global minimum (if the learning rate is not too large and you wait long enough). The partial derivatives of the cost function with regard to the j^{th} model parameter θ_j are given by [Equation 4-18](#).

Equation 4-18. Logistic cost function partial derivatives

$$\frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^m (\sigma(\boldsymbol{\theta}^\top \mathbf{x}^{(i)}) - y^{(i)}) x_j^{(i)}$$

This equation looks very much like [Equation 4-5](#): for each instance it computes the prediction error and multiplies it by the j^{th} feature value, and then it computes the average over all training instances. Once you have the gradient vector containing all the partial derivatives, you can use it in the Batch Gradient Descent algorithm. That's it: you now know how to train a Logistic Regression model. For Stochastic GD you would take one instance at a time, and for Mini-batch GD you would use a mini-batch at a time.

Decision Boundaries

Let's use the iris dataset to illustrate Logistic Regression. This is a famous dataset that contains the sepal and petal length and width of 150 iris flowers of three different species: *Iris setosa*, *Iris versicolor*, and *Iris virginica* (see Figure 4-22).

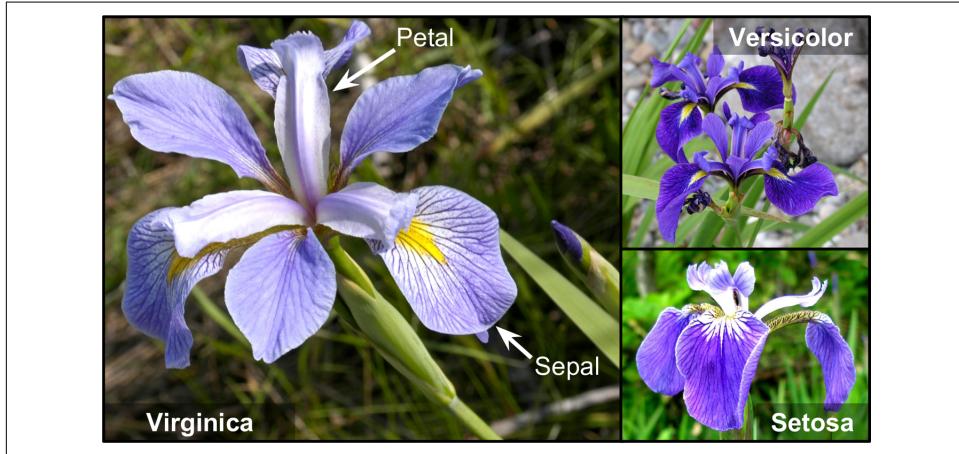


Figure 4-22. Flowers of three iris plant species¹²

Let's try to build a classifier to detect the *Iris virginica* type based only on the petal width feature. First let's load the data and take a quick peek:

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris(as_frame=True)
>>> list(iris)
['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names',
 'filename', 'data_module']
>>> iris.data.head(3)
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0              5.1          3.5            1.4           0.2
1              4.9          3.0            1.4           0.2
2              4.7          3.2            1.3           0.2
>>> iris.target.head(3) # note that the instances are not shuffled
0    0
1    0
2    0
Name: target, dtype: int64
>>> iris.target_names
array(['setosa', 'versicolor', 'virginica'], dtype='<U10')
```

¹² Photos reproduced from the corresponding Wikipedia pages. *Iris virginica* photo by Frank Mayfield ([Creative Commons BY-SA 2.0](#)), *Iris versicolor* photo by D. Gordon E. Robertson ([Creative Commons BY-SA 3.0](#)), *Iris setosa* photo public domain.

Now let's split the data and train a Logistic Regression model on the training set:

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

X = iris.data[["petal width (cm)"]].values
y = iris.target_names[iris.target] == 'virginica'
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

log_reg = LogisticRegression(random_state=42)
log_reg.fit(X_train, y_train)
```

Let's look at the model's estimated probabilities for flowers with petal widths varying from 0 cm to 3 cm (Figure 4-23):¹³

```
X_new = np.linspace(0, 3, 1000).reshape(-1, 1) # reshape to get a column vector
y_proba = log_reg.predict_proba(X_new)
decision_boundary = X_new[y_proba[:, 1] >= 0.5][0, 0]

plt.plot(X_new, y_proba[:, 0], "b--", linewidth=2,
          label="Not Iris virginica proba")
plt.plot(X_new, y_proba[:, 1], "g-", linewidth=2, label="Iris virginica proba")
plt.plot([decision_boundary, decision_boundary], [0, 1], "k:", linewidth=2,
          label="Decision boundary")
[...] # beautify the figure: add grid, labels, axis, legend, arrows and samples
plt.show()
```

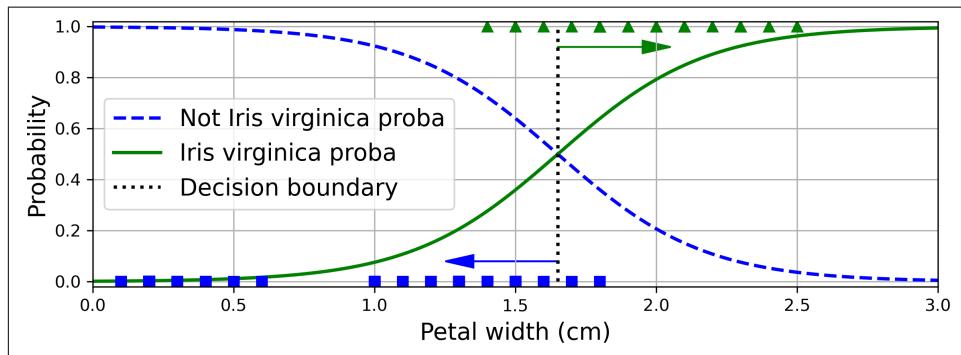


Figure 4-23. Estimated probabilities and decision boundary

The petal width of *Iris virginica* flowers (represented as triangles) ranges from 1.4 cm to 2.5 cm, while the other iris flowers (represented by squares) generally have a smaller petal width, ranging from 0.1 cm to 1.8 cm. Notice that there is a bit of overlap. Above about 2 cm the classifier is highly confident that the flower is an *Iris virginica* (it outputs a high probability for that class), while below 1 cm it is highly

¹³ NumPy's `reshape()` function allows one dimension to be -1, which means "automatic": the value is inferred from the length of the array and the remaining dimensions.

confident that it is not an *Iris virginica* (high probability for the “Not Iris virginica” class). In between these extremes, the classifier is unsure. However, if you ask it to predict the class (using the `predict()` method rather than the `predict_proba()` method), it will return whichever class is the most likely. Therefore, there is a *decision boundary* at around 1.6 cm where both probabilities are equal to 50%: if the petal width is higher than 1.6 cm, the classifier will predict that the flower is an *Iris virginica*, and otherwise it will predict that it is not (even if it is not very confident):

```
>>> decision_boundary
1.6516516516516517
>>> log_reg.predict([[1.7], [1.5]])
array([ True, False])
```

Figure 4-24 shows the same dataset, but this time displaying two features: petal width and length. Once trained, the Logistic Regression classifier can, based on these two features, estimate the probability that a new flower is an *Iris virginica*. The dashed line represents the points where the model estimates a 50% probability: this is the model’s decision boundary. Note that it is a linear boundary.¹⁴ Each parallel line represents the points where the model outputs a specific probability, from 15% (bottom left) to 90% (top right). All the flowers beyond the top-right line have over 90% chance of being *Iris virginica*, according to the model.

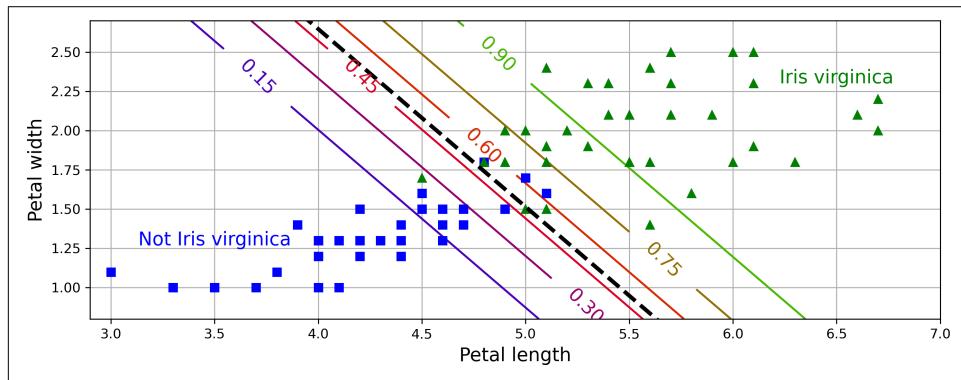


Figure 4-24. Linear decision boundary

Just like the other linear models, Logistic Regression models can be regularized using ℓ_1 or ℓ_2 penalties. Scikit-Learn actually adds an ℓ_2 penalty by default.

¹⁴ It is the set of points \mathbf{x} such that $\theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0$, which defines a straight line.



The hyperparameter controlling the regularization strength of a Scikit-Learn `LogisticRegression` model is not `alpha` (as in other linear models), but its inverse: `C`. The higher the value of `C`, the *less* the model is regularized.

Softmax Regression

The Logistic Regression model can be generalized to support multiple classes directly, without having to train and combine multiple binary classifiers (as discussed in [Chapter 3](#)). This is called *Softmax Regression*, or *Multinomial Logistic Regression*.

The idea is simple: when given an instance \mathbf{x} , the Softmax Regression model first computes a score $s_k(\mathbf{x})$ for each class k , then estimates the probability of each class by applying the *softmax function* (also called the *normalized exponential*) to the scores. The equation to compute $s_k(\mathbf{x})$ should look familiar, as it is just like the equation for Linear Regression prediction (see [Equation 4-19](#)).

Equation 4-19. Softmax score for class k

$$s_k(\mathbf{x}) = (\boldsymbol{\theta}^{(k)})^\top \mathbf{x}$$

Note that each class has its own dedicated parameter vector $\boldsymbol{\theta}^{(k)}$. All these vectors are typically stored as rows in a *parameter matrix* $\boldsymbol{\Theta}$.

Once you have computed the score of every class for the instance \mathbf{x} , you can estimate the probability \hat{p}_k that the instance belongs to class k by running the scores through the softmax function ([Equation 4-20](#)). The function computes the exponential of every score, then normalizes them (dividing by the sum of all the exponentials). The scores are generally called logits or log-odds (although they are actually unnormalized log-odds).

Equation 4-20. Softmax function

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp(s_k(\mathbf{x}))}{\sum_{j=1}^K \exp(s_j(\mathbf{x}))}$$

In this equation:

- K is the number of classes.
- $\mathbf{s}(\mathbf{x})$ is a vector containing the scores of each class for the instance \mathbf{x} .
- $\sigma(\mathbf{s}(\mathbf{x}))_k$ is the estimated probability that the instance \mathbf{x} belongs to class k , given the scores of each class for that instance.

Just like the Logistic Regression classifier, by default the Softmax Regression classifier predicts the class with the highest estimated probability (which is simply the class with the highest score), as shown in [Equation 4-21](#).

Equation 4-21. Softmax Regression classifier prediction

$$\hat{y} = \operatorname{argmax}_k \sigma(s(\mathbf{x}))_k = \operatorname{argmax}_k s_k(\mathbf{x}) = \operatorname{argmax}_k \left((\boldsymbol{\theta}^{(k)})^\top \mathbf{x} \right)$$

The *argmax* operator returns the value of a variable that maximizes a function. In this equation, it returns the value of k that maximizes the estimated probability $\sigma(s(\mathbf{x}))_k$.



The Softmax Regression classifier predicts only one class at a time (i.e., it is multiclass, not multioutput), so it should be used only with mutually exclusive classes, such as different species of plants. You cannot use it to recognize multiple people in one picture.

Now that you know how the model estimates probabilities and makes predictions, let's take a look at training. The objective is to have a model that estimates a high probability for the target class (and consequently a low probability for the other classes). Minimizing the cost function shown in [Equation 4-22](#), called the *cross entropy*, should lead to this objective because it penalizes the model when it estimates a low probability for a target class. Cross entropy is frequently used to measure how well a set of estimated class probabilities matches the target classes.

Equation 4-22. Cross entropy cost function

$$J(\boldsymbol{\Theta}) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log(\hat{p}_k^{(i)})$$

In this equation:

- $y_k^{(i)}$ is the target probability that the i^{th} instance belongs to class k . In general, it is either equal to 1 or 0, depending on whether the instance belongs to the class or not.

Notice that when there are just two classes ($K = 2$), this cost function is equivalent to the Logistic Regression's cost function (log loss; see [Equation 4-17](#)).

Cross Entropy

Cross entropy originated from Claude Shannon's *information theory*. Suppose you want to efficiently transmit information about the weather every day. If there are eight

options (sunny, rainy, etc.), you could encode each option using three bits because $2^3 = 8$. However, if you think it will be sunny almost every day, it would be much more efficient to code “sunny” on just one bit (0) and the other seven options on four bits (starting with a 1). Cross entropy measures the average number of bits you actually send per option. If your assumption about the weather is perfect, cross entropy will be equal to the entropy of the weather itself (i.e., its intrinsic unpredictability). But if your assumptions are wrong (e.g., if it rains often), cross entropy will be greater by an amount called the *Kullback–Leibler (KL) divergence*.

The cross entropy between two probability distributions p and q is defined as $H(p, q) = -\sum_x p(x) \log q(x)$ (at least when the distributions are discrete). For more details, check out [my video on the subject](#).

The gradient vector of this cost function with regard to $\Theta^{(k)}$ is given by [Equation 4-23](#).

Equation 4-23. Cross entropy gradient vector for class k

$$\nabla_{\Theta^{(k)}} J(\Theta) = \frac{1}{m} \sum_{i=1}^m (\hat{p}_k^{(i)} - y_k^{(i)}) \mathbf{x}^{(i)}$$

Now you can compute the gradient vector for every class, then use Gradient Descent (or any other optimization algorithm) to find the parameter matrix Θ that minimizes the cost function.

Let’s use Softmax Regression to classify the iris plants into all three classes. Scikit-Learn’s `LogisticRegression` uses Softmax Regression automatically when you train it on more than two classes (assuming you use `solver="lbfgs"`, which is the default). It also applies ℓ_2 regularization by default, which you can control using the hyperparameter `C`, as earlier:

```
X = iris.data[["petal length (cm)", "petal width (cm)"]].values
y = iris["target"]
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

softmax_reg = LogisticRegression(C=30, random_state=42)
softmax_reg.fit(X_train, y_train)
```

So the next time you find an iris with petals that are 5 cm long and 2 cm wide, you can ask your model to tell you what type of iris it is, and it will answer *Iris virginica* (class 2) with 96% probability (or *Iris versicolor* with 4% probability):

```
>>> softmax_reg.predict([[5, 2]])
array([2])
>>> softmax_reg.predict_proba([[5, 2]]).round(2)
array([[0.   , 0.04, 0.96]])
```

Figure 4-25 shows the resulting decision boundaries, represented by the background colors. Notice that the decision boundaries between any two classes are linear. The figure also shows the probabilities for the *Iris versicolor* class, represented by the curved lines (e.g., the line labeled with 0.30 represents the 30% probability boundary). Notice that the model can predict a class that has an estimated probability below 50%. For example, at the point where all decision boundaries meet, all classes have an equal estimated probability of 33%.

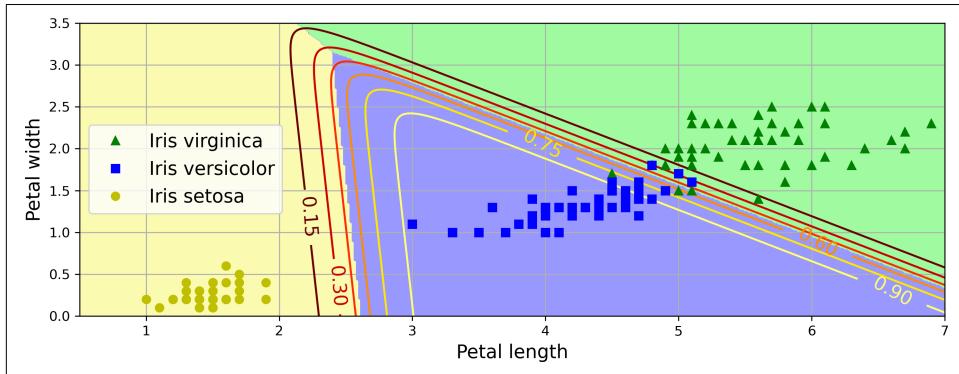


Figure 4-25. Softmax Regression decision boundaries

In this chapter, you learned various ways to train linear models, both for regression and classification. You used a closed-form equation to solve Linear Regression, as well as Gradient Descent, and you learned how various penalties can be added to the cost function during training to regularize the model. Along the way, you also learned how to plot learning curves and analyze them, and how to implement early stopping. Finally, you learned how Logistic Regression and Softmax Regression work. We've opened up the first Machine Learning black boxes! In the next chapters, we will open many more, starting with Support Vector Machines.

Exercises

1. Which Linear Regression training algorithm can you use if you have a training set with millions of features?
2. Suppose the features in your training set have very different scales. Which algorithms might suffer from this, and how? What can you do about it?
3. Can Gradient Descent get stuck in a local minimum when training a Logistic Regression model?
4. Do all Gradient Descent algorithms lead to the same model, provided you let them run long enough?

5. Suppose you use Batch Gradient Descent and you plot the validation error at every epoch. If you notice that the validation error consistently goes up, what is likely going on? How can you fix this?
6. Is it a good idea to stop Mini-batch Gradient Descent immediately when the validation error goes up?
7. Which Gradient Descent algorithm (among those we discussed) will reach the vicinity of the optimal solution the fastest? Which will actually converge? How can you make the others converge as well?
8. Suppose you are using Polynomial Regression. You plot the learning curves and you notice that there is a large gap between the training error and the validation error. What is happening? What are three ways to solve this?
9. Suppose you are using Ridge Regression and you notice that the training error and the validation error are almost equal and fairly high. Would you say that the model suffers from high bias or high variance? Should you increase the regularization hyperparameter α or reduce it?
10. Why would you want to use:
 - a. Ridge Regression instead of plain Linear Regression (i.e., without any regularization)?
 - b. Lasso instead of Ridge Regression?
 - c. Elastic Net instead of Lasso?
11. Suppose you want to classify pictures as outdoor/indoor and daytime/nighttime. Should you implement two Logistic Regression classifiers or one Softmax Regression classifier?
12. Implement Batch Gradient Descent with early stopping for Softmax Regression without using Scikit-Learn, only NumPy. Use it on a classification task such as the iris dataset.

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab3>.

Support Vector Machines

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. The GitHub repo is <https://github.com/ageron/handsonml3>.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

A *Support Vector Machine* (SVM) is a powerful and versatile Machine Learning model, capable of performing linear or nonlinear classification, regression, and even novelty detection. SVMs shine with small to medium-sized nonlinear datasets (i.e., hundreds to thousands of instances), especially for classification tasks. However, they don’t scale very well to very large datasets, as we will see.

This chapter will explain the core concepts of SVMs, how to use them, and how they work. Let’s jump right in!

Linear SVM Classification

The fundamental idea behind SVMs is best explained with some visuals. [Figure 5-1](#) shows part of the iris dataset that was introduced at the end of [Chapter 4](#). The two classes can clearly be separated easily with a straight line (they are *linearly separable*). The left plot shows the decision boundaries of three possible linear classifiers. The model whose decision boundary is represented by the dashed line is so bad that it

does not even separate the classes properly. The other two models work perfectly on this training set, but their decision boundaries come so close to the instances that these models will probably not perform as well on new instances. In contrast, the solid line in the plot on the right represents the decision boundary of an SVM classifier; this line not only separates the two classes but also stays as far away from the closest training instances as possible. You can think of an SVM classifier as fitting the widest possible street (represented by the parallel dashed lines) between the classes. This is called *large margin classification*.

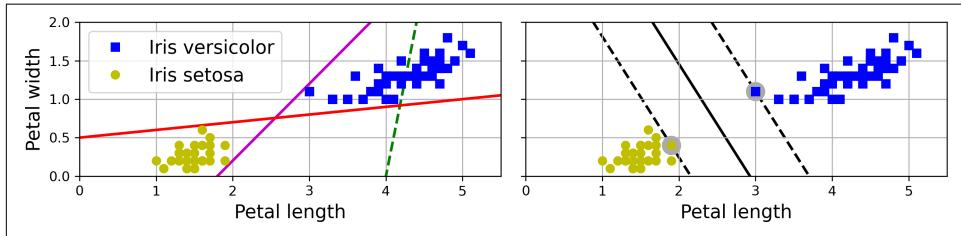


Figure 5-1. Large margin classification

Notice that adding more training instances “off the street” will not affect the decision boundary at all: it is fully determined (or “supported”) by the instances located on the edge of the street. These instances are called the *support vectors* (they are circled in Figure 5-1).



SVMs are sensitive to the feature scales, as you can see in Figure 5-2: in the left plot, the vertical scale is much larger than the horizontal scale, so the widest possible street is close to horizontal. After feature scaling (e.g., using Scikit-Learn’s `StandardScaler`), the decision boundary in the right plot looks much better.

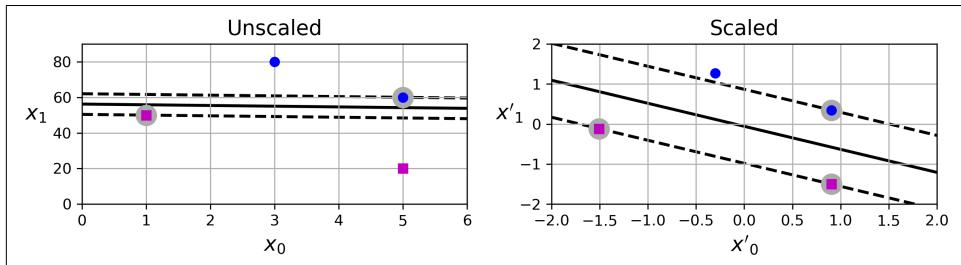


Figure 5-2. Sensitivity to feature scales

Soft Margin Classification

If we strictly impose that all instances must be off the street and on the correct side, this is called *hard margin classification*. There are two main issues with hard

margin classification. First, it only works if the data is linearly separable. Second, it is sensitive to outliers. [Figure 5-3](#) shows the iris dataset with just one additional outlier: on the left, it is impossible to find a hard margin; on the right, the decision boundary ends up very different from the one we saw in [Figure 5-1](#) without the outlier, and it will probably not generalize as well.

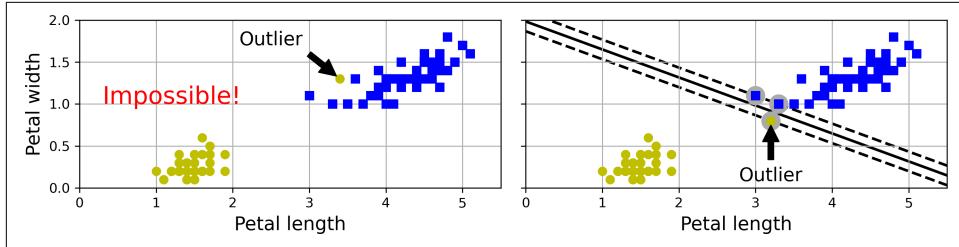


Figure 5-3. Hard margin sensitivity to outliers

To avoid these issues, use a more flexible model. The objective is to find a good balance between keeping the street as large as possible and limiting the *margin violations* (i.e., instances that end up in the middle of the street or even on the wrong side). This is called *soft margin classification*.

When creating an SVM model using Scikit-Learn, you can specify several hyperparameters, including a regularization hyperparameter called C . If you set it to a low value, then you end up with the model on the left of [Figure 5-4](#). With a high value, you get the model on the right. As you can see, reducing C makes the street larger, but it also leads to more margin violations. In other words, reducing C results in more instances supporting the street, so there's less risk of overfitting. But if you reduce it too much, then the model ends up underfitting, as seems to be the case here: the model with $C=100$ looks like it will generalize better than the one with $C=1$.

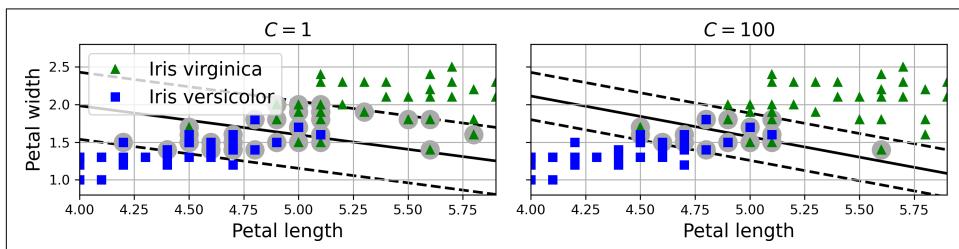


Figure 5-4. Large margin (left) versus fewer margin violations (right)



If your SVM model is overfitting, you can try regularizing it by reducing C .

The following Scikit-Learn code loads the iris dataset, and trains a linear SVM classifier to detect *Iris virginica* flowers. The pipeline first scales the features, then uses a LinearSVC with C=1:

```
from sklearn.datasets import load_iris
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

iris = load_iris(as_frame=True)
X = iris.data[["petal length (cm)", "petal width (cm)"]].values
y = (iris.target == 2) # Iris virginica

svm_clf = make_pipeline(StandardScaler(),
                       LinearSVC(C=1, random_state=42))
svm_clf.fit(X, y)
```

The resulting model is represented on the left in [Figure 5-4](#).

Then, as usual, you can use the model to make predictions:

```
>>> X_new = [[5.5, 1.7], [5.0, 1.5]]
>>> svm_clf.predict(X_new)
array([ True, False])
```

The first plant is classified as an Iris virginia, while the second is not. Let's look at the scores that the SVM used to make these predictions. These measure the signed distance between each instance and the decision boundary:

```
>>> svm_clf.decision_function(X_new)
array([ 0.66163411, -0.22036063])
```

Unlike the `LogisticRegression` classifier, `LinearSVC` does not have a `predict_proba()` method to estimate the class probabilities. That said, if you use the `SVC` class (discussed shortly) instead of `LinearSVC`, and if you set its `probability` to `True`, then the model will fit an extra model at the end of training to map the SVM decision function scores to estimated probabilities. Under the hood, this requires using 5-fold cross-validation to generate out-of-sample predictions for every instance in the training set, then training a `LogisticRegression` model, so it will slow down training considerably. After that, the `predict_proba()` and `predict_log_proba()` methods will be available.

Nonlinear SVM Classification

Although linear SVM classifiers are efficient and often work surprisingly well, many datasets are not even close to being linearly separable. One approach to handling nonlinear datasets is to add more features, such as polynomial features (as you did in [Chapter 4](#)); in some cases this can result in a linearly separable dataset. Consider the left plot in [Figure 5-5](#): it represents a simple dataset with just one feature, x_1 . This

dataset is not linearly separable, as you can see. But if you add a second feature $x_2 = (x_1)^2$, the resulting 2D dataset is perfectly linearly separable.

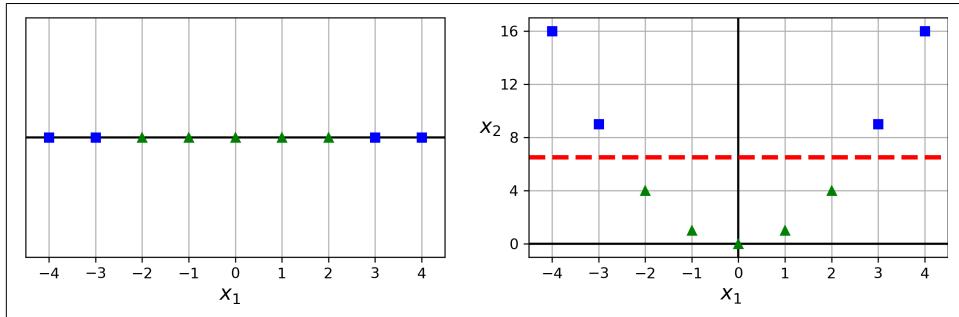


Figure 5-5. Adding features to make a dataset linearly separable

To implement this idea using Scikit-Learn, create a pipeline containing a `PolyomialFeatures` transformer (discussed in “[Polynomial Regression](#)” on page 152), followed by a `StandardScaler` and a `LinearSVC`. Let’s test this on the moons dataset: this is a toy dataset for binary classification in which the data points are shaped as two interleaving crescent moons (see Figure 5-6). You can generate this dataset using the `make_moons()` function:

```
from sklearn.datasets import make_moons
from sklearn.preprocessing import PolynomialFeatures

X, y = make_moons(n_samples=100, noise=0.15, random_state=42)

polynomial_svm_clf = make_pipeline(
    PolynomialFeatures(degree=3),
    StandardScaler(),
    LinearSVC(C=10, max_iter=10_000, random_state=42)
)
polynomial_svm_clf.fit(X, y)
```

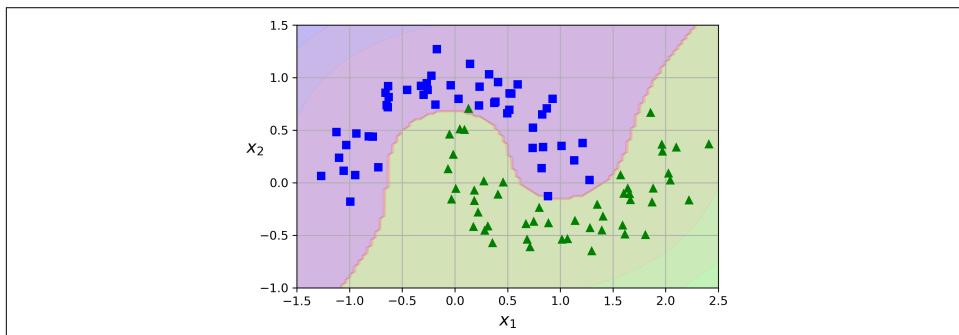


Figure 5-6. Linear SVM classifier using polynomial features

Polynomial Kernel

Adding polynomial features is simple to implement and can work great with all sorts of Machine Learning algorithms (not just SVMs). That said, at a low polynomial degree, this method cannot deal with very complex datasets, and with a high polynomial degree it creates a huge number of features, making the model too slow.

Fortunately, when using SVMs you can apply an almost miraculous mathematical technique called the *kernel trick* (which is explained later in this chapter). The kernel trick makes it possible to get the same result as if you had added many polynomial features, even with a very high-degree, without actually having to add them. This means there's no combinatorial explosion of the number of features. This trick is implemented by the SVC class. Let's test it on the moons dataset:

```
from sklearn.svm import SVC

poly_kernel_svm_clf = make_pipeline(StandardScaler(),
                                    SVC(kernel="poly", degree=3, coef0=1, C=5))
poly_kernel_svm_clf.fit(X, y)
```

This code trains an SVM classifier using a third-degree polynomial kernel. It is represented on the left in Figure 5-7. On the right is another SVM classifier using a 10th-degree polynomial kernel. Obviously, if your model is overfitting, you might want to reduce the polynomial degree. Conversely, if it is underfitting, you can try increasing it. The hyperparameter `coef0` controls how much the model is influenced by high-degree terms versus low-degree terms.

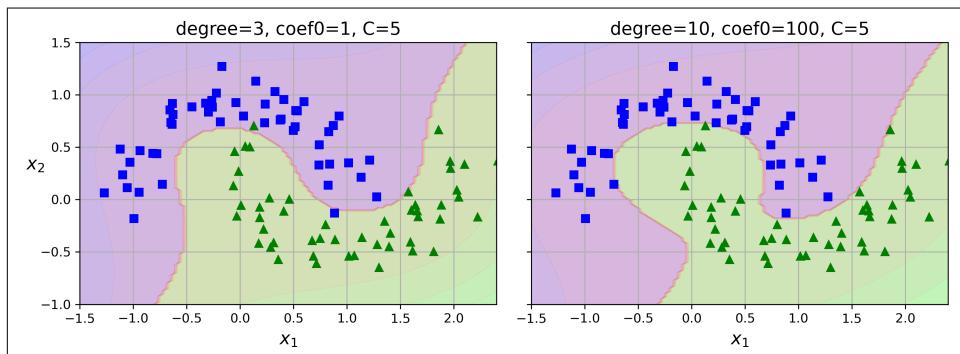


Figure 5-7. SVM classifiers with a polynomial kernel



Although hyperparameters will generally be tuned automatically (e.g., using randomized search), it's good to have a sense of what each hyperparameter actually does, and how it may interact with other hyperparameters: this way, you can narrow the search to a much smaller space.

Similarity Features

Another technique to tackle nonlinear problems is to add features computed using a similarity function, which measures how much each instance resembles a particular *landmark*, as we did in [Chapter 2](#) when we added the geographic similarity features. For example, let's take the 1D dataset discussed earlier and add two landmarks to it at $x_1 = -2$ and $x_1 = 1$ (see the left plot in [Figure 5-8](#)). Next, let's define the similarity function to be the Gaussian RBF with $\gamma = 0.3$. This is a bell-shaped function varying from 0 (very far away from the landmark) to 1 (at the landmark).

Now we are ready to compute the new features. For example, let's look at the instance $x_1 = -1$: it is located at a distance of 1 from the first landmark and 2 from the second landmark. Therefore its new features are $x_2 = \exp(-0.3 \times 1^2) \approx 0.74$ and $x_3 = \exp(-0.3 \times 2^2) \approx 0.30$. The plot on the right in [Figure 5-8](#) shows the transformed dataset (dropping the original features). As you can see, it is now linearly separable.

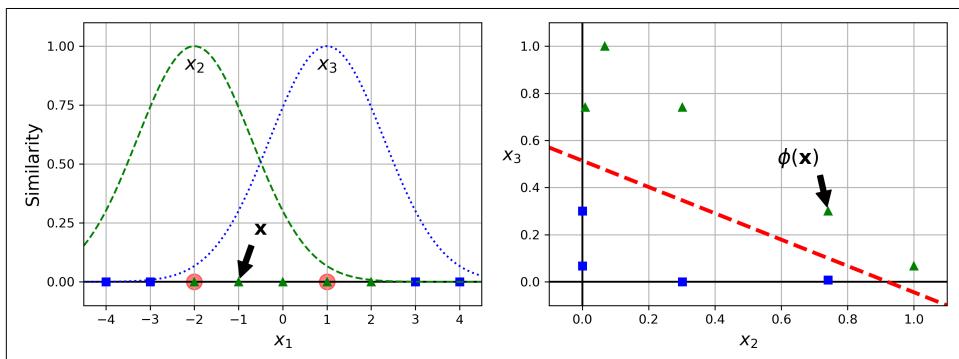


Figure 5-8. Similarity features using the Gaussian RBF

You may wonder how to select the landmarks. The simplest approach is to create a landmark at the location of each and every instance in the dataset. Doing that creates many dimensions and thus increases the chances that the transformed training set will be linearly separable. The downside is that a training set with m instances and n features gets transformed into a training set with m instances and m features (assuming you drop the original features). If your training set is very large, you end up with an equally large number of features.

Gaussian RBF Kernel

Just like the polynomial features method, the similarity features method can be useful with any Machine Learning algorithm, but it may be computationally expensive to compute all the additional features, especially on large training sets. Once again the kernel trick does its SVM magic, making it possible to obtain a similar result as if you

had added many similarity features, but without actually doing so. Let's try the SVC class with the Gaussian RBF kernel:

```
rbf_kernel_svm_clf = make_pipeline(StandardScaler(),
                                    SVC(kernel="rbf", gamma=5, C=0.001))
rbf_kernel_svm_clf.fit(X, y)
```

This model is represented at the bottom left in [Figure 5-9](#). The other plots show models trained with different values of hyperparameters γ (gamma) and C . Increasing γ makes the bell-shaped curve narrower (see the lefthand plots in [Figure 5-8](#)). As a result, each instance's range of influence is smaller: the decision boundary ends up being more irregular, wiggling around individual instances. Conversely, a small γ value makes the bell-shaped curve wider: instances have a larger range of influence, and the decision boundary ends up smoother. So γ acts like a regularization hyperparameter: if your model is overfitting, you should reduce γ ; if it is underfitting, you should increase γ (similar to the C hyperparameter).

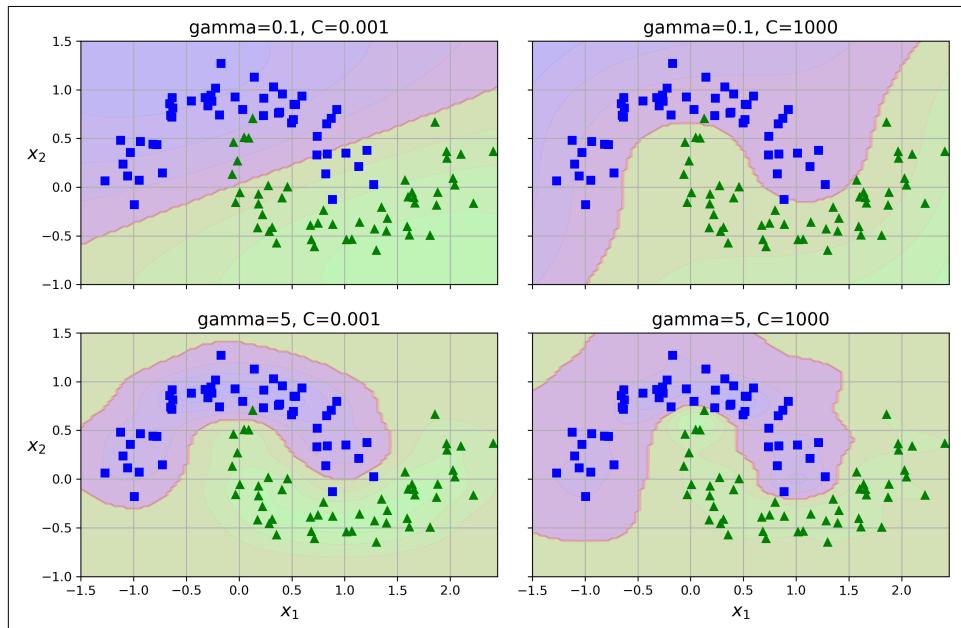


Figure 5-9. SVM classifiers using an RBF kernel

Other kernels exist but are used much more rarely. Some kernels are specialized for specific data structures. *String kernels* are sometimes used when classifying text documents or DNA sequences (e.g., using the *string subsequence kernel* or kernels based on the *Levenshtein distance*).



With so many kernels to choose from, how can you decide which one to use? As a rule of thumb, you should always try the linear kernel first. The `LinearSVC` class is much faster than `SVC(kernel="linear")`, especially if the training set is very large. If it is not too large, you should also try kernelized SVMs, starting with the Gaussian RBF kernel; it often works really well. Then if you have spare time and computing power, you can experiment with a few other kernels, using hyperparameter search. If there are kernels specialized for your training set's data structure, make sure to give them a try too.

SVM Classes and Computational Complexity

The `LinearSVC` class is based on the `liblinear` library, which implements an [optimized algorithm](#) for linear SVMs.¹ It does not support the kernel trick, but it scales almost linearly with the number of training instances and the number of features. Its training time complexity is roughly $O(m \times n)$. The algorithm takes longer if you require very high precision. This is controlled by the tolerance hyperparameter ϵ (called `tol` in Scikit-Learn). In most classification tasks, the default tolerance is fine.

The `SVC` class is based on the `libsvm` library, which implements [an algorithm](#) that supports the kernel trick.² The training time complexity is usually between $O(m^2 \times n)$ and $O(m^3 \times n)$. Unfortunately, this means that it gets dreadfully slow when the number of training instances gets large (e.g., hundreds of thousands of instances). This algorithm is best for small or medium-sized nonlinear training sets. It scales well with the number of features, especially with *sparse features* (i.e., when each instance has few nonzero features). In this case, the algorithm scales roughly with the average number of nonzero features per instance. [Table 5-1](#) compares Scikit-Learn's SVM classification classes.

Lastly, the `SGDClassifier` class also performs large-margin classification by default, and its hyperparameters can be adjusted to produce similar results as the linear SVMs, especially the regularization hyperparameters (`alpha` and `penalty`) and the `learning_rate`. For training, it uses Stochastic Gradient Descent (see [Chapter 4](#)), which allows incremental learning and uses little memory, so you can use it to train a model on a large dataset that does not fit in RAM (i.e., out-of-core). Moreover, it scales very well, as its computational complexity is $O(m \times n)$.

¹ Chih-Jen Lin et al., “A Dual Coordinate Descent Method for Large-Scale Linear SVM,” *Proceedings of the 25th International Conference on Machine Learning* (2008): 408–415.

² John Platt, “Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines” (Microsoft Research technical report, April 21, 1998), <https://homl.info/smo>.

Table 5-1. Comparison of Scikit-Learn classes for SVM classification

Class	Time Complexity	Out-of-core support	Scaling required	Kernel trick
LinearSVC	$O(m \times n)$	No	Yes	No
SVC	$O(m^2 \times n)$ to $O(m^3 \times n)$	No	Yes	Yes
SGDClassifier	$O(m \times n)$	Yes	Yes	No

Now let's see how the SVM algorithm can also be used for linear and nonlinear regression.

SVM Regression

To use SVMs for regression instead of classification, the trick is to tweak the objective: instead of trying to fit the largest possible street between two classes while limiting margin violations, SVM Regression tries to fit as many instances as possible *on* the street while limiting margin violations (i.e., instances *off* the street). The width of the street is controlled by a hyperparameter, ϵ . Figure 5-10 shows two linear SVM Regression models trained on some linear data, one with a small margin ($\epsilon = 0.5$) and the other with a larger margin ($\epsilon = 1.2$).

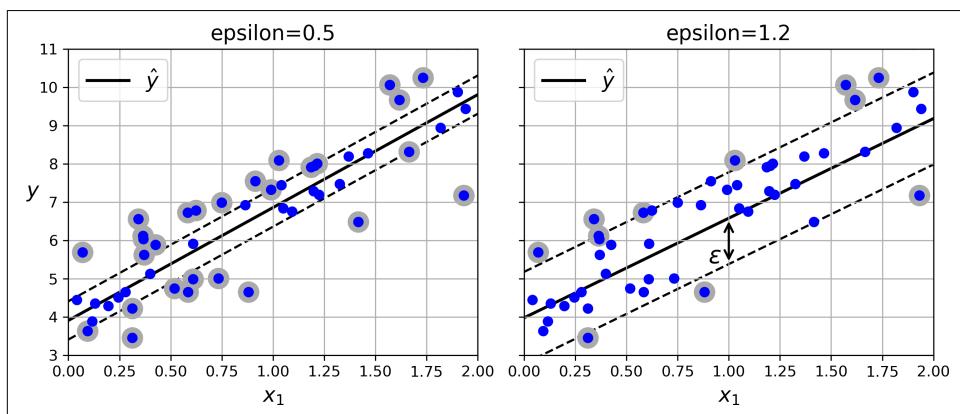


Figure 5-10. SVM Regression

Reducing ϵ increases the number of support vectors, which regularizes the model. Moreover, if you add more training instances within the margin, it will not affect the model's predictions; thus, the model is said to be *ϵ -insensitive*.

You can use Scikit-Learn's `LinearSVR` class to perform linear SVM Regression. The following code produces the model represented on the left in Figure 5-10:

```
from sklearn.svm import LinearSVR
X, y = [...] # a linear dataset
```

```

svm_reg = make_pipeline(StandardScaler(),
                       LinearSVR(epsilon=0.5, random_state=42))
svm_reg.fit(X, y)

```

To tackle nonlinear regression tasks, you can use a kernelized SVM model. [Figure 5-11](#) shows SVM Regression on a random quadratic training set, using a second-degree polynomial kernel. There is some regularization in the left plot (i.e., a small C value), and much less in the right plot (i.e., a large C value).

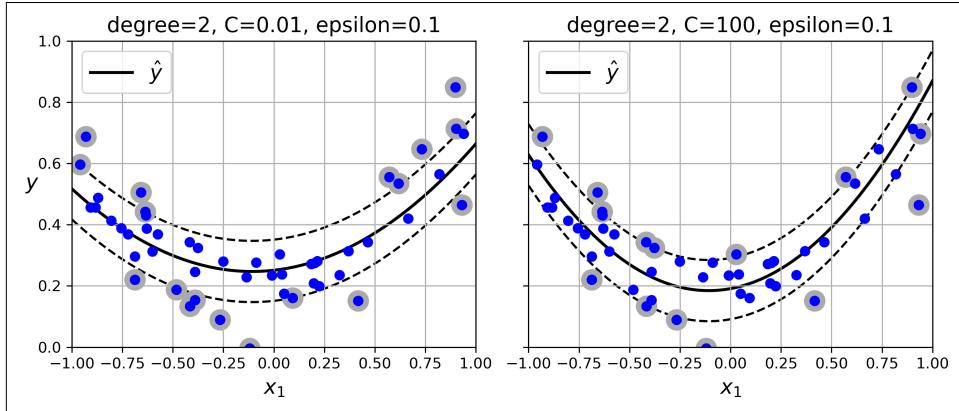


Figure 5-11. SVM Regression using a second-degree polynomial kernel

The following code uses Scikit-Learn’s SVR class (which supports the kernel trick) to produce the model represented on the left in [Figure 5-11](#):

```

from sklearn.svm import SVR

X, y = [...] # a quadratic dataset
svm_poly_reg = make_pipeline(StandardScaler(),
                            SVR(kernel="poly", degree=2, C=0.01, epsilon=0.1))
svm_poly_reg.fit(X, y)

```

The SVR class is the regression equivalent of the SVC class, and the LinearSVR class is the regression equivalent of the LinearSVC class. The LinearSVR class scales linearly with the size of the training set (just like the LinearSVC class), while the SVR class gets much too slow when the training set grows very large (just like the SVC class).



SVMs can also be used for novelty detection, as we will see in [Chapter 9](#).

The rest of this chapter explains how SVMs make predictions and how their training algorithms work, starting with linear SVM classifiers. If you are just getting started

with Machine Learning, you can safely skip this and go straight to the exercises at the end of this chapter, and come back later when you want to get a deeper understanding of SVMs.

Under The Hood of Linear SVM Classifiers

The linear SVM classifier model predicts the class of a new instance \mathbf{x} by first computing the decision function $\boldsymbol{\theta}^\top \mathbf{x} = \theta_0 x_0 + \dots + \theta_n x_n$, where x_0 is the bias feature (always equal to 1). If the result is positive, then the predicted class \hat{y} is the positive class (1), otherwise it is the negative class (0). This is exactly like Logistic Regression (discussed in [Chapter 4](#)).



Up to now, I have used the convention of putting all the model parameters in one vector $\boldsymbol{\theta}$, including the bias term θ_0 and the input feature weights θ_1 to θ_n . This required adding a bias input $x_0 = 1$ to all instances. Another very common convention is to separate the bias term b (equal to θ_0), and the feature weights vector \mathbf{w} (containing θ_1 to θ_n). In this case, no bias feature needs to be added to the input feature vectors, and the linear SVM's decision function is equal to $\mathbf{w}^\top \mathbf{x} + b = w_1 x_1 + \dots + w_n x_n + b$. I will use this convention throughout the rest of this book.

So making predictions with a linear SVM classifier is quite straightforward. How about training? Well, it requires finding the weights vector \mathbf{w} and the bias term b that make the “street” (i.e., the margin) as wide as possible while limiting the number of margin violations. Let’s start with the width of the street: to make it larger, we need to make \mathbf{w} smaller. This may be easier to visualize in 2D, as shown in [Figure 5-12](#). Let’s define the borders of the street as the points where the decision function is equal to -1 or $+1$. On the left plot, the weight w_1 is 1, so the points at which $w_1 x_1 = -1$ or $+1$ are $x_1 = -1$ and $+1$: therefore the margin’s size is 2. On the right plot, the weight is 0.5, so the points at which $w_1 x_1 = -1$ or $+1$ are $x_1 = -2$ and $+2$: the margin’s size is 4. So we need to keep \mathbf{w} as small as possible. Note that the bias term b has no influence on the size of the margin: tweaking it just shifts the margin around, without affecting its size.

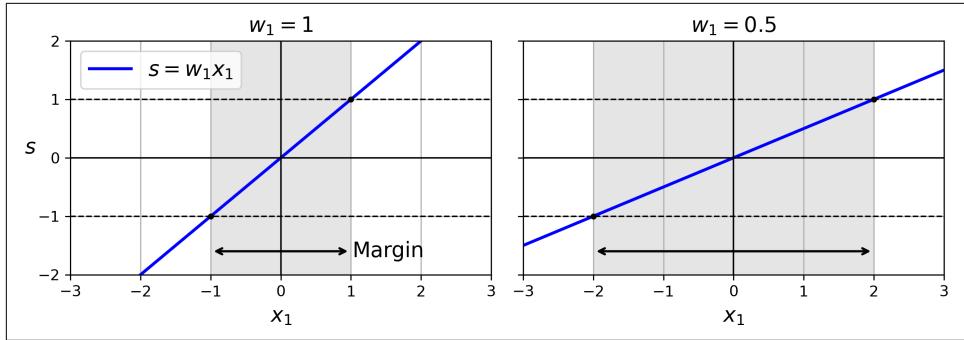


Figure 5-12. A smaller weight vector results in a larger margin

We also want to avoid margin violations, so we need the decision function to be greater than 1 for all positive training instances and lower than -1 for negative training instances. If we define $t^{(i)} = -1$ for negative instances (when $y^{(i)} = 0$) and $t^{(i)} = 1$ for positive instances (when $y^{(i)} = 1$), then we can write this constraint as $t^{(i)}(\mathbf{w}^\top \mathbf{x}^{(i)} + b) \geq 1$ for all instances.

We can therefore express the hard margin linear SVM classifier objective as the constrained optimization problem in [Equation 5-1](#).

Equation 5-1. Hard margin linear SVM classifier objective

$$\begin{aligned} & \underset{\mathbf{w}, b}{\text{minimize}} \quad \frac{1}{2} \mathbf{w}^\top \mathbf{w} \\ & \text{subject to} \quad t^{(i)} (\mathbf{w}^\top \mathbf{x}^{(i)} + b) \geq 1 \quad \text{for } i = 1, 2, \dots, m \end{aligned}$$



We are minimizing $\frac{1}{2} \mathbf{w}^\top \mathbf{w}$, which is equal to $\frac{1}{2} \|\mathbf{w}\|^2$, rather than minimizing $\|\mathbf{w}\|$ (the norm of \mathbf{w}). Indeed, $\frac{1}{2} \|\mathbf{w}\|^2$ has a nice, simple derivative (it is just \mathbf{w}), while $\|\mathbf{w}\|$ is not differentiable at $\mathbf{w} = 0$. Optimization algorithms often work much better on differentiable functions.

To get the soft margin objective, we need to introduce a *slack variable* $\zeta^{(i)} \geq 0$ for each instance:³ $\zeta^{(i)}$ measures how much the i^{th} instance is allowed to violate the margin. We now have two conflicting objectives: make the slack variables as small as possible to reduce the margin violations, and make $\frac{1}{2} \mathbf{w}^\top \mathbf{w}$ as small as possible to increase the margin. This is where the C hyperparameter comes in: it allows us to define the

³ Zeta (ζ) is the sixth letter of the Greek alphabet.

tradeoff between these two objectives. This gives us the constrained optimization problem in [Equation 5-2](#).

Equation 5-2. Soft margin linear SVM classifier objective

$$\begin{aligned} & \underset{\mathbf{w}, b, \zeta}{\text{minimize}} \quad \frac{1}{2} \mathbf{w}^\top \mathbf{w} + C \sum_{i=1}^m \zeta^{(i)} \\ & \text{subject to} \quad t^{(i)} (\mathbf{w}^\top \mathbf{x}^{(i)} + b) \geq 1 - \zeta^{(i)} \quad \text{and} \quad \zeta^{(i)} \geq 0 \quad \text{for } i = 1, 2, \dots, m \end{aligned}$$

The hard margin and soft margin problems are both convex quadratic optimization problems with linear constraints. Such problems are known as *Quadratic Programming* (QP) problems. Many off-the-shelf solvers are available to solve QP problems by using a variety of techniques that are outside the scope of this book.⁴

So using a QP solver is one way to train an SVM. Another is to use Gradient Descent to minimize the *Hinge loss* or the *squared Hinge loss* (see [Figure 5-13](#)). Given an instance \mathbf{x} of the positive class (i.e., with $t = 1$), the loss is zero if the output s of the decision function ($s = \mathbf{w}^\top \mathbf{x} + b$) is greater or equal to 1. This happens when the instance is off the street and on the positive side. Given an instance of the negative class (i.e., with $t = -1$), the loss is zero if $s \leq -1$. This happens when the instance is off the street and on the negative side. The further away an instance is from the correct side of the margin, the higher the loss: it grows linearly for the Hinge loss, and quadratically for the squared Hinge loss. This makes the squared Hinge loss more sensitive to outliers. However, if the dataset is clean, it tends to converge faster. By default, `LinearSVC` uses the squared Hinge loss, while `SGDClassifier` uses the Hinge loss. Both classes let you choose the loss by setting the `loss` hyperparameter to "hinge" or "squared_hinge". The `SVC` class's optimization algorithm finds a similar solution as minimizing the Hinge loss.

⁴ To learn more about Quadratic Programming, you can start by reading Stephen Boyd and Lieven Vandenberghe's book [Convex Optimization](#) (Cambridge University Press, 2004) or watch Richard Brown's [series of video lectures](#).

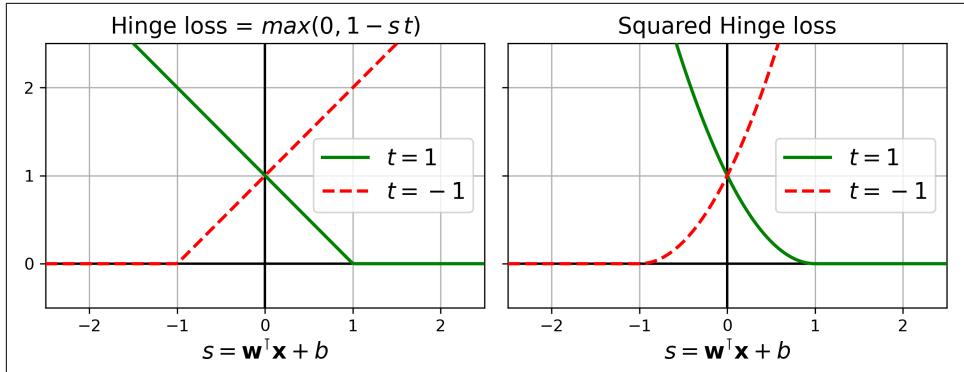


Figure 5-13. The Hinge loss (left) and the Squared Hinge loss (right)

There's yet another way to train a linear SVM classifier: solving the dual problem.

The Dual Problem

Given a constrained optimization problem, known as the *primal problem*, it is possible to express a different but closely related problem, called its *dual problem*. The solution to the dual problem typically gives a lower bound to the solution of the primal problem, but under some conditions it can have the same solution as the primal problem. Luckily, the SVM problem happens to meet these conditions,⁵ so you can choose to solve the primal problem or the dual problem; both will have the same solution. [Equation 5-3](#) shows the dual form of the linear SVM objective. If you are interested in knowing how to derive the dual problem from the primal problem, see the extra material section in the notebook.

[Equation 5-3. Dual form of the linear SVM objective](#)

$$\begin{aligned} \underset{\alpha}{\text{minimize}} \quad & \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)\top} \mathbf{x}^{(j)} - \sum_{i=1}^m \alpha^{(i)} \\ \text{subject to } & \alpha^{(i)} \geq 0 \text{ for all } i = 1, 2, \dots, m \text{ and } \sum_{i=1}^m \alpha^{(i)} t^{(i)} = 0 \end{aligned}$$

⁵ The objective function is convex, and the inequality constraints are continuously differentiable and convex functions.

Once you find the vector $\hat{\mathbf{a}}$ that minimizes this equation (using a QP solver), use [Equation 5-4](#) to compute $\hat{\mathbf{w}}$ and \hat{b} that minimize the primal problem. In this equation, n_s represents the number of support vectors.

Equation 5-4. From the dual solution to the primal solution

$$\begin{aligned}\hat{\mathbf{w}} &= \sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} \mathbf{x}^{(i)} \\ \hat{b} &= \frac{1}{n_s} \sum_{i=1}^m \left(t^{(i)} - \hat{\mathbf{w}}^\top \mathbf{x}^{(i)} \right) \\ \hat{\alpha}^{(i)} &> 0\end{aligned}$$

The dual problem is faster to solve than the primal one when the number of training instances is smaller than the number of features. More importantly, the dual problem makes the kernel trick possible, while the primal does not. So what is this kernel trick, anyway?

Kernelized SVMs

Suppose you want to apply a second-degree polynomial transformation to a two-dimensional training set (such as the moons training set), then train a linear SVM classifier on the transformed training set. [Equation 5-5](#) shows the second-degree polynomial mapping function ϕ that you want to apply.

Equation 5-5. Second-degree polynomial mapping

$$\varphi(\mathbf{x}) = \varphi\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix}$$

Notice that the transformed vector is 3D instead of 2D. Now let's look at what happens to a couple of 2D vectors, \mathbf{a} and \mathbf{b} , if we apply this second-degree polynomial

mapping and then compute the dot product⁶ of the transformed vectors (See [Equation 5-6](#)).

Equation 5-6. Kernel trick for a second-degree polynomial mapping

$$\begin{aligned}\varphi(\mathbf{a})^\top \varphi(\mathbf{b}) &= \begin{pmatrix} a_1^2 \\ \sqrt{2} a_1 a_2 \\ a_2^2 \end{pmatrix}^\top \begin{pmatrix} b_1^2 \\ \sqrt{2} b_1 b_2 \\ b_2^2 \end{pmatrix} = a_1^2 b_1^2 + 2a_1 b_1 a_2 b_2 + a_2^2 b_2^2 \\ &= (a_1 b_1 + a_2 b_2)^2 = \left(\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^\top \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \right)^2 = (\mathbf{a}^\top \mathbf{b})^2\end{aligned}$$

How about that? The dot product of the transformed vectors is equal to the square of the dot product of the original vectors: $\phi(\mathbf{a})^\top \phi(\mathbf{b}) = (\mathbf{a}^\top \mathbf{b})^2$.

Here is the key insight: if you apply the transformation ϕ to all training instances, then the dual problem (see [Equation 5-3](#)) will contain the dot product $\phi(\mathbf{x}^{(i)})^\top \phi(\mathbf{x}^{(j)})$. But if ϕ is the second-degree polynomial transformation defined in [Equation 5-5](#), then you can replace this dot product of transformed vectors simply by $(\mathbf{x}^{(i)^\top} \mathbf{x}^{(j)})^2$. So, you don't need to transform the training instances at all; just replace the dot product by its square in [Equation 5-3](#). The result will be strictly the same as if you had gone through the trouble of transforming the training set then fitting a linear SVM algorithm, but this trick makes the whole process much more computationally efficient.

The function $K(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^\top \mathbf{b})^2$ is a second-degree polynomial kernel. In Machine Learning, a *kernel* is a function capable of computing the dot product $\phi(\mathbf{a})^\top \phi(\mathbf{b})$, based only on the original vectors \mathbf{a} and \mathbf{b} , without having to compute (or even to

⁶ As explained in [Chapter 4](#), the dot product of two vectors \mathbf{a} and \mathbf{b} is normally noted $\mathbf{a} \cdot \mathbf{b}$. However, in Machine Learning, vectors are frequently represented as column vectors (i.e., single-column matrices), so the dot product is achieved by computing $\mathbf{a}^\top \mathbf{b}$. To remain consistent with the rest of the book, we will use this notation here, ignoring the fact that this technically results in a single-cell matrix rather than a scalar value.

know about) the transformation ϕ . [Equation 5-7](#) lists some of the most commonly used kernels.

Equation 5-7. Common kernels

- | | |
|---------------|---|
| Linear: | $K(\mathbf{a}, \mathbf{b}) = \mathbf{a}^T \mathbf{b}$ |
| Polynomial: | $K(\mathbf{a}, \mathbf{b}) = (\gamma \mathbf{a}^T \mathbf{b} + r)^d$ |
| Gaussian RBF: | $K(\mathbf{a}, \mathbf{b}) = \exp(-\gamma \ \mathbf{a} - \mathbf{b}\ ^2)$ |
| Sigmoid: | $K(\mathbf{a}, \mathbf{b}) = \tanh(\gamma \mathbf{a}^T \mathbf{b} + r)$ |

Mercer's Theorem

According to *Mercer's theorem*, if a function $K(\mathbf{a}, \mathbf{b})$ respects a few mathematical conditions called *Mercer's conditions* (e.g., K must be continuous and symmetric in its arguments so that $K(\mathbf{a}, \mathbf{b}) = K(\mathbf{b}, \mathbf{a})$, etc.), then there exists a function ϕ that maps \mathbf{a} and \mathbf{b} into another space (possibly with much higher dimensions) such that $K(\mathbf{a}, \mathbf{b}) = \phi(\mathbf{a})^T \phi(\mathbf{b})$. You can use K as a kernel because you know ϕ exists, even if you don't know what ϕ is. In the case of the Gaussian RBF kernel, it can be shown that ϕ maps each training instance to an infinite-dimensional space, so it's a good thing you don't need to actually perform the mapping!

Note that some frequently used kernels (such as the sigmoid kernel) don't respect all of Mercer's conditions, yet they generally work well in practice.

There is still one loose end we must tie up. [Equation 5-4](#) shows how to go from the dual solution to the primal solution in the case of a linear SVM classifier. But if you apply the kernel trick, you end up with equations that include $\phi(x^{(i)})$. In fact, $\widehat{\mathbf{w}}$ must have the same number of dimensions as $\phi(x^{(i)})$, which may be huge or even infinite, so you can't compute it. But how can you make predictions without knowing $\widehat{\mathbf{w}}$? Well, the good news is that you can plug the formula for $\widehat{\mathbf{w}}$ from [Equation 5-4](#) into the decision function for a new instance $\mathbf{x}^{(n)}$, and you get an equation with only

dot products between input vectors. This makes it possible to use the kernel trick (Equation 5-8).

Equation 5-8. Making predictions with a kernelized SVM

$$\begin{aligned}
 h_{\widehat{\mathbf{w}}, \widehat{b}}(\varphi(\mathbf{x}^{(n)})) &= \widehat{\mathbf{w}}^\top \varphi(\mathbf{x}^{(n)}) + \widehat{b} = \left(\sum_{i=1}^m \widehat{\alpha}^{(i)} t^{(i)} \varphi(\mathbf{x}^{(i)}) \right)^\top \varphi(\mathbf{x}^{(n)}) + \widehat{b} \\
 &= \sum_{i=1}^m \widehat{\alpha}^{(i)} t^{(i)} (\varphi(\mathbf{x}^{(i)})^\top \varphi(\mathbf{x}^{(n)})) + \widehat{b} \\
 &= \sum_{\substack{i=1 \\ \widehat{\alpha}^{(i)} > 0}}^m \widehat{\alpha}^{(i)} t^{(i)} K(\mathbf{x}^{(i)}, \mathbf{x}^{(n)}) + \widehat{b}
 \end{aligned}$$

Note that since $\alpha^{(i)} \neq 0$ only for support vectors, making predictions involves computing the dot product of the new input vector $\mathbf{x}^{(n)}$ with only the support vectors, not all the training instances. Of course, you need to use the same trick to compute the bias term \widehat{b} (Equation 5-9).

Equation 5-9. Using the kernel trick to compute the bias term

$$\begin{aligned}
 \widehat{b} &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \widehat{\alpha}^{(i)} > 0}}^m \left(t^{(i)} - \widehat{\mathbf{w}}^\top \varphi(\mathbf{x}^{(i)}) \right) = \frac{1}{n_s} \sum_{\substack{i=1 \\ \widehat{\alpha}^{(i)} > 0}}^m \left(t^{(i)} - \left(\sum_{j=1}^m \widehat{\alpha}^{(j)} t^{(j)} \varphi(\mathbf{x}^{(j)}) \right)^\top \varphi(\mathbf{x}^{(i)}) \right) \\
 &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \widehat{\alpha}^{(i)} > 0}}^m \left(t^{(i)} - \sum_{\substack{j=1 \\ \widehat{\alpha}^{(j)} > 0}}^m \widehat{\alpha}^{(j)} t^{(j)} K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) \right)
 \end{aligned}$$

If you are starting to get a headache, it's perfectly normal: it's an unfortunate side effect of the kernel trick.



It is also possible to implement online kernelized SVMs, capable of incremental learning, as described in the papers “[Incremental and Decremental Support Vector Machine Learning](#)”⁷ and “[Fast Kernel Classifiers with Online and Active Learning](#)”⁸. These kernelized SVMs are implemented in Matlab and C++. But for large-scale nonlinear problems, you may want to consider using Random Forests (see [Chapter 7](#)) or neural networks (see [Part II](#)).

Exercises

1. What is the fundamental idea behind Support Vector Machines?
2. What is a support vector?
3. Why is it important to scale the inputs when using SVMs?
4. Can an SVM classifier output a confidence score when it classifies an instance? What about a probability?
5. How can you choose between `LinearSVC`, `SVC`, or `SGDClassifier`?
6. Say you’ve trained an SVM classifier with an RBF kernel, but it seems to underfit the training set. Should you increase or decrease γ (`gamma`)? What about C ?
7. What does it mean for a model to be ϵ -insensitive?
8. What is the point of using the kernel trick?
9. Train a `LinearSVC` on a linearly separable dataset. Then train an `SVC` and a `SGDClassifier` on the same dataset. See if you can get them to produce roughly the same model.
10. Train an SVM classifier on the Wine dataset, which you can load using `sklearn.datasets.load_wine()`. This dataset contains the chemical analysis of 178 wine samples produced by 3 different cultivators: the goal is to train a classification model capable of predicting the cultivator based on the wine’s chemical analysis. Since SVM classifiers are binary classifiers, you will need to use one-versus-all to classify all 3 classes. What accuracy can you reach?
11. Train and fine-tune an SVM regressor on the California housing dataset. You can use the original dataset rather than the tweaked version we used in [Chapter 2](#). The original dataset can be fetched using `sklearn.datasets.fetch_california_housing()`. The targets represent hundreds of thousands of dollars. Since there are over 20,000 instances, SVMs can be slow, so for hyperparameter tuning

⁷ Gert Cauwenberghs and Tomaso Poggio, “[Incremental and Decremental Support Vector Machine Learning](#),” *Proceedings of the 13th International Conference on Neural Information Processing Systems* (2000): 388–394.

⁸ Antoine Bordes et al., “[Fast Kernel Classifiers with Online and Active Learning](#),” *Journal of Machine Learning Research* 6 (2005): 1579–1619.

you should use far fewer instances (e.g., 2,000), to test many more hyperparameter combinations. What is your best model's RMSE?

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab3>.

CHAPTER 6

Decision Trees

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. The GitHub repo is <https://github.com/ageron/handsonml3>.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

Decision Trees are versatile Machine Learning algorithms that can perform both classification and regression tasks, and even multioutput tasks. They are powerful algorithms, capable of fitting complex datasets. For example, in [Chapter 2](#) you trained a `DecisionTreeRegressor` model on the California housing dataset, fitting it perfectly (actually, overfitting it).

Decision Trees are also the fundamental components of Random Forests (see [Chapter 7](#)), which are among the most powerful Machine Learning algorithms available today.

In this chapter we will start by discussing how to train, visualize, and make predictions with Decision Trees. Then we will go through the CART training algorithm used by Scikit-Learn, and we will discuss how to regularize trees and use them for regression tasks. Finally, we will discuss some of the limitations of Decision Trees.

Training and Visualizing a Decision Tree

To understand Decision Trees, let's build one and take a look at how it makes predictions. The following code trains a `DecisionTreeClassifier` on the iris dataset (see [Chapter 4](#)):

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris(as_frame=True)
X_iris = iris.data[["petal length (cm)", "petal width (cm)"]].values
y_iris = iris.target

tree_clf = DecisionTreeClassifier(max_depth=2, random_state=42)
tree_clf.fit(X_iris, y_iris)
```

You can visualize the trained Decision Tree by first using the `export_graphviz()` method to output a graph definition file called `iris_tree.dot`:

```
from sklearn.tree import export_graphviz

export_graphviz(
    tree_clf,
    out_file="iris_tree.dot",
    feature_names=["petal length (cm)", "petal width (cm)"],
    class_names=iris.target_names,
    rounded=True,
    filled=True
)
```

Then you can use `graphviz.Source.from_file()` to load and display the file in a Jupyter notebook:

```
from graphviz import Source

Source.from_file("iris_tree.dot")
```

`Graphviz` is an open source graph visualization software package. It also includes a dot command-line tool to convert `.dot` files to a variety of formats, such as PDF or PNG.

Your first Decision Tree looks like [Figure 6-1](#).

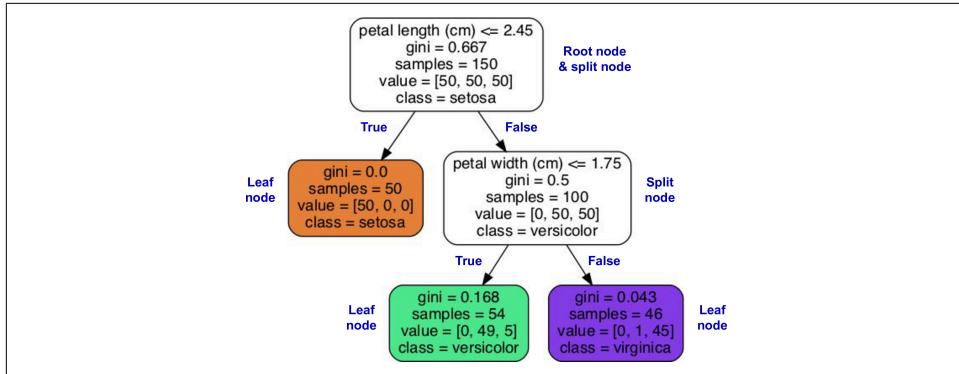


Figure 6-1. Iris Decision Tree

Making Predictions

Let's see how the tree represented in Figure 6-1 makes predictions. Suppose you find an iris plant and you want to classify it based on its petals. You start at the *root node* (depth 0, at the top): this node asks whether the flower's petal length is smaller than 2.45 cm. If it is, then you move down to the root's left child node (depth 1, left). In this case, it is a *leaf node* (i.e., it does not have any child nodes), so it does not ask any questions: simply look at the predicted class for that node, and the Decision Tree predicts that your flower is an *Iris setosa* (`class=setosa`).

Now suppose you find another flower, and this time the petal length is greater than 2.45 cm. You must move down to the root's right child node (depth 1, right), which is not a leaf node, it's a *split node*, so it asks another question: is the petal width smaller than 1.75 cm? If it is, then your flower is most likely an *Iris versicolor* (depth 2, left). If not, it is likely an *Iris virginica* (depth 2, right). It's really that simple.



One of the many qualities of Decision Trees is that they require very little data preparation. In fact, they don't require feature scaling or centering at all.

A node's `samples` attribute counts how many training instances it applies to. For example, 100 training instances have a petal length greater than 2.45 cm (depth 1, right), and of those 100, 54 have a petal width smaller than 1.75 cm (depth 2, left). A node's `value` attribute tells you how many training instances of each class this node applies to: for example, the bottom-right node applies to 0 *Iris setosa*, 1 *Iris versicolor*, and 45 *Iris virginica*. Finally, a node's `gini` attribute measures its *Gini impurity*: a node is "pure" ($\text{gini}=0$) if all training instances it applies to belong to the same class. For example, since the depth-1 left node applies only to *Iris setosa* training instances, it

is pure and its Gini impurity is 0. **Equation 6-1** shows how the training algorithm computes the Gini impurity G_i of the i^{th} node. The depth-2 left node has a Gini impurity equal to $1 - (0/54)^2 - (49/54)^2 - (5/54)^2 \approx 0.168$.

Equation 6-1. Gini impurity

$$G_i = 1 - \sum_{k=1}^n p_{i,k}^2$$

In this equation:

- G_i is the Gini impurity of the i^{th} node.
- $p_{i,k}$ is the ratio of class k instances among the training instances in the i^{th} node.



Scikit-Learn uses the CART algorithm, which produces only *binary trees*, meaning trees where split nodes always have exactly two children (i.e., questions only have yes/no answers). However, other algorithms such as ID3 can produce Decision Trees with nodes that have more than two children.

Figure 6-2 shows this Decision Tree's decision boundaries. The thick vertical line represents the decision boundary of the root node (depth 0): petal length = 2.45 cm. Since the lefthand area is pure (only *Iris setosa*), it cannot be split any further. However, the righthand area is impure, so the depth-1 right node splits it at petal width = 1.75 cm (represented by the dashed line). Since `max_depth` was set to 2, the Decision Tree stops right there. If you set `max_depth` to 3, then the two depth-2 nodes would each add another decision boundary (represented by the two vertical dotted lines).

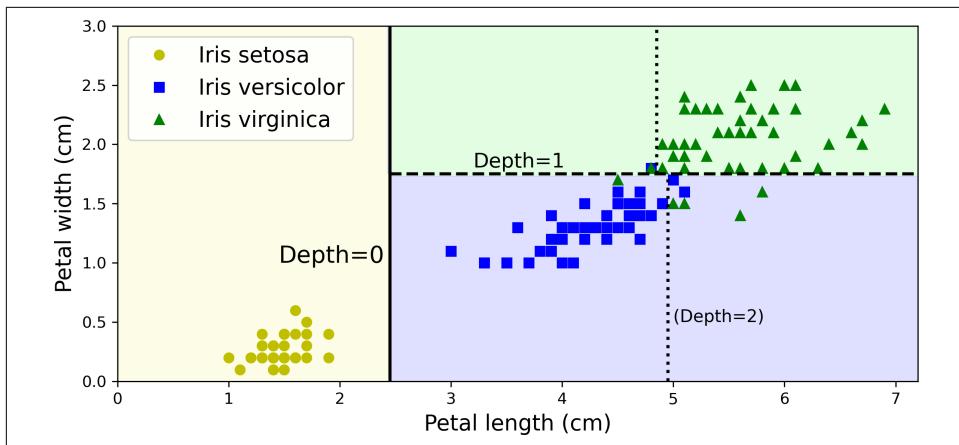


Figure 6-2. Decision Tree decision boundaries

Model Interpretation: White Box Versus Black Box

Decision Trees are intuitive, and their decisions are easy to interpret. Such models are often called *white box models*. In contrast, as we will see, Random Forests or neural networks are generally considered *black box models*. They make great predictions, and you can easily check the calculations that they performed to make these predictions; nevertheless, it is usually hard to explain in simple terms why the predictions were made. For example, if a neural network says that a particular person appears on a picture, it is hard to know what contributed to this prediction: did the model recognize that person's eyes? Their mouth? Their nose? Their shoes? Or even the couch that they were sitting on? Conversely, Decision Trees provide nice, simple classification rules that can even be applied manually if need be (e.g., for flower classification). The field of *Interpretable ML* aims at creating ML systems that can explain their decisions in a way humans can understand. This is important in many domains, for example to ensure the system does not make unfair decisions.



The tree structure, including all the information shown in Figure 6-1, is available via the classifier's `tree_` attribute. Type `help(tree_clf.tree_)` for details, and see the notebook for an example.

Estimating Class Probabilities

A Decision Tree can also estimate the probability that an instance belongs to a particular class k . First it traverses the tree to find the leaf node for this instance, and then it returns the ratio of training instances of class k in this node. For example,

suppose you have found a flower whose petals are 5 cm long and 1.5 cm wide. The corresponding leaf node is the depth-2 left node, so the Decision Tree outputs the following probabilities: 0% for *Iris setosa* (0/54), 90.7% for *Iris versicolor* (49/54), and 9.3% for *Iris virginica* (5/54). And if you ask it to predict the class, it outputs *Iris versicolor* (class 1) because it has the highest probability. Let's check this:

```
>>> tree_clf.predict_proba([[5, 1.5]]).round(3)
array([[0.    , 0.907, 0.093]])
>>> tree_clf.predict([[5, 1.5]])
array([1])
```

Perfect! Notice that the estimated probabilities would be identical anywhere else in the bottom-right rectangle of [Figure 6-2](#)—for example, if the petals were 6 cm long and 1.5 cm wide (even though it seems obvious that it would most likely be an *Iris virginica* in this case).

The CART Training Algorithm

Scikit-Learn uses the *Classification and Regression Tree* (CART) algorithm to train Decision Trees (also called “growing” trees). The algorithm works by first splitting the training set into two subsets using a single feature k and a threshold t_k (e.g., “petal length ≤ 2.45 cm”). How does it choose k and t_k ? It searches for the pair (k, t_k) that produces the purest subsets, weighted by their size. [Equation 6-2](#) gives the cost function that the algorithm tries to minimize.

Equation 6-2. CART cost function for classification

$$J(k, t_k) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}}$$

where $\begin{cases} G_{\text{left/right}} \text{ measures the impurity of the left/right subset,} \\ m_{\text{left/right}} \text{ is the number of instances in the left/right subset.} \end{cases}$

Once the CART algorithm has successfully split the training set in two, it splits the subsets using the same logic, then the sub-subsets, and so on, recursively. It stops recursing once it reaches the maximum depth (defined by the `max_depth` hyperparameter), or if it cannot find a split that will reduce impurity. A few other hyperparameters (described in a moment) control additional stopping conditions: `min_samples_split`, `min_samples_leaf`, `min_weight_fraction_leaf`, and `max_leaf_nodes`.



As you can see, the CART algorithm is a *greedy algorithm*: it greedily searches for an optimum split at the top level, then repeats the process at each subsequent level. It does not check whether or not the split will lead to the lowest possible impurity several levels down. A greedy algorithm often produces a solution that's reasonably good but not guaranteed to be optimal.

Unfortunately, finding the optimal tree is known to be an *NP-Complete* problem.¹ It requires $O(\exp(m))$ time, making the problem intractable even for small training sets. This is why we must settle for a “reasonably good” solution when training decision trees.

Computational Complexity

Making predictions requires traversing the Decision Tree from the root to a leaf. Decision Trees generally are approximately balanced, so traversing the Decision Tree requires going through roughly $O(\log_2(m))$ nodes, where $\log_2(m)$ is the *binary logarithm* of m , equal to $\log(m) / \log(2)$. Since each node only requires checking the value of one feature, the overall prediction complexity is $O(\log_2(m))$, independent of the number of features. So predictions are very fast, even when dealing with large training sets.

The training algorithm compares all features (or less if `max_features` is set) on all samples at each node. Comparing all features on all samples at each node results in a training complexity of $O(n \times m \log_2(m))$.

Gini Impurity or Entropy?

By default, the `DecisionTreeClassifier` class uses the Gini impurity measure, but you can select the `entropy` impurity measure instead by setting the `criterion` hyper-parameter to "entropy". The concept of entropy originated in thermodynamics as a measure of molecular disorder: entropy approaches zero when molecules are still and well ordered. Entropy later spread to a wide variety of domains, including in Shannon’s information theory, where it measures the average information content of a message, as we saw in [Chapter 4](#). Entropy is zero when all messages are identical. In Machine Learning, entropy is frequently used as an impurity measure: a set’s entropy is zero when it contains instances of only one class. [Equation 6-3](#) shows the definition

¹ P is the set of problems that can be solved in *polynomial time* (i.e., a polynomial of the dataset size). NP is the set of problems whose solutions can be verified in polynomial time. An NP-Hard problem is a problem that can be reduced to a known NP-Hard problem in polynomial time. An NP-Complete problem is both NP and NP-Hard. A major open mathematical question is whether or not P = NP. If P ≠ NP (which seems likely), then no polynomial algorithm will ever be found for any NP-Complete problem (except perhaps one day on a quantum computer).

of the entropy of the i^{th} node. For example, the depth-2 left node in [Figure 6-1](#) has an entropy equal to $-(49/54) \log_2 (49/54) - (5/54) \log_2 (5/54) \approx 0.445$.

Equation 6-3. Entropy

$$H_i = - \sum_{\substack{k=1 \\ p_{i,k} \neq 0}}^n p_{i,k} \log_2 (p_{i,k})$$

So, should you use Gini impurity or entropy? The truth is, most of the time it does not make a big difference: they lead to similar trees. Gini impurity is slightly faster to compute, so it is a good default. However, when they differ, Gini impurity tends to isolate the most frequent class in its own branch of the tree, while entropy tends to produce slightly more balanced trees.²

Regularization Hyperparameters

Decision Trees make very few assumptions about the training data (as opposed to linear models, which assume that the data is linear, for example). If left unconstrained, the tree structure will adapt itself to the training data, fitting it very closely—indeed, most likely overfitting it. Such a model is often called a *nonparametric model*, not because it does not have any parameters (it often has a lot) but because the number of parameters is not determined prior to training, so the model structure is free to stick closely to the data. In contrast, a *parametric model*, such as a linear model, has a predetermined number of parameters, so its degree of freedom is limited, reducing the risk of overfitting (but increasing the risk of underfitting).

To avoid overfitting the training data, you need to restrict the Decision Tree’s freedom during training. As you know by now, this is called regularization. The regularization hyperparameters depend on the algorithm used, but generally you can at least restrict the maximum depth of the Decision Tree. In Scikit-Learn, this is controlled by the `max_depth` hyperparameter. The default value is `None`, which means unlimited. Reducing `max_depth` will regularize the model and thus reduce the risk of overfitting.

The `DecisionTreeClassifier` class has a few other parameters that similarly restrict the shape of the Decision Tree:

- `max_features`: maximum number of features that are evaluated for splitting at each node
- `max_leaf_nodes`: maximum number of leaf nodes

² See Sebastian Raschka’s [interesting analysis](#) for more details.

- `min_samples_split`: minimum number of samples a node must have before it can be split
- `min_samples_leaf`: minimum number of samples a leaf node must have to be created
- `min_weight_fraction_leaf`: same as `min_samples_leaf` but expressed as a fraction of the total number of weighted instances

Increasing `min_*` hyperparameters or reducing `max_*` hyperparameters will regularize the model.



Other algorithms work by first training the Decision Tree without restrictions, then *pruning* (deleting) unnecessary nodes. A node whose children are all leaf nodes is considered unnecessary if the purity improvement it provides is not statistically significant. Standard statistical tests, such as the χ^2 test (chi-squared test), are used to estimate the probability that the improvement is purely the result of chance (which is called the *null hypothesis*). If this probability, called the *p-value*, is higher than a given threshold (typically 5%, controlled by a hyperparameter), then the node is considered unnecessary and its children are deleted. The pruning continues until all unnecessary nodes have been pruned.

Let's test regularization on the *moons dataset*, introduced in [Chapter 5](#). We'll train one decision tree without regularization, and another with `min_samples_leaf=5`. [Figure 6-3](#) shows the decision boundaries of each tree.

```
from sklearn.datasets import make_moons

X_moons, y_moons = make_moons(n_samples=150, noise=0.2, random_state=42)

tree_clf1 = DecisionTreeClassifier(random_state=42)
tree_clf2 = DecisionTreeClassifier(min_samples_leaf=5, random_state=42)
tree_clf1.fit(X_moons, y_moons)
tree_clf2.fit(X_moons, y_moons)
```

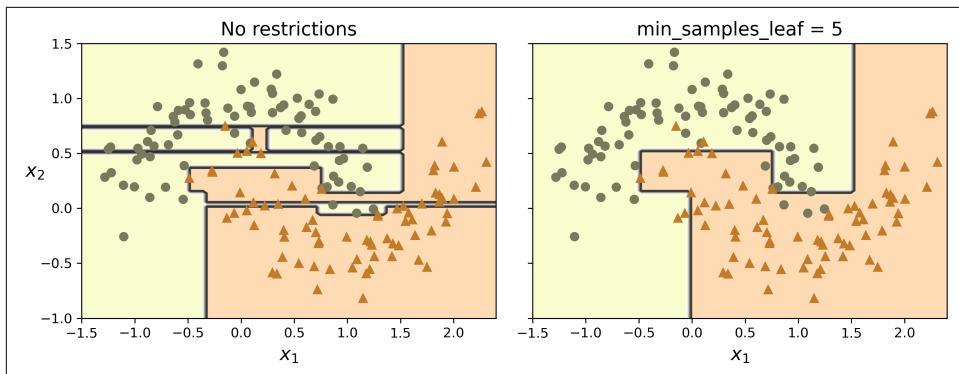


Figure 6-3. Decision boundaries of an unregularized tree (left) and a regularized tree (right)

The unregularized model on the left is clearly overfitting, and the regularized model on the right will probably generalize better. We can verify this by evaluating both trees on a test set generated using a different random seed:

```
>>> X_moons_test, y_moons_test = make_moons(n_samples=1000, noise=0.2,
...                                              random_state=43)
...
>>> tree_clf1.score(X_moons_test, y_moons_test)
0.898
>>> tree_clf2.score(X_moons_test, y_moons_test)
0.92
```

Indeed, the second tree has a better accuracy on the test set.

Regression

Decision Trees are also capable of performing regression tasks. Let's build a regression tree using Scikit-Learn's `DecisionTreeRegressor` class, training it on a noisy quadratic dataset with `max_depth=2`:

```
import numpy as np
from sklearn.tree import DecisionTreeRegressor

np.random.seed(42)
X_quad = np.random.rand(200, 1) - 0.5 # a single random input feature
y_quad = X_quad ** 2 + 0.025 * np.random.randn(200, 1)

tree_reg = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg.fit(X_quad, y_quad)
```

The resulting tree is represented in [Figure 6-4](#).

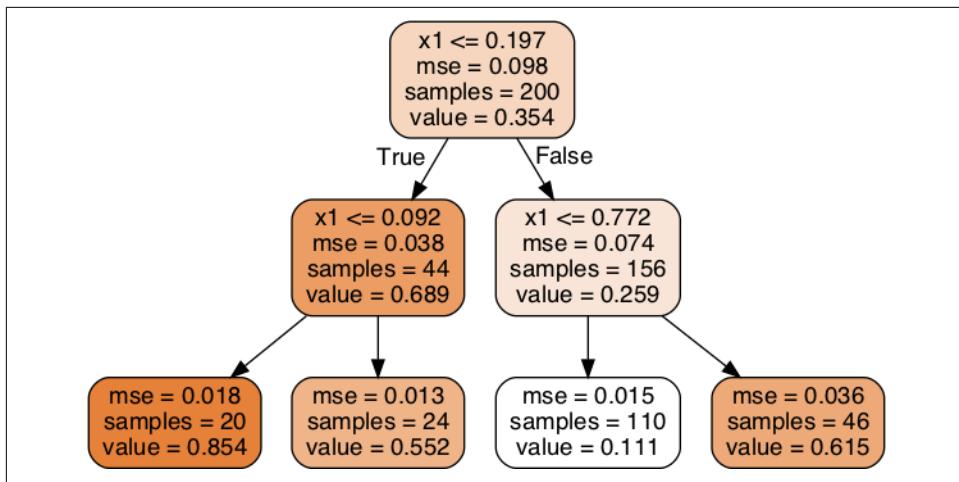


Figure 6-4. A Decision Tree for regression

This tree looks very similar to the classification tree you built earlier. The main difference is that instead of predicting a class in each node, it predicts a value. For example, suppose you want to make a prediction for a new instance with $x_1 = 0.2$. The root node asks whether $x_1 \leq 0.197$. Since it is not, the algorithm goes to the right child node, which asks whether $x_1 \leq 0.772$. Since it is, the algorithm goes to the left child node. This is a leaf node, and it predicts $\text{value}=0.111$. This prediction is the average target value of the 110 training instances associated with this leaf node, and it results in a mean squared error equal to 0.015 over these 110 instances.

This model's predictions are represented on the left in Figure 6-5. If you set `max_depth=3`, you get the predictions represented on the right. Notice how the predicted value for each region is always the average target value of the instances in that region. The algorithm splits each region in a way that makes most training instances as close as possible to that predicted value.

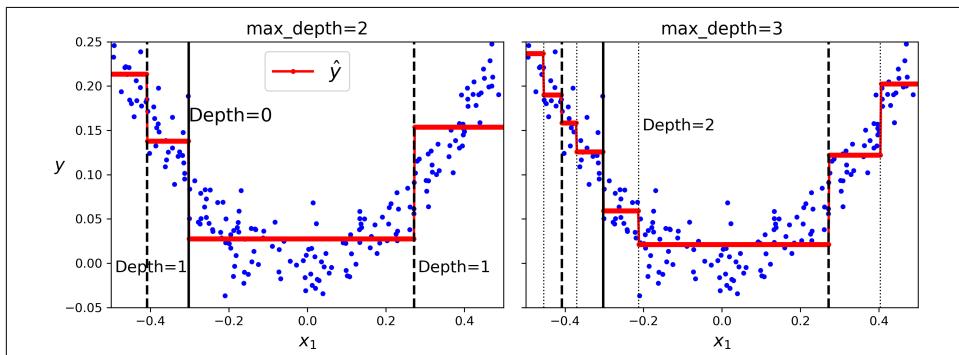


Figure 6-5. Predictions of two Decision Tree regression models

The CART algorithm works mostly the same way as earlier, except that instead of trying to split the training set in a way that minimizes impurity, it now tries to split the training set in a way that minimizes the MSE. [Equation 6-4](#) shows the cost function that the algorithm tries to minimize.

Equation 6-4. CART cost function for regression

$$J(k, t_k) = \frac{m_{\text{left}}}{m} \text{MSE}_{\text{left}} + \frac{m_{\text{right}}}{m} \text{MSE}_{\text{right}} \quad \text{where} \quad \begin{cases} \text{MSE}_{\text{node}} = \frac{\sum_{i \in \text{node}} (\hat{y}_{\text{node}} - y^{(i)})^2}{m_{\text{node}}} \\ \hat{y}_{\text{node}} = \frac{\sum_{i \in \text{node}} y^{(i)}}{m_{\text{node}}} \end{cases}$$

Just like for classification tasks, Decision Trees are prone to overfitting when dealing with regression tasks. Without any regularization (i.e., using the default hyperparameters), you get the predictions on the left in [Figure 6-6](#). These predictions are obviously overfitting the training set very badly. Just setting `min_samples_leaf=10` results in a much more reasonable model, represented on the right in [Figure 6-6](#).

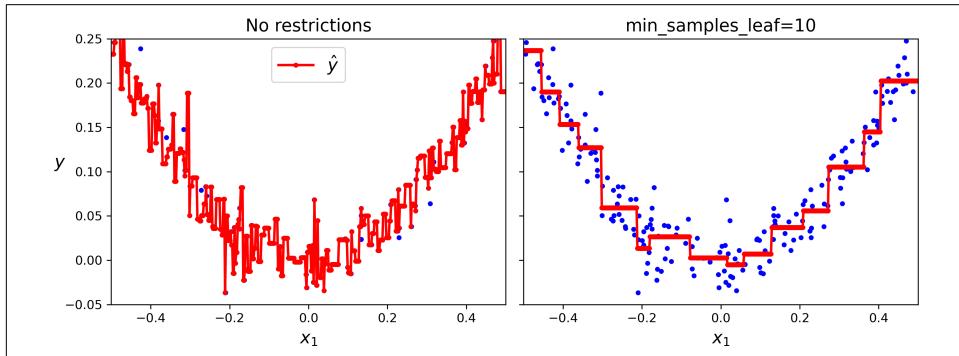


Figure 6-6. Predictions of an unregularized regression tree (left) and a regularized tree (right)

Sensitivity to axis orientation

Hopefully by now you are convinced that Decision Trees have a lot going for them: they are relatively easy to understand and interpret, simple to use, versatile, and powerful. However, they do have a few limitations. First, as you may have noticed, Decision Trees love orthogonal decision boundaries (all splits are perpendicular to an axis), which makes them sensitive to the data's orientation. For example, [Figure 6-7](#) shows a simple linearly separable dataset: on the left, a Decision Tree can split it easily, while on the right, after the dataset is rotated by 45°, the decision boundary

looks unnecessarily convoluted. Although both Decision Trees fit the training set perfectly, it is very likely that the model on the right will not generalize well.

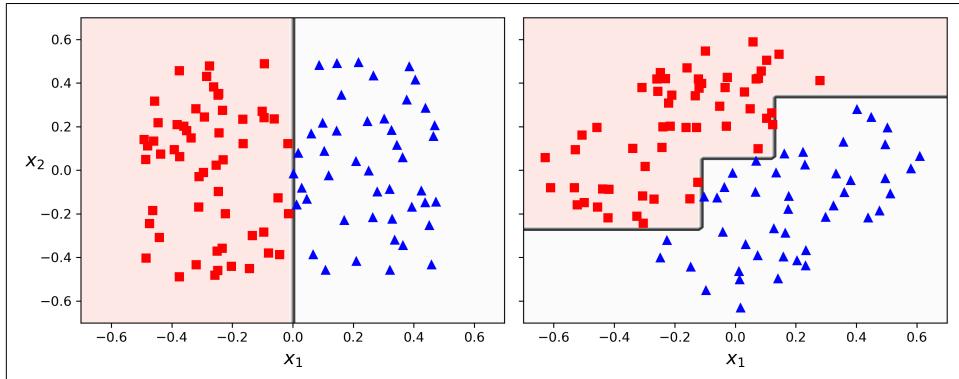


Figure 6-7. Sensitivity to training set rotation

One way to limit this problem is to scale the data then apply a Principal Component Analysis transformation. We will look at PCA in detail in [Chapter 8](#), but for now you only need to know that it rotates the data in a way that reduces the correlation between the features, which often (not always) makes things easier for trees. Let's create a small pipeline that scales the data and rotates it using PCA, then let's train a `DecisionTreeClassifier` on that data. [Figure 6-8](#) shows the decision boundaries of that tree: as you can see, the rotation makes it possible to fit the dataset pretty well using only one feature, z_1 , which is a linear function of the original petal length and width.

```
from sklearn.decomposition import PCA
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

pca_pipeline = make_pipeline(StandardScaler(), PCA())
X_iris_rotated = pca_pipeline.fit_transform(X_iris)
tree_clf_pca = DecisionTreeClassifier(max_depth=2, random_state=42)
tree_clf_pca.fit(X_iris_rotated, y_iris)
```

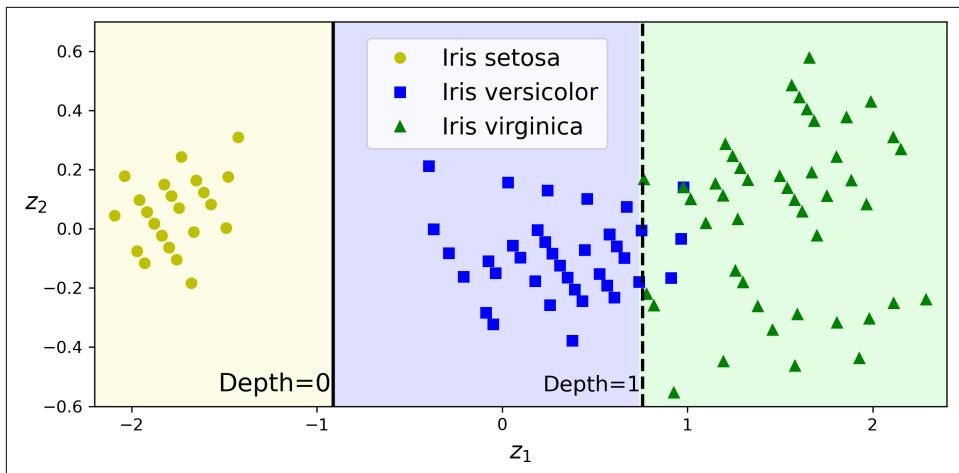


Figure 6-8. A tree's decision boundaries on the scaled and PCA-rotated iris dataset

Decision Trees have a high variance

More generally, the main issue with Decision Trees is that they have quite a high variance: small changes to the hyperparameters or to the data may produce very different models. In fact, since the training algorithm used by Scikit-Learn is stochastic—it randomly selects the set of features to evaluate at each node—even retraining the same Decision Tree on the exact same data may produce a very different model, such as the one represented in [Figure 6-9](#) (unless you set the `random_state` hyperparameter). As you can see, it looks very different from the previous Decision Tree ([Figure 6-2](#)).

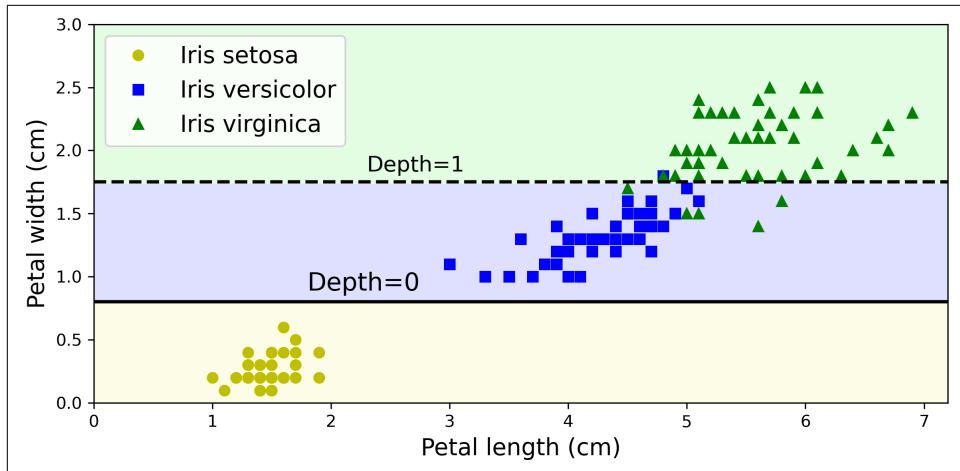


Figure 6-9. Retraining the same model on the same data may produce a very different model

Luckily, by averaging predictions over many trees, it's possible to reduce variance significantly. Such an *ensemble* of trees is called a *Random Forest*, and it's one of the most powerful types of models available today, as we will see in the next chapter.

Exercises

1. What is the approximate depth of a Decision Tree trained (without restrictions) on a training set with one million instances?
2. Is a node's Gini impurity generally lower or greater than its parent's? Is it *generally* lower/greater, or *always* lower/greater?
3. If a Decision Tree is overfitting the training set, is it a good idea to try decreasing `max_depth`?
4. If a Decision Tree is underfitting the training set, is it a good idea to try scaling the input features?
5. If it takes one hour to train a Decision Tree on a training set containing 1 million instances, roughly how much time will it take to train another Decision Tree on a training set containing 10 million instances? Hint: consider the CART algorithm's computational complexity.
6. If it takes one hour to train a Decision Tree on a given training set, roughly how much time will it take if you double the number of features?
7. Train and fine-tune a Decision Tree for the moons dataset by following these steps:

- a. Use `make_moons(n_samples=10000, noise=0.4)` to generate a moons dataset.
 - b. Use `train_test_split()` to split the dataset into a training set and a test set.
 - c. Use grid search with cross-validation (with the help of the `GridSearchCV` class) to find good hyperparameter values for a `DecisionTreeClassifier`. Hint: try various values for `max_leaf_nodes`.
 - d. Train it on the full training set using these hyperparameters, and measure your model's performance on the test set. You should get roughly 85% to 87% accuracy.
8. Grow a forest by following these steps:
- a. Continuing the previous exercise, generate 1,000 subsets of the training set, each containing 100 instances selected randomly. Hint: you can use Scikit-Learn's `ShuffleSplit` class for this.
 - b. Train one Decision Tree on each subset, using the best hyperparameter values found in the previous exercise. Evaluate these 1,000 Decision Trees on the test set. Since they were trained on smaller sets, these Decision Trees will likely perform worse than the first Decision Tree, achieving only about 80% accuracy.
 - c. Now comes the magic. For each test set instance, generate the predictions of the 1,000 Decision Trees, and keep only the most frequent prediction (you can use SciPy's `mode()` function for this). This approach gives you *majority-vote predictions* over the test set.
 - d. Evaluate these predictions on the test set: you should obtain a slightly higher accuracy than your first model (about 0.5 to 1.5% higher). Congratulations, you have trained a Random Forest classifier!

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab3>.

Ensemble Learning and Random Forests

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 7th chapter of the final book. The GitHub repo is <https://github.com/ageron/handsonml3>.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

Suppose you pose a complex question to thousands of random people, then aggregate their answers. In many cases you will find that this aggregated answer is better than an expert’s answer. This is called the *wisdom of the crowd*. Similarly, if you aggregate the predictions of a group of predictors (such as classifiers or regressors), you will often get better predictions than with the best individual predictor. A group of predictors is called an *ensemble*; thus, this technique is called *Ensemble Learning*, and an Ensemble Learning algorithm is called an *Ensemble method*.

As an example of an Ensemble method, you can train a group of Decision Tree classifiers, each on a different random subset of the training set. You can then obtain the predictions of all the individual trees, and the class that gets the most votes is the ensemble’s prediction (see the last exercise in [Chapter 6](#)). Such an ensemble of Decision Trees is called a *Random Forest*, and despite its simplicity, this is one of the most powerful Machine Learning algorithms available today.

As discussed in [Chapter 2](#), you will often use Ensemble methods near the end of a project, once you have already built a few good predictors, to combine them

into an even better predictor. In fact, the winning solutions in Machine Learning competitions often involve several Ensemble methods—most famously in the [Netflix Prize competition](#).

In this chapter we will discuss the most popular Ensemble methods, including voting classifiers, *bagging*, *pasting*, Random Forests, *boosting*, and *stacking*.

Voting Classifiers

Suppose you have trained a few classifiers, each one achieving about 80% accuracy. You may have a Logistic Regression classifier, an SVM classifier, a Random Forest classifier, a K-Nearest Neighbors classifier, and perhaps a few more (see [Figure 7-1](#)).

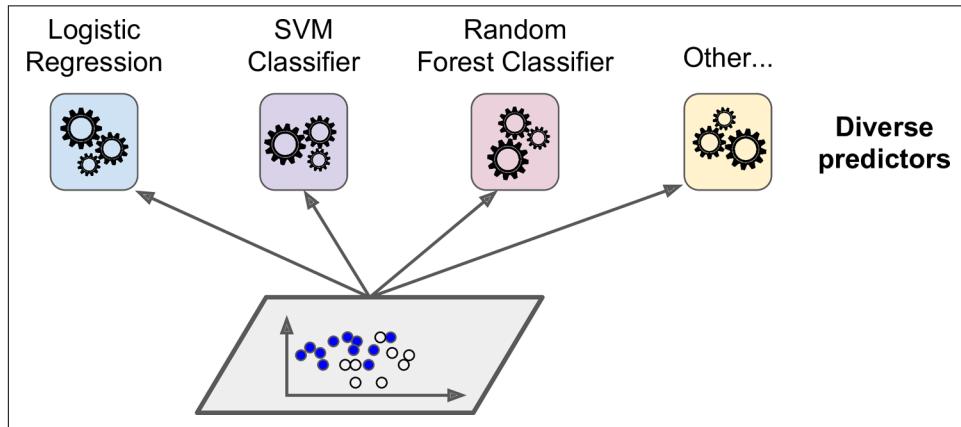


Figure 7-1. Training diverse classifiers

A very simple way to create an even better classifier is to aggregate the predictions of each classifier: the class that gets the most votes is the ensemble's prediction. This majority-vote classifier is called a *hard voting* classifier (see [Figure 7-2](#)).

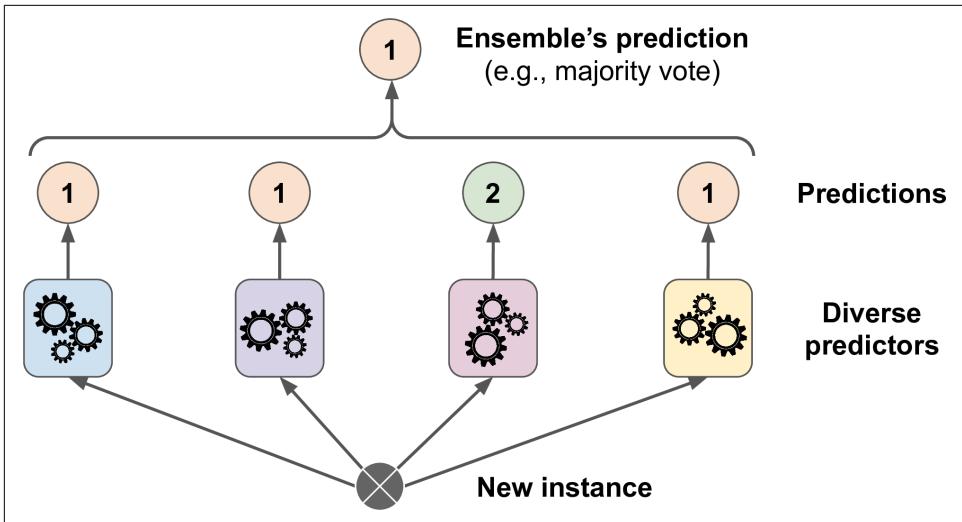


Figure 7-2. Hard voting classifier predictions

Somewhat surprisingly, this voting classifier often achieves a higher accuracy than the best classifier in the ensemble. In fact, even if each classifier is a *weak learner* (meaning it does only slightly better than random guessing), the ensemble can still be a *strong learner* (achieving high accuracy), provided there are a sufficient number of weak learners in the ensemble, and they are sufficiently diverse.

How is this possible? The following analogy can help shed some light on this mystery. Suppose you have a slightly biased coin that has a 51% chance of coming up heads and 49% chance of coming up tails. If you toss it 1,000 times, you will generally get more or less 510 heads and 490 tails, and hence a majority of heads. If you do the math, you will find that the probability of obtaining a majority of heads after 1,000 tosses is close to 75%. The more you toss the coin, the higher the probability (e.g., with 10,000 tosses, the probability climbs over 97%). This is due to the *law of large numbers*: as you keep tossing the coin, the ratio of heads gets closer and closer to the probability of heads (51%). Figure 7-3 shows 10 series of biased coin tosses. You can see that as the number of tosses increases, the ratio of heads approaches 51%. Eventually all 10 series end up so close to 51% that they are consistently above 50%.

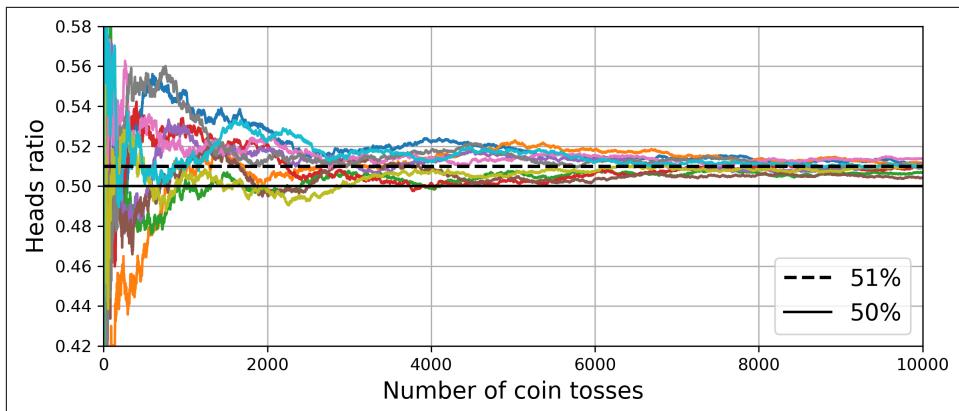


Figure 7-3. The law of large numbers

Similarly, suppose you build an ensemble containing 1,000 classifiers that are individually correct only 51% of the time (barely better than random guessing). If you predict the majority voted class, you can hope for up to 75% accuracy! However, this is only true if all classifiers are perfectly independent, making uncorrelated errors, which is clearly not the case because they are trained on the same data. They are likely to make the same types of errors, so there will be many majority votes for the wrong class, reducing the ensemble's accuracy.



Ensemble methods work best when the predictors are as independent from one another as possible. One way to get diverse classifiers is to train them using very different algorithms. This increases the chance that they will make very different types of errors, improving the ensemble's accuracy.

Scikit-Learn provides a `VotingClassifier` class that's quite easy to use: just give it a list of name/predictor pairs, and use it like a normal classifier, that's it! Let's try it on the moons dataset (introduced in [Chapter 6](#)): we will load and split the moons dataset into a training set and a test set, then we'll create and train a voting classifier composed of three diverse classifiers:

```
from sklearn.datasets import make_moons
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC

X, y = make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

voting_clf = VotingClassifier(
    estimators=[
```

```

        ('lr', LogisticRegression(random_state=42)),
        ('rf', RandomForestClassifier(random_state=42)),
        ('svc', SVC(random_state=42))
    ]
)
voting_clf.fit(X_train, y_train)

```

When you fit a `VotingClassifier`, it clones every estimator and fits the clones. The original estimators are available via the `estimators` attribute, while the fitted clones are available via the `estimators_` attribute. If you prefer a dict rather than a list, you can use `named_estimators` or `named_estimators_` instead. For example, let's look at each fitted classifier's accuracy on the test set:

```

>>> for name, clf in voting_clf.named_estimators_.items():
...     print(name, "=", clf.score(X_test, y_test))
...
lr = 0.864
rf = 0.896
svc = 0.896

```

When you call the voting classifier's `predict()` method, it performs hard voting. For example, the voting classifier predicts class 1 for the first instance of the test set, because 2 out of 3 classifiers predict that class:

```

>>> voting_clf.predict(X_test[:1])
array([1])
>>> [clf.predict(X_test[:1]) for clf in voting_clf.estimators_]
[array([1]), array([1]), array([0])]

```

Now let's look at the performance of the voting classifier on the test set:

```

>>> voting_clf.score(X_test, y_test)
0.912

```

There you have it! The voting classifier outperforms all the individual classifiers.

If all classifiers are able to estimate class probabilities (i.e., they all have a `predict_proba()` method), then you can tell Scikit-Learn to predict the class with the highest class probability, averaged over all the individual classifiers. This is called *soft voting*. It often achieves higher performance than hard voting because it gives more weight to highly confident votes. All you need to do is set the voting classifier's `voting` hyperparameter to "soft", and ensure that all classifiers can estimate class probabilities. This is not the case for the `SVC` class by default, so you need to set its `probability` hyperparameter to `True` (this will make the `SVC` class use cross-validation to estimate class probabilities, slowing down training, and it will add a `predict_proba()` method). Let's try that:

```

>>> voting_clf.voting = "soft"
>>> voting_clf.named_estimators["svc"].probability = True
>>> voting_clf.fit(X_train, y_train)

```

```
>>> voting_clf.score(X_test, y_test)  
0.92
```

We reach 92% accuracy simply by using soft voting, not bad!

Bagging and Pasting

One way to get a diverse set of classifiers is to use very different training algorithms, as just discussed. Another approach is to use the same training algorithm for every predictor but train them on different random subsets of the training set. When sampling is performed *with replacement*,¹ this method is called *bagging*² (short for *bootstrap aggregating*³). When sampling is performed *without replacement*, it is called *pasting*.⁴

In other words, both bagging and pasting allow training instances to be sampled several times across multiple predictors, but only bagging allows training instances to be sampled several times for the same predictor. This sampling and training process is represented in [Figure 7-4](#).

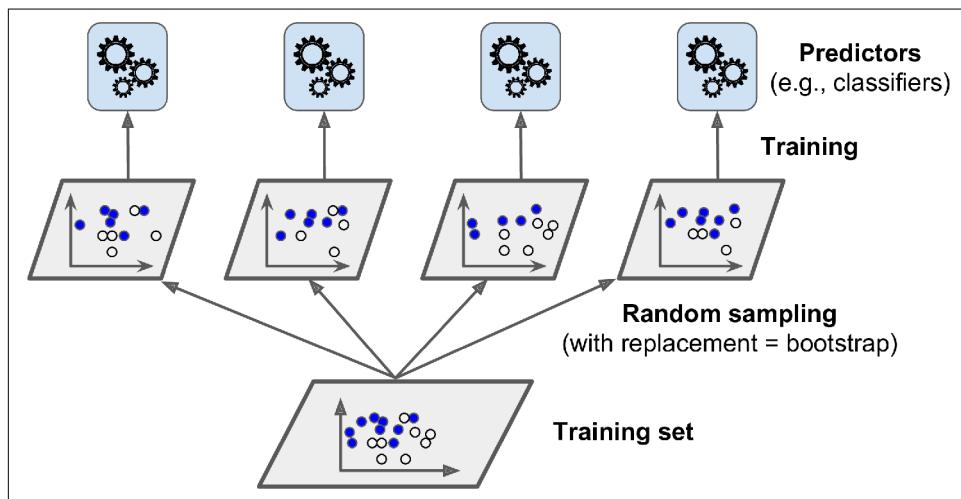


Figure 7-4. Bagging and pasting involve training several predictors on different random samples of the training set

¹ Imagine picking a card randomly from a deck of cards, writing it down, then placing it back in the deck before picking the next card: the same card could be sampled multiple times.

² Leo Breiman, “Bagging Predictors,” *Machine Learning* 24, no. 2 (1996): 123–140.

³ In statistics, resampling with replacement is called *bootstrapping*.

⁴ Leo Breiman, “Pasting Small Votes for Classification in Large Databases and On-Line,” *Machine Learning* 36, no. 1–2 (1999): 85–103.

Once all predictors are trained, the ensemble can make a prediction for a new instance by simply aggregating the predictions of all predictors. The aggregation function is typically the *statistical mode* for classification (i.e., the most frequent prediction, just like a hard voting classifier), or the average for regression. Each individual predictor has a higher bias than if it were trained on the original training set, but aggregation reduces both bias and variance.⁵ Generally, the net result is that the ensemble has a similar bias but a lower variance than a single predictor trained on the original training set.

As you can see in [Figure 7-4](#), predictors can all be trained in parallel, via different CPU cores or even different servers. Similarly, predictions can be made in parallel. This is one of the reasons bagging and pasting are such popular methods: they scale very well.

Bagging and Pasting in Scikit-Learn

Scikit-Learn offers a simple API for both bagging and pasting with the `BaggingClassifier` class (or `BaggingRegressor` for regression). The following code trains an ensemble of 500 Decision Tree classifiers:⁶ each is trained on 100 training instances randomly sampled from the training set with replacement (this is an example of bagging, but if you want to use pasting instead, just set `bootstrap=False`). The `n_jobs` parameter tells Scikit-Learn the number of CPU cores to use for training and predictions, and `-1` tells Scikit-Learn to use all available cores.

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=500,
                           max_samples=100, n_jobs=-1, random_state=42)
bag_clf.fit(X_train, y_train)
```



The `BaggingClassifier` automatically performs soft voting instead of hard voting if the base classifier can estimate class probabilities (i.e., if it has a `predict_proba()` method), which is the case with Decision Tree classifiers.

[Figure 7-5](#) compares the decision boundary of a single Decision Tree with the decision boundary of a bagging ensemble of 500 trees (from the preceding code), both trained on the moons dataset. As you can see, the ensemble's predictions will likely generalize much better than the single Decision Tree's predictions: the ensemble has a

⁵ Bias and variance were introduced in [Chapter 4](#).

⁶ `max_samples` can alternatively be set to a float between 0.0 and 1.0, in which case the max number of sampled instances is equal to the size of the training set times `max_samples`.

comparable bias but a smaller variance (it makes roughly the same number of errors on the training set, but the decision boundary is less irregular).

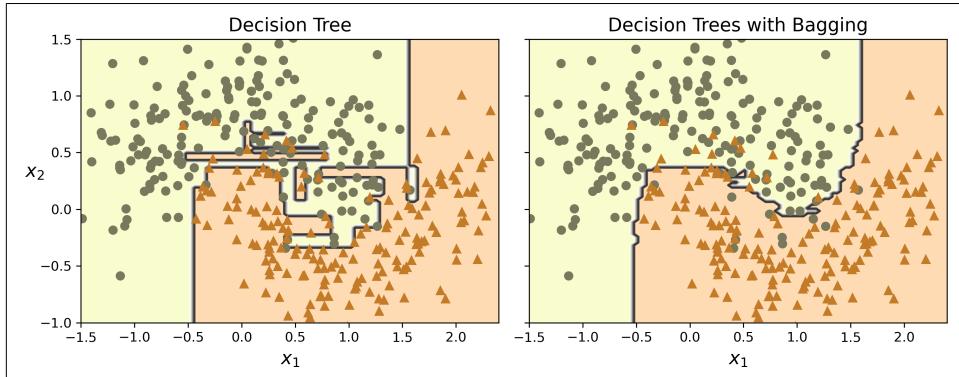


Figure 7-5. A single Decision Tree (left) versus a bagging ensemble of 500 trees (right)

Bagging introduces a bit more diversity in the subsets that each predictor is trained on, so bagging ends up with a slightly higher bias than pasting; but the extra diversity also means that the predictors end up being less correlated, so the ensemble's variance is reduced. Overall, bagging often results in better models, which explains why it is generally preferred. However, if you have spare time and CPU power, you can use cross-validation to evaluate both bagging and pasting and select the one that works best.

Out-of-Bag Evaluation

With bagging, some training instances may be sampled several times for any given predictor, while others may not be sampled at all. By default a `BaggingClassifier` samples m training instances with replacement (`bootstrap=True`), where m is the size of the training set. With this process, it can be shown mathematically that only about 63% of the training instances are sampled on average for each predictor.⁷ The remaining 37% of the training instances that are not sampled are called *out-of-bag* (oob) instances. Note that they are not the same 37% for all predictors.

A bagging ensemble can be evaluated using oob instances, without the need for a separate validation set: indeed, if there are enough estimators, then each instance in the training set will likely be an oob instance of several estimators, so these estimators can be used to make a fair ensemble prediction for that instance. Once you have a prediction for each instance, you can compute the ensemble's prediction accuracy (or any other metric).

⁷ As m grows, this ratio approaches $1 - \exp(-1) \approx 63\%$.

In Scikit-Learn, you can set `oob_score=True` when creating a `BaggingClassifier` to request an automatic oob evaluation after training. The following code demonstrates this. The resulting evaluation score is available in the `oob_score_` attribute:

```
>>> bag_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=500,
...                                oob_score=True, n_jobs=-1, random_state=42)
...
>>> bag_clf.fit(X_train, y_train)
>>> bag_clf.oob_score_
0.896
```

According to this oob evaluation, this `BaggingClassifier` is likely to achieve about 89.6% accuracy on the test set. Let's verify this:

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = bag_clf.predict(X_test)
>>> accuracy_score(y_test, y_pred)
0.92
```

We get 92% accuracy on the test. The oob evaluation was a bit too pessimistic, a bit over 2% too low.

The oob decision function for each training instance is also available through the `oob_decision_function_` attribute. Since the base estimator has a `predict_proba()` method, the decision function returns the class probabilities for each training instance. For example, the oob evaluation estimates that the first training instance has a 67.6% probability of belonging to the positive class, and 32.4% of belonging to the negative class:

```
>>> bag_clf.oob_decision_function_[:3] # probas for the first 3 instances
array([[0.32352941, 0.67647059],
       [0.3375    , 0.6625    ],
       [1.        , 0.        ]])
```

Random Patches and Random Subspaces

The `BaggingClassifier` class supports sampling the features as well. Sampling is controlled by two hyperparameters: `max_features` and `bootstrap_features`. They work the same way as `max_samples` and `bootstrap`, but for feature sampling instead of instance sampling. Thus, each predictor will be trained on a random subset of the input features.

This technique is particularly useful when you are dealing with high-dimensional inputs (such as images), as it can considerably speed-up training. Sampling both training instances and features is called the *Random Patches* method.⁸ Keeping all

⁸ Gilles Louppe and Pierre Geurts, “Ensembles on Random Patches,” *Lecture Notes in Computer Science* 7523 (2012): 346–361.

training instances (by setting `bootstrap=False` and `max_samples=1.0`) but sampling features (by setting `bootstrap_features` to `True` and/or `max_features` to a value smaller than `1.0`) is called the *Random Subspaces* method.⁹

Sampling features results in even more predictor diversity, trading a bit more bias for a lower variance.

Random Forests

As we have discussed, a `Random Forest`¹⁰ is an ensemble of Decision Trees, generally trained via the bagging method (or sometimes pasting), typically with `max_samples` set to the size of the training set. Instead of building a `BaggingClassifier` and passing it a `DecisionTreeClassifier`, you can use the `RandomForestClassifier` class, which is more convenient and optimized for Decision Trees¹¹ (similarly, there is a `RandomForestRegressor` class for regression tasks). The following code trains a Random Forest classifier with 500 trees, each limited to maximum 16 nodes, and using all available CPU cores:

```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16,
                                  n_jobs=-1, random_state=42)
rnd_clf.fit(X_train, y_train)

y_pred_rf = rnd_clf.predict(X_test)
```

With a few exceptions, a `RandomForestClassifier` has all the hyperparameters of a `DecisionTreeClassifier` (to control how trees are grown), plus all the hyperparameters of a `BaggingClassifier` to control the ensemble itself.

The Random Forest algorithm introduces extra randomness when growing trees; instead of searching for the very best feature when splitting a node (see [Chapter 6](#)), it searches for the best feature among a random subset of features. By default, it samples \sqrt{n} features (where n is the total number of features). The algorithm results in greater tree diversity, which (again) trades a higher bias for a lower variance, generally yielding an overall better model. So the following `BaggingClassifier` is equivalent to the previous `RandomForestClassifier`:

⁹ Tin Kam Ho, “The Random Subspace Method for Constructing Decision Forests,” *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20, no. 8 (1998): 832–844.

¹⁰ Tin Kam Ho, “Random Decision Forests,” *Proceedings of the Third International Conference on Document Analysis and Recognition* 1 (1995): 278.

¹¹ The `BaggingClassifier` class remains useful if you want a bag of something other than Decision Trees.

```
bag_clf = BaggingClassifier(  
    DecisionTreeClassifier(max_features="sqrt", max_leaf_nodes=16),  
    n_estimators=500, n_jobs=-1, random_state=42)
```

Extra-Trees

When you are growing a tree in a Random Forest, at each node only a random subset of the features is considered for splitting (as discussed earlier). It is possible to make trees even more random by also using random thresholds for each feature rather than searching for the best possible thresholds (like regular Decision Trees do). For this, simply set `splitter="random"` when creating a `DecisionTreeClassifier`.

A forest of such extremely random trees is called an *Extremely Randomized Trees* ensemble¹² (or *Extra-Trees* for short). Once again, this technique trades more bias for a lower variance. It also makes Extra-Trees much faster to train than regular Random Forests, because finding the best possible threshold for each feature at every node is one of the most time-consuming tasks of growing a tree.

You can create an Extra-Trees classifier using Scikit-Learn's `ExtraTreesClassifier` class. Its API is identical to the `RandomForestClassifier` class, except `bootstrap` defaults to `False`. Similarly, the `ExtraTreesRegressor` class has the same API as the `RandomForestRegressor` class, except `bootstrap` defaults to `False`.



It is hard to tell in advance whether a `RandomForestClassifier` will perform better or worse than an `ExtraTreesClassifier`. Generally, the only way to know is to try both and compare them using cross-validation.

Feature Importance

Yet another great quality of Random Forests is that they make it easy to measure the relative importance of each feature. Scikit-Learn measures a feature's importance by looking at how much the tree nodes that use that feature reduce impurity on average, across all trees in the forest. More precisely, it is a weighted average, where each node's weight is equal to the number of training samples that are associated with it (see [Chapter 6](#)).

Scikit-Learn computes this score automatically for each feature after training, then it scales the results so that the sum of all importances is equal to 1. You can access the result using the `feature_importances_` variable. For example, the following code trains a `RandomForestClassifier` on the iris dataset (introduced in [Chapter 4](#)) and outputs each feature's importance. It seems that the most important features are

¹² Pierre Geurts et al., “Extremely Randomized Trees,” *Machine Learning* 63, no. 1 (2006): 3–42.

the petal length (44%) and width (42%), while sepal length and width are rather unimportant in comparison (11% and 2%, respectively):

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris(as_frame=True)
>>> rnd_clf = RandomForestClassifier(n_estimators=500, random_state=42)
>>> rnd_clf.fit(iris.data, iris.target)
>>> for score, name in zip(rnd_clf.feature_importances_, iris.data.columns):
...     print(round(score, 2), name)
...
0.11 sepal length (cm)
0.02 sepal width (cm)
0.44 petal length (cm)
0.42 petal width (cm)
```

Similarly, if you train a Random Forest classifier on the MNIST dataset (introduced in [Chapter 3](#)) and plot each pixel's importance, you get the image represented in Figure 7-6.

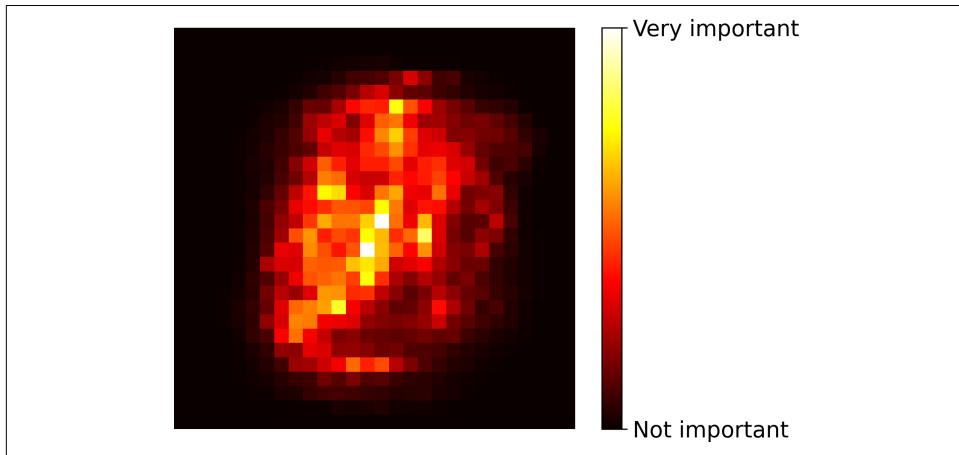


Figure 7-6. MNIST pixel importance (according to a Random Forest classifier)

Random Forests are very handy to get a quick understanding of what features actually matter, in particular if you need to perform feature selection.

Boosting

Boosting (originally called *hypothesis boosting*) refers to any Ensemble method that can combine several weak learners into a strong learner. The general idea of most boosting methods is to train predictors sequentially, each trying to correct its predecessor. There are many boosting methods available, but by far the most popular

are *AdaBoost*¹³ (short for *Adaptive Boosting*) and *Gradient Boosting*. Let's start with AdaBoost.

AdaBoost

One way for a new predictor to correct its predecessor is to pay a bit more attention to the training instances that the predecessor underfitted. This results in new predictors focusing more and more on the hard cases. This is the technique used by AdaBoost.

For example, when training an AdaBoost classifier, the algorithm first trains a base classifier (such as a Decision Tree) and uses it to make predictions on the training set. The algorithm then increases the relative weight of misclassified training instances. Then it trains a second classifier, using the updated weights, and again makes predictions on the training set, updates the instance weights, and so on (see Figure 7-7).

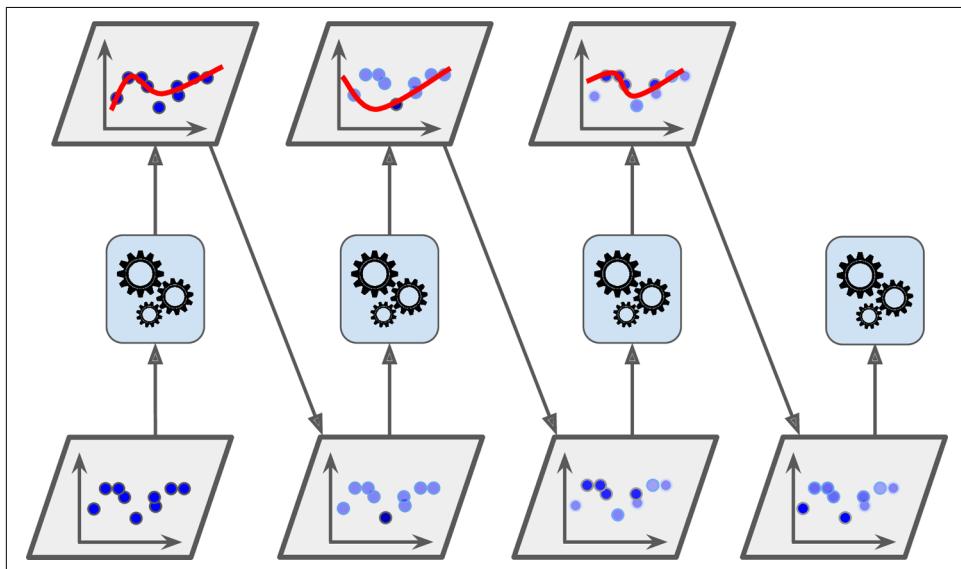


Figure 7-7. AdaBoost sequential training with instance weight updates

Figure 7-8 shows the decision boundaries of five consecutive predictors on the moons dataset (in this example, each predictor is a highly regularized SVM classifier with an RBF kernel¹⁴). The first classifier gets many instances wrong, so their weights

¹³ Yoav Freund and Robert E. Schapire, "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting," *Journal of Computer and System Sciences* 55, no. 1 (1997): 119–139.

¹⁴ This is just for illustrative purposes. SVMs are generally not good base predictors for AdaBoost; they are slow and tend to be unstable with it.

get boosted. The second classifier therefore does a better job on these instances, and so on. The plot on the right represents the same sequence of predictors, except that the learning rate is halved (i.e., the misclassified instance weights are boosted much less at every iteration). As you can see, this sequential learning technique has some similarities with Gradient Descent, except that instead of tweaking a single predictor's parameters to minimize a cost function, AdaBoost adds predictors to the ensemble, gradually making it better.

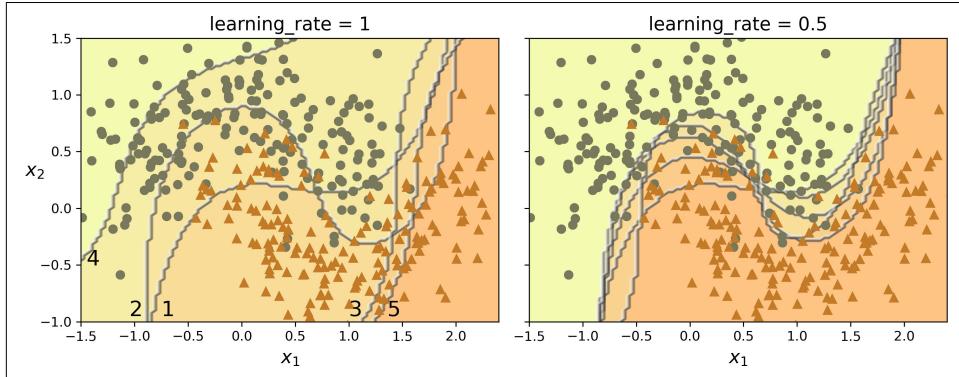


Figure 7-8. Decision boundaries of consecutive predictors

Once all predictors are trained, the ensemble makes predictions very much like bagging or pasting, except that predictors have different weights depending on their overall accuracy on the weighted training set.



There is one important drawback to this sequential learning technique: training cannot be parallelized since each predictor can only be trained after the previous predictor has been trained and evaluated. As a result, it does not scale as well as bagging or pasting.

Let's take a closer look at the AdaBoost algorithm. Each instance weight $w^{(i)}$ is initially set to $1/m$. A first predictor is trained, and its weighted error rate r_1 is computed on the training set; see [Equation 7-1](#).

Equation 7-1. Weighted error rate of the j^{th} predictor

$$r_j = \sum_{\substack{i=1 \\ \hat{y}_j^{(i)} \neq y^{(i)}}}^m w^{(i)} \quad \text{where } \hat{y}_j^{(i)} \text{ is the } j^{\text{th}} \text{ predictor's prediction for the } i^{\text{th}} \text{ instance.}$$

The predictor's weight α_j is then computed using [Equation 7-2](#), where η is the learning rate hyperparameter (defaults to 1).¹⁵ The more accurate the predictor is, the higher its weight will be. If it is just guessing randomly, then its weight will be close to zero. However, if it is most often wrong (i.e., less accurate than random guessing), then its weight will be negative.

Equation 7-2. Predictor weight

$$\alpha_j = \eta \log \frac{1 - r_j}{r_j}$$

Next, the AdaBoost algorithm updates the instance weights, using [Equation 7-3](#), which boosts the weights of the misclassified instances.

Equation 7-3. Weight update rule

for $i = 1, 2, \dots, m$

$$w^{(i)} \leftarrow \begin{cases} w^{(i)} & \text{if } \hat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} \exp(\alpha_j) & \text{if } \hat{y}_j^{(i)} \neq y^{(i)} \end{cases}$$

Then all the instance weights are normalized (i.e., divided by $\sum_{i=1}^m w^{(i)}$).

Finally, a new predictor is trained using the updated weights, and the whole process is repeated: the new predictor's weight is computed, the instance weights are updated, then another predictor is trained, and so on. The algorithm stops when the desired number of predictors is reached, or when a perfect predictor is found.

To make predictions, AdaBoost simply computes the predictions of all the predictors and weighs them using the predictor weights α_j . The predicted class is the one that receives the majority of weighted votes (see [Equation 7-4](#)).

Equation 7-4. AdaBoost predictions

$$\hat{y}(\mathbf{x}) = \underset{k}{\operatorname{argmax}} \sum_{j=1}^N \alpha_j \quad \text{where } N \text{ is the number of predictors.}$$

$$\hat{y}_j(\mathbf{x}) = k$$

Scikit-Learn uses a multiclass version of AdaBoost called **SAMME**¹⁶ (which stands for *Stagewise Additive Modeling using a Multiclass Exponential loss function*). When there

¹⁵ The original AdaBoost algorithm does not use a learning rate hyperparameter.

¹⁶ For more details, see Ji Zhu et al., "Multi-Class AdaBoost," *Statistics and Its Interface* 2, no. 3 (2009): 349–360.

are just two classes, SAMME is equivalent to AdaBoost. If the predictors can estimate class probabilities (i.e., if they have a `predict_proba()` method), Scikit-Learn can use a variant of SAMME called SAMME.R (the *R* stands for “Real”), which relies on class probabilities rather than predictions and generally performs better.

The following code trains an AdaBoost classifier based on 30 *Decision Stumps* using Scikit-Learn’s `AdaBoostClassifier` class (as you might expect, there is also an `AdaBoostRegressor` class). A Decision Stump is a Decision Tree with `max_depth=1`—in other words, a tree composed of a single decision node plus two leaf nodes. This is the default base estimator for the `AdaBoostClassifier` class:

```
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=30,
    learning_rate=0.5, random_state=42)
ada_clf.fit(X_train, y_train)
```



If your AdaBoost ensemble is overfitting the training set, you can try reducing the number of estimators or more strongly regularizing the base estimator.

Gradient Boosting

Another very popular boosting algorithm is *Gradient Boosting*.¹⁷ Just like AdaBoost, Gradient Boosting works by sequentially adding predictors to an ensemble, each one correcting its predecessor. However, instead of tweaking the instance weights at every iteration like AdaBoost does, this method tries to fit the new predictor to the *residual errors* made by the previous predictor.

Let’s go through a simple regression example, using Decision Trees as the base predictors: this is called *Gradient Tree Boosting*, or *Gradient Boosted Regression Trees* (GBRT). First, let’s generate a noisy quadratic dataset and fit a `DecisionTreeRegressor` to it:

```
import numpy as np
from sklearn.tree import DecisionTreeRegressor

np.random.seed(42)
X = np.random.rand(100, 1) - 0.5
y = 3 * X[:, 0] ** 2 + 0.05 * np.random.randn(100) # y = 3x^2 + Gaussian noise
```

¹⁷ Gradient Boosting was first introduced in Leo Breiman’s 1997 paper “Arcing the Edge” and was further developed in the 1999 paper “Greedy Function Approximation: A Gradient Boosting Machine” by Jerome H. Friedman.

```
tree_reg1 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg1.fit(X, y)
```

Next, we'll train a second `DecisionTreeRegressor` on the residual errors made by the first predictor:

```
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2, random_state=43)
tree_reg2.fit(X, y2)
```

Then we train a third regressor on the residual errors made by the second predictor:

```
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2, random_state=44)
tree_reg3.fit(X, y3)
```

Now we have an ensemble containing three trees. It can make predictions on a new instance simply by adding up the predictions of all the trees:

```
>>> X_new = np.array([[-0.4], [0.], [0.5]])
>>> sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
array([0.49484029, 0.04021166, 0.75026781])
```

Figure 7-9 represents the predictions of these three trees in the left column, and the ensemble's predictions in the right column. In the first row, the ensemble has just one tree, so its predictions are exactly the same as the first tree's predictions. In the second row, a new tree is trained on the residual errors of the first tree. On the right you can see that the ensemble's predictions are equal to the sum of the predictions of the first two trees. Similarly, in the third row another tree is trained on the residual errors of the second tree. You can see that the ensemble's predictions gradually get better as trees are added to the ensemble.

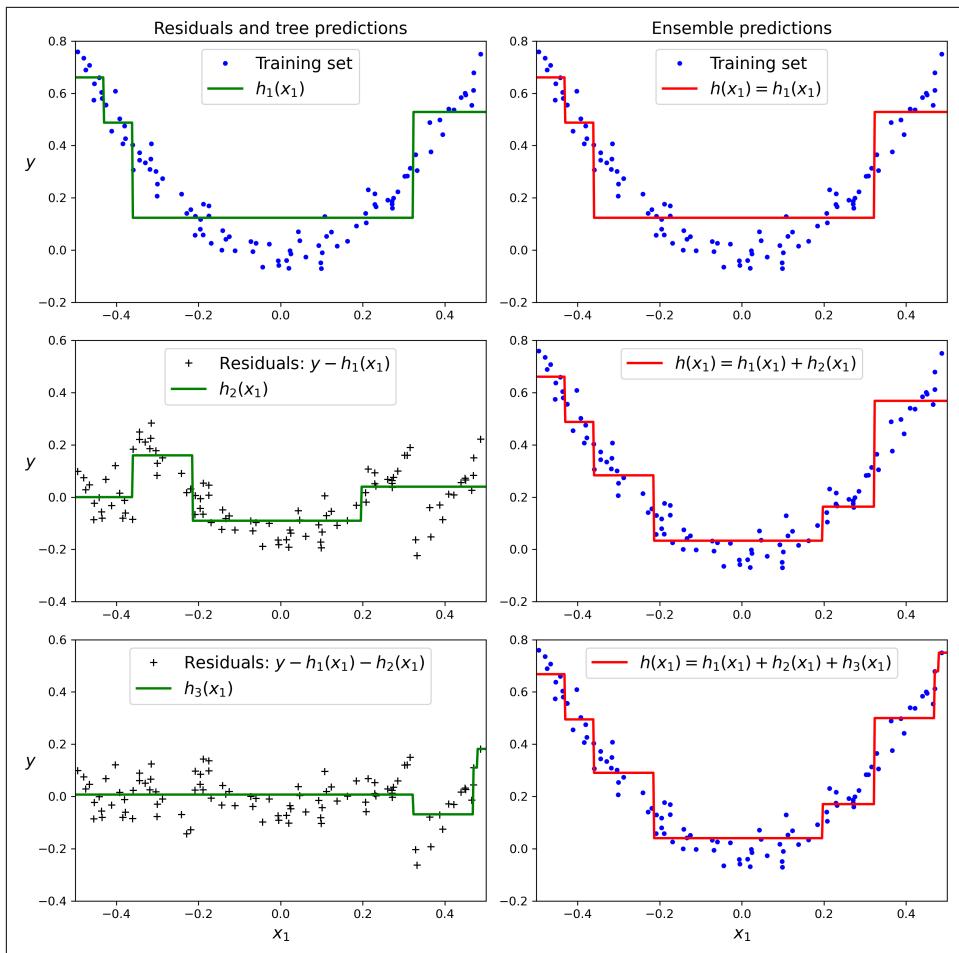


Figure 7-9. In this depiction of Gradient Boosting, the first predictor (top left) is trained normally, then each consecutive predictor (middle left and lower left) is trained on the previous predictor's residuals; the right column shows the resulting ensemble's predictions

A simpler way to train GBRT ensembles is to use Scikit-Learn's `GradientBoostingRegressor` class (there's also a `GradientBoostingClassifier` class for classification). Much like the `RandomForestRegressor` class, it has hyperparameters to control the growth of Decision Trees (e.g., `max_depth`, `min_samples_leaf`), as well as hyperparameters to control the ensemble training, such as the number of trees (`n_estimators`). The following code creates the same ensemble as the previous one:

```
from sklearn.ensemble import GradientBoostingRegressor

gbdt = GradientBoostingRegressor(max_depth=2, n_estimators=3,
```

```
learning_rate=1.0, random_state=42)
gbdt.fit(X, y)
```

The `learning_rate` hyperparameter scales the contribution of each tree. If you set it to a low value, such as `0.05`, you will need more trees in the ensemble to fit the training set, but the predictions will usually generalize better. This is a regularization technique called *shrinkage*. Figure 7-10 shows two GBRT ensembles trained with different hyperparameters: the one on the left does not have enough trees to fit the training set, while the one on the right has about the right amount. If we added more trees, the GBRT would start to overfit the training set.

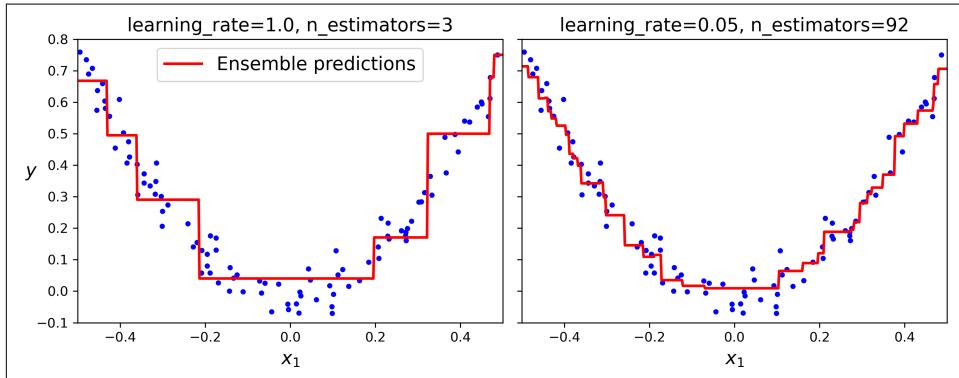


Figure 7-10. GBRT ensembles with not enough predictors (left) and just enough (right)

To find the optimal number of trees, you could perform cross-validation using `GridSearchCV` or `RandomizedSearchCV`, as usual, but there's a simpler way: if you set the `n_iter_no_change` hyperparameter to an integer value, say `10`, then the `GradientBoostingRegressor` will automatically stop adding more trees during training if it sees that the last `10` trees didn't help. This is simply early stopping (introduced in Chapter 4), but with a little bit of patience: it tolerates having no progress for a few iterations before it stops. Let's train the ensemble using early stopping:

```
gbdt_best = GradientBoostingRegressor(
    max_depth=2, learning_rate=0.05, n_estimators=500,
    n_iter_no_change=10, random_state=42)
gbdt_best.fit(X, y)
```

If you set `n_iter_no_change` too low, training may stop too early and the model will underfit. But if you set it too high, it will overfit instead. We also set a fairly small learning rate and a high number of estimators, but the actual number of estimators in the trained ensemble is much lower, thanks to early stopping:

```
>>> gbdt_best.n_estimators_
92
```

When `n_iter_no_change` is set, the `fit()` method automatically splits the training set into a smaller training set and a validation set: this allows it to evaluate the model's performance each time it adds a new tree. The size of the validation set is controlled by the `validation_fraction` hyperparameter, which is 10% by default. The `tol` hyperparameter determines the maximum performance improvement that still counts as negligible. It defaults to 0.0001.

The `GradientBoostingRegressor` class also supports a `subsample` hyperparameter, which specifies the fraction of training instances to be used for training each tree. For example, if `subsample=0.25`, then each tree is trained on 25% of the training instances, selected randomly. As you can probably guess by now, this technique trades a higher bias for a lower variance. It also speeds up training considerably. This is called *Stochastic Gradient Boosting*.

Histogram-Based Gradient Boosting

Scikit-Learn also provides another GBRT implementation, optimized for large datasets: *Histogram-based Gradient Boosting* (HGB). It works by binning the inputs features, replacing them with integers. The number of bins is controlled by the `max_bins` hyperparameter, which defaults to 255 and cannot be set any higher than this. Binning can greatly reduce the number of possible thresholds that the training algorithm needs to evaluate. Moreover, working with integers makes it possible to use faster and more memory-efficient data structures. And the way the bins are built removes the need for sorting the features when training each tree.

As a result, this implementation has a computational complexity of $O(n \times m)$ instead of $O(n \times m \times \log(m))$, where m is the number of training instances, and n is the number of features. In practice, this means that HGB can train hundreds of times faster than regular GBRT on large datasets. However, binning causes a precision loss, which acts as a regularizer: depending on the dataset, this may help reduce overfitting, or it may cause underfitting.

Scikit-Learn provides two classes for HGB: `HistGradientBoostingRegressor` and `HistGradientBoostingClassifier`. They have a similar API as `GradientBoostingRegressor` and `GradientBoostingClassifier`, with a few notable differences:

- Early stopping is automatically activated if the number of instances is greater than 10,000. You can turn early-stopping always on or off by setting the `early_stopping` hyperparameter to `True` or `False`.
- Subsampling is not supported.
- `n_estimators` is renamed to `max_iter`.

- The only Decision Tree hyperparameters that can be tweaked are `max_leaf_nodes`, `min_samples_leaf`, and `max_depth`.

The HGB classes also have two nice features: they support both categorical features and missing values. This simplifies preprocessing quite a bit. However, the categorical features must be represented as integers ranging from 0 to a number lower than `max_bins`: you can use an `OrdinalEncoder` for this. For example, here's how to build and train a complete pipeline for the California housing dataset introduced in [Chapter 2](#):

```
from sklearn.pipeline import make_pipeline
from sklearn.compose import make_column_transformer
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.preprocessing import OrdinalEncoder

hgb_reg = make_pipeline(
    make_column_transformer((OrdinalEncoder(), ["ocean_proximity"]),
                           remainder="passthrough"),
    HistGradientBoostingRegressor(categorical_features=[0], random_state=42)
)
hgb_reg.fit(housing, housing_labels)
```

The whole pipeline is just as short as the imports! No need for an imputer, or a scaler, or a one-hot encoder, it's really convenient. Note that `categorical_features` must be set to the categorical column indices (or a boolean array). Without any hyperparameter tuning, this model yields an RMSE of about 47,600, which is not too bad.



Several other optimized implementations of Gradient Boosting are available in the Python ML ecosystem, in particular: [XGBoost](#), [CatBoost](#), and [LightGBM](#). These libraries have been around for several years, they are all specialized for Gradient Boosting, their APIs are very similar to Scikit-Learn's, and they provide many additional features, including GPU-acceleration: you should definitely check them out! Moreover, the [TensorFlow Random Forests library](#) provides optimized implementations of many Random Forest algorithms: plain Random Forests, Extra Trees, GBRT, and several more.

Stacking

The last Ensemble method we will discuss in this chapter is called *stacking* (short for *stacked generalization*).¹⁸ It is based on a simple idea: instead of using trivial functions

¹⁸ David H. Wolpert, “Stacked Generalization,” *Neural Networks* 5, no. 2 (1992): 241–259.

(such as hard voting) to aggregate the predictions of all predictors in an ensemble, why don't we train a model to perform this aggregation? [Figure 7-11](#) shows such an ensemble performing a regression task on a new instance. Each of the bottom three predictors predicts a different value (3.1, 2.7, and 2.9), and then the final predictor (called a *blender*, or a *meta learner*) takes these predictions as inputs and makes the final prediction (3.0).

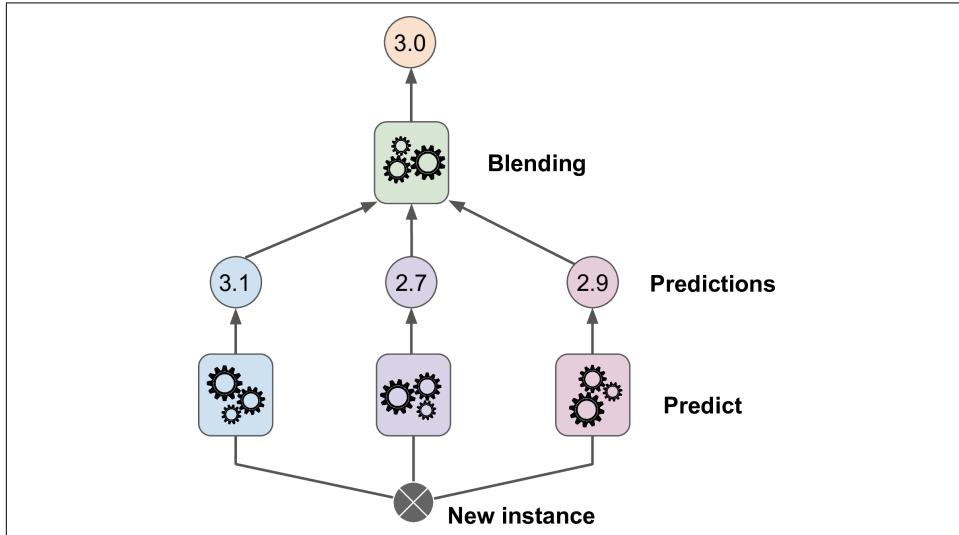


Figure 7-11. Aggregating predictions using a blending predictor

To train the blender, you first need to build the blending training set: you can use `cross_val_predict()` on every predictor in the ensemble to get out-of-sample predictions for each instance in the original training set ([Figure 7-12](#)). These can be used as the input features to train the blender, and the targets can simply be copied from the original training set. Note that regardless of the number of features in the original training set (just one in this example), the blending training set will contain one input feature per predictor (three in this example). Once the blender is trained, the base predictors are retrained one last time on the full original training set.

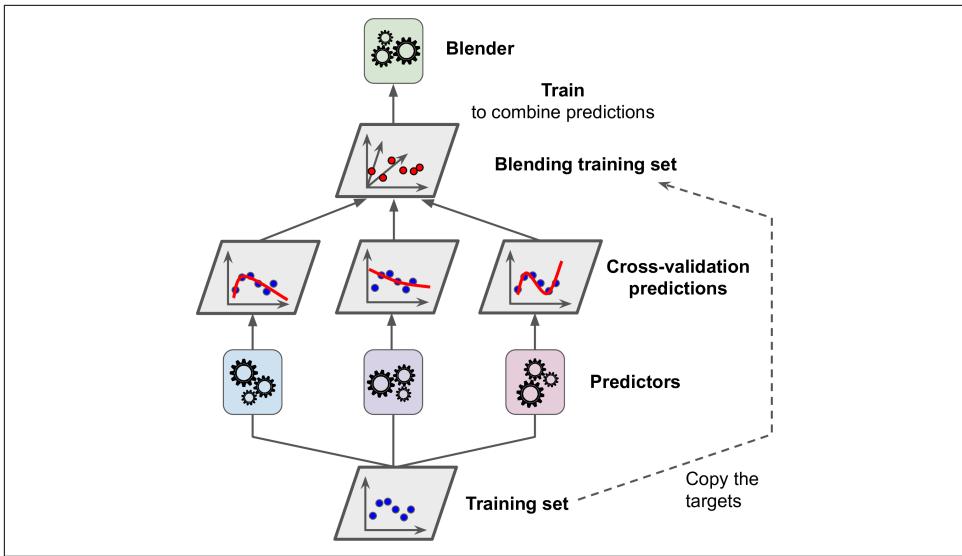


Figure 7-12. Training the blender in a stacking ensemble

It is actually possible to train several different blenders this way (e.g., one using Linear Regression, another using Random Forest Regression), to get a whole layer of blenders, and then add another blender on top of that to produce the final prediction, as shown in [Figure 7-13](#). You may be able to squeeze out a few more drops of performance doing this, but it will cost you in both training time and system complexity.

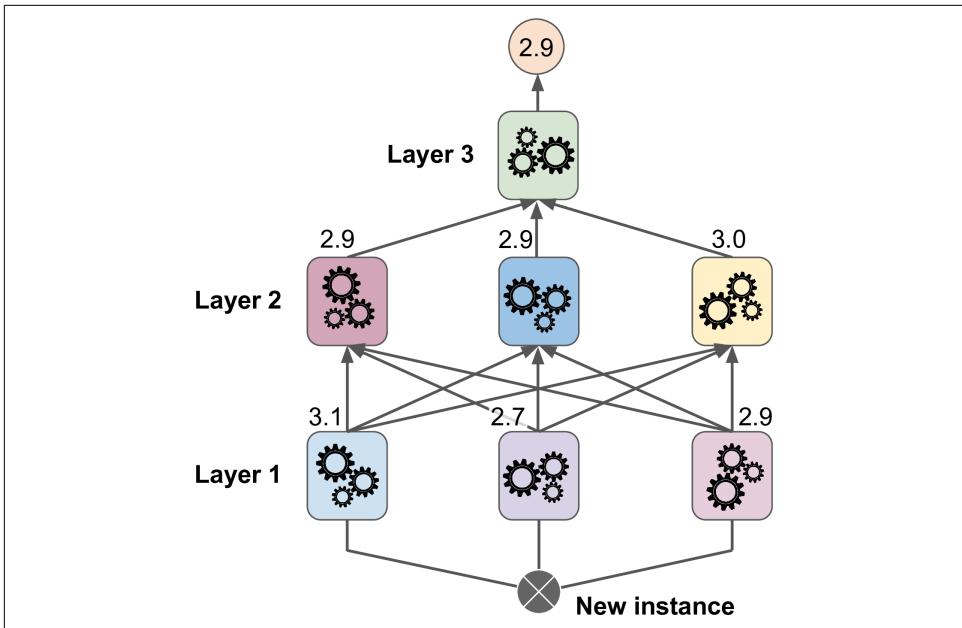


Figure 7-13. Predictions in a multilayer stacking ensemble

Scikit-Learn provides two classes for stacking ensembles: `StackingClassifier` and `StackingRegressor`. For example, you can replace the `VotingClassifier` you used at the beginning of this chapter on the moons dataset with a `StackingClassifier`:

```
from sklearn.ensemble import StackingClassifier

stacking_clf = StackingClassifier(
    estimators=[
        ('lr', LogisticRegression(random_state=42)),
        ('rf', RandomForestClassifier(random_state=42)),
        ('svc', SVC(probability=True, random_state=42))
    ],
    final_estimator=RandomForestClassifier(random_state=43),
    cv=5 # number of cross-validation folds
)
stacking_clf.fit(X_train, y_train)
```

For each predictor, the stacking classifier will call `predict_proba()` if available, or it will fallback to `decision_function()` if available, or as a last resort it will call `predict()`. If you don't provide a final estimator, `StackingClassifier` will use `LogisticRegression`, and `StackingRegressor` will use `RidgeCV`.

If you evaluate this stacking model on the test set, you will find 92.8% accuracy, which is a bit better than the voting classifier using soft voting, which got 92%.

In conclusion, ensemble methods are versatile, powerful, and fairly simple to use. Random Forests, AdaBoost and GBRT are among the first models you should test on most Machine Learning tasks, and they particularly shine with heterogeneous tabular data. Moreover, as they require very little preprocessing, they're great for getting a prototype up and running quickly. Lastly, ensemble methods like voting classifiers and stacking classifiers can help push your system's performance to its limits.

Exercises

1. If you have trained five different models on the exact same training data, and they all achieve 95% precision, is there any chance that you can combine these models to get better results? If so, how? If not, why?
2. What is the difference between hard and soft voting classifiers?
3. Is it possible to speed up training of a bagging ensemble by distributing it across multiple servers? What about pasting ensembles, boosting ensembles, Random Forests, or stacking ensembles?
4. What is the benefit of out-of-bag evaluation?
5. What makes Extra-Trees more random than regular Random Forests? How can this extra randomness help? Are Extra-Trees slower or faster than regular Random Forests?
6. If your AdaBoost ensemble underfits the training data, which hyperparameters should you tweak and how?
7. If your Gradient Boosting ensemble overfits the training set, should you increase or decrease the learning rate?
8. Load the MNIST data (introduced in [Chapter 3](#)), and split it into a training set, a validation set, and a test set (e.g., use 50,000 instances for training, 10,000 for validation, and 10,000 for testing). Then train various classifiers, such as a Random Forest classifier, an Extra-Trees classifier, and an SVM classifier. Next, try to combine them into an ensemble that outperforms each individual classifier on the validation set, using soft or hard voting. Once you have found one, try it on the test set. How much better does it perform compared to the individual classifiers?
9. Run the individual classifiers from the previous exercise to make predictions on the validation set, and create a new training set with the resulting predictions: each training instance is a vector containing the set of predictions from all your classifiers for an image, and the target is the image's class. Train a classifier on this new training set. Congratulations, you have just trained a blender, and together with the classifiers it forms a stacking ensemble! Now evaluate the ensemble on the test set. For each image in the test set, make predictions with all

your classifiers, then feed the predictions to the blender to get the ensemble's predictions. How does it compare to the voting classifier you trained earlier? Now try again using a `StackingClassifier` instead: do you get better performance? If so, why?

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab3>.

Dimensionality Reduction

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 8th chapter of the final book. The GitHub repo is <https://github.com/ageron/handsonml3>.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

Many Machine Learning problems involve thousands or even millions of features for each training instance. Not only do all these features make training extremely slow, but they can also make it much harder to find a good solution, as we will see. This problem is often referred to as the *curse of dimensionality*.

Fortunately, in real-world problems, it is often possible to reduce the number of features considerably, turning an intractable problem into a tractable one. For example, consider the MNIST images (introduced in [Chapter 3](#)): the pixels on the image borders are almost always white, so you could completely drop these pixels from the training set without losing much information. As we saw in the previous chapter, [Figure 7-6](#) confirms that these pixels are utterly unimportant for the classification task. Additionally, two neighboring pixels are often highly correlated: if you merge them into a single pixel (e.g., by taking the mean of the two pixel intensities), you will not lose much information.



Reducing dimensionality does cause some information loss, just like compressing an image to JPEG can degrade its quality, so even though it will speed up training, it may make your system perform slightly worse. It also makes your pipelines a bit more complex and thus harder to maintain. Therefore I recommend you first try to train your system with the original data before considering using dimensionality reduction. In some cases, reducing the dimensionality of the training data may filter out some noise and unnecessary details and thus result in higher performance, but in general it won't; it will just speed up training.

Apart from speeding up training, dimensionality reduction is also extremely useful for data visualization (or *DataViz*). Reducing the number of dimensions down to two (or three) makes it possible to plot a condensed view of a high-dimensional training set on a graph and often gain some important insights by visually detecting patterns, such as clusters. Moreover, DataViz is essential to communicate your conclusions to people who are not data scientists—in particular, decision makers who will use your results.

In this chapter we will discuss the curse of dimensionality and get a sense of what goes on in high-dimensional space. Then, we will consider the two main approaches to dimensionality reduction (projection and Manifold Learning), and we will go through three of the most popular dimensionality reduction techniques: PCA, random projection, and LLE.

The Curse of Dimensionality

We are so used to living in three dimensions¹ that our intuition fails us when we try to imagine a high-dimensional space. Even a basic 4D hypercube is incredibly hard to picture in our minds (see [Figure 8-1](#)), let alone a 200-dimensional ellipsoid bent in a 1,000-dimensional space.

¹ Well, four dimensions if you count time, and a few more if you are a string theorist.

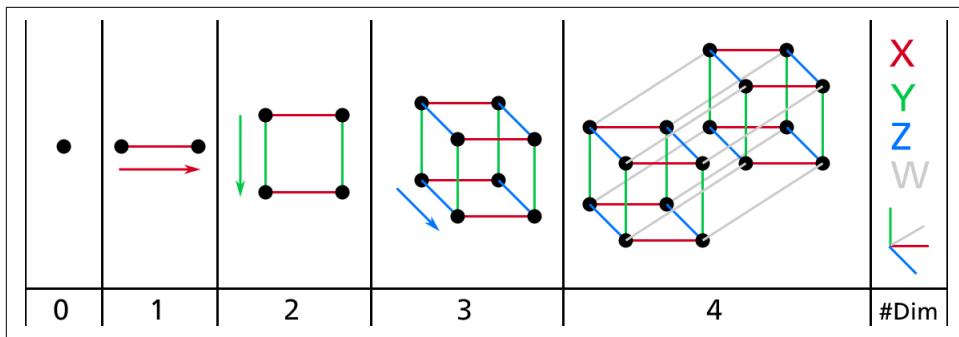


Figure 8-1. Point, segment, square, cube, and tesseract (0D to 4D hypercubes)²

It turns out that many things behave very differently in high-dimensional space. For example, if you pick a random point in a unit square (a 1×1 square), it will have only about a 0.4% chance of being located less than 0.001 from a border (in other words, it is very unlikely that a random point will be “extreme” along any dimension). But in a 10,000-dimensional unit hypercube, this probability is greater than 99.999999%. Most points in a high-dimensional hypercube are very close to the border.³

Here is a more troublesome difference: if you pick two points randomly in a unit square, the distance between these two points will be, on average, roughly 0.52. If you pick two random points in a unit 3D cube, the average distance will be roughly 0.66. But what about two points picked randomly in a 1,000,000-dimensional hypercube? The average distance, believe it or not, will be about 408.25 (roughly $\sqrt{1,000,000/6}$)! This is counterintuitive: how can two points be so far apart when they both lie within the same unit hypercube? Well, there’s just plenty of space in high dimensions. As a result, high-dimensional datasets are at risk of being very sparse: most training instances are likely to be far away from each other. This also means that a new instance will likely be far away from any training instance, making predictions much less reliable than in lower dimensions, since they will be based on much larger extrapolations. In short, the more dimensions the training set has, the greater the risk of overfitting it.

In theory, one solution to the curse of dimensionality could be to increase the size of the training set to reach a sufficient density of training instances. Unfortunately, in practice, the number of training instances required to reach a given density grows exponentially with the number of dimensions. With just 100 features—significantly fewer than in the MNIST problem—all ranging from 0 to 1, you would need

² Watch a rotating tesseract projected into 3D space at <https://homl.info/30>. Image by Wikipedia user NerdBoy1392 (Creative Commons BY-SA 3.0). Reproduced from <https://en.wikipedia.org/wiki/Tesseract>.

³ Fun fact: anyone you know is probably an extremist in at least one dimension (e.g., how much sugar they put in their coffee), if you consider enough dimensions.

more training instances than atoms in the observable universe in order for training instances to be within 0.1 of each other on average, assuming they were spread out uniformly across all dimensions.

Main Approaches for Dimensionality Reduction

Before we dive into specific dimensionality reduction algorithms, let's take a look at the two main approaches to reducing dimensionality: projection and Manifold Learning.

Projection

In most real-world problems, training instances are *not* spread out uniformly across all dimensions. Many features are almost constant, while others are highly correlated (as discussed earlier for MNIST). As a result, all training instances lie within (or close to) a much lower-dimensional *subspace* of the high-dimensional space. This sounds very abstract, so let's look at an example. In [Figure 8-2](#) you can see a 3D dataset represented by small spheres.

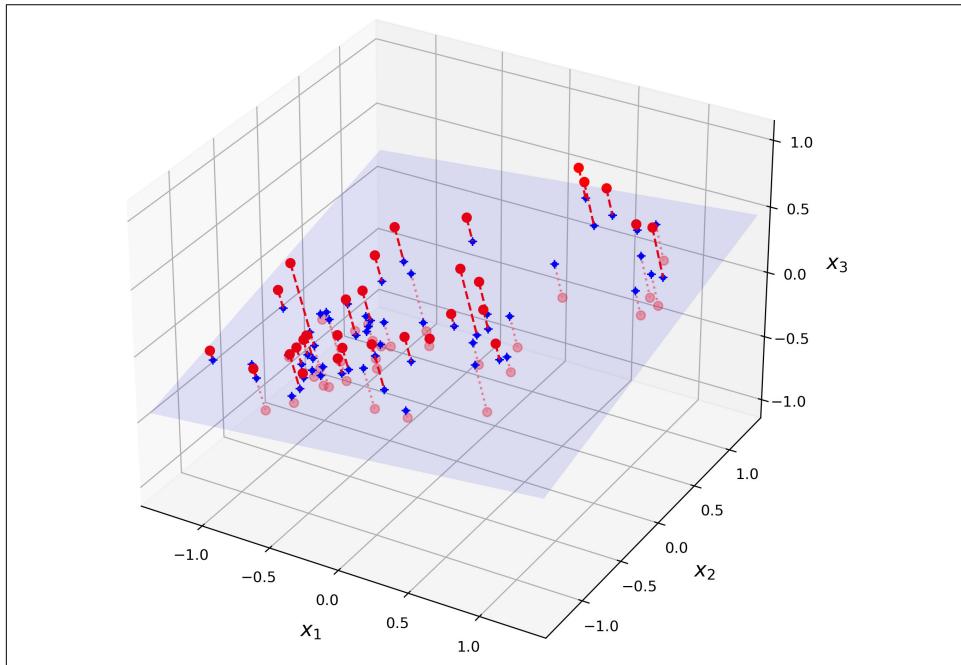


Figure 8-2. A 3D dataset lying close to a 2D subspace

Notice that all training instances lie close to a plane: this is a lower-dimensional (2D) subspace of the higher-dimensional (3D) space. If we project every training

instance perpendicularly onto this subspace (as represented by the short dashed lines connecting the instances to the plane), we get the new 2D dataset shown in [Figure 8-3](#). Ta-da! We have just reduced the dataset's dimensionality from 3D to 2D. Note that the axes correspond to new features z_1 and z_2 : they are the coordinates of the projections on the plane.

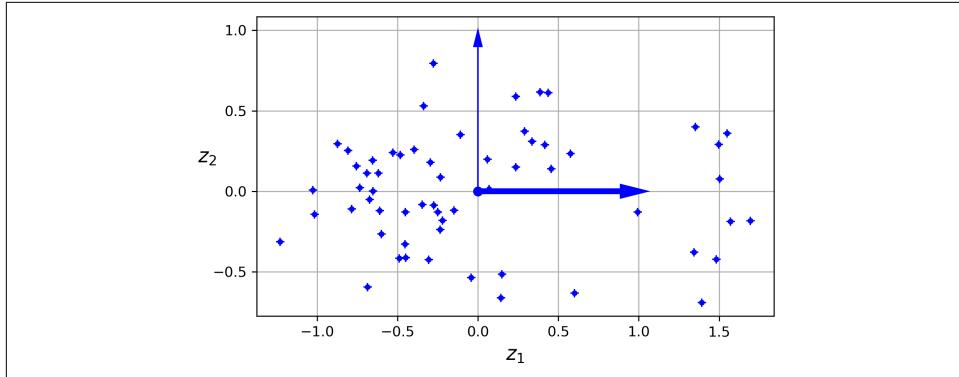


Figure 8-3. The new 2D dataset after projection

Manifold Learning

However, projection is not always the best approach to dimensionality reduction. In many cases the subspace may twist and turn, such as in the famous *Swiss roll* toy dataset represented in [Figure 8-4](#).

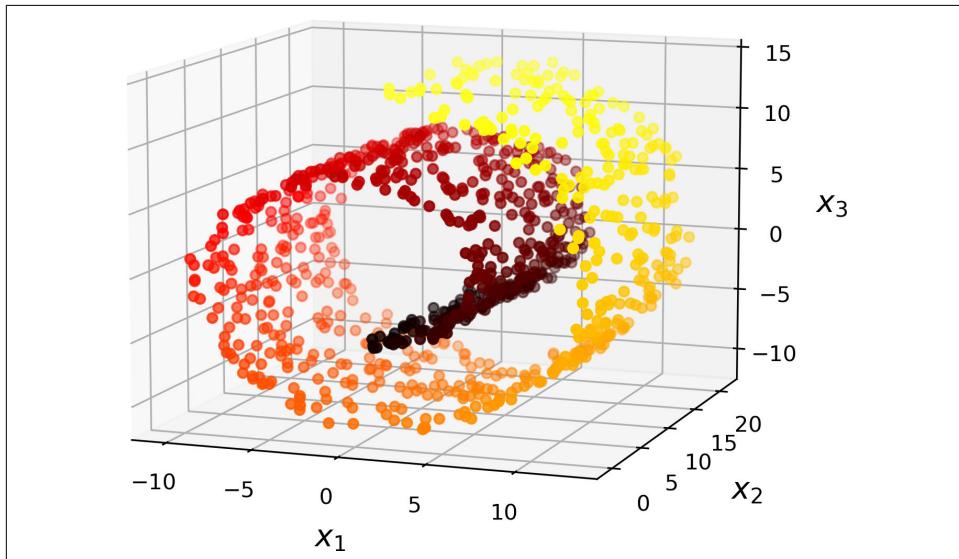


Figure 8-4. Swiss roll dataset

Simply projecting onto a plane (e.g., by dropping x_3) would squash different layers of the Swiss roll together, as shown on the left side of [Figure 8-5](#). What you probably want instead is to unroll the Swiss roll to obtain the 2D dataset on the right side of [Figure 8-5](#).

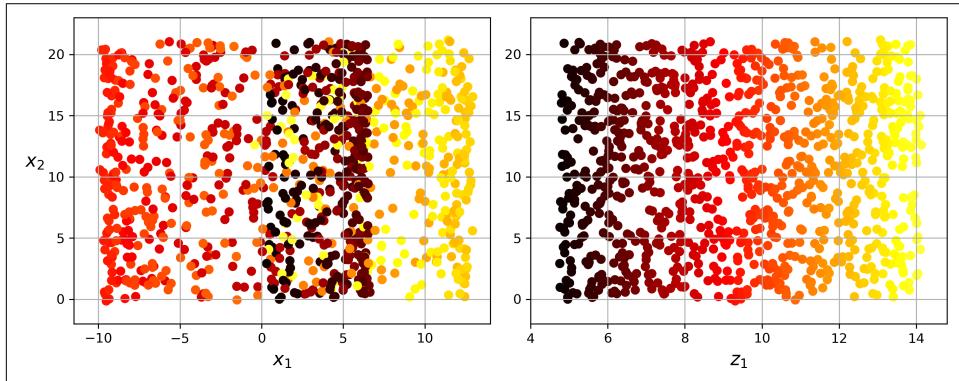


Figure 8-5. Squashing by projecting onto a plane (left) versus unrolling the Swiss roll (right)

The Swiss roll is an example of a 2D *manifold*. Put simply, a 2D manifold is a 2D shape that can be bent and twisted in a higher-dimensional space. More generally, a d -dimensional manifold is a part of an n -dimensional space (where $d < n$) that locally resembles a d -dimensional hyperplane. In the case of the Swiss roll, $d = 2$ and $n = 3$: it locally resembles a 2D plane, but it is rolled in the third dimension.

Many dimensionality reduction algorithms work by modeling the manifold on which the training instances lie; this is called *Manifold Learning*. It relies on the *manifold assumption*, also called the *manifold hypothesis*, which holds that most real-world high-dimensional datasets lie close to a much lower-dimensional manifold. This assumption is very often empirically observed.

Once again, think about the MNIST dataset: all handwritten digit images have some similarities. They are made of connected lines, the borders are white, and they are more or less centered. If you randomly generated images, only a ridiculously tiny fraction of them would look like handwritten digits. In other words, the degrees of freedom available to you if you try to create a digit image are dramatically lower than the degrees of freedom you have if you are allowed to generate any image you want. These constraints tend to squeeze the dataset into a lower-dimensional manifold.

The manifold assumption is often accompanied by another implicit assumption: that the task at hand (e.g., classification or regression) will be simpler if expressed in the lower-dimensional space of the manifold. For example, in the top row of [Figure 8-6](#) the Swiss roll is split into two classes: in the 3D space (on the left), the decision

boundary would be fairly complex, but in the 2D unrolled manifold space (on the right), the decision boundary is a straight line.

However, this implicit assumption does not always hold. For example, in the bottom row of [Figure 8-6](#), the decision boundary is located at $x_1 = 5$. This decision boundary looks very simple in the original 3D space (a vertical plane), but it looks more complex in the unrolled manifold (a collection of four independent line segments).

In short, reducing the dimensionality of your training set before training a model will usually speed up training, but it may not always lead to a better or simpler solution; it all depends on the dataset.

Hopefully you now have a good sense of what the curse of dimensionality is and how dimensionality reduction algorithms can fight it, especially when the manifold assumption holds. The rest of this chapter will go through some of the most popular algorithms for dimensionality reduction.

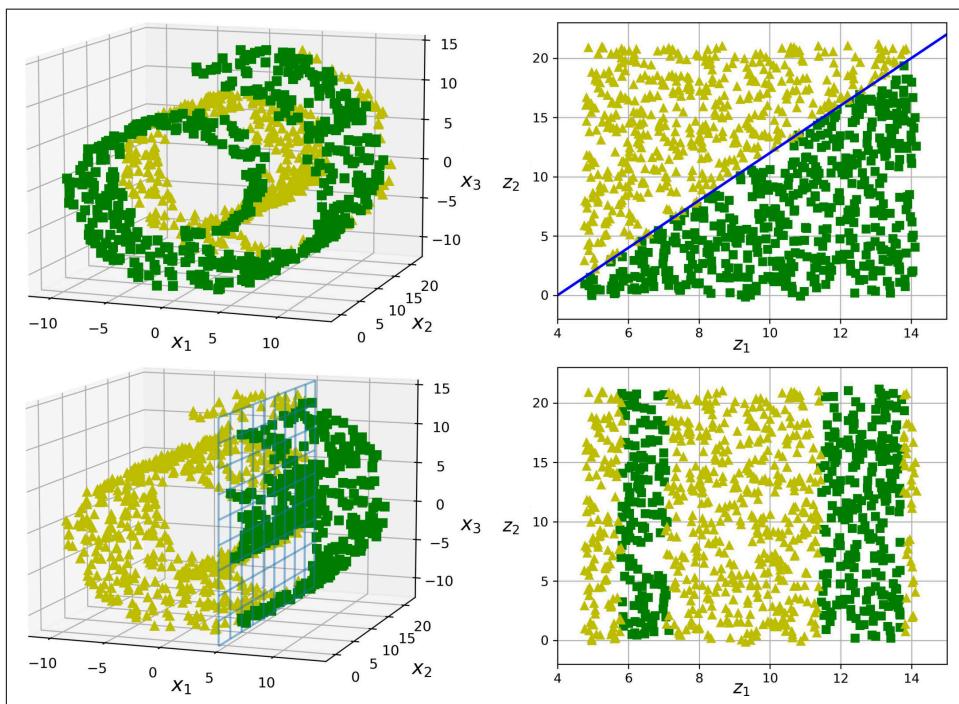


Figure 8-6. The decision boundary may not always be simpler with lower dimensions

PCA

Principal Component Analysis (PCA) is by far the most popular dimensionality reduction algorithm. First it identifies the hyperplane that lies closest to the data, and then it projects the data onto it, just like in Figure 8-2.

Preserving the Variance

Before you can project the training set onto a lower-dimensional hyperplane, you first need to choose the right hyperplane. For example, a simple 2D dataset is represented on the left in Figure 8-7, along with three different axes (i.e., 1D hyperplanes). On the right is the result of the projection of the dataset onto each of these axes. As you can see, the projection onto the solid line preserves the maximum variance (top), while the projection onto the dotted line preserves very little variance (bottom) and the projection onto the dashed line preserves an intermediate amount of variance (middle).

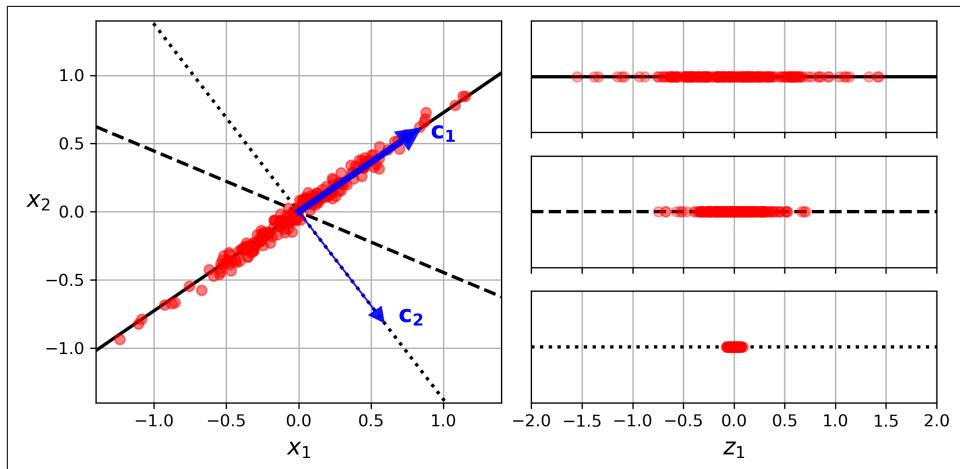


Figure 8-7. Selecting the subspace on which to project

It seems reasonable to select the axis that preserves the maximum amount of variance, as it will most likely lose less information than the other projections. Another way to justify this choice is that it is the axis that minimizes the mean squared distance between the original dataset and its projection onto that axis. This is the rather simple idea behind PCA.⁴

⁴ Karl Pearson, "On Lines and Planes of Closest Fit to Systems of Points in Space," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2, no. 11 (1901): 559–572, <https://homl.info/pca>.

Principal Components

PCA identifies the axis that accounts for the largest amount of variance in the training set. In [Figure 8-7](#), it is the solid line. It also finds a second axis, orthogonal to the first one, that accounts for the largest amount of remaining variance. In this 2D example there is no choice: it is the dotted line. If it were a higher-dimensional dataset, PCA would also find a third axis, orthogonal to both previous axes, and a fourth, a fifth, and so on—as many axes as the number of dimensions in the dataset.

The i^{th} axis is called the i^{th} *principal component* (PC) of the data. In [Figure 8-7](#), the first PC is the axis on which vector \mathbf{c}_1 lies, and the second PC is the axis on which vector \mathbf{c}_2 lies. In [Figure 8-2](#) the first two PCs are on the projection plane, and the third PC is the axis orthogonal to that plane. After the projection, in [Figure 8-3](#), the first PC corresponds to the z_1 axis, and the second PC corresponds to the z_2 axis.



For each principal component, PCA finds a zero-centered unit vector pointing in the direction of the PC. Since two opposing unit vectors lie on the same axis, the direction of the unit vectors returned by PCA is not stable: if you perturb the training set slightly and run PCA again, the unit vectors may point in the opposite direction as the original vectors. However, they will generally still lie on the same axes. In some cases, a pair of unit vectors may even rotate or swap (if the variances along these two axes are very close), but the plane they define will generally remain the same.

So how can you find the principal components of a training set? Luckily, there is a standard matrix factorization technique called *Singular Value Decomposition* (SVD) that can decompose the training set matrix \mathbf{X} into the matrix multiplication of three matrices $\mathbf{U} \Sigma \mathbf{V}^T$, where \mathbf{V} contains the unit vectors that define all the principal components that we are looking for, as shown in [Equation 8-1](#).

Equation 8-1. Principal components matrix

$$\mathbf{V} = \begin{pmatrix} | & | & | \\ \mathbf{c}_1 & \mathbf{c}_2 & \cdots & \mathbf{c}_n \\ | & | & | \end{pmatrix}$$

The following Python code uses NumPy’s `svd()` function to obtain all the principal components of the 3D training set represented in [Figure 8-2](#), then it extracts the two unit vectors that define the first two PCs:

```
import numpy as np

X = [...] # create a small 3D dataset
X_centered = X - X.mean(axis=0)
```

```
U, s, Vt = np.linalg.svd(X_centered)
c1 = Vt[0]
c2 = Vt[1]
```



PCA assumes that the dataset is centered around the origin. As we will see, Scikit-Learn's PCA classes take care of centering the data for you. If you implement PCA yourself (as in the preceding example), or if you use other libraries, don't forget to center the data first.

Projecting Down to d Dimensions

Once you have identified all the principal components, you can reduce the dimensionality of the dataset down to d dimensions by projecting it onto the hyperplane defined by the first d principal components. Selecting this hyperplane ensures that the projection will preserve as much variance as possible. For example, in [Figure 8-2](#) the 3D dataset is projected down to the 2D plane defined by the first two principal components, preserving a large part of the dataset's variance. As a result, the 2D projection looks very much like the original 3D dataset.

To project the training set onto the hyperplane and obtain a reduced dataset $\mathbf{X}_{d\text{-proj}}$ of dimensionality d , compute the matrix multiplication of the training set matrix \mathbf{X} by the matrix \mathbf{W}_d , defined as the matrix containing the first d columns of \mathbf{V} , as shown in [Equation 8-2](#).

Equation 8-2. Projecting the training set down to d dimensions

$$\mathbf{X}_{d\text{-proj}} = \mathbf{X}\mathbf{W}_d$$

The following Python code projects the training set onto the plane defined by the first two principal components:

```
W2 = Vt[:2].T
X2D = X_centered @ W2
```

There you have it! You now know how to reduce the dimensionality of any dataset by projecting it down to any number of dimensions, while preserving as much variance as possible.

Using Scikit-Learn

Scikit-Learn's PCA class uses SVD to implement PCA, just like we did earlier in this chapter. The following code applies PCA to reduce the dimensionality of the dataset down to two dimensions (note that it automatically takes care of centering the data):

```
from sklearn.decomposition import PCA
```

```
pca = PCA(n_components=2)
X2D = pca.fit_transform(X)
```

After fitting the PCA transformer to the dataset, its `components_` attribute holds the transpose of \mathbf{W}_d : it contains one row for each of the first d principal components.

Explained Variance Ratio

Another useful piece of information is the *explained variance ratio* of each principal component, available via the `explained_variance_ratio_` variable. The ratio indicates the proportion of the dataset's variance that lies along each principal component. For example, let's look at the explained variance ratios of the first two components of the 3D dataset represented in [Figure 8-2](#):

```
>>> pca.explained_variance_ratio_
array([0.7578477 , 0.15186921])
```

This output tells you that about 76% of the dataset's variance lies along the first PC, and about 15% lies along the second PC. This leaves about 9% for the third PC, so it is reasonable to assume that the third PC probably carries little information.

Choosing the Right Number of Dimensions

Instead of arbitrarily choosing the number of dimensions to reduce down to, it is simpler to choose the number of dimensions that add up to a sufficiently large portion of the variance (e.g., 95%). Unless, of course, you are reducing dimensionality for data visualization, in which case you will want to reduce the dimensionality down to 2 or 3.

The following code loads and splits the MNIST dataset (introduced in [Chapter 3](#)) and performs PCA without reducing dimensionality, then it computes the minimum number of dimensions required to preserve 95% of the training set's variance:

```
from sklearn.datasets import fetch_openml

mnist = fetch_openml('mnist_784', as_frame=False)
X_train, y_train = mnist.data[:60_000], mnist.target[:60_000]
X_test, y_test = mnist.data[60_000:], mnist.target[60_000:]

pca = PCA()
pca.fit(X_train)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1 # d equals 154
```

You could then set `n_components=d` and run PCA again. But there is a better option: instead of specifying the number of principal components you want to preserve, you can set `n_components` to be a float between 0.0 and 1.0, indicating the ratio of variance you wish to preserve:

```
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X_train)
```

The actual number of components is determined during training, and it is stored in the `n_components_` attribute:

```
>>> pca.n_components_
154
```

Yet another option is to plot the explained variance as a function of the number of dimensions (simply plot `cumsum`; see [Figure 8-8](#)). There will usually be an elbow in the curve, where the explained variance stops growing fast. In this case, you can see that reducing the dimensionality down to about 100 dimensions wouldn't lose too much explained variance.

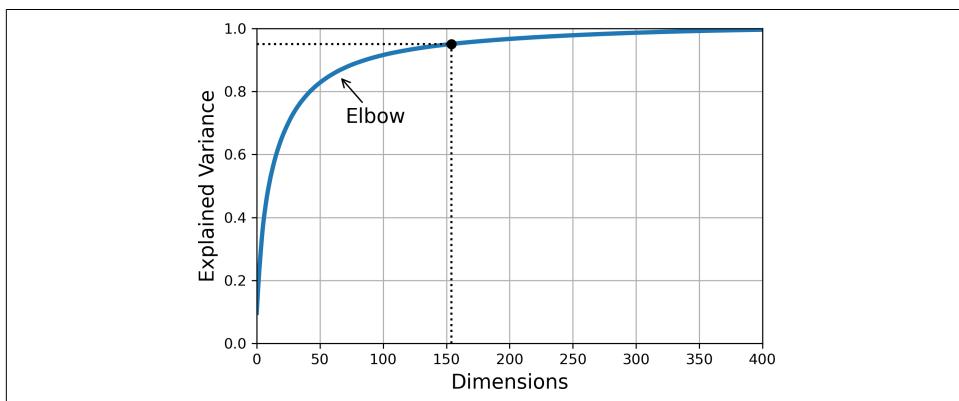


Figure 8-8. Explained variance as a function of the number of dimensions

Lastly, if you are using dimensionality reduction as a preprocessing step for a supervised learning task (e.g., classification), then you can tune the number of dimensions as you would any other hyperparameter (see [Chapter 2](#)). For example, the following code example creates a two-step pipeline, first reducing dimensionality using PCA, then classifying using a Random Forest. Next, it uses `RandomizedSearchCV` to find a good combination of hyperparameters for both PCA and the Random Forest classifier. This example does a quick search, tuning only two hyperparameters, training on just 1,000 instances, and running for just 10 iterations, but feel free to do a more thorough search if you have the time.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV
from sklearn.pipeline import make_pipeline

clf = make_pipeline(PCA(random_state=42),
                    RandomForestClassifier(random_state=42))
param_distrib = {
    "pca_n_components": np.arange(10, 80),
```

```
"randomforestclassifier__n_estimators": np.arange(50, 500)
}
rnd_search = RandomizedSearchCV(clf, param_distrib, n_iter=10, cv=3,
                                 random_state=42)
rnd_search.fit(X_train[:1000], y_train[:1000])
```

Let's look at the best hyperparameters found:

```
>>> print(rnd_search.best_params_)
{'randomforestclassifier__n_estimators': 465, 'pca__n_components': 23}
```

It's interesting to note how low the optimal number of components is: we reduced a 784-dimensional dataset to just 23 dimensions! This is tied to the fact that we used a Random Forest, which is a pretty powerful model. If we used a linear model instead, such as an `SGDClassifier`, the search would find that we need to preserve more dimensions (about 70).

PCA for Compression

After dimensionality reduction, the training set takes up much less space. For example, after applying PCA to the MNIST dataset while preserving 95% of its variance, we are left with 154 features, instead of the original 784 features. So the dataset is now less than 20% of its original size, and we only lost 5% of its variance! This is a reasonable compression ratio, and it's easy to see how such a size reduction would speed up a classification algorithm tremendously.

It is also possible to decompress the reduced dataset back to 784 dimensions by applying the inverse transformation of the PCA projection. This won't give you back the original data, since the projection lost a bit of information (within the 5% variance that was dropped), but it will likely be close to the original data. The mean squared distance between the original data and the reconstructed data (compressed and then decompressed) is called the *reconstruction error*.

The `inverse_transform()` method lets us decompress the reduced MNIST dataset back to 784 dimensions:

```
X_recovered = pca.inverse_transform(X_reduced)
```

Figure 8-9 shows a few digits from the original training set (on the left), and the corresponding digits after compression and decompression. You can see that there is a slight image quality loss, but the digits are still mostly intact.

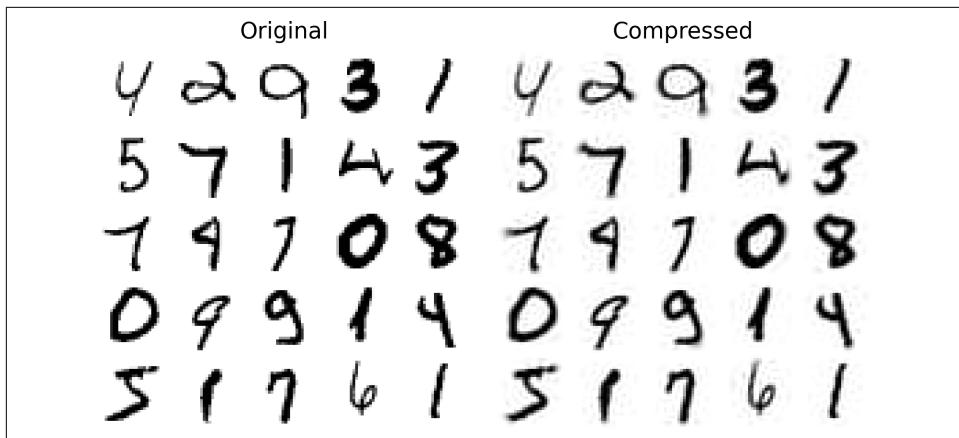


Figure 8-9. MNIST compression that preserves 95% of the variance

The equation of the inverse transformation is shown in [Equation 8-3](#).

Equation 8-3. PCA inverse transformation, back to the original number of dimensions

$$\mathbf{X}_{\text{recovered}} = \mathbf{X}_{d\text{-proj}} \mathbf{W}_d^\top$$

Randomized PCA

If you set the `svd_solver` hyperparameter to "randomized", Scikit-Learn uses a stochastic algorithm called *Randomized PCA* that quickly finds an approximation of the first d principal components. Its computational complexity is $O(m \times d^2) + O(d^3)$, instead of $O(m \times n^2) + O(n^3)$ for the full SVD approach, so it is dramatically faster than full SVD when d is much smaller than n :

```
rnd_pca = PCA(n_components=154, svd_solver="randomized", random_state=42)
X_reduced = rnd_pca.fit_transform(X_train)
```



By default, `svd_solver` is actually set to "auto": Scikit-Learn automatically uses the randomized PCA algorithm if $\max(m, n) > 500$ and `n_components` is an integer smaller than 80% of $\min(m, n)$, or else it uses the full SVD approach. So the preceding code would use the randomized PCA algorithm even if you removed `svd_solver="randomized"` argument, since $154 < 0.8 \times 784$. If you want to force Scikit-Learn to use full SVD for a slightly more precise result, you can set the `svd_solver` hyperparameter to "full".

Incremental PCA

One problem with the preceding implementations of PCA is that they require the whole training set to fit in memory in order for the algorithm to run. Fortunately, *Incremental PCA* (IPCA) algorithms have been developed. They allow you to split the training set into mini-batches and feed an IPCA algorithm one mini-batch at a time. This is useful for large training sets and for applying PCA online (i.e., on the fly, as new instances arrive).

The following code splits the MNIST training set into 100 mini-batches (using NumPy's `array_split()` function) and feeds them to Scikit-Learn's `IncrementalPCA` class⁵ to reduce the dimensionality of the MNIST dataset down to 154 dimensions, just like before. Note that you must call the `partial_fit()` method with each mini-batch, rather than the `fit()` method with the whole training set:

```
from sklearn.decomposition import IncrementalPCA

n_batches = 100
inc_pca = IncrementalPCA(n_components=154)
for X_batch in np.array_split(X_train, n_batches):
    inc_pca.partial_fit(X_batch)

X_reduced = inc_pca.transform(X_train)
```

Alternatively, you can use NumPy's `memmap` class, which allows you to manipulate a large array stored in a binary file on disk as if it were entirely in memory; the class loads only the data it needs in memory, when it needs it. To demonstrate this, let's first create a `memmap` file and copy the MNIST training set to it, then call `flush()` to ensure that any data still in cache gets saved to disk. In real life, `X_train` would typically not fit in memory so you would load it chunk by chunk and save each chunk to the right part of the `memmap` array:

```
filename = "my_mnist.memmap"
X_mmap = np.memmap(filename, dtype='float32', mode='write', shape=X_train.shape)
X_mmap[:] = X_train # could be a loop instead, saving the data chunk by chunk
X_mmap.flush()
```

Next, we can load the `memmap` file and use it like a regular NumPy array. Let's use the `IncrementalPCA` class to reduce its dimensionality. Since this algorithm uses only a small part of the array at any given time, memory usage remains under control. This makes it possible to call the usual `fit()` method instead of `partial_fit()`, which is quite convenient:

```
X_mmap = np.memmap(filename, dtype="float32", mode="readonly").reshape(-1, 784)
batch_size = X_mmap.shape[0] // n_batches
```

⁵ Scikit-Learn uses the algorithm described in David A. Ross et al., “Incremental Learning for Robust Visual Tracking,” *International Journal of Computer Vision* 77, no. 1–3 (2008): 125–141.

```
inc_pca = IncrementalPCA(n_components=154, batch_size=batch_size)
inc_pca.fit(X_mmmap)
```



Only the raw binary data is saved to disk, so you need to specify the data type and shape of the array when you load it. If you omit the shape, `np.memmap()` returns a 1D array.

For very high-dimensional datasets, PCA can be too slow. As we saw earlier, even if you use Randomized PCA, its computational complexity is still $O(m \times d^2) + O(d^3)$, so the target number of dimensions d must not be too large. If you are dealing with a dataset with tens of thousands of features or more (e.g., images), then training may become much too slow: in this case, you should consider using Random Projection instead.

Random Projection

As its name suggests, the Random Projection algorithm projects the data to a lower-dimensional space using a random linear projection. This may sound crazy, but it turns out that such a random projection is actually very likely to preserve distances fairly well, as was demonstrated mathematically by William B. Johnson and Joram Lindenstrauss in a famous lemma. So two similar instances will remain similar after the projection, and two very different instances will remain very different.

Obviously, the more dimensions you drop, the more information is lost, and the more distances get distorted. So how can you choose the optimal number of dimensions? Well, Johnson and Lindenstrauss came up with an equation that determines the minimum number of dimensions to preserve in order to ensure—with high probability—that distances won’t change by more than a given tolerance. For example, if you have a dataset containing $m = 5,000$ instances with $n = 20,000$ features each, and you don’t want the squared distance between any two instances to change by more than $\epsilon = 10\%$,⁶ then you should project the data down to d dimensions, with $d \geq 4 \log(m) / (\frac{1}{2} \epsilon^2 - \frac{1}{3} \epsilon^3)$, which is 7,300 dimensions. That’s quite a significant dimensionality reduction! Notice that the equation does not use n , it only relies on m and ϵ . This equation is implemented by the `johnson_lindenstrauss_min_dim()` function:

```
>>> from sklearn.random_projection import johnson_lindenstrauss_min_dim
>>> m, ε = 5_000, 0.1
>>> d = johnson_lindenstrauss_min_dim(m, eps=ε)
>>> d
7300
```

⁶ ϵ is the greek letter epsilon, often used for tiny values.

Now we can just generate a random matrix \mathbf{P} of shape $[d, n]$, where each item is sampled randomly from a Gaussian distribution with mean 0 and variance $1/d$, and we use it to project a dataset from n dimensions down to d :

```
n = 20_000
np.random.seed(42)
P = np.random.randn(d, n) / np.sqrt(d) # std dev = square root of variance

X = np.random.randn(m, n) # generate a fake dataset
X_reduced = X @ P.T
```

That's all there is to it! It's simple and efficient, and no training is required: the only thing the algorithm needs to create the random matrix is the dataset's shape, that's it. The data itself is not used at all.

Scikit-Learn offers a `GaussianRandomProjection` class to do exactly what we just did: when you call its `fit()` method, it uses `johnson_lindenstrauss_min_dim()` to determine the output dimensionality, then it generates a random matrix, which it stores in the `components_` attribute. Then when you call `transform()`, it uses this matrix to perform the projection. When creating the transformer, you can set `eps` if you want to tweak ϵ (it defaults to 0.1), and `n_components` if you want to force a specific target dimensionality d . The following code example gives the same result as above (you can also verify that `gaussian_rnd_proj.components_` is equal to \mathbf{P}):

```
from sklearn.random_projection import GaussianRandomProjection

gaussian_rnd_proj = GaussianRandomProjection(eps=epsilon, random_state=42)
X_reduced = gaussian_rnd_proj.fit_transform(X) # same result as above
```

Scikit-Learn provides a second random projection transformer: `SparseRandomProjection`. It determines the target dimensionality in the same way, it generates a random matrix of the same shape, and it performs the projection identically. The main difference is that the random matrix is sparse. This means it uses much less memory: about 25 MB instead of almost 1.2GB in the preceding example! And it's also much faster, both to generate the random matrix and to reduce dimensionality: about 50% faster on the preceding example. Moreover, if the input is sparse, the transformation keeps it sparse (unless you set `dense_output=True`). Lastly, it enjoys the same distance-preserving property as the previous approach, and the quality of the dimensionality reduction is comparable. In short, it's usually preferable to use this transformer instead of the first one, especially for large or sparse datasets.

The ratio r of nonzero items in the sparse random matrix is called its *density*. By default, it is equal to $1/\sqrt{n}$. With 20,000 features, this means that only one in ~ 141 cells in the random matrix is nonzero: that's quite sparse! You can set the `density` hyperparameter to another value if you prefer. Each cell in the sparse random matrix has a probability r of being nonzero, and each nonzero value is either $-v$ or $+v$ (both equally likely), where $v = 1/\sqrt{dr}$.



Random Projection is not always used to reduce the dimensionality of large datasets. For example, a [2017 paper](#)⁷ by Sanjoy Dasgupta et al. showed that the brain of fruit flies implements an analog of Random Projection to map dense low-dimensional olfactory inputs to sparse high-dimensional binary outputs: for each odor, only a small fraction of the output neurons get activated, but similar odors activate many of the same neurons. This is similar to a well-known algorithm called *Locality Sensitive Hashing* (LSH) which is typically used in search engines to group similar documents.

If you want to perform the inverse transform, you first need to compute the pseudo-inverse of the components matrix, using SciPy's `pinv()` function, then multiply the reduced data by the transpose of the pseudo-inverse:

```
components_pinv = np.linalg.pinv(gaussian_rnd_proj.components_)
X_recovered = X_reduced @ components_pinv.T
```



Computing the pseudo-inverse may take a very long time if the components matrix is large, as the computational complexity of `pinv()` is $O(dn^2)$ if $d < n$, or $O(nd^2)$ otherwise.

In summary, Random Projection is a simple, fast, memory-efficient and surprisingly powerful dimensionality reduction algorithm that you should keep in mind, especially when you deal with high-dimensional datasets.

LLE

Locally Linear Embedding (LLE)⁸ is a *nonlinear dimensionality reduction* (NLDR) technique. It is a Manifold Learning technique that does not rely on projections, unlike PCA and Random Projection. In a nutshell, LLE works by first measuring how each training instance linearly relates to its nearest neighbors, and then looking for a low-dimensional representation of the training set where these local relationships are best preserved (more details shortly). This approach makes it particularly good at unrolling twisted manifolds, especially when there is not too much noise.

The following code makes a Swiss roll then uses Scikit-Learn's `LocallyLinearEmbedding` class to unroll it:

⁷ Sanjoy Dasgupta et al., “A neural algorithm for a fundamental computing problem,” *Science* 358, no. 6364 (2017): 793–796.

⁸ Sam T. Roweis and Lawrence K. Saul, “Nonlinear Dimensionality Reduction by Locally Linear Embedding,” *Science* 290, no. 5500 (2000): 2323–2326.

```

from sklearn.datasets import make_swiss_roll
from sklearn.manifold import LocallyLinearEmbedding

X_swiss, t = make_swiss_roll(n_samples=1000, noise=0.2, random_state=42)
lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10, random_state=42)
X_unrolled = lle.fit_transform(X_swiss)

```

The variable t is a 1D NumPy array containing the position of each instance along the rolled axis of Swiss roll. We don't use it in this example, but it can be used as a target for a non-linear regression task.

The resulting 2D dataset is shown in [Figure 8-10](#). As you can see, the Swiss roll is completely unrolled, and the distances between instances are locally well preserved. However, distances are not preserved on a larger scale: the unrolled Swiss roll should be a rectangle, not this kind of stretched and twisted band. Nevertheless, LLE did a pretty good job at modeling the manifold.

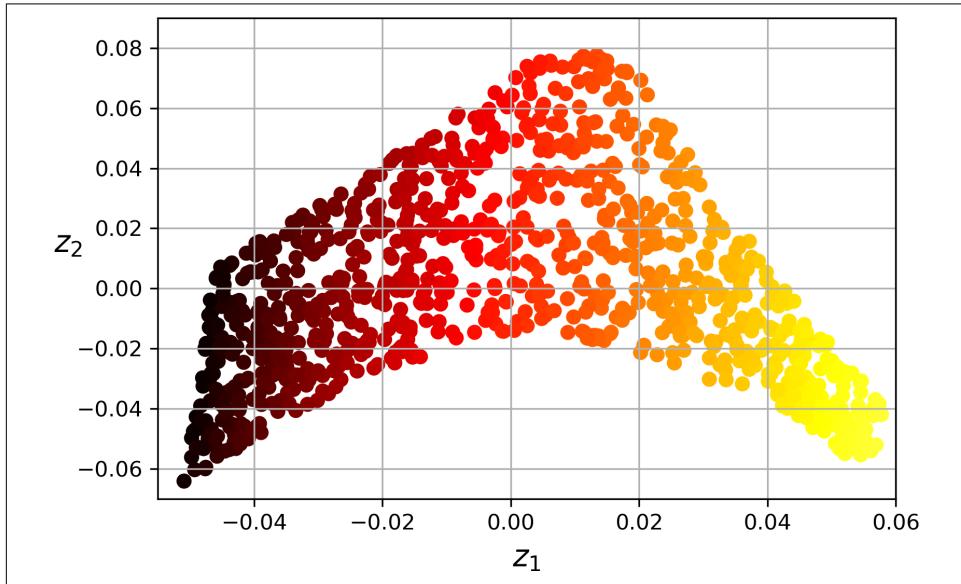


Figure 8-10. Unrolled Swiss roll using LLE

Here's how LLE works: for each training instance $\mathbf{x}^{(i)}$, the algorithm identifies its k nearest neighbors (in the preceding code $k = 10$), then tries to reconstruct $\mathbf{x}^{(i)}$ as a linear function of these neighbors. More specifically, it tries to find the weights $w_{i,j}$ such that the squared distance between $\mathbf{x}^{(i)}$ and $\sum_{j=1}^m w_{i,j} \mathbf{x}^{(j)}$ is as small as possible, assuming $w_{i,j} = 0$ if $\mathbf{x}^{(j)}$ is not one of the k nearest neighbors of $\mathbf{x}^{(i)}$. Thus the first step of LLE is the constrained optimization problem described in [Equation 8-4](#), where

\mathbf{W} is the weight matrix containing all the weights $w_{i,j}$. The second constraint simply normalizes the weights for each training instance $\mathbf{x}^{(i)}$.

Equation 8-4. LLE step one: linearly modeling local relationships

$$\begin{aligned}\widehat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmin}} \sum_{i=1}^m \left(\mathbf{x}^{(i)} - \sum_{j=1}^m w_{i,j} \mathbf{x}^{(j)} \right)^2 \\ \text{subject to } \begin{cases} w_{i,j} = 0 & \text{if } \mathbf{x}^{(j)} \text{ is not one of the } k \text{ n.n. of } \mathbf{x}^{(i)} \\ \sum_{j=1}^m w_{i,j} = 1 & \text{for } i = 1, 2, \dots, m \end{cases}\end{aligned}$$

After this step, the weight matrix $\widehat{\mathbf{W}}$ (containing the weights $\widehat{w}_{i,j}$) encodes the local linear relationships between the training instances. The second step is to map the training instances into a d -dimensional space (where $d < n$) while preserving these local relationships as much as possible. If $\mathbf{z}^{(i)}$ is the image of $\mathbf{x}^{(i)}$ in this d -dimensional space, then we want the squared distance between $\mathbf{z}^{(i)}$ and $\sum_{j=1}^m \widehat{w}_{i,j} \mathbf{z}^{(j)}$ to be as small as possible. This idea leads to the unconstrained optimization problem described in [Equation 8-5](#). It looks very similar to the first step, but instead of keeping the instances fixed and finding the optimal weights, we are doing the reverse: keeping the weights fixed and finding the optimal position of the instances' images in the low-dimensional space. Note that \mathbf{Z} is the matrix containing all $\mathbf{z}^{(i)}$.

Equation 8-5. LLE step two: reducing dimensionality while preserving relationships

$$\widehat{\mathbf{Z}} = \underset{\mathbf{Z}}{\operatorname{argmin}} \sum_{i=1}^m \left(\mathbf{z}^{(i)} - \sum_{j=1}^m \widehat{w}_{i,j} \mathbf{z}^{(j)} \right)^2$$

Scikit-Learn's LLE implementation has the following computational complexity: $O(m \log(m)n \log(k))$ for finding the k nearest neighbors, $O(mnk^3)$ for optimizing the weights, and $O(dm^2)$ for constructing the low-dimensional representations. Unfortunately, the m^2 in the last term makes this algorithm scale poorly to very large datasets.

As you can see, LLE is quite different from the projection techniques, and it's significantly more complex, but it can also construct much better low-dimensional representations, especially if the data is non-linear.

Other Dimensionality Reduction Techniques

Before we conclude this chapter, let's take a quick look at a few other popular dimensionality reduction techniques available in Scikit-Learn:

`sklearn.manifold.MDS`

Multidimensional Scaling (MDS) reduces dimensionality while trying to preserve the distances between the instances. Random Projection does that for high-dimensional data, but it doesn't work well on low-dimensional data.

`sklearn.manifold.Isomap`

Isomap creates a graph by connecting each instance to its nearest neighbors, then reduces dimensionality while trying to preserve the *geodesic distances* between the instances. The geodesic distance between two nodes in a graph is the number of nodes on the shortest path between these nodes.

`sklearn.manifold.TSNE`

t-Distributed Stochastic Neighbor Embedding (t-SNE) reduces dimensionality while trying to keep similar instances close and dissimilar instances apart. It is mostly used for visualization, in particular to visualize clusters of instances in high-dimensional space. For example, in the exercises at the end of this chapter you will use t-SNE to visualize a 2D map of the MNIST images.

`sklearn.discriminant_analysis.LinearDiscriminantAnalysis`

Linear Discriminant Analysis (LDA) is a linear classification algorithm, and during training it learns the most discriminative axes between the classes. These axes can then be used to define a hyperplane onto which to project the data. The benefit of this approach is that the projection will keep classes as far apart as possible, so LDA is a good technique to reduce dimensionality before running another classification algorithm (unless LDA alone is sufficient).

Figure 8-11 shows the results of MDS, Isomap and t-SNE on the Swiss roll. MDS manages to flatten the Swiss roll without losing its global curvature, while Isomap drops it entirely. Depending on the downstream task, preserving the large scale structure may be good or bad. As for t-SNE, it did a reasonable job in flattening the Swiss roll, preserving a bit of curvature, and it also amplified clusters, tearing the roll apart. Again, this might be good or bad, depending on the downstream task.

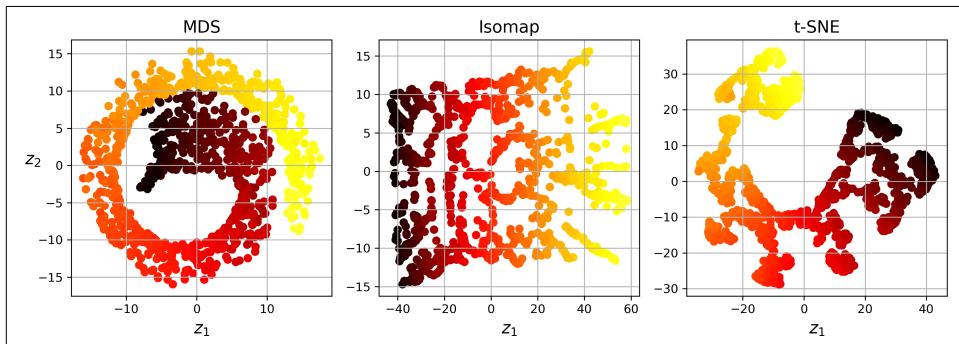


Figure 8-11. Using various techniques to reduce the Swiss roll to 2D

Exercises

1. What are the main motivations for reducing a dataset's dimensionality? What are the main drawbacks?
2. What is the curse of dimensionality?
3. Once a dataset's dimensionality has been reduced, is it possible to reverse the operation? If so, how? If not, why?
4. Can PCA be used to reduce the dimensionality of a highly nonlinear dataset?
5. Suppose you perform PCA on a 1,000-dimensional dataset, setting the explained variance ratio to 95%. How many dimensions will the resulting dataset have?
6. In what cases would you use regular PCA, Incremental PCA, Randomized PCA, or Random Projection?
7. How can you evaluate the performance of a dimensionality reduction algorithm on your dataset?
8. Does it make any sense to chain two different dimensionality reduction algorithms?
9. Load the MNIST dataset (introduced in [Chapter 3](#)) and split it into a training set and a test set (take the first 60,000 instances for training, and the remaining 10,000 for testing). Train a Random Forest classifier on the dataset and time how long it takes, then evaluate the resulting model on the test set. Next, use PCA to reduce the dataset's dimensionality, with an explained variance ratio of 95%. Train a new Random Forest classifier on the reduced dataset and see how long it takes. Was training much faster? Next, evaluate the classifier on the test set. How does it compare to the previous classifier? Try again with an `SGDClassifier`. How much does PCA help now?
10. Use t-SNE to reduce the first 5,000 images of the MNIST dataset down to two dimensions and plot the result using Matplotlib. You can use a scatterplot using 10 different colors to represent each image's target class. Alternatively, you can replace each dot in the scatterplot with the corresponding instance's class (a digit from 0 to 9), or even plot scaled-down versions of the digit images themselves (if you plot all digits, the visualization will be too cluttered, so you should either draw a random sample or plot an instance only if no other instance has already been plotted at a close distance). You should get a nice visualization with well-separated clusters of digits. Try using other dimensionality reduction algorithms such as PCA, LLE, or MDS and compare the resulting visualizations.

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab3>.

Unsupervised Learning Techniques

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 9th chapter of the final book. The GitHub repo is [https://git
hub.com/ageron/handsonml3](https://github.com/ageron/handsonml3).

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

Although most of the applications of Machine Learning today are based on supervised learning (and as a result, this is where most of the investments go to), the vast majority of the available data is unlabeled: we have the input features X , but we do not have the labels y . The computer scientist Yann LeCun famously said that “if intelligence was a cake, unsupervised learning would be the cake, supervised learning would be the icing on the cake, and reinforcement learning would be the cherry on the cake.” In other words, there is a huge potential in unsupervised learning that we have only barely started to sink our teeth into.

Say you want to create a system that will take a few pictures of each item on a manufacturing production line and detect which items are defective. You can fairly easily create a system that will take pictures automatically, and this might give you thousands of pictures every day. You can then build a reasonably large dataset in just a few weeks. But wait, there are no labels! If you want to train a regular binary classifier that will predict whether an item is defective or not, you will need to label every single picture as “defective” or “normal.” This will generally require human experts to

sit down and manually go through all the pictures. This is a long, costly, and tedious task, so it will usually only be done on a small subset of the available pictures. As a result, the labeled dataset will be quite small, and the classifier's performance will be disappointing. Moreover, every time the company makes any change to its products, the whole process will need to be started over from scratch. Wouldn't it be great if the algorithm could just exploit the unlabeled data without needing humans to label every picture? Enter unsupervised learning.

In [Chapter 8](#) we looked at the most common unsupervised learning task: dimensionality reduction. In this chapter we will look at a few more unsupervised tasks:

Clustering

The goal is to group similar instances together into *clusters*. Clustering is a great tool for data analysis, customer segmentation, recommender systems, search engines, image segmentation, semi-supervised learning, dimensionality reduction, and more.

Anomaly detection (also called outlier detection)

The objective is to learn what “normal” data looks like, and then use that to detect abnormal instances. These instances are called *anomalies*, or *outliers*, while the normal instances are called *inliers*. Anomaly detection is useful in a wide variety of applications, such as fraud detection, detecting defective products in manufacturing, identifying a new trend in a time series, or removing outliers from a dataset before training another model, which can significantly improve the performance of the resulting model.

Density estimation

This is the task of estimating the *probability density function* (PDF) of the random process that generated the dataset. Density estimation is commonly used for anomaly detection: instances located in very low-density regions are likely to be anomalies. It is also useful for data analysis and visualization.

Ready for some cake? We will start with two clustering algorithms, K-Means and DBSCAN, then we'll discuss Gaussian mixture models and see how they can be used for density estimation, clustering, and anomaly detection.

Clustering Algorithms: K-Means and DBSCAN

As you enjoy a hike in the mountains, you stumble upon a plant you have never seen before. You look around and you notice a few more. They are not identical, yet they are sufficiently similar for you to know that they most likely belong to the same species (or at least the same genus). You may need a botanist to tell you what species that is, but you certainly don't need an expert to identify groups of similar-looking objects. This is called *clustering*: it is the task of identifying similar instances and assigning them to *clusters*, or groups of similar instances.

Just like in classification, each instance gets assigned to a group. However, unlike classification, clustering is an unsupervised task. Consider Figure 9-1: on the left is the iris dataset (introduced in Chapter 4), where each instance's species (i.e., its class) is represented with a different marker. It is a labeled dataset, for which classification algorithms such as Logistic Regression, SVMs, or Random Forest classifiers are well suited. On the right is the same dataset, but without the labels, so you cannot use a classification algorithm anymore. This is where clustering algorithms step in: many of them can easily detect the lower-left cluster. It is also quite easy to see with our own eyes, but it is not so obvious that the upper-right cluster is composed of two distinct sub-clusters. That said, the dataset has two additional features (sepal length and width), not represented here, and clustering algorithms can make good use of all features, so in fact they identify the three clusters fairly well (e.g., using a Gaussian mixture model, only 5 instances out of 150 are assigned to the wrong cluster).

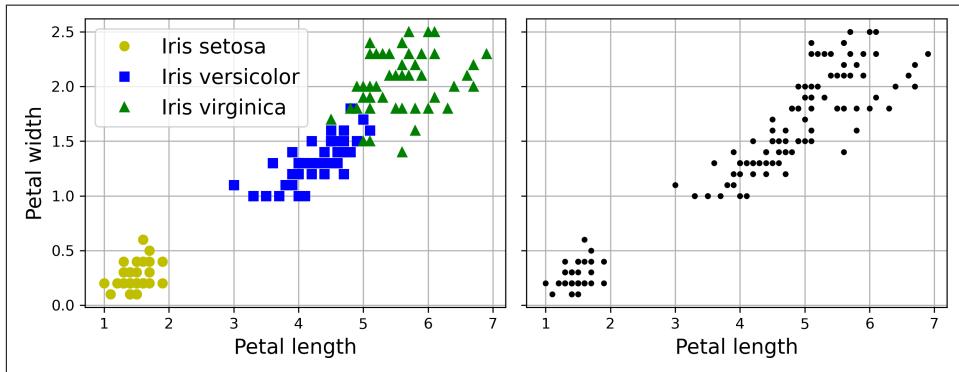


Figure 9-1. Classification (left) versus clustering (right)

Clustering is used in a wide variety of applications, including these:

For customer segmentation

You can cluster your customers based on their purchases and their activity on your website. This is useful to understand who your customers are and what they need, so you can adapt your products and marketing campaigns to each segment. For example, customer segmentation can be useful in *recommender systems* to suggest content that other users in the same cluster enjoyed.

For data analysis

When you analyze a new dataset, it can be helpful to run a clustering algorithm, and then analyze each cluster separately.

As a dimensionality reduction technique

Once a dataset has been clustered, it is usually possible to measure each instance's *affinity* with each cluster: affinity is any measure of how well an instance fits into a cluster. Each instance's feature vector \mathbf{x} can then be replaced with the vector of

its cluster affinities. If there are k clusters, then this vector is k -dimensional. This vector is typically much lower-dimensional than the original feature vector, but it can preserve enough information for further processing.

For feature engineering

The cluster affinities can often be useful as extra features. For example, we used K-Means in [Chapter 2](#) to add geographic cluster affinity features to the California housing dataset, and they helped us get better performance.

For anomaly detection (also called outlier detection)

Any instance that has a low affinity to all the clusters is likely to be an anomaly. For example, if you have clustered the users of your website based on their behavior, you can detect users with unusual behavior, such as an unusual number of requests per second.

For semi-supervised learning

If you only have a few labels, you could perform clustering and propagate the labels to all the instances in the same cluster. This technique can greatly increase the number of labels available for a subsequent supervised learning algorithm, and thus improve its performance.

For search engines

Some search engines let you search for images that are similar to a reference image. To build such a system, you would first apply a clustering algorithm to all the images in your database; similar images would end up in the same cluster. Then when a user provides a reference image, all you need to do is use the trained clustering model to find this image's cluster, and you can then simply return all the images from this cluster.

To segment an image

By clustering pixels according to their color, then replacing each pixel's color with the mean color of its cluster, it is possible to considerably reduce the number of different colors in the image. Image segmentation is used in many object detection and tracking systems, as it makes it easier to detect the contour of each object.

There is no universal definition of what a cluster is: it really depends on the context, and different algorithms will capture different kinds of clusters. Some algorithms look for instances centered around a particular point, called a *centroid*. Others look for continuous regions of densely packed instances: these clusters can take on any shape. Some algorithms are hierarchical, looking for clusters of clusters. And the list goes on.

In this section, we will look at two popular clustering algorithms, K-Means and DBSCAN, and explore some of their applications, such as nonlinear dimensionality reduction, semi-supervised learning, and anomaly detection.

K-Means

Consider the unlabeled dataset represented in [Figure 9-2](#): you can clearly see five blobs of instances. The K-Means algorithm is a simple algorithm capable of clustering this kind of dataset very quickly and efficiently, often in just a few iterations. It was proposed by Stuart Lloyd at Bell Labs in 1957 as a technique for pulse-code modulation, but it was only published outside of the company in 1982.¹ In 1965, Edward W. Forgy had published virtually the same algorithm, so K-Means is sometimes referred to as Lloyd–Forgy.

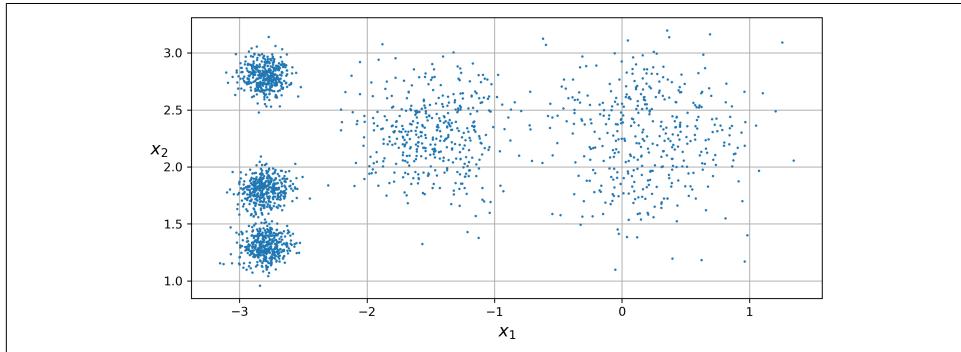


Figure 9-2. An unlabeled dataset composed of five blobs of instances

Let's train a K-Means clusterer on this dataset. It will try to find each blob's center and assign each instance to the closest blob:

```
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

X, y = make_blobs([...]) # make the blobs: y contains the cluster ids, but we
                        # will not use them, it's what we want to predict
k = 5
kmeans = KMeans(n_clusters=k, random_state=42)
y_pred = kmeans.fit_predict(X)
```

Note that you have to specify the number of clusters k that the algorithm must find. In this example, it is pretty obvious from looking at the data that k should be set to 5, but in general it is not that easy. We will discuss this shortly.

Each instance was assigned to one of the five clusters. In the context of clustering, an instance's *label* is the index of the cluster that this instance gets assigned to by the algorithm: this is not to be confused with the class labels in classification, which are used as targets (remember that clustering is an unsupervised learning task). The

¹ Stuart P. Lloyd, "Least Squares Quantization in PCM," *IEEE Transactions on Information Theory* 28, no. 2 (1982): 129–137.

KMeans instance preserves the predicted labels of the instances it was trained on, available via the `labels_` instance variable:

```
>>> y_pred
array([4, 0, 1, ..., 2, 1, 0], dtype=int32)
>>> y_pred is kmeans.labels_
True
```

We can also take a look at the five centroids that the algorithm found:

```
>>> kmeans.cluster_centers_
array([[-2.80389616,  1.80117999],
       [ 0.20876306,  2.25551336],
       [-2.79290307,  2.79641063],
       [-1.46679593,  2.28585348],
       [-2.80037642,  1.30082566]])
```

You can easily assign new instances to the cluster whose centroid is closest:

```
>>> import numpy as np
>>> X_new = np.array([[0, 2], [3, 2], [-3, 3], [-3, 2.5]])
>>> kmeans.predict(X_new)
array([1, 1, 2, 2], dtype=int32)
```

If you plot the cluster's decision boundaries, you get a Voronoi tessellation: see [Figure 9-3](#), where each centroid is represented with an X.

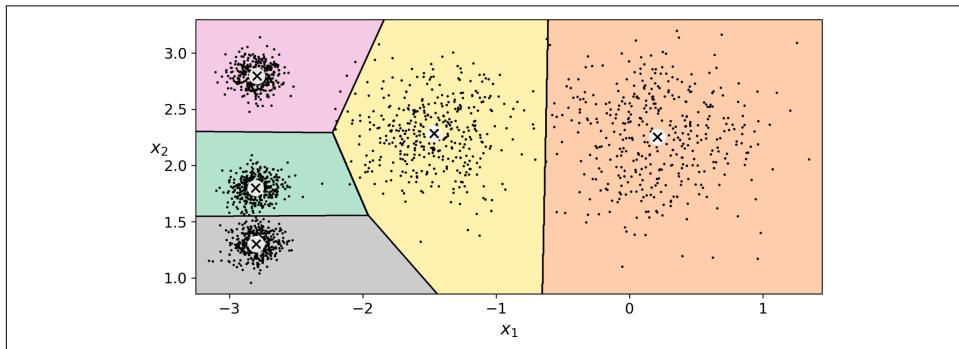


Figure 9-3. K-Means decision boundaries (Voronoi tessellation)

The vast majority of the instances were clearly assigned to the appropriate cluster, but a few instances were probably mislabeled, especially near the boundary between the top-left cluster and the central cluster. Indeed, the K-Means algorithm does not behave very well when the blobs have very different diameters because all it cares about when assigning an instance to a cluster is the distance to the centroid.

Instead of assigning each instance to a single cluster, which is called *hard clustering*, it can be useful to give each instance a score per cluster, which is called *soft clustering*. The score can be the distance between the instance and the centroid; conversely, it

can be a similarity score (or affinity), such as the Gaussian Radial Basis Function we used in [Chapter 2](#). In the `KMeans` class, the `transform()` method measures the distance from each instance to every centroid:

```
>>> kmeans.transform(X_new).round(2)
array([[2.81, 0.33, 2.9 , 1.49, 2.89],
       [5.81, 2.8 , 5.85, 4.48, 5.84],
       [1.21, 3.29, 0.29, 1.69, 1.71],
       [0.73, 3.22, 0.36, 1.55, 1.22]])
```

In this example, the first instance in `X_new` is located at a distance of about 2.81 from the first centroid, 0.33 from the second centroid, 2.90 from the third centroid, 1.49 from the fourth centroid, and 2.89 from the fifth centroid. If you have a high-dimensional dataset and you transform it this way, you end up with a k -dimensional dataset: this transformation can be a very efficient nonlinear dimensionality reduction technique. Alternatively, you can use these distances as extra features to train another model, as we did in [Chapter 2](#).

The K-Means algorithm

So, how does the algorithm work? Well, suppose you were given the centroids. You could easily label all the instances in the dataset by assigning each of them to the cluster whose centroid is closest. Conversely, if you were given all the instance labels, you could easily locate each cluster's centroid by computing the mean of the instances in that cluster. But you are given neither the labels nor the centroids, so how can you proceed? Well, just start by placing the centroids randomly (e.g., by picking k instances at random from the dataset and using their locations as centroids). Then label the instances, update the centroids, label the instances, update the centroids, and so on until the centroids stop moving. The algorithm is guaranteed to converge in a finite number of steps (usually quite small). That's because the mean squared distance between the instances and their closest centroid can only go down at each step, and since it cannot be negative, it's guaranteed to converge.

You can see the algorithm in action in [Figure 9-4](#): the centroids are initialized randomly (top left), then the instances are labeled (top right), then the centroids are updated (center left), the instances are relabeled (center right), and so on. As you can see, in just three iterations, the algorithm has reached a clustering that seems close to optimal.



The computational complexity of the algorithm is generally linear with regard to the number of instances m , the number of clusters k , and the number of dimensions n . However, this is only true when the data has a clustering structure. If it does not, then in the worst-case scenario the complexity can increase exponentially with the number of instances. In practice, this rarely happens, and K-Means is generally one of the fastest clustering algorithms.

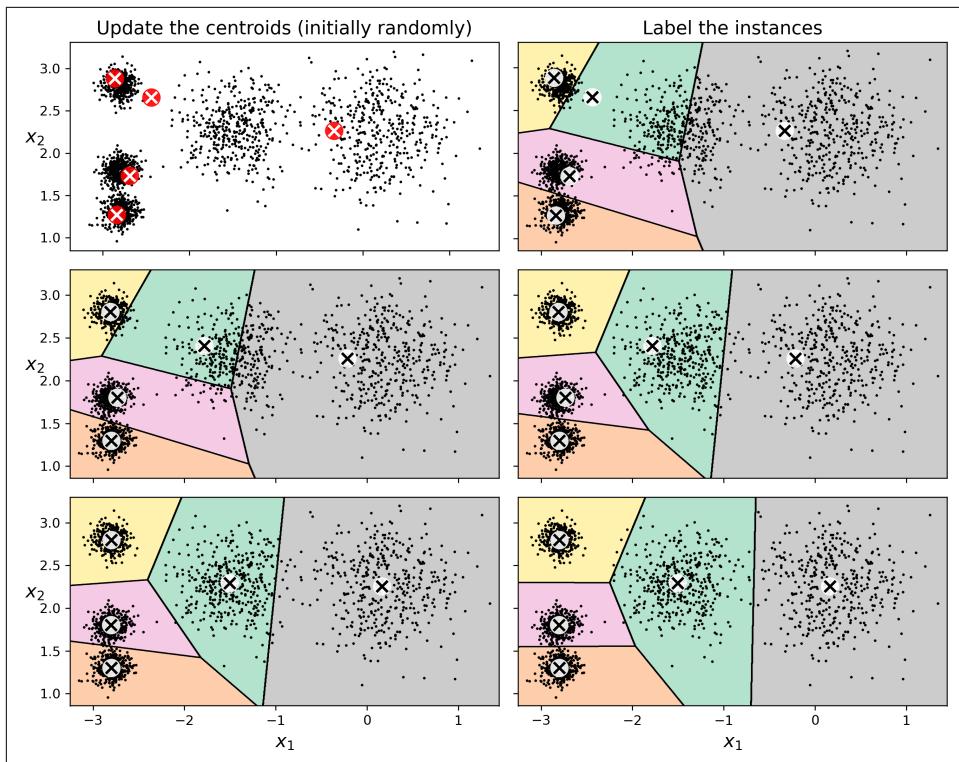


Figure 9-4. The K-Means algorithm

Although the algorithm is guaranteed to converge, it may not converge to the right solution (i.e., it may converge to a local optimum): whether it does or not depends on the centroid initialization. Figure 9-5 shows two suboptimal solutions that the algorithm can converge to if you are not lucky with the random initialization step.

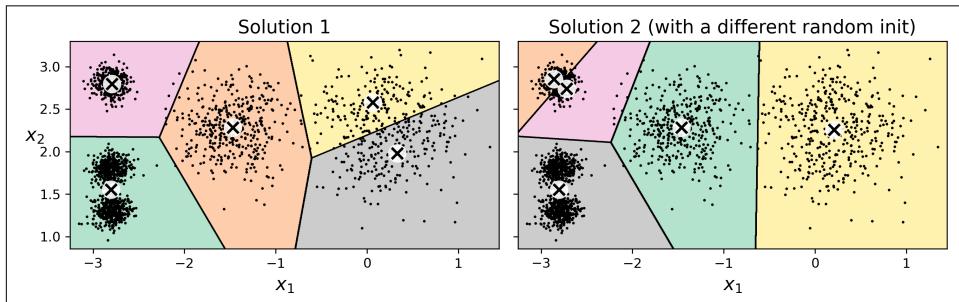


Figure 9-5. Suboptimal solutions due to unlucky centroid initializations

Let's look at a few ways you can mitigate this risk by improving the centroid initialization.

Centroid initialization methods

If you happen to know approximately where the centroids should be (e.g., if you ran another clustering algorithm earlier), then you can set the `init` hyperparameter to a NumPy array containing the list of centroids, and set `n_init` to 1:

```
good_init = np.array([[-3, 3], [-3, 2], [-3, 1], [-1, 2], [0, 2]])
kmeans = KMeans(n_clusters=5, init=good_init, n_init=1, random_state=42)
kmeans.fit(X)
```

Another solution is to run the algorithm multiple times with different random initializations and keep the best solution. The number of random initializations is controlled by the `n_init` hyperparameter: by default, it is equal to 10, which means that the whole algorithm described earlier runs 10 times when you call `fit()`, and Scikit-Learn keeps the best solution. But how exactly does it know which solution is the best? It uses a performance metric! That metric is called the model's *inertia*, which is the sum of the squared distances between the instances and their closest centroid. It is roughly equal to 219.4 for the model on the left in [Figure 9-5](#), 258.6 for the model on the right in [Figure 9-5](#), and only 211.6 for the model in [Figure 9-3](#). The `KMeans` class runs the algorithm `n_init` times and keeps the model with the lowest inertia. In this example, the model in [Figure 9-3](#) will be selected (unless we are very unlucky with `n_init` consecutive random initializations). If you are curious, a model's inertia is accessible via the `inertia_` instance variable:

```
>>> kmeans.inertia_
211.59853725816836
```

The `score()` method returns the negative inertia. Why negative? Because a predictor's `score()` method must always respect Scikit-Learn's “greater is better” rule: if a predictor is better than another, its `score()` method should return a greater score.

```
>>> kmeans.score(X)
-211.5985372581684
```

An important improvement to the K-Means algorithm, *K-Means++*, was proposed in a [2006 paper](#) by David Arthur and Sergei Vassilvitskii.² They introduced a smarter initialization step that tends to select centroids that are distant from one another, and this improvement makes the K-Means algorithm much less likely to converge to a suboptimal solution. They showed that the additional computation required for the smarter initialization step is well worth it because it makes it possible to drastically reduce the number of times the algorithm needs to be run to find the optimal solution. The K-Means++ initialization algorithm works like this:

1. Take one centroid $c^{(1)}$, chosen uniformly at random from the dataset.

² David Arthur and Sergei Vassilvitskii, “k-Means++: The Advantages of Careful Seeding,” *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms* (2007): 1027–1035.

- Take a new centroid $\mathbf{c}^{(i)}$, choosing an instance $\mathbf{x}^{(i)}$ with probability $D(\mathbf{x}^{(i)})^2 / \sum_{j=1}^m D(\mathbf{x}^{(j)})^2$, where $D(\mathbf{x}^{(i)})$ is the distance between the instance $\mathbf{x}^{(i)}$ and the closest centroid that was already chosen. This probability distribution ensures that instances farther away from already chosen centroids are much more likely to be selected as centroids.
- Repeat the previous step until all k centroids have been chosen.

The KMeans class uses this initialization method by default.

Accelerated K-Means and mini-batch K-Means

Another improvement to the K-Means algorithm was proposed in a [2003 paper](#) by Charles Elkan.³ On some large datasets with many clusters, the algorithm can be accelerated by avoiding many unnecessary distance calculations. Elkan achieved this by exploiting the triangle inequality (i.e., that a straight line is always the shortest distance between two points⁴) and by keeping track of lower and upper bounds for distances between instances and centroids. However, Elkan's algorithm does not always accelerate training, it depends on the dataset, and sometimes it can even slow down training significantly. Still, if you want to give it a try, set `algorithm="elkan"`.

Yet another important variant of the K-Means algorithm was proposed in a [2010 paper](#) by David Sculley.⁵ Instead of using the full dataset at each iteration, the algorithm is capable of using mini-batches, moving the centroids just slightly at each iteration. This speeds up the algorithm typically by a factor of three to four and makes it possible to cluster huge datasets that do not fit in memory. Scikit-Learn implements this algorithm in the `MiniBatchKMeans` class. You can just use this class like the `KMeans` class:

```
from sklearn.cluster import MiniBatchKMeans

minibatch_kmeans = MiniBatchKMeans(n_clusters=5, random_state=42)
minibatch_kmeans.fit(X)
```

If the dataset does not fit in memory, the simplest option is to use the `memmap` class, as we did for incremental PCA in [Chapter 8](#). Alternatively, you can pass one mini-batch at a time to the `partial_fit()` method, but this will require much more work, since you will need to perform multiple initializations and select the best one yourself.

³ Charles Elkan, “Using the Triangle Inequality to Accelerate k-Means,” *Proceedings of the 20th International Conference on Machine Learning* (2003): 147–153.

⁴ The triangle inequality is $AC \leq AB + BC$ where A, B and C are three points and AB, AC, and BC are the distances between these points.

⁵ David Sculley, “Web-Scale K-Means Clustering,” *Proceedings of the 19th International Conference on World Wide Web* (2010): 1177–1178.

Although the Mini-batch K-Means algorithm is much faster than the regular K-Means algorithm, its inertia is generally slightly worse. You can see this in [Figure 9-6](#): the plot on the left compares the inertias of Mini-batch K-Means and regular K-Means models trained on the previous five-blobs dataset using various numbers of clusters k . The difference between the two curves is small, but visible. In the plot on the right, you can see that Mini-batch K-Means is roughly 3.5 times faster than regular K-Means on this dataset.

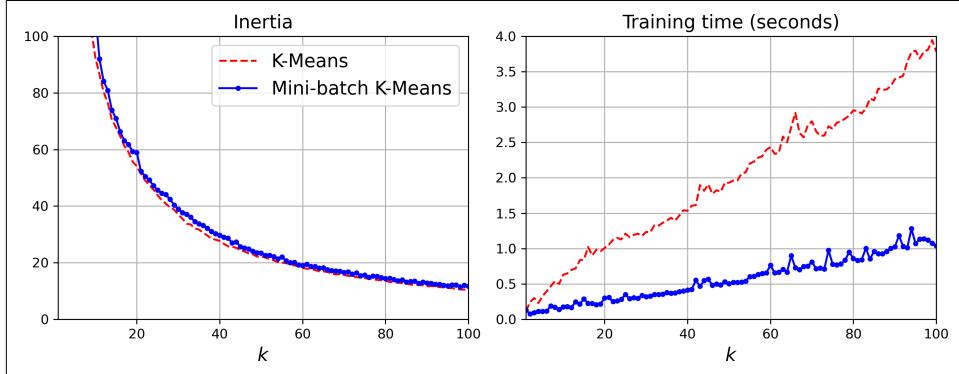


Figure 9-6. Mini-batch K-Means has a higher inertia than K-Means (left) but it is much faster (right), especially as k increases

Finding the optimal number of clusters

So far, we have set the number of clusters k to 5 because it was obvious by looking at the data that this was the correct number of clusters. But in general, it will not be so easy to know how to set k , and the result might be quite bad if you set it to the wrong value. As you can see in [Figure 9-7](#), setting k to 3 or 8 results in fairly bad models.

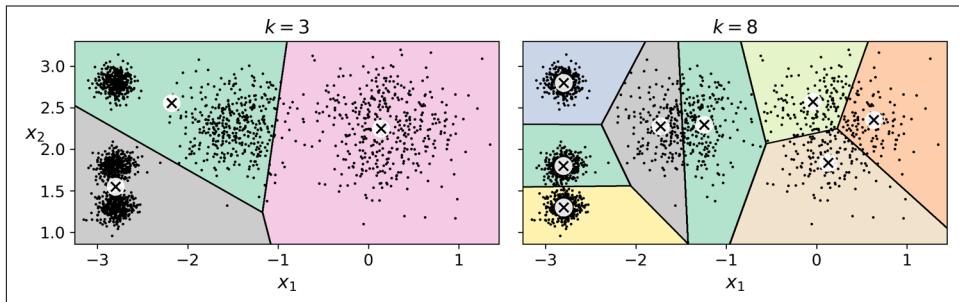


Figure 9-7. Bad choices for the number of clusters: when k is too small, separate clusters get merged (left), and when k is too large, some clusters get chopped into multiple pieces (right)

You might be thinking that we could just pick the model with the lowest inertia, right? Unfortunately, it is not that simple. The inertia for $k=3$ is about 653.2, which is much higher than for $k=5$ (which was 211.6). But with $k=8$, the inertia is just 119.1. The inertia is not a good performance metric when trying to choose k because it keeps getting lower as we increase k . Indeed, the more clusters there are, the closer each instance will be to its closest centroid, and therefore the lower the inertia will be. Let's plot the inertia as a function of k (see Figure 9-8).

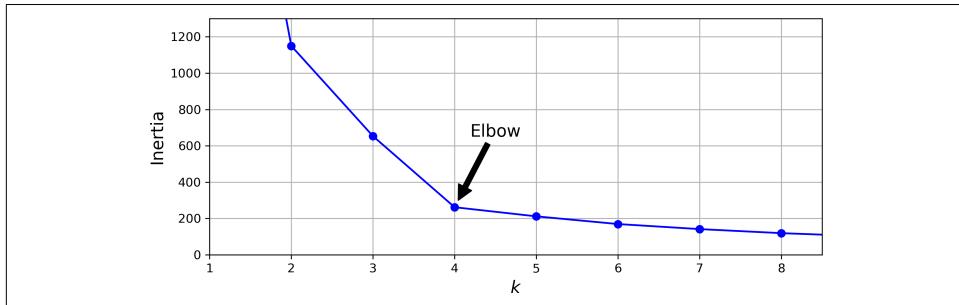


Figure 9-8. When plotting the inertia as a function of the number of clusters k , the curve often contains an inflection point called the “elbow”

As you can see, the inertia drops very quickly as we increase k up to 4, but then it decreases much more slowly as we keep increasing k . This curve has roughly the shape of an arm, and there is an “elbow” at $k = 4$. So, if we did not know better, 4 would be a good choice: any lower value would be dramatic, while any higher value would not help much, and we might just be splitting perfectly good clusters in half for no good reason.

This technique for choosing the best value for the number of clusters is rather coarse. A more precise approach (but also more computationally expensive) is to use the *silhouette score*, which is the mean *silhouette coefficient* over all the instances. An instance's silhouette coefficient is equal to $(b - a) / \max(a, b)$, where a is the mean distance to the other instances in the same cluster (i.e., the mean intra-cluster distance) and b is the mean nearest-cluster distance (i.e., the mean distance to the instances of the next closest cluster, defined as the one that minimizes b , excluding the instance's own cluster). The silhouette coefficient can vary between -1 and $+1$. A coefficient close to $+1$ means that the instance is well inside its own cluster and far from other clusters, while a coefficient close to 0 means that it is close to a cluster boundary, and finally a coefficient close to -1 means that the instance may have been assigned to the wrong cluster.

To compute the silhouette score, you can use Scikit-Learn's `silhouette_score()` function, giving it all the instances in the dataset and the labels they were assigned:

```
>>> from sklearn.metrics import silhouette_score  
>>> silhouette_score(X, kmeans.labels_)  
0.655517642572828
```

Let's compare the silhouette scores for different numbers of clusters (see [Figure 9-9](#)).

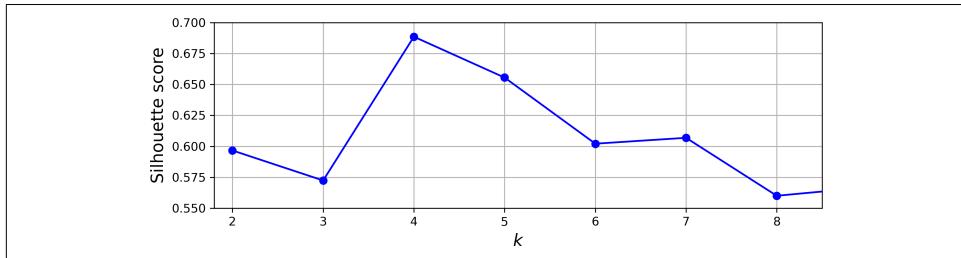


Figure 9-9. Selecting the number of clusters k using the silhouette score

As you can see, this visualization is much richer than the previous one: although it confirms that $k = 4$ is a very good choice, it also underlines the fact that $k = 5$ is quite good as well, and much better than $k = 6$ or 7 . This was not visible when comparing inertias.

An even more informative visualization is obtained when you plot every instance's silhouette coefficient, sorted by the cluster they are assigned to and by the value of the coefficient. This is called a *silhouette diagram* (see [Figure 9-10](#)). Each diagram contains one knife shape per cluster. The shape's height indicates the number of instances in the cluster, and its width represents the sorted silhouette coefficients of the instances in the cluster (wider is better).

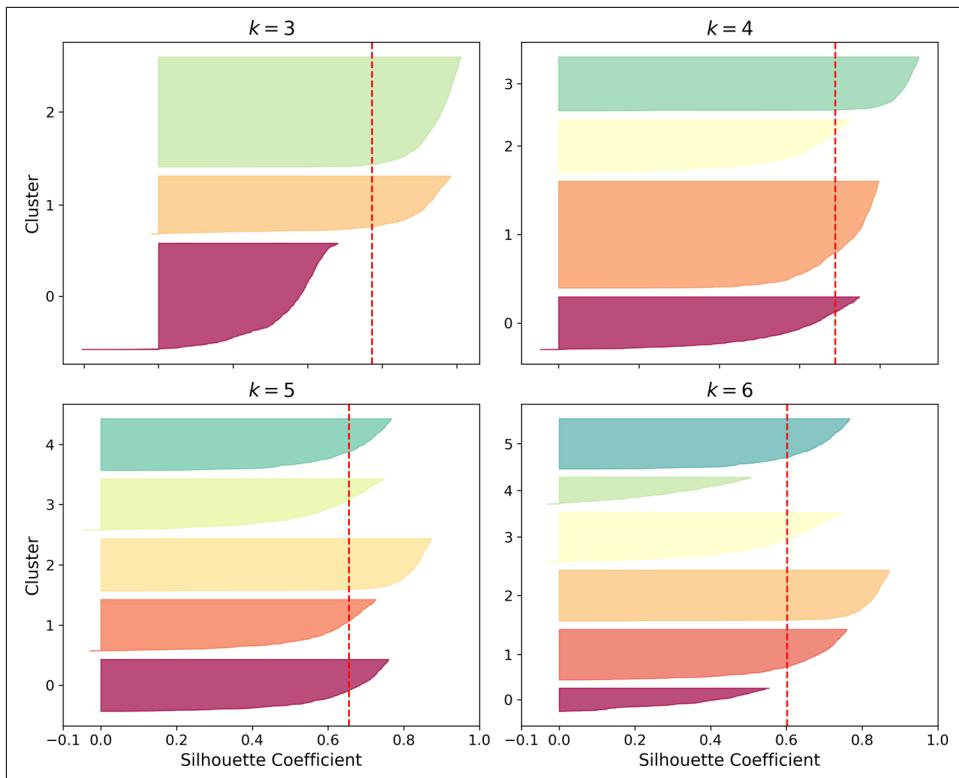


Figure 9-10. Analyzing the silhouette diagrams for various values of k

The vertical dashed lines represent the mean silhouette score for each number of clusters. When most of the instances in a cluster have a lower coefficient than this score (i.e., if many of the instances stop short of the dashed line, ending to the left of it), then the cluster is rather bad since this means its instances are much too close to other clusters. We can see that when $k = 3$ or 6 , we get bad clusters. But when $k = 4$ or 5 , the clusters look pretty good: most instances extend beyond the dashed line, to the right and closer to 1.0. When $k = 4$, the cluster at index 1 (the second from the bottom) is rather big. When $k = 5$, all clusters have similar sizes. So, even though the overall silhouette score from $k = 4$ is slightly greater than for $k = 5$, it seems like a good idea to use $k = 5$ to get clusters of similar sizes.

Limits of K-Means

Despite its many merits, most notably being fast and scalable, K-Means is not perfect. As we saw, it is necessary to run the algorithm several times to avoid suboptimal solutions, plus you need to specify the number of clusters, which can be quite a hassle. Moreover, K-Means does not behave very well when the clusters have varying

sizes, different densities, or nonspherical shapes. For example, [Figure 9-11](#) shows how K-Means clusters a dataset containing three ellipsoidal clusters of different dimensions, densities, and orientations.

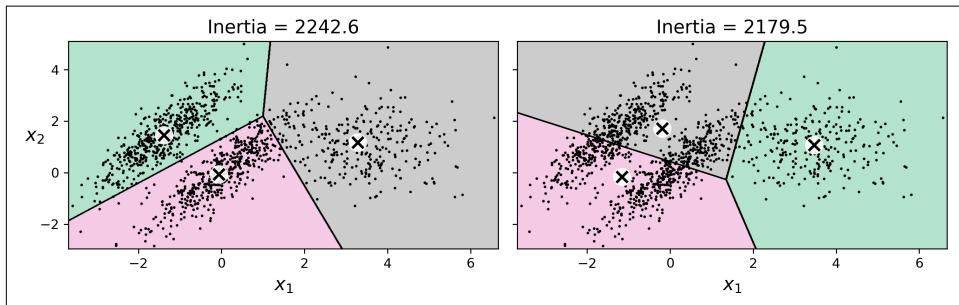


Figure 9-11. K-Means fails to cluster these ellipsoidal blobs properly

As you can see, neither of these solutions is any good. The solution on the left is better, but it still chops off 25% of the middle cluster and assigns it to the cluster on the right. The solution on the right is just terrible, even though its inertia is lower. So, depending on the data, different clustering algorithms may perform better. On these types of elliptical clusters, Gaussian mixture models work great.



It is important to scale the input features (see [Chapter 2](#)) before you run K-Means, or the clusters may be very stretched and K-Means will perform poorly. Scaling the features does not guarantee that all the clusters will be nice and spherical, but it generally helps K-Means.

Now let's look at a few ways we can benefit from clustering. We will use K-Means, but feel free to experiment with other clustering algorithms.

Using Clustering for Image Segmentation

Image segmentation is the task of partitioning an image into multiple segments. There are several variants:

- In *color segmentation*, pixels with a similar color get assigned to the same segment. This is sufficient in many applications. For example, if you want to analyze satellite images to measure how much total forest area there is in a region, color segmentation may be just fine.
- In *semantic segmentation*, all pixels that are part of the same object type get assigned to the same segment. For example, in a self-driving car's vision system, all pixels that are part of a pedestrian's image might be assigned to the "pedestrian" segment (there would be one segment containing all the pedestrians).

- In *instance segmentation*, all pixels that are part of the same individual object are assigned to the same segment. In this case there would be a different segment for each pedestrian.

The state of the art in semantic or instance segmentation today is achieved using complex architectures based on convolutional neural networks (see [Chapter 14](#)). In this chapter, we are going to focus on the (much simpler) *color segmentation* task, using K-Means.

Let's import the Pillow package (Python Image Library), and use it to load the *ladybug.png* image (see the upper-left image in [Figure 9-12](#)), assuming it's located at `filepath`:

```
>>> import PIL
>>> image = np.asarray(PIL.Image.open(filepath))
>>> image.shape
(533, 800, 3)
```

The image is represented as a 3D array. The first dimension's size is the height; the second is the width; and the third is the number of color channels, in this case red, green, and blue (RGB). In other words, for each pixel there is a 3D vector containing the intensities of red, green, and blue, as unsigned 8-bit integers between 0 and 255. Some images may have fewer channels, such as grayscale images (one channel). And some images may have more channels, such as images with an additional *alpha channel* for transparency, or satellite images which often contain channels for many light frequencies (e.g., infrared).

The following code reshapes the array to get a long list of RGB colors, then it clusters these colors using K-Means with 8 clusters, and it creates a `segmented_img` array containing the nearest cluster center for each pixel (i.e., the mean color of each pixel's cluster), and lastly it reshapes this array to the original image shape. The third line uses advanced NumPy indexing: for example, if the first 10 labels in `kmeans_.labels_` are equal to 1, then the first 10 colors in `segmented_img` are equal to `kmeans.cluster_centers_[1]`.

```
X = image.reshape(-1, 3)
kmeans = KMeans(n_clusters=8, random_state=42).fit(X)
segmented_img = kmeans.cluster_centers_[kmeans.labels_]
segmented_img = segmented_img.reshape(image.shape)
```

This outputs the image shown in the upper right of [Figure 9-12](#). You can experiment with various numbers of clusters, as shown in the figure. When you use fewer than eight clusters, notice that the ladybug's flashy red color fails to get a cluster of its own: it gets merged with colors from the environment. This is because K-Means prefers clusters of similar sizes. The ladybug is small—much smaller than the rest of the image—so even though its color is flashy, K-Means fails to dedicate a cluster to it.



Figure 9-12. Image segmentation using K-Means with various numbers of color clusters

That wasn't too hard, was it? Now let's look at another application of clustering.

Using Clustering for Semi-Supervised Learning

Another use case for clustering is in semi-supervised learning, when we have plenty of unlabeled instances and very few labeled instances. In this section, we'll use the digits dataset, which is a simple MNIST-like dataset containing 1,797 grayscale 8×8 images representing the digits 0 to 9. First, let's load and split the dataset (it's already shuffled):

```
from sklearn.datasets import load_digits

X_digits, y_digits = load_digits(return_X_y=True)
X_train, y_train = X_digits[:1400], y_digits[:1400]
X_test, y_test = X_digits[1400:], y_digits[1400:]
```

We will pretend we only have labels for 50 instances. To get a baseline performance, let's train a Logistic Regression model on these 50 labeled instances:

```
from sklearn.linear_model import LogisticRegression

n_labeled = 50
log_reg = LogisticRegression(max_iter=10_000)
log_reg.fit(X_train[:n_labeled], y_train[:n_labeled])
```

Let's measure the accuracy of this model on the test set (note that the test set must be labeled):

```
>>> log_reg.score(X_test, y_test)
0.7481108312342569
```

The model's accuracy is just 74.8%. That's not great: indeed, if you try training the model on the full training set, you will find that it will reach about 90.7% accuracy. Let's see how we can do better. First, let's cluster the training set into 50 clusters. Then for each cluster, let's find the image closest to the centroid. Let's call these images the *representative images*:

```
k = 50
kmeans = KMeans(n_clusters=k, random_state=42)
X_digits_dist = kmeans.fit_transform(X_train)
representative_digit_idx = np.argmin(X_digits_dist, axis=0)
X_representative_digits = X_train[representative_digit_idx]
```

Figure 9-13 shows these 50 representative images.

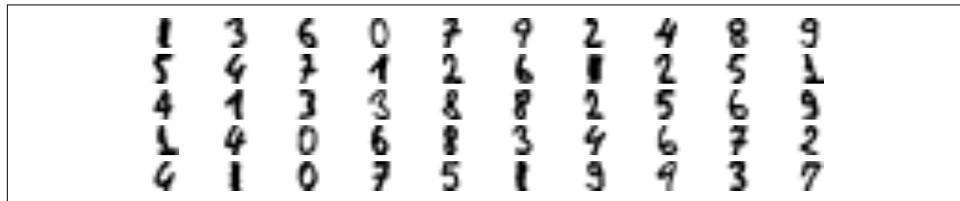


Figure 9-13. Fifty representative digit images (one per cluster)

Let's look at each image and manually label it:

```
y_representative_digits = np.array([1, 3, 6, 0, 7, 9, 2, ..., 5, 1, 9, 9, 3, 7])
```

Now we have a dataset with just 50 labeled instances, but instead of being random instances, each of them is a representative image of its cluster. Let's see if the performance is any better:

```
>>> log_reg = LogisticRegression(max_iter=10_000)
>>> log_reg.fit(X_representative_digits, y_representative_digits)
>>> log_reg.score(X_test, y_test)
0.8488664987405542
```

Wow! We jumped from 74.8% accuracy to 84.9%, although we are still only training the model on 50 instances. Since it is often costly and painful to label instances, especially when it has to be done manually by experts, it is a good idea to label representative instances rather than just random instances.

But perhaps we can go one step further: what if we propagated the labels to all the other instances in the same cluster? This is called *label propagation*:

```
y_train_propagated = np.empty(len(X_train), dtype=np.int64)
for i in range(k):
    y_train_propagated[kmeans.labels_ == i] = y_representative_digits[i]
```

Now let's train the model again and look at its performance:

```
>>> log_reg = LogisticRegression()
>>> log_reg.fit(X_train, y_train_propagated)
```

```
>>> log_reg.score(X_test, y_test)
0.8942065491183879
```

We got another significant accuracy boost! Let's see if we can do even better by ignoring the 1% instances that are farthest from their cluster center: this should eliminate some outliers. The following code first computes the distance from each instance to its closest cluster center, then for each cluster it sets the 1% largest distances to -1 . Lastly, it creates a set without these instances marked with a -1 distance.

```
percentile_closest = 99

X_cluster_dist = X_digits_dist[np.arange(len(X_train)), kmeans.labels_]
for i in range(k):
    in_cluster = (kmeans.labels_ == i)
    cluster_dist = X_cluster_dist[in_cluster]
    cutoff_distance = np.percentile(cluster_dist, percentile_closest)
    above_cutoff = (X_cluster_dist > cutoff_distance)
    X_cluster_dist[in_cluster & above_cutoff] = -1

partially_propagated = (X_cluster_dist != -1)
X_train_partially_propagated = X_train[partially_propagated]
y_train_partially_propagated = y_train_propagated[partially_propagated]
```

Now let's train the model again on this partially propagated dataset and see what accuracy we get:

```
>>> log_reg = LogisticRegression(max_iter=10_000)
>>> log_reg.fit(X_train_partially_propagated, y_train_partially_propagated)
>>> log_reg.score(X_test, y_test)
0.9093198992443325
```

Nice! With just 50 labeled instances (only 5 examples per class on average!), we got 90.9% accuracy, which is actually slightly higher than the performance we got on the fully labeled digits dataset (90.7%). This is partly thanks to the fact that we dropped some outliers, and partly because the propagated labels are actually pretty good—their accuracy is about 97.5%, as the following code shows:

```
>>> (y_train_partially_propagated == y_train[partially_propagated]).mean()
0.9755555555555555
```



Scikit-Learn also offers two classes that can propagate labels automatically: `LabelSpreading` and `LabelPropagation` in the `sklearn.semi_supervised` package. They both construct a similarity matrix between all the instances, and iteratively propagate labels from labeled instances to similar unlabeled instances. There's also a very different class called `SelfTrainingClassifier` in the same package: you give it a base classifier (such as a `RandomForestClassifier`) and it trains it on the labeled instances, then uses it to predict labels for the unlabeled samples. It then updates the training set with the labels it is most confident about. Lastly, it repeats this process of training and labeling until it cannot add labels anymore. These techniques are not magic bullets, but they can occasionally give your model a little boost.

Active Learning

To continue improving your model and your training set, the next step could be to do a few rounds of *active learning*, which is when a human expert interacts with the learning algorithm, providing labels for specific instances when the algorithm requests them. There are many different strategies for active learning, but one of the most common ones is called *uncertainty sampling*. Here is how it works:

1. The model is trained on the labeled instances gathered so far, and this model is used to make predictions on all the unlabeled instances.
2. The instances for which the model is most uncertain (i.e., when its estimated probability is lowest) are given to the expert for labeling.
3. You iterate this process until the performance improvement stops being worth the labeling effort.

Other active learning strategies include labeling the instances that would result in the largest model change, or the largest drop in the model's validation error, or the instances that different models disagree on (e.g., an SVM and a Random Forest).

Before we move on to Gaussian mixture models, let's take a look at DBSCAN, another popular clustering algorithm that illustrates a very different approach based on local density estimation. This approach allows the algorithm to identify clusters of arbitrary shapes.

DBSCAN

This algorithm defines clusters as continuous regions of high density. Here is how it works:

- For each instance, the algorithm counts how many instances are located within a small distance ϵ (epsilon) from it. This region is called the instance's ϵ -neighborhood.
- If an instance has at least `min_samples` instances in its ϵ -neighborhood (including itself), then it is considered a *core instance*. In other words, core instances are those that are located in dense regions.
- All instances in the neighborhood of a core instance belong to the same cluster. This neighborhood may include other core instances; therefore, a long sequence of neighboring core instances forms a single cluster.
- Any instance that is not a core instance and does not have one in its neighborhood is considered an anomaly.

This algorithm works well if all the clusters are well separated by low-density regions. The `DBSCAN` class in Scikit-Learn is as simple to use as you might expect. Let's test it on the moons dataset, introduced in [Chapter 6](#):

```
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=1000, noise=0.05)
dbSCAN = DBSCAN(eps=0.05, min_samples=5)
dbSCAN.fit(X)
```

The labels of all the instances are now available in the `labels_` instance variable:

```
>>> dbSCAN.labels_
array([ 0,  2, -1, -1,  1,  0,  0,  0,  2,  5, [...], 3,  3,  4,  2,  6,  3])
```

Notice that some instances have a cluster index equal to -1 , which means that they are considered as anomalies by the algorithm. The indices of the core instances are available in the `core_sample_indices_` instance variable, and the core instances themselves are available in the `components_` instance variable:

```
>>> dbSCAN.core_sample_indices_
array([ 0,  4,  5,  6,  7,  8, 10, 11, [...], 993, 995, 997, 998, 999])
>>> dbSCAN.components_
array([[[-0.02137124,  0.40618608],
       [-0.84192557,  0.53058695],
       [...],
       [ 0.79419406,  0.60777171]])
```

This clustering is represented in the lefthand plot of [Figure 9-14](#). As you can see, it identified quite a lot of anomalies, plus seven different clusters. How disappointing! Fortunately, if we widen each instance's neighborhood by increasing `eps` to 0.2, we get the clustering on the right, which looks perfect. Let's continue with this model.

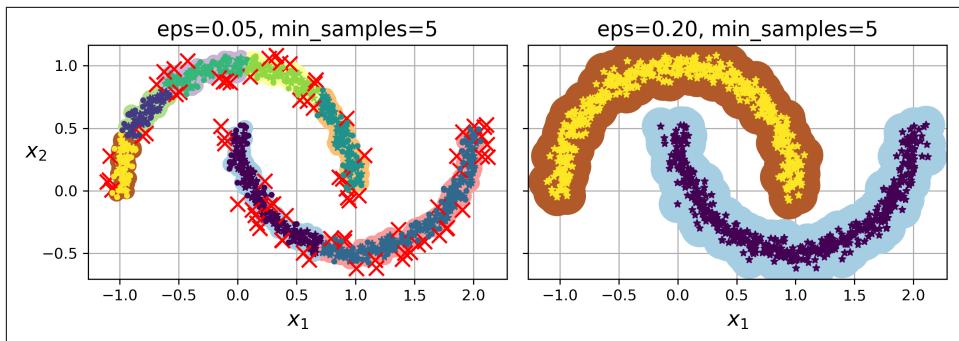


Figure 9-14. DBSCAN clustering using two different neighborhood radii

Surprisingly, the DBSCAN class does not have a `predict()` method, although it has a `fit_predict()` method. In other words, it cannot predict which cluster a new instance belongs to. This decision was made because different classification algorithms can be better for different tasks, so the authors decided to let the user choose which one to use. Moreover, it's not hard to implement. For example, let's train a `KNeighborsClassifier`:

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=50)
knn.fit(dbSCAN.components_, dbSCAN.labels_[dbSCAN.core_sample_indices_])
```

Now, given a few new instances, we can predict which cluster they most likely belong to and even estimate a probability for each cluster:

```
>>> X_new = np.array([[-0.5, 0], [0, 0.5], [1, -0.1], [2, 1]])
>>> knn.predict(X_new)
array([1, 0, 1, 0])
>>> knn.predict_proba(X_new)
array([[0.18, 0.82],
       [1. , 0. ],
       [0.12, 0.88],
       [1. , 0. ]])
```

Note that we only trained the classifier on the core instances, but we could also have chosen to train it on all the instances, or all but the anomalies: this choice depends on the final task.

The decision boundary is represented in Figure 9-15 (the crosses represent the four instances in `X_new`). Notice that since there is no anomaly in the training set, the classifier always chooses a cluster, even when that cluster is far away. It is fairly straightforward to introduce a maximum distance, in which case the two instances that are far away from both clusters are classified as anomalies. To do this, use the `kneighbors()` method of the `KNeighborsClassifier`. Given a set of instances, it

returns the distances and the indices of the k nearest neighbors in the training set (two matrices, each with k columns):

```
>>> y_dist, y_pred_idx = knn.kneighbors(X_new, n_neighbors=1)
>>> y_pred = dbSCAN.labels_[dbSCAN.core_sample_indices_][y_pred_idx]
>>> y_pred[y_dist > 0.2] = -1
>>> y_pred.ravel()
array([-1,  0,  1, -1])
```

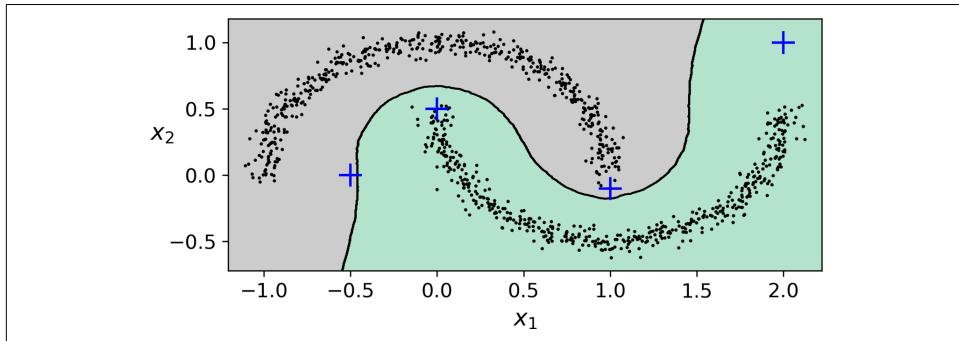


Figure 9-15. Decision boundary between two clusters

In short, DBSCAN is a very simple yet powerful algorithm capable of identifying any number of clusters of any shape. It is robust to outliers, and it has just two hyperparameters (`eps` and `min_samples`). If the density varies significantly across the clusters, or if there's no sufficiently low-density region around some clusters, DBSCAN can struggle to capture all the clusters properly. Moreover, its computational complexity is roughly $O(m^2n)$, so it does not scale well to large datasets.



You may also want to try *Hierarchical DBSCAN* (HDBSCAN), which is implemented in the [scikit-learn-contrib project](#), as it is usually better than DBSCAN at finding clusters of varying densities.

Other Clustering Algorithms

Scikit-Learn implements several more clustering algorithms that you should take a look at. I cannot cover them all in detail here, but here is a brief overview:

Agglomerative clustering

A hierarchy of clusters is built from the bottom up. Think of many tiny bubbles floating on water and gradually attaching to each other until there's one big group of bubbles. Similarly, at each iteration, agglomerative clustering connects the nearest pair of clusters (starting with individual instances). If you drew a tree with a branch for every pair of clusters that merged, you would get a binary

tree of clusters, where the leaves are the individual instances. This approach can capture clusters of various shapes, it also produces a flexible and informative cluster tree instead of forcing you to choose a particular cluster scale, and it can be used with any pairwise distance. It can scale nicely to large numbers of instances if you provide a connectivity matrix, which is a sparse $m \times m$ matrix that indicates which pairs of instances are neighbors (e.g., returned by `sklearn.neighbors.kneighbors_graph()`). Without a connectivity matrix, the algorithm does not scale well to large datasets.

BIRCH

The BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies) algorithm was designed specifically for very large datasets, and it can be faster than batch K-Means, with similar results, as long as the number of features is not too large (<20). During training, it builds a tree structure containing just enough information to quickly assign each new instance to a cluster, without having to store all the instances in the tree: this approach allows it to use limited memory, while handling huge datasets.

Mean-Shift

This algorithm starts by placing a circle centered on each instance; then for each circle it computes the mean of all the instances located within it, and it shifts the circle so that it is centered on the mean. Next, it iterates this mean-shifting step until all the circles stop moving (i.e., until each of them is centered on the mean of the instances it contains). Mean-Shift shifts the circles in the direction of higher density, until each of them has found a local density maximum. Finally, all the instances whose circles have settled in the same place (or close enough) are assigned to the same cluster. Mean-Shift has some of the same features as DBSCAN, like how it can find any number of clusters of any shape, it has very few hyperparameters (just one—the radius of the circles, called the *bandwidth*), and it relies on local density estimation. But unlike DBSCAN, Mean-Shift tends to chop clusters into pieces when they have internal density variations. Unfortunately, its computational complexity is $O(m^2n)$, so it is not suited for large datasets.

Affinity propagation

In this algorithm, instances repeatedly exchange messages between one another until every instance has elected another instance (or itself) to represent it. These elected instances are called *exemplars*. Each exemplar and all the instances that elected it form one cluster. In real-life politics, you typically want to vote for a candidate whose opinions are similar to yours, but you also want them to win the election, so you might choose a candidate you don't fully agree with, but who is more popular. You typically evaluate popularity through polls. Affinity propagation works in a similar way, and it tends to choose exemplars located near the center of clusters, similar to K-Means. But unlike with K-Means, you

don't have to pick a number of clusters ahead of time: it is determined during training. Moreover, affinity propagation can deal nicely with clusters of different sizes. Sadly, this algorithm has a computational complexity of $O(m^2)$, so it is not suited for large datasets.

Spectral clustering

This algorithm takes a similarity matrix between the instances and creates a low-dimensional embedding from it (i.e., it reduces the matrix's dimensionality), then it uses another clustering algorithm in this low-dimensional space (Scikit-Learn's implementation uses K-Means). Spectral clustering can capture complex cluster structures, and it can also be used to cut graphs (e.g., to identify clusters of friends on a social network). It does not scale well to large numbers of instances, and it does not behave well when the clusters have very different sizes.

Now let's dive into Gaussian mixture models, which can be used for density estimation, clustering, and anomaly detection.

Gaussian Mixtures

A *Gaussian mixture model* (GMM) is a probabilistic model that assumes that the instances were generated from a mixture of several Gaussian distributions whose parameters are unknown. All the instances generated from a single Gaussian distribution form a cluster that typically looks like an ellipsoid. Each cluster can have a different ellipsoidal shape, size, density, and orientation, just like in [Figure 9-11](#). When you observe an instance, you know it was generated from one of the Gaussian distributions, but you are not told which one, and you do not know what the parameters of these distributions are.

There are several GMM variants. In the simplest variant, implemented in the `GaussianMixture` class, you must know in advance the number k of Gaussian distributions. The dataset \mathbf{X} is assumed to have been generated through the following probabilistic process:

- For each instance, a cluster is picked randomly from among k clusters. The probability of choosing the j^{th} cluster is the cluster's weight $\phi^{(j)}$.⁶ The index of the cluster chosen for the i^{th} instance is noted $z^{(i)}$.
- If the i^{th} instance was assigned to the j^{th} cluster (i.e., $z^{(i)} = j$), then the location $\mathbf{x}^{(i)}$ of this instance is sampled randomly from the Gaussian distribution with mean $\boldsymbol{\mu}^{(j)}$ and covariance matrix $\Sigma^{(j)}$. This is noted $\mathbf{x}^{(i)} \sim \boldsymbol{\mu}^{(j)}, \Sigma^{(j)}$.

⁶ Phi (ϕ or φ) is the 21st letter of the Greek alphabet.

So, what can you do with such a model? Well, given the dataset \mathbf{X} , you typically want to start by estimating the weights ϕ and all the distribution parameters $\mu^{(1)}$ to $\mu^{(k)}$ and $\Sigma^{(1)}$ to $\Sigma^{(k)}$. Scikit-Learn's `GaussianMixture` class makes this super easy:

```
from sklearn.mixture import GaussianMixture

gm = GaussianMixture(n_components=3, n_init=10)
gm.fit(X)
```

Let's look at the parameters that the algorithm estimated:

```
>>> gm.weights_
array([0.39025715, 0.40007391, 0.20966893])
>>> gm.means_
array([[ 0.05131611,  0.07521837],
       [-1.40763156,  1.42708225],
       [ 3.39893794,  1.05928897]])
>>> gm.covariances_
array([[[[ 0.68799922,  0.79606357],
         [ 0.79606357,  1.21236106]],
        [[ 0.63479409,  0.72970799],
         [ 0.72970799,  1.1610351 ]],
        [[ 1.14833585, -0.03256179],
         [-0.03256179,  0.95490931]]]])
```

Great, it worked fine! Indeed, two of the three clusters were generated with 500 instances each, while the third cluster only contains 250 instances. So the true cluster weights are 0.4, 0.4, and 0.2, respectively, and that's roughly what the algorithm found. Similarly, the true means and covariance matrices are quite close to those found by the algorithm. But how? This class relies on the *Expectation-Maximization* (EM) algorithm, which has many similarities with the K-Means algorithm: it also initializes the cluster parameters randomly, then it repeats two steps until convergence, first assigning instances to clusters (this is called the *expectation step*) and then updating the clusters (this is called the *maximization step*). Sounds familiar, right? In the context of clustering, you can think of EM as a generalization of K-Means that not only finds the cluster centers ($\mu^{(1)}$ to $\mu^{(k)}$), but also their size, shape, and orientation ($\Sigma^{(1)}$ to $\Sigma^{(k)}$), as well as their relative weights ($\phi^{(1)}$ to $\phi^{(k)}$). Unlike K-Means, though, EM uses soft cluster assignments, not hard assignments. For each instance, during the expectation step, the algorithm estimates the probability that it belongs to each cluster (based on the current cluster parameters). Then, during the maximization step, each cluster is updated using *all* the instances in the dataset, with each instance weighted by the estimated probability that it belongs to that cluster. These probabilities are called the *responsibilities* of the clusters for the instances. During the maximization step, each cluster's update will mostly be impacted by the instances it is most responsible for.



Unfortunately, just like K-Means, EM can end up converging to poor solutions, so it needs to be run several times, keeping only the best solution. This is why we set `n_init` to 10. Be careful: by default `n_init` is set to 1.

You can check whether or not the algorithm converged and how many iterations it took:

```
>>> gm.converged_
True
>>> gm.n_iter_
4
```

Now that you have an estimate of the location, size, shape, orientation, and relative weight of each cluster, the model can easily assign each instance to the most likely cluster (hard clustering) or estimate the probability that it belongs to a particular cluster (soft clustering). Just use the `predict()` method for hard clustering, or the `predict_proba()` method for soft clustering:

```
>>> gm.predict(X)
array([0, 0, 1, ..., 2, 2, 2])
>>> gm.predict_proba(X).round(3)
array([[0.977, 0., 0.023],
       [0.983, 0.001, 0.016],
       [0., 1., 0.],
       ...,
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.]])
```

A Gaussian mixture model is a *generative model*, meaning you can sample new instances from it (note that they are ordered by cluster index):

```
>>> X_new, y_new = gm.sample(6)
>>> X_new
array([[-0.86944074, -0.32767626],
       [ 0.29836051,  0.28297011],
       [-2.8014927 , -0.09047309],
       [ 3.98203732,  1.49951491],
       [ 3.81677148,  0.53095244],
       [ 2.84104923, -0.73858639]])
>>> y_new
array([0, 0, 1, 2, 2, 2])
```

It is also possible to estimate the density of the model at any given location. This is achieved using the `score_samples()` method: for each instance it is given, this method estimates the log of the *probability density function* (PDF) at that location. The greater the score, the higher the density:

```
>>> gm.score_samples(X).round(2)
array([-2.61, -3.57, -3.33, ..., -3.51, -4.4 , -3.81])
```

If you compute the exponential of these scores, you get the value of the PDF at the location of the given instances. These are not probabilities, but probability *densities*: they can take on any positive value, not just a value between 0 and 1. To estimate the probability that an instance will fall within a particular region, you would have to integrate the PDF over that region (if you do so over the entire space of possible instance locations, the result will be 1).

Figure 9-16 shows the cluster means, the decision boundaries (dashed lines), and the density contours of this model.

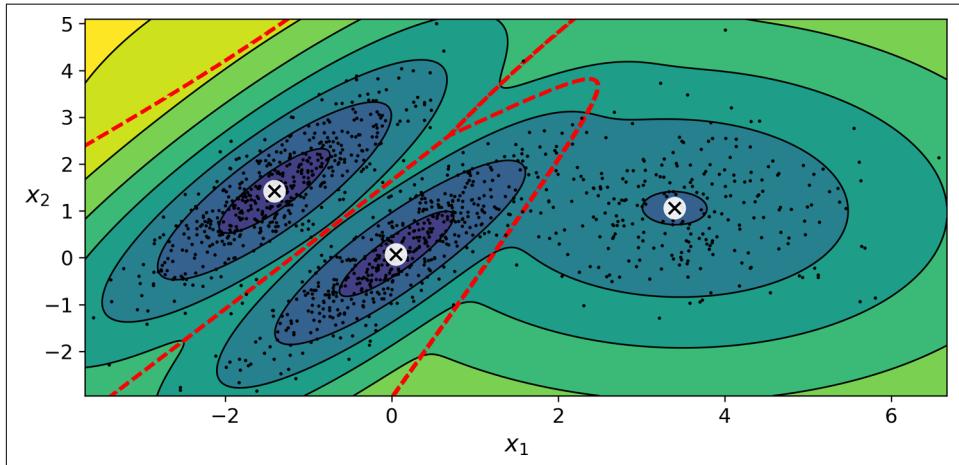


Figure 9-16. Cluster means, decision boundaries, and density contours of a trained Gaussian mixture model

Nice! The algorithm clearly found an excellent solution. Of course, we made its task easy by generating the data using a set of 2D Gaussian distributions (unfortunately, real-life data is not always so Gaussian and low-dimensional). We also gave the algorithm the correct number of clusters. When there are many dimensions, or many clusters, or few instances, EM can struggle to converge to the optimal solution. You might need to reduce the difficulty of the task by limiting the number of parameters that the algorithm has to learn. One way to do this is to limit the range of shapes and orientations that the clusters can have. This can be achieved by imposing constraints on the covariance matrices. To do this, set the `covariance_type` hyperparameter to one of the following values:

`"spherical"`

All clusters must be spherical, but they can have different diameters (i.e., different variances).

"diag"

Clusters can take on any ellipsoidal shape of any size, but the ellipsoid's axes must be parallel to the coordinate axes (i.e., the covariance matrices must be diagonal).

"tied"

All clusters must have the same ellipsoidal shape, size, and orientation (i.e., all clusters share the same covariance matrix).

By default, `covariance_type` is equal to "full", which means that each cluster can take on any shape, size, and orientation (it has its own unconstrained covariance matrix). [Figure 9-17](#) plots the solutions found by the EM algorithm when `covariance_type` is set to "tied" or "spherical".

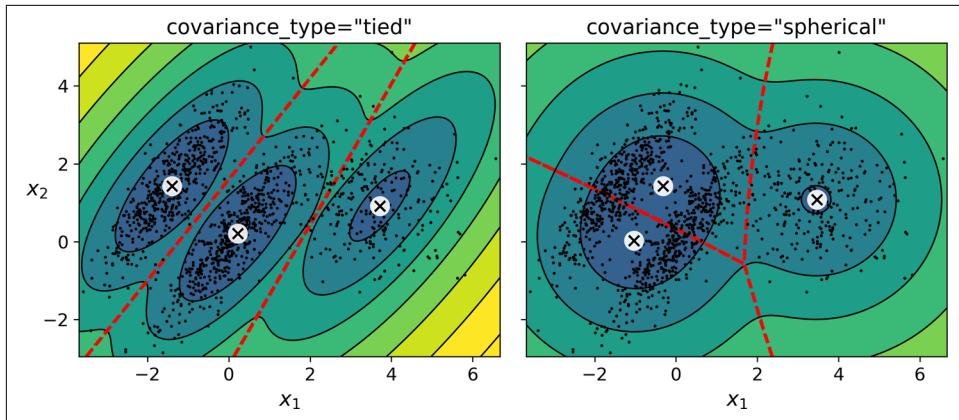


Figure 9-17. Gaussian mixtures for tied clusters (left) and spherical clusters (right)



The computational complexity of training a `GaussianMixture` model depends on the number of instances m , the number of dimensions n , the number of clusters k , and the constraints on the covariance matrices. If `covariance_type` is "spherical" or "diag", it is $O(kmn)$, assuming the data has a clustering structure. If `covariance_type` is "tied" or "full", it is $O(kmn^2 + kn^3)$, so it will not scale to large numbers of features.

Gaussian mixture models can also be used for anomaly detection. Let's see how.

Using Gaussian Mixtures for Anomaly Detection

Using a Gaussian mixture model for anomaly detection is quite simple: any instance located in a low-density region can be considered an anomaly. You must define what density threshold you want to use. For example, in a manufacturing company that tries to detect defective products, the ratio of defective products is usually well

known. Say it is equal to 2%. You then set the density threshold to be the value that results in having 2% of the instances located in areas below that threshold density. If you notice that you get too many false positives (i.e., perfectly good products that are flagged as defective), you can lower the threshold. Conversely, if you have too many false negatives (i.e., defective products that the system does not flag as defective), you can increase the threshold. This is the usual precision/recall trade-off (see [Chapter 3](#)). Here is how you would identify the outliers using the fourth percentile lowest density as the threshold (i.e., approximately 4% of the instances will be flagged as anomalies):

```
densities = gm.score_samples(X)
density_threshold = np.percentile(densities, 2)
anomalies = X[densities < density_threshold]
```

[Figure 9-18](#) represents these anomalies as stars.

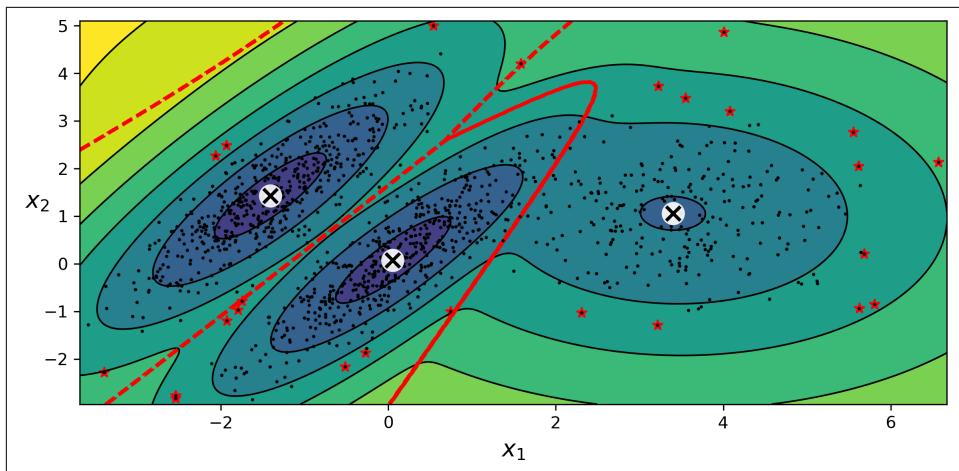


Figure 9-18. Anomaly detection using a Gaussian mixture model

A closely related task is *novelty detection*: it differs from anomaly detection in that the algorithm is assumed to be trained on a “clean” dataset, uncontaminated by outliers, whereas anomaly detection does not make this assumption. Indeed, outlier detection is often used to clean up a dataset.



Gaussian mixture models try to fit all the data, including the outliers, so if you have too many of them, this will bias the model’s view of “normality,” and some outliers may wrongly be considered as normal. If this happens, you can try to fit the model once, use it to detect and remove the most extreme outliers, then fit the model again on the cleaned-up dataset. Another approach is to use robust covariance estimation methods (see the `EllipticEnvelope` class).

Just like K-Means, the `GaussianMixture` algorithm requires you to specify the number of clusters. So, how can you find it?

Selecting the Number of Clusters

With K-Means, you could use the inertia or the silhouette score to select the appropriate number of clusters. But with Gaussian mixtures, it is not possible to use these metrics because they are not reliable when the clusters are not spherical or have different sizes. Instead, you can try to find the model that minimizes a *theoretical information criterion*, such as the *Bayesian information criterion* (BIC) or the *Akaike information criterion* (AIC), defined in [Equation 9-1](#).

Equation 9-1. Bayesian information criterion (BIC) and Akaike information criterion (AIC)

$$BIC = -\log(m)p - 2 \log(\widehat{\mathcal{L}})$$

$$AIC = -2p - 2 \log(\widehat{\mathcal{L}})$$

In these equations:

- m is the number of instances, as always.
- p is the number of parameters learned by the model.
- $\widehat{\mathcal{L}}$ is the maximized value of the *likelihood function* of the model.

Both the BIC and the AIC penalize models that have more parameters to learn (e.g., more clusters) and reward models that fit the data well. They often end up selecting the same model. When they differ, the model selected by the BIC tends to be simpler (fewer parameters) than the one selected by the AIC, but tends to not fit the data quite as well (this is especially true for larger datasets).

Likelihood Function

The terms “probability” and “likelihood” are often used interchangeably in everyday language, but they have very different meanings in statistics. Given a statistical model with some parameters θ , the word “probability” is used to describe how plausible a future outcome x is (knowing the parameter values θ), while the word “likelihood” is used to describe how plausible a particular set of parameter values θ are, after the outcome x is known.

Consider a 1D mixture model of two Gaussian distributions centered at -4 and $+1$. For simplicity, this toy model has a single parameter θ that controls the standard deviations of both distributions. The top-left contour plot in [Figure 9-19](#) shows the entire model $f(x; \theta)$ as a function of both x and θ . To estimate the probability distribution of

a future outcome x , you need to set the model parameter θ . For example, if you set θ to 1.3 (the horizontal line), you get the probability density function $f(x; \theta=1.3)$ shown in the lower-left plot. Say you want to estimate the probability that x will fall between -2 and +2. You must calculate the integral of the PDF on this range (i.e., the surface of the shaded region). But what if you don't know θ , and instead if you have observed a single instance $x=2.5$ (the vertical line in the upper-left plot)? In this case, you get the likelihood function $\mathcal{L}(\theta|x=2.5) = f(x=2.5; \theta)$, represented in the upper-right plot.

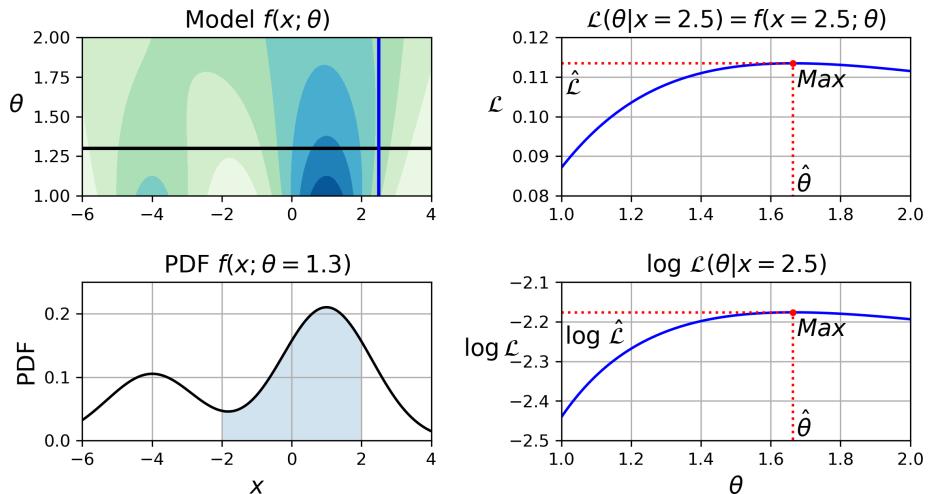


Figure 9-19. A model's parametric function (top left), and some derived functions: a PDF (lower left), a likelihood function (top right), and a log likelihood function (lower right)

In short, the PDF is a function of x (with θ fixed), while the likelihood function is a function of θ (with x fixed). It is important to understand that the likelihood function is *not* a probability distribution: if you integrate a probability distribution over all possible values of x , you always get 1; but if you integrate the likelihood function over all possible values of θ , the result can be any positive value.

Given a dataset \mathbf{X} , a common task is to try to estimate the most likely values for the model parameters. To do this, you must find the values that maximize the likelihood function, given \mathbf{X} . In this example, if you have observed a single instance $x=2.5$, the *maximum likelihood estimate* (MLE) of θ is $\hat{\theta}=1.5$. If a prior probability distribution g over θ exists, it is possible to take it into account by maximizing $\mathcal{L}(\theta|x)g(\theta)$ rather than just maximizing $\mathcal{L}(\theta|x)$. This is called *maximum a-posteriori* (MAP) estimation. Since MAP constrains the parameter values, you can think of it as a regularized version of MLE.

Notice that maximizing the likelihood function is equivalent to maximizing its logarithm (represented in the lower-righthand plot in Figure 9-19). Indeed the logarithm is a strictly increasing function, so if θ maximizes the log likelihood, it also maximizes

the likelihood. It turns out that it is generally easier to maximize the log likelihood. For example, if you observed several independent instances $x^{(1)}$ to $x^{(m)}$, you would need to find the value of θ that maximizes the product of the individual likelihood functions. But it is equivalent, and much simpler, to maximize the sum (not the product) of the log likelihood functions, thanks to the magic of the logarithm which converts products into sums: $\log(ab)=\log(a)+\log(b)$.

Once you have estimated $\hat{\theta}$, the value of θ that maximizes the likelihood function, then you are ready to compute $\hat{\mathcal{L}} = \mathcal{L}(\hat{\theta}, \mathbf{X})$, which is the value used to compute the AIC and BIC; you can think of it as a measure of how well the model fits the data.

To compute the BIC and AIC, call the `bic()` and `aic()` methods:

```
>>> gm.bic(X)
8189.747000497186
>>> gm.aic(X)
8102.521720382148
```

[Figure 9-20](#) shows the BIC for different numbers of clusters k . As you can see, both the BIC and the AIC are lowest when $k=3$, so it is most likely the best choice.

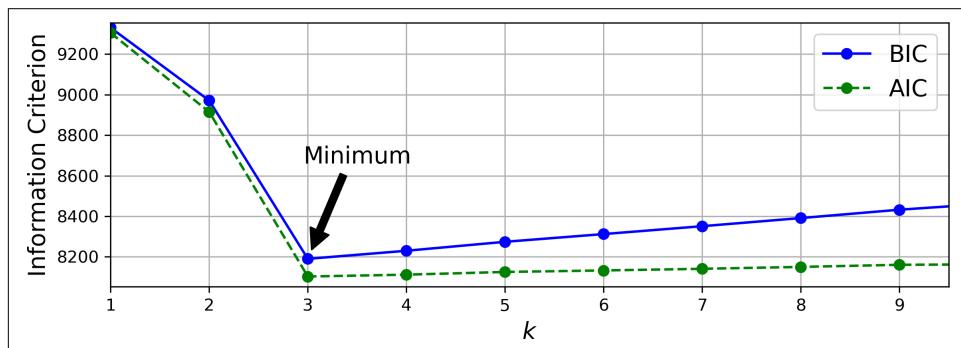


Figure 9-20. AIC and BIC for different numbers of clusters k

Bayesian Gaussian Mixture Models

Rather than manually searching for the optimal number of clusters, you can use the `BayesianGaussianMixture` class, which is capable of giving weights equal (or close) to zero to unnecessary clusters. Set the number of clusters `n_components` to a value that you have good reason to believe is greater than the optimal number of clusters (this assumes some minimal knowledge about the problem at hand), and the algorithm will eliminate the unnecessary clusters automatically. For example, let's set the number of clusters to 10 and see what happens:

```
>>> from sklearn.mixture import BayesianGaussianMixture
>>> bgm = BayesianGaussianMixture(n_components=10, n_init=10, random_state=42)
```

```

>>> bgm.fit(X)
>>> bgm.weights_.round(2)
array([0.4 , 0.21, 0.4 , 0. , 0. , 0. , 0. , 0. , 0. , 0. ])

```

Perfect: the algorithm automatically detected that only three clusters are needed, and the resulting clusters are almost identical to the ones in [Figure 9-16](#).

A final note about Gaussian mixture models: although they work great on clusters with ellipsoidal shapes, they don't do so well with clusters of very different shapes. For example, let's see what happens if we use a Bayesian Gaussian mixture model to cluster the moons dataset (see [Figure 9-21](#)).

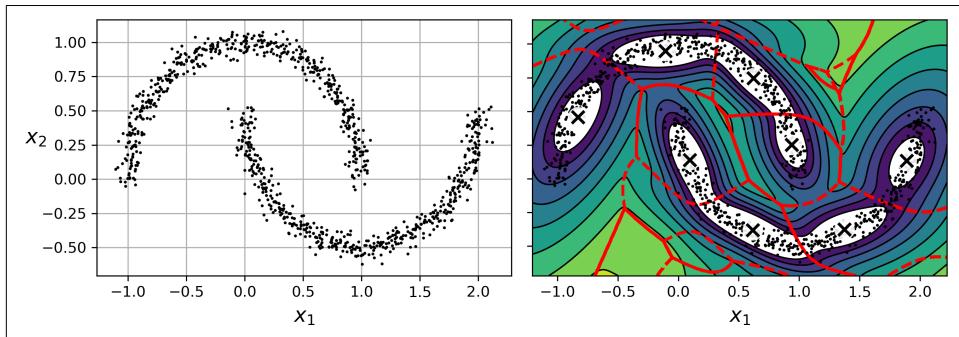


Figure 9-21. Fitting a Gaussian mixture to nonellipsoidal clusters

Oops! The algorithm desperately searched for ellipsoids, so it found eight different clusters instead of two. The density estimation is not too bad, so this model could perhaps be used for anomaly detection, but it failed to identify the two moons. To conclude this chapter, let's take a quick look at a few algorithms capable of dealing with arbitrarily shaped clusters.

Other Algorithms for Anomaly and Novelty Detection

Scikit-Learn implements other algorithms dedicated to anomaly detection or novelty detection:

Fast-MCD (minimum covariance determinant)

Implemented by the `EllipticEnvelope` class, this algorithm is useful for outlier detection, in particular to clean up a dataset. It assumes that the normal instances (inliers) are generated from a single Gaussian distribution (not a mixture). It also assumes that the dataset is contaminated with outliers that were not generated from this Gaussian distribution. When the algorithm estimates the parameters of the Gaussian distribution (i.e., the shape of the elliptic envelope around the inliers), it is careful to ignore the instances that are most likely outliers. This technique gives a better estimation of the elliptic envelope and thus makes the algorithm better at identifying the outliers.

Isolation Forest

This is an efficient algorithm for outlier detection, especially in high-dimensional datasets. The algorithm builds a Random Forest in which each Decision Tree is grown randomly: at each node, it picks a feature randomly, then it picks a random threshold value (between the min and max values) to split the dataset in two. The dataset gradually gets chopped into pieces this way, until all instances end up isolated from the other instances. Anomalies are usually far from other instances, so on average (across all the Decision Trees) they tend to get isolated in fewer steps than normal instances.

Local Outlier Factor (LOF)

This algorithm is also good for outlier detection. It compares the density of instances around a given instance to the density around its neighbors. An anomaly is often more isolated than its k nearest neighbors.

One-class SVM

This algorithm is better suited for novelty detection. Recall that a kernelized SVM classifier separates two classes by first (implicitly) mapping all the instances to a high-dimensional space, then separating the two classes using a linear SVM classifier within this high-dimensional space (see [Chapter 5](#)). Since we just have one class of instances, the one-class SVM algorithm instead tries to separate the instances in high-dimensional space from the origin. In the original space, this will correspond to finding a small region that encompasses all the instances. If a new instance does not fall within this region, it is an anomaly. There are a few hyperparameters to tweak: the usual ones for a kernelized SVM, plus a margin hyperparameter that corresponds to the probability of a new instance being mistakenly considered as novel when it is in fact normal. It works great, especially with high-dimensional datasets, but like all SVMs it does not scale to large datasets.

PCA and other dimensionality reduction techniques with an `inverse_transform()` method

If you compare the reconstruction error of a normal instance with the reconstruction error of an anomaly, the latter will usually be much larger. This is a simple and often quite efficient anomaly detection approach (see this chapter's exercises for an example).

Exercises

1. How would you define clustering? Can you name a few clustering algorithms?
2. What are some of the main applications of clustering algorithms?
3. Describe two techniques to select the right number of clusters when using K-Means.

4. What is label propagation? Why would you implement it, and how?
5. Can you name two clustering algorithms that can scale to large datasets? And two that look for regions of high density?
6. Can you think of a use case where active learning would be useful? How would you implement it?
7. What is the difference between anomaly detection and novelty detection?
8. What is a Gaussian mixture? What tasks can you use it for?
9. Can you name two techniques to find the right number of clusters when using a Gaussian mixture model?
10. The classic Olivetti faces dataset contains 400 grayscale 64×64 -pixel images of faces. Each image is flattened to a 1D vector of size 4,096. 40 different people were photographed (10 times each), and the usual task is to train a model that can predict which person is represented in each picture. Load the dataset using the `sklearn.datasets.fetch_olivetti_faces()` function, then split it into a training set, a validation set, and a test set (note that the dataset is already scaled between 0 and 1). Since the dataset is quite small, you probably want to use stratified sampling to ensure that there are the same number of images per person in each set. Next, cluster the images using K-Means, and ensure that you have a good number of clusters (using one of the techniques discussed in this chapter). Visualize the clusters: do you see similar faces in each cluster?
11. Continuing with the Olivetti faces dataset, train a classifier to predict which person is represented in each picture, and evaluate it on the validation set. Next, use K-Means as a dimensionality reduction tool, and train a classifier on the reduced set. Search for the number of clusters that allows the classifier to get the best performance: what performance can you reach? What if you append the features from the reduced set to the original features (again, searching for the best number of clusters)?
12. Train a Gaussian mixture model on the Olivetti faces dataset. To speed up the algorithm, you should probably reduce the dataset's dimensionality (e.g., use PCA, preserving 99% of the variance). Use the model to generate some new faces (using the `sample()` method), and visualize them (if you used PCA, you will need to use its `inverse_transform()` method). Try to modify some images (e.g., rotate, flip, darken) and see if the model can detect the anomalies (i.e., compare the output of the `score_samples()` method for normal images and for anomalies).
13. Some dimensionality reduction techniques can also be used for anomaly detection. For example, take the Olivetti faces dataset and reduce it with PCA, preserving 99% of the variance. Then compute the reconstruction error for each image. Next, take some of the modified images you built in the previous exercise, and

look at their reconstruction error: notice how much larger the reconstruction error is. If you plot a reconstructed image, you will see why: it tries to reconstruct a normal face.

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab3>.

PART II

Neural Networks and Deep Learning

CHAPTER 10

Introduction to Artificial Neural Networks with Keras

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 10th chapter of the final book. The GitHub repo is <https://github.com/ageron/handsonml3>.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

Birds inspired us to fly, burdock plants inspired Velcro, and nature has inspired countless more inventions. It seems only logical, then, to look at the brain’s architecture for inspiration on how to build an intelligent machine. This is the logic that sparked *artificial neural networks* (ANNs): an ANN is a Machine Learning model inspired by the networks of biological neurons found in our brains. However, although planes were inspired by birds, they don’t have to flap their wings. Similarly, ANNs have gradually become quite different from their biological cousins. Some researchers even argue that we should drop the biological analogy altogether (e.g., by saying “units” rather than “neurons”), lest we restrict our creativity to biologically plausible systems.¹

¹ You can get the best of both worlds by being open to biological inspirations without being afraid to create biologically unrealistic models, as long as they work well.

ANNs are at the very core of Deep Learning. They are versatile, powerful, and scalable, making them ideal to tackle large and highly complex Machine Learning tasks such as classifying billions of images (e.g., Google Images), powering speech recognition services (e.g., Apple's Siri), recommending the best videos to watch to hundreds of millions of users every day (e.g., YouTube), or learning to beat the world champion at the game of Go (DeepMind's AlphaGo).

The first part of this chapter introduces artificial neural networks, starting with a quick tour of the very first ANN architectures and leading up to *Multilayer Perceptrons* (MLPs), which are heavily used today (other architectures will be explored in the next chapters). In the second part, we will look at how to implement neural networks using TensorFlow's Keras API. This is a beautifully designed and simple high-level API for building, training, evaluating, and running neural networks. But don't be fooled by its simplicity: it is expressive and flexible enough to let you build a wide variety of neural network architectures. In fact, it will probably be sufficient for most of your use cases. And should you ever need extra flexibility, you can always write custom Keras components using its lower-level API, or even use TensorFlow directly, as we will see in [Chapter 12](#).

But first, let's go back in time to see how artificial neural networks came to be!

From Biological to Artificial Neurons

Surprisingly, ANNs have been around for quite a while: they were first introduced back in 1943 by the neurophysiologist Warren McCulloch and the mathematician Walter Pitts. In their [landmark paper](#)² "A Logical Calculus of Ideas Immanent in Nervous Activity," McCulloch and Pitts presented a simplified computational model of how biological neurons might work together in animal brains to perform complex computations using *propositional logic*. This was the first artificial neural network architecture. Since then many other architectures have been invented, as we will see.

The early successes of ANNs led to the widespread belief that we would soon be conversing with truly intelligent machines. When it became clear in the 1960s that this promise would go unfulfilled (at least for quite a while), funding flew elsewhere, and ANNs entered a long winter. In the early 1980s, new architectures were invented and better training techniques were developed, sparking a revival of interest in *connectionism*, the study of neural networks. But progress was slow, and by the 1990s other powerful Machine Learning techniques were invented, such as Support Vector Machines (see [Chapter 5](#)). These techniques seemed to offer better results

² Warren S. McCulloch and Walter Pitts, "A Logical Calculus of the Ideas Immanent in Nervous Activity," *The Bulletin of Mathematical Biology* 5, no. 4 (1943): 115–113.

and stronger theoretical foundations than ANNs, so once again the study of neural networks was put on hold.

We are now witnessing yet another wave of interest in ANNs. Will this wave die out like the previous ones did? Well, here are a few good reasons to believe that this time is different and that the renewed interest in ANNs will have a much more profound impact on our lives:

- There is now a huge quantity of data available to train neural networks, and ANNs frequently outperform other ML techniques on very large and complex problems.
- The tremendous increase in computing power since the 1990s now makes it possible to train large neural networks in a reasonable amount of time. This is in part due to Moore's law (the number of components in integrated circuits has doubled about every 2 years over the last 50 years), but also thanks to the gaming industry, which has stimulated the production of powerful GPU cards by the millions. Moreover, cloud platforms have made this power accessible to everyone.
- The training algorithms have been improved. To be fair they are only slightly different from the ones used in the 1990s, but these relatively small tweaks have had a huge positive impact.
- Some theoretical limitations of ANNs have turned out to be benign in practice. For example, many people thought that ANN training algorithms were doomed because they were likely to get stuck in local optima, but it turns out that this is not a big problem in practice, especially for larger neural networks: the local optima often perform almost as well as the global optimum.
- ANNs seem to have entered a virtuous circle of funding and progress. Amazing products based on ANNs regularly make the headline news, which pulls more and more attention and funding toward them, resulting in more and more progress and even more amazing products.

Biological Neurons

Before we discuss artificial neurons, let's take a quick look at a biological neuron (represented in [Figure 10-1](#)). It is an unusual-looking cell mostly found in animal brains. It's composed of a *cell body* containing the nucleus and most of the cell's complex components, many branching extensions called *dendrites*, plus one very long extension called the *axon*. The axon's length may be just a few times longer than the cell body, or up to tens of thousands of times longer. Near its extremity the axon splits off into many branches called *telodendria*, and at the tip of these branches are minuscule structures called *synaptic terminals* (or simply *synapses*), which are connected

to the dendrites or cell bodies of other neurons.³ Biological neurons produce short electrical impulses called *action potentials* (APs, or just *signals*) which travel along the axons and make the synapses release chemical signals called *neurotransmitters*. When a neuron receives a sufficient amount of these neurotransmitters within a few milliseconds, it fires its own electrical impulses (actually, it depends on the neurotransmitters, as some of them inhibit the neuron from firing).

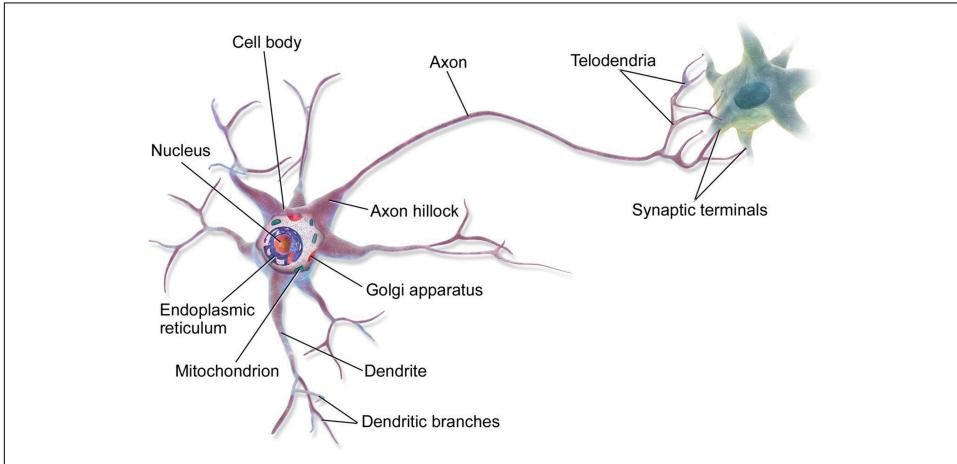


Figure 10-1. Biological neuron⁴

Thus, individual biological neurons seem to behave in a rather simple way, but they are organized in a vast network of billions, with each neuron typically connected to thousands of other neurons. Highly complex computations can be performed by a network of fairly simple neurons, much like a complex anthill can emerge from the combined efforts of simple ants. The architecture of biological neural networks (BNNs)⁵ is still the subject of active research, but some parts of the brain have been mapped, and it seems that neurons are often organized in consecutive layers, especially in the cerebral cortex (i.e., the outer layer of your brain), as shown in Figure 10-2.

³ They are not actually attached, just so close that they can very quickly exchange chemical signals.

⁴ Image by Bruce Blaus ([Creative Commons 3.0](https://en.wikipedia.org/wiki/Neuron)). Reproduced from <https://en.wikipedia.org/wiki/Neuron>.

⁵ In the context of Machine Learning, the phrase “neural networks” generally refers to ANNs, not BNNs.

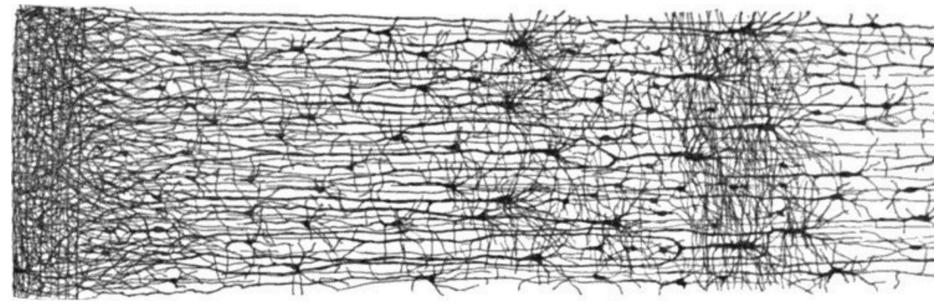


Figure 10-2. Multiple layers in a biological neural network (human cortex)⁶

Logical Computations with Neurons

McCulloch and Pitts proposed a very simple model of the biological neuron, which later became known as an *artificial neuron*: it has one or more binary (on/off) inputs and one binary output. The artificial neuron activates its output when more than a certain number of its inputs are active. In their paper, they showed that even with such a simplified model it is possible to build a network of artificial neurons that computes any logical proposition you want. To see how such a network works, let's build a few ANNs that perform various logical computations (see Figure 10-3), assuming that a neuron is activated when at least two of its input connections are active.

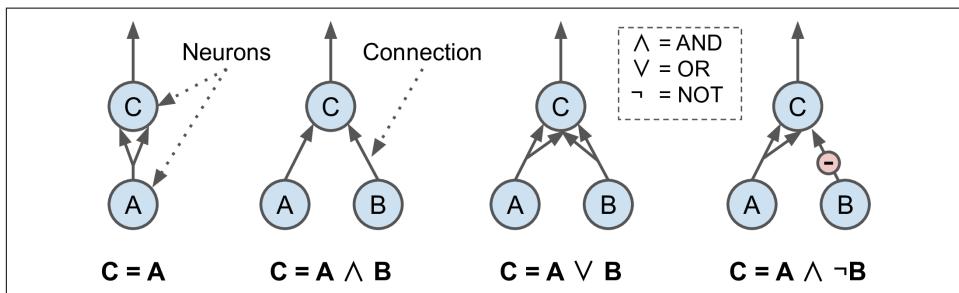


Figure 10-3. ANNs performing simple logical computations

Let's see what these networks do:

- The first network on the left is the identity function: if neuron A is activated, then neuron C gets activated as well (since it receives two input signals from neuron A); but if neuron A is off, then neuron C is off as well.

⁶ Drawing of a cortical lamination by S. Ramon y Cajal (public domain). Reproduced from https://en.wikipedia.org/wiki/Cerebral_cortex.

- The second network performs a logical AND: neuron C is activated only when both neurons A and B are activated (a single input signal is not enough to activate neuron C).
- The third network performs a logical OR: neuron C gets activated if either neuron A or neuron B is activated (or both).
- Finally, if we suppose that an input connection can inhibit the neuron's activity (which is the case with biological neurons), then the fourth network computes a slightly more complex logical proposition: neuron C is activated only if neuron A is active and neuron B is off. If neuron A is active all the time, then you get a logical NOT: neuron C is active when neuron B is off, and vice versa.

You can imagine how these networks can be combined to compute complex logical expressions (see the exercises at the end of the chapter for an example).

The Perceptron

The *Perceptron* is one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt. It is based on a slightly different artificial neuron (see [Figure 10-4](#)) called a *threshold logic unit* (TLU), or sometimes a *linear threshold unit* (LTU). The inputs and output are numbers (instead of binary on/off values), and each input connection is associated with a weight. The TLU first computes a linear function of its inputs: $z = w_1 x_1 + w_2 x_2 + \dots + w_n x_n + b = \mathbf{w}^T \mathbf{x} + b$. Then it applies a *step function* to the result: $h_{\mathbf{w}}(\mathbf{x}) = \text{step}(z)$. So it's almost like Logistic Regression, except it uses a step function instead of the logistic function ([Chapter 4](#)). Just like in Logistic Regression, the model parameters are the input weights \mathbf{w} and the bias term b .

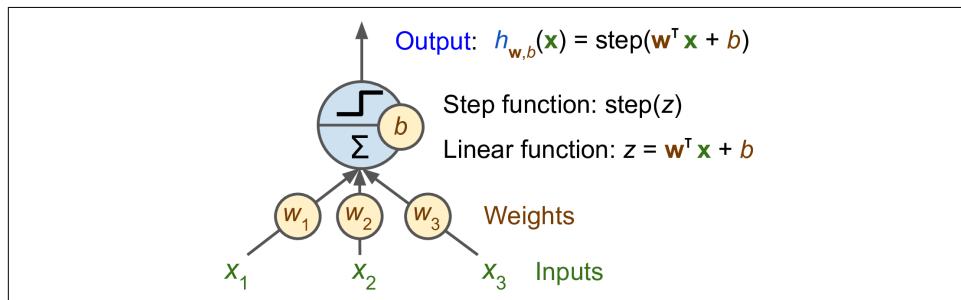


Figure 10-4. Threshold logic unit: an artificial neuron which computes a weighted sum of its inputs $\mathbf{w}^T \mathbf{x}$, plus a bias term b , then applies a step function

The most common step function used in Perceptrons is the *Heaviside step function* (see [Equation 10-1](#)). Sometimes the sign function is used instead.

Equation 10-1. Common step functions used in Perceptrons (assuming threshold = 0)

$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \quad \text{sgn}(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

A single TLU can be used for simple linear binary classification. It computes a linear function of its inputs, and if the result exceeds a threshold, it outputs the positive class. Otherwise it outputs the negative class. This may remind you of Logistic Regression ([Chapter 4](#)) or linear SVM classification ([Chapter 5](#)). You could, for example, use a single TLU to classify iris flowers based on petal length and width. Training such a TLU would require finding the right values for w_1 , w_2 and b (the training algorithm is discussed shortly).

A Perceptron is composed of one or more TLUs organized in a single layer, where every TLU is connected to every input. Such a layer is called a *fully connected layer*, or a *dense layer*. The inputs constitute the *input layer*. And since the layer of TLUs produces the final outputs, it is called the *output layer*. For example, a Perceptron with two inputs and three outputs is represented in [Figure 10-5](#).

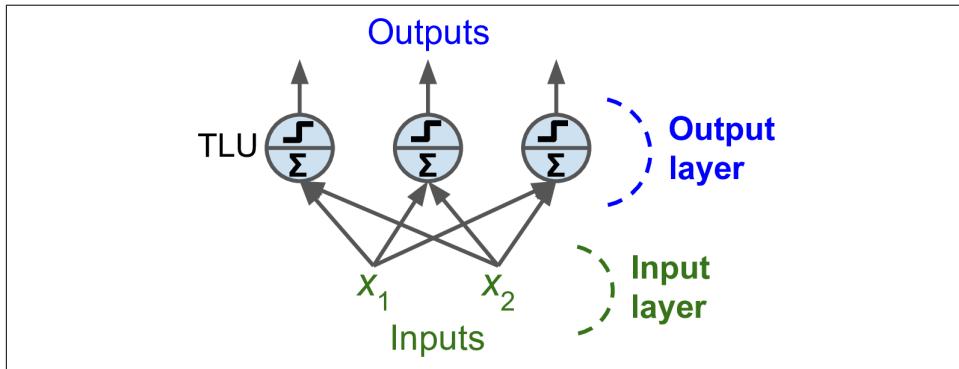


Figure 10-5. Architecture of a Perceptron with two inputs and three output neurons

This Perceptron can classify instances simultaneously into three different binary classes, which makes it a multilabel classifier. It may also be used for multiclass classification.

Thanks to the magic of linear algebra, [Equation 10-2](#) can be used to efficiently compute the outputs of a layer of artificial neurons for several instances at once.

Equation 10-2. Computing the outputs of a fully connected layer

$$h_{\mathbf{W}, \mathbf{b}}(\mathbf{X}) = \phi(\mathbf{X}\mathbf{W} + \mathbf{b})$$

In this equation:

- As always, \mathbf{X} represents the matrix of input features. It has one row per instance and one column per feature.
- The weight matrix \mathbf{W} contains all the connection weights. It has one row per input and one column per neuron.
- The bias vector \mathbf{b} contains all the bias terms: one per neuron.
- The function ϕ is called the *activation function*: when the artificial neurons are TLUs, it is a step function (but we will discuss other activation functions shortly).



In mathematics, the sum of a matrix and a vector is undefined. However, in Data Science, we allow “broadcasting”: adding a vector to a matrix means adding it to every row in the matrix. So $\mathbf{X}\mathbf{W} + \mathbf{b}$ first multiplies \mathbf{X} by \mathbf{W} —which results in a matrix with one row per instance and one column per output—then adds the vector \mathbf{b} to every row of that matrix, which adds each bias term to the corresponding output, for every instance. Moreover, ϕ is applied itemwise to each item in the resulting matrix.

So, how is a Perceptron trained? The Perceptron training algorithm proposed by Rosenblatt was largely inspired by *Hebb’s rule*. In his 1949 book *The Organization of Behavior* (Wiley), Donald Hebb suggested that when a biological neuron triggers another neuron often, the connection between these two neurons grows stronger. Siegrid Löwel later summarized Hebb’s idea in the catchy phrase, “Cells that fire together, wire together”; that is, the connection weight between two neurons tends to increase when they fire simultaneously. This rule later became known as Hebb’s rule (or *Hebbian learning*). Perceptrons are trained using a variant of this rule that takes into account the error made by the network when it makes a prediction; the Perceptron learning rule reinforces connections that help reduce the error. More specifically, the Perceptron is fed one training instance at a time, and for each instance it makes its predictions. For every output neuron that produced a wrong prediction, it reinforces

the connection weights from the inputs that would have contributed to the correct prediction. The rule is shown in [Equation 10-3](#).

Equation 10-3. Perceptron learning rule (weight update)

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i$$

In this equation:

- $w_{i,j}$ is the connection weight between the i^{th} input and the j^{th} neuron.
- x_i is the i^{th} input value of the current training instance.
- \hat{y}_j is the output of the j^{th} output neuron for the current training instance.
- y_j is the target output of the j^{th} output neuron for the current training instance.
- η is the learning rate (see [Chapter 4](#)).

The decision boundary of each output neuron is linear, so Perceptrons are incapable of learning complex patterns (just like Logistic Regression classifiers). However, if the training instances are linearly separable, Rosenblatt demonstrated that this algorithm would converge to a solution.⁷ This is called the *Perceptron convergence theorem*.

Scikit-Learn provides a `Perceptron` class which can be used pretty much as you would expect—for example, on the iris dataset (introduced in [Chapter 4](#)):

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris(as_frame=True)
X = iris.data[["petal length (cm)", "petal width (cm)"]].values
y = (iris.target == 0) # Iris setosa

per_clf = Perceptron(random_state=42)
per_clf.fit(X, y)

X_new = [[2, 0.5], [3, 1]]
y_pred = per_clf.predict(X_new) # predicts True and False for these 2 flowers
```

You may have noticed that the Perceptron learning algorithm strongly resembles Stochastic Gradient Descent (introduced in [Chapter 4](#)). In fact, Scikit-Learn's `Perceptron` class is equivalent to using an `SGDClassifier` with the following hyperparameters: `loss="perceptron"`, `learning_rate="constant"`, `eta0=1` (the learning rate), and `penalty=None` (no regularization).

⁷ Note that this solution is not unique: when data points are linearly separable, there is an infinity of hyperplanes that can separate them.



Contrary to Logistic Regression classifiers, Perceptrons do not output a class probability. This is one reason to prefer Logistic Regression over Perceptrons. Moreover, Perceptrons do not use any regularization by default, and training stops as soon as there are no more prediction errors on the training set, so the model typically does not generalize as well as Logistic Regression or a linear SVM classifier. However, Perceptrons may train a bit faster.

In their 1969 monograph *Perceptrons*, Marvin Minsky and Seymour Papert highlighted a number of serious weaknesses of Perceptrons—in particular, the fact that they are incapable of solving some trivial problems (e.g., the *Exclusive OR* (XOR) classification problem; see the left side of Figure 10-6). This is true of any other linear classification model (such as Logistic Regression classifiers), but researchers had expected much more from Perceptrons, and some were so disappointed that they dropped neural networks altogether in favor of higher-level problems such as logic, problem solving, and search. The lack of practical applications also didn't help.

It turns out that some of the limitations of Perceptrons can be eliminated by stacking multiple Perceptrons. The resulting ANN is called a *Multilayer Perceptron* (MLP). An MLP can solve the XOR problem, as you can verify by computing the output of the MLP represented on the right side of Figure 10-6: with inputs $(0, 0)$ or $(1, 1)$, the network outputs 0, and with inputs $(0, 1)$ or $(1, 0)$ it outputs 1. Try verifying that this network indeed solves the XOR problem!⁸

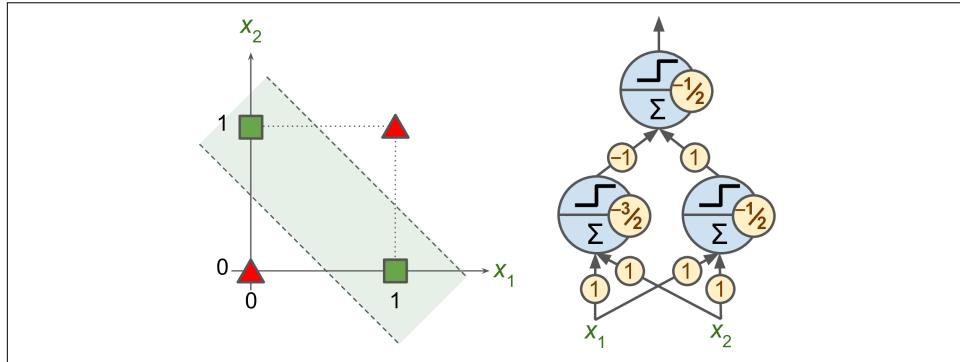


Figure 10-6. XOR classification problem and an MLP that solves it

⁸ For example, when the inputs are $(0, 1)$, the lower left neuron computes $0 \times 1 + 1 \times 1 - 3/2 = -1/2$, which is negative so it outputs 0. The lower right neuron computes $0 \times 1 + 1 \times 1 - 1/2 = 1/2$ which is positive so it outputs 1. The output neuron receives the outputs of the first two neurons as its inputs, so it computes $0 \times (-1) + 1 \times 1 - 1/2 = 1/2$, which is positive so it outputs 1.

The Multilayer Perceptron and Backpropagation

An MLP is composed of one input layer, one or more layers of TLUs, called *hidden layers*, and one final layer of TLUs called the *output layer* (see [Figure 10-7](#)). The layers close to the input layer are usually called the *lower layers*, and the ones close to the outputs are usually called the *upper layers*.

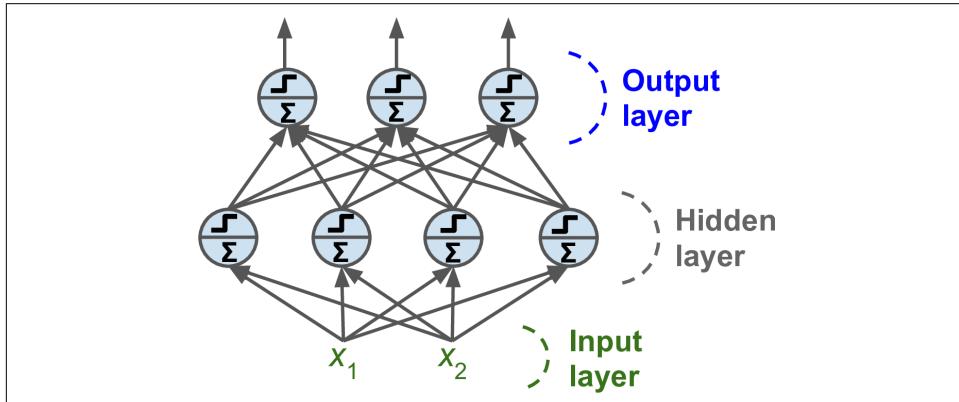


Figure 10-7. Architecture of a Multilayer Perceptron with two inputs, one hidden layer of four neurons, and three output neurons



The signal flows only in one direction (from the inputs to the outputs), so this architecture is an example of a *feedforward neural network* (FNN).

When an ANN contains a deep stack of hidden layers,⁹ it is called a *deep neural network* (DNN). The field of Deep Learning studies DNNs, and more generally it is interested in models containing deep stacks of computations. Even so, many people talk about Deep Learning whenever neural networks are involved (even shallow ones).

For many years researchers struggled to find a way to train MLPs, without success. In the early 1960s, several researchers discussed the possibility to use Gradient Descent to train neural networks, but as we saw in [Chapter 4](#), Gradient Descent requires computing the gradients of the model's error with regards to the model parameters: it wasn't clear at the time how to do this efficiently with such a complex model containing so many parameters, especially with the computers they had back then.

⁹ In the 1990s, an ANN with more than two hidden layers was considered deep. Nowadays, it is common to see ANNs with dozens of layers, or even hundreds, so the definition of "deep" is quite fuzzy.

Then in 1970, a researcher named Seppo Linnainmaa introduced, in his master thesis, a technique to compute all the gradients automatically and efficiently. This algorithm is now called *reverse-mode automatic differentiation* (or *reverse-mode autodiff* for short). In just two passes through the network (one forward, one backward), it is able to compute the gradients of the neural network's error with regard to every single model parameter. In other words, it can find out how each connection weight and each bias should be tweaked in order to reduce the neural network's error. These gradients can then be used to perform a Gradient Descent step. If you repeat this process of computing the gradients automatically and taking a Gradient Descent step, the neural network's error will gradually drop, until it eventually reaches a minimum. This combination of reverse-mode autodiff and Gradient Descent is now called *backpropagation* (or *backprop* for short).



There are various autodiff techniques, with different pros and cons. *Reverse-mode autodiff* is well suited when the function to differentiate has many variables (e.g., connection weights and biases) and few outputs (e.g., one loss). If you want to learn more about autodiff, check out [Appendix B](#).

Backpropagation can actually be applied to all sorts of computational graphs, not just neural networks: indeed, Linnainmaa's master thesis was not about neural nets, it was more general. It took several more years before backprop started to be used to train neural networks, but it still wasn't mainstream. Then David Rumelhart reinvented backpropagation (as did several other researchers in other fields), and he published a [groundbreaking paper¹⁰](#) in 1986, along with Geoffrey Hinton and Ronald Williams, analyzing how backpropagation allowed neural networks to learn useful internal representations. Their results were so impressive that backpropagation was quickly popularized in the field. Today, it is by far the most popular training technique for neural networks.

So let's run through backpropagation again in a bit more detail:

- It handles one mini-batch at a time (for example, containing 32 instances each), and it goes through the full training set multiple times. Each pass is called an *epoch*.
- Each mini-batch enters the network through the input layer. The algorithm then computes the output of all the neurons in the first hidden layer, for every instance in the mini-batch. The result is passed on to the next layer, its output is computed and passed to the next layer, and so on until we get the output of

¹⁰ David Rumelhart et al. "Learning Internal Representations by Error Propagation," (Defense Technical Information Center technical report, September 1985).

the last layer, the output layer. This is the *forward pass*: it is exactly like making predictions, except all intermediate results are preserved since they are needed for the backward pass.

- Next, the algorithm measures the network's output error (i.e., it uses a loss function that compares the desired output and the actual output of the network, and returns some measure of the error).
- Then it computes how much each output bias and each connection to the output layer contributed to the error. This is done analytically by applying the *chain rule* (perhaps the most fundamental rule in calculus), which makes this step fast and precise.
- The algorithm then measures how much of these error contributions came from each connection in the layer below, again using the chain rule, working backward until the algorithm reaches the input layer. As explained earlier, this reverse pass efficiently measures the error gradient across all the connection weights and biases in the network by propagating the error gradient backward through the network (hence the name of the algorithm).
- Finally, the algorithm performs a Gradient Descent step to tweak all the connection weights in the network, using the error gradients it just computed.

In short, backpropagation makes predictions for a mini-batch (forward pass), measures the error, then goes through each layer in reverse to measure the error contribution from each parameter (reverse pass), and finally tweaks the connection weights and biases to reduce the error (Gradient Descent step).



It is important to initialize all the hidden layers' connection weights randomly, or else training will fail. For example, if you initialize all weights and biases to zero, then all neurons in a given layer will be perfectly identical, and thus backpropagation will affect them in exactly the same way, so they will remain identical. In other words, despite having hundreds of neurons per layer, your model will act as if it had only one neuron per layer: it won't be too smart. If instead you randomly initialize the weights, you *break the symmetry* and allow backpropagation to train a diverse team of neurons.

In order for backprop to work properly, Rumelhart and his colleagues made a key change to the MLP's architecture: they replaced the step function with the logistic function, $\sigma(z) = 1 / (1 + \exp(-z))$, also called the *sigmoid* function. This was essential because the step function contains only flat segments, so there is no gradient to work with (Gradient Descent cannot move on a flat surface), while the sigmoid function has a well-defined nonzero derivative everywhere, allowing Gradient Descent to make some progress at every step. In fact, the backpropagation algorithm works well

with many other activation functions, not just the sigmoid function. Here are two other popular choices:

The hyperbolic tangent function: $\tanh(z) = 2\sigma(2z) - 1$

Just like the sigmoid function, this activation function is S-shaped, continuous, and differentiable, but its output value ranges from -1 to 1 (instead of 0 to 1 in the case of the sigmoid function). That range tends to make each layer's output more or less centered around 0 at the beginning of training, which often helps speed up convergence.

The Rectified Linear Unit function: $\text{ReLU}(z) = \max(0, z)$

The ReLU function is continuous but unfortunately not differentiable at $z = 0$ (the slope changes abruptly, which can make Gradient Descent bounce around), and its derivative is 0 for $z < 0$. In practice, however, it works very well and has the advantage of being fast to compute, so it has become the default.¹¹ Importantly, the fact that it does not have a maximum output value helps reduce some issues during Gradient Descent (we will come back to this in [Chapter 11](#)).

These popular activation functions and their derivatives are represented in [Figure 10-8](#). But wait! Why do we need activation functions in the first place? Well, if you chain several linear transformations, all you get is a linear transformation. For example, if $f(x) = 2x + 3$ and $g(x) = 5x - 1$, then chaining these two linear functions gives you another linear function: $f(g(x)) = 2(5x - 1) + 3 = 10x + 1$. So if you don't have some nonlinearity between layers, then even a deep stack of layers is equivalent to a single layer, and you can't solve very complex problems with that. Conversely, a large enough DNN with nonlinear activations can theoretically approximate any continuous function.

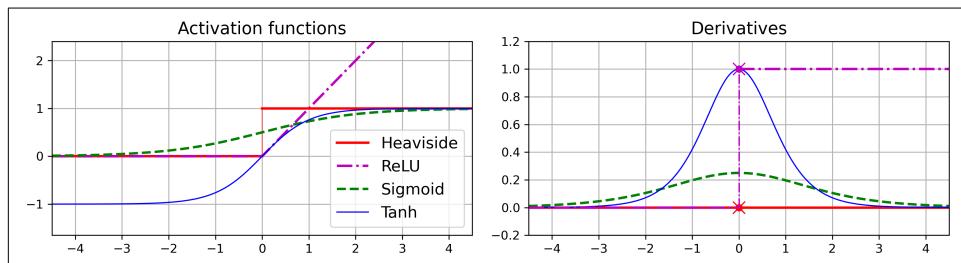


Figure 10-8. Activation functions (left) and their derivatives (right)

¹¹ Biological neurons seem to implement a roughly sigmoid (S-shaped) activation function, so researchers stuck to sigmoid functions for a very long time. But it turns out that ReLU generally works better in ANNs. This is one of the cases where the biological analogy was perhaps misleading.

OK! You know where neural nets came from, what their architecture is, and how to compute their outputs. You've also learned about the backpropagation algorithm. But what exactly can you do with neural nets?

Regression MLPs

First, MLPs can be used for regression tasks. If you want to predict a single value (e.g., the price of a house, given many of its features), then you just need a single output neuron: its output is the predicted value. For multivariate regression (i.e., to predict multiple values at once), you need one output neuron per output dimension. For example, to locate the center of an object in an image, you need to predict 2D coordinates, so you need two output neurons. If you also want to place a bounding box around the object, then you need two more numbers: the width and the height of the object. So, you end up with four output neurons.

Scikit-Learn includes an `MLPRegressor` class, so let's use it to build an MLP with three hidden layers composed of 50 neurons each, and train it on the California housing dataset. For simplicity, we will use Scikit-Learn's `fetch_california_housing()` function to load the data. This dataset is simpler than the one we used in [Chapter 2](#), since it contains only numerical features (there is no `ocean_proximity` feature), and there are no missing values. The following code starts by fetching and splitting the dataset, then it creates a pipeline to standardize the input features before sending them to the `MLPRegressor`. This is very important for neural networks because they are trained using Gradient Descent, and as we saw in [Chapter 4](#), Gradient Descent does not converge very well when the features have very different scales. Finally, the code trains the model and evaluates its validation error. The model uses the ReLU activation function in the hidden layers, and it uses a variant of Gradient Descent called *Adam* (see [Chapter 11](#)) to minimize the mean squared error, with a little bit of ℓ_2 regularization (which you can control via the `alpha` hyperparameter).

```
from sklearn.datasets import fetch_california_housing
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPRegressor
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()
X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target, random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full, random_state=42)

mlp_reg = MLPRegressor(hidden_layer_sizes=[50, 50, 50], random_state=42)
pipeline = make_pipeline(StandardScaler(), mlp_reg)
pipeline.fit(X_train, y_train)
```

```

y_pred = pipeline.predict(X_valid)
rmse = mean_squared_error(y_valid, y_pred, squared=False) # about 0.505

```

We get a validation RMSE of about 0.505, which is comparable to what you would get with a Random Forest classifier. Not too bad for a first try!

Note that this MLP does not use any activation function for the output layer, so it's free to output any value it wants. This is generally fine, but if you want to guarantee that the output will always be positive, then you should use the ReLU activation function in the output layer, or the *softplus* activation function, which is a smooth variant of ReLU: $\text{softplus}(z) = \log(1 + \exp(z))$. Softplus is close to 0 when z is negative, and close to z when z is positive. Finally, if you want to guarantee that the predictions will always fall within a given range of values, then you should use the sigmoid function or the hyperbolic tangent, and scale the targets to the appropriate range: 0 to 1 for sigmoid and -1 to 1 for tanh. Sadly, the `MLPRegressor` class does not support activation functions in the output layer.



Building and training a standard MLP with Scikit-Learn in just a few lines of code is very convenient, but the neural net features are limited. This is why we will switch to Keras in the second part of this chapter.

The `MLPRegressor` class uses the mean squared error, which is usually what you want for regression, but if you have a lot of outliers in the training set, you may prefer to use the mean absolute error instead. Alternatively, you may want to use the Huber loss, which is a combination of both. It is quadratic when the error is smaller than a threshold δ (typically 1) but linear when the error is larger than δ . The linear part makes it less sensitive to outliers than the mean squared error, and the quadratic part allows it to converge faster and be more precise than the mean absolute error. However, `MLPRegressor` only supports the MSE.

Table 10-1 summarizes the typical architecture of a regression MLP.

Table 10-1. Typical regression MLP architecture

Hyperparameter	Typical value
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per prediction dimension
Hidden activation	ReLU
Output activation	None, or ReLU/softplus (if positive outputs) or sigmoid/tanh (if bounded outputs)
Loss function	MSE, or Huber if outliers

Classification MLPs

MLPs can also be used for classification tasks. For a binary classification problem, you just need a single output neuron using the sigmoid activation function: the output will be a number between 0 and 1, which you can interpret as the estimated probability of the positive class. The estimated probability of the negative class is equal to one minus that number.

MLPs can also easily handle multilabel binary classification tasks (see [Chapter 3](#)). For example, you could have an email classification system that predicts whether each incoming email is ham or spam, and simultaneously predicts whether it is an urgent or nonurgent email. In this case, you would need two output neurons, both using the sigmoid activation function: the first would output the probability that the email is spam, and the second would output the probability that it is urgent. More generally, you would dedicate one output neuron for each positive class. Note that the output probabilities do not necessarily add up to 1. This lets the model output any combination of labels: you can have nonurgent ham, urgent ham, nonurgent spam, and perhaps even urgent spam (although that would probably be an error).

If each instance can belong only to a single class, out of three or more possible classes (e.g., classes 0 through 9 for digit image classification), then you need to have one output neuron per class, and you should use the softmax activation function for the whole output layer (see [Figure 10-9](#)). The softmax function (introduced in [Chapter 4](#)) will ensure that all the estimated probabilities are between 0 and 1, and that they add up to 1 since the classes are exclusive. This is called multiclass classification.

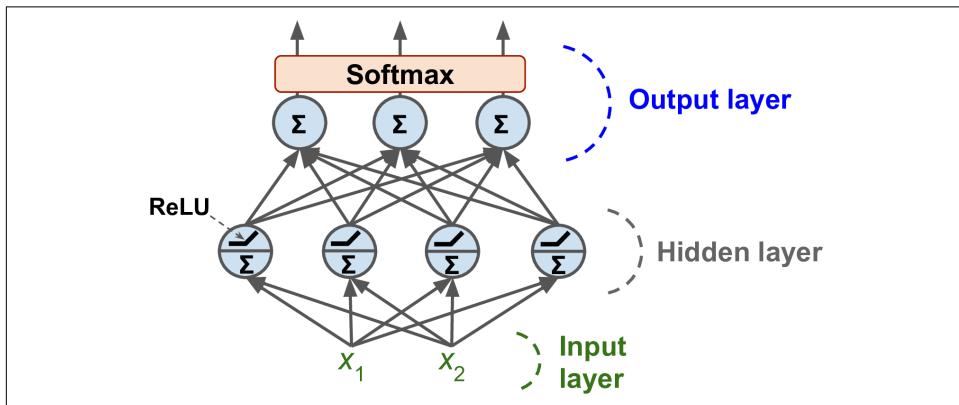


Figure 10-9. A modern MLP (including ReLU and softmax) for classification

Regarding the loss function, since we are predicting probability distributions, the cross-entropy loss (or x -entropy or log loss for short, see [Chapter 4](#)) is generally a good choice.

Scikit-Learn has an `MLPClassifier` class in the `sklearn.neural_network` package. It is almost identical to the `MLPRegressor` class, except that it minimizes the cross-entropy rather than the MSE. Give it a try now, for example on the Iris dataset. It's almost a linear task, so a single layer with 5 to 10 neurons should suffice (and make sure to scale the features).

Table 10-2 summarizes the typical architecture of a classification MLP.

Table 10-2. Typical classification MLP architecture

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
# hidden layers	Depends on the task, but typically 1 to 5		
# output neurons	1	1 per binary label	1 per class
Output layer activation	Sigmoid	Sigmoid	Softmax
Loss function	x-entropy	x-entropy	x-entropy



Before we go on, I recommend you go through exercise 1 at the end of this chapter. You will play with various neural network architectures and visualize their outputs using the *TensorFlow Playground*. This will be very useful to better understand MLPs, including the effects of all the hyperparameters (number of layers and neurons, activation functions, and more).

Now you have all the concepts you need to start implementing MLPs with Keras!

Implementing MLPs with Keras

Keras is TensorFlow's high-level Deep Learning API: it allows you to build, train, evaluate, and execute all sorts of neural networks. The original Keras library was developed by François Chollet as part of a research project¹² and was released as a standalone open source project in March 2015. It quickly gained popularity, owing to its ease of use, flexibility, and beautiful design.

¹² Project ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System).



Keras used to support multiple backends, including TensorFlow, PlaidML, Theano and Microsoft Cognitive Toolkit (CNTK) (the last two are sadly deprecated), but since version 2.4, Keras is TensorFlow-only. Conversely, TensorFlow used to include multiple high-level APIs, but Keras was officially chosen as its preferred high-level API when TensorFlow 2 came out. Installing TensorFlow will automatically install Keras as well, and Keras will not work without TensorFlow installed. In short, Keras and TensorFlow fell in love and got married. François Chollet joined Google where he continues to lead the Keras project. Other popular Deep Learning libraries include [PyTorch by Facebook](#) and [JAX by Google](#).¹³

All right, now let's use Keras! We will start by building an MLP for image classification.



Colab Runtimes come with recent versions of TensorFlow and Keras preinstalled. However, if you want to install them on your own machine, please see the installation instructions at <https://colab.info/install>.

Building an Image Classifier Using the Sequential API

First, we need to load a dataset. We will use Fashion MNIST, which is a drop-in replacement of MNIST (introduced in [Chapter 3](#)). It has the exact same format as MNIST (70,000 grayscale images of 28×28 pixels each, with 10 classes), but the images represent fashion items rather than handwritten digits, so each class is more diverse, and the problem turns out to be significantly more challenging than MNIST. For example, a simple linear model reaches about 92% accuracy on MNIST, but only about 83% on Fashion MNIST.

Using Keras to load the dataset

Keras provides some utility functions to fetch and load common datasets, including MNIST, Fashion MNIST, and a few more. Let's load Fashion MNIST. It's already

¹³ PyTorch's API is quite similar to Keras's, so once you know Keras, it is not difficult to switch to PyTorch, if you ever want to. PyTorch's popularity grew exponentially in 2018, largely thanks to its simplicity and excellent documentation, which were not TensorFlow 1.x's main strengths back then. However, TensorFlow 2 is just as simple as PyTorch, in part because it has adopted Keras as its official high-level API, and also because the developers have greatly simplified and cleaned up the rest of the API. The documentation has also been completely reorganized, and it is much easier to find what you need now. Similarly, PyTorch's main weaknesses (e.g., limited portability and no computation graph analysis) have been largely addressed in PyTorch 1.0. Healthy competition seems beneficial to everyone.

shuffled and split into a training set (60,000 images) and a test set (10,000 images), but let's hold out the last 5,000 images from the training set for validation:

```
import tensorflow as tf

fashion_mnist = tf.keras.datasets.fashion_mnist.load_data()
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist
X_train, y_train = X_train_full[:-5000], y_train_full[:-5000]
X_valid, y_valid = X_train_full[-5000:], y_train_full[-5000:]
```



TensorFlow is usually imported as `tf`, and the Keras API is available via `tf.keras`.

When loading MNIST or Fashion MNIST using Keras rather than Scikit-Learn, one important difference is that every image is represented as a 28×28 array rather than a 1D array of size 784. Moreover, the pixel intensities are represented as integers (from 0 to 255) rather than floats (from 0.0 to 255.0). Let's take a look at the shape and data type of the training set:

```
>>> X_train.shape
(55000, 28, 28)
>>> X_train.dtype
dtype('uint8')
```

For simplicity, we'll scale the pixel intensities down to the 0–1 range by dividing them by 255.0 (this also converts them to floats):

```
X_train, X_valid, X_test = X_train / 255., X_valid / 255., X_test / 255.
```

With MNIST, when the label is equal to 5, it means that the image represents the handwritten digit 5. Easy. For Fashion MNIST, however, we need the list of class names to know what we are dealing with:

```
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

For example, the first image in the training set represents an ankle boot:

```
>>> class_names[y_train[0]]
'Ankle boot'
```

Figure 10-10 shows some samples from the Fashion MNIST dataset.



Figure 10-10. Samples from Fashion MNIST

Creating the model using the Sequential API

Now let's build the neural network! Here is a classification MLP with two hidden layers:

```
tf.random.set_seed(42)
model = tf.keras.Sequential()
model.add(tf.keras.layers.Input(shape=[28, 28]))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(300, activation="relu"))
model.add(tf.keras.layers.Dense(100, activation="relu"))
model.add(tf.keras.layers.Dense(10, activation="softmax"))
```

Let's go through this code line by line:

- We first set TensorFlow's random seed to make the results reproducible: the random weights of the hidden layers and the output layer will be the same every time you run the notebook. You could also choose to use the `tf.keras.utils.set_random_seed()` function, which conveniently sets the random seeds for TensorFlow, Python (`random.seed()`), and NumPy (`np.random.seed()`).
- The next line creates a `Sequential` model. This is the simplest kind of Keras model for neural networks that are just composed of a single stack of layers connected sequentially. This is called the Sequential API.
- Next, we build the first layer and add it to the model: it is an `Input` layer. We specify the input `shape`, which doesn't include the batch size, only the shape of the instances. Keras needs to know the shape of the inputs so it can determine the shape of the connection weight matrix of the first hidden layer.
- Then we add a `Flatten` layer whose role is to convert each input image into a 1D array: for example, if it receives a batch of shape [32, 28, 28], it will reshape it to [32, 784]. In other words, if it receives input data `X`, it computes `X.reshape(-1,`

784). This layer does not have any parameters; it is just there to do some simple preprocessing.

- Next we add a `Dense` hidden layer with 300 neurons. It will use the `ReLU` activation function. Each `Dense` layer manages its own weight matrix, containing all the connection weights between the neurons and their inputs. It also manages a vector of bias terms (one per neuron). When it receives some input data, it computes [Equation 10-2](#).
- Then we add a second `Dense` hidden layer with 100 neurons, also using the `ReLU` activation function.
- Finally, we add a `Dense` output layer with 10 neurons (one per class), using the softmax activation function because the classes are exclusive.



Specifying `activation="relu"` is equivalent to specifying `activation=tf.keras.activations.relu`. Other activation functions are available in the `tf.keras.activations` package, we will use many of them in this book. See <https://keras.io/api/layers/activations/> for the full list. We will also define our own custom activation functions in [Chapter 12](#).

Instead of adding the layers one by one as we just did, it's often more convenient to pass a list of layers when creating the `Sequential` model. You can also drop the `Input` layer and instead specify the `input_shape` in the first layer:

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(300, activation="relu"),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax")
])
```

The model's `summary()` method displays all the model's layers,¹⁴ including each layer's name (which is automatically generated unless you set it when creating the layer), its output shape (None means the batch size can be anything), and its number of parameters. The summary ends with the total number of parameters, including trainable and non-trainable parameters. Here we only have trainable parameters (we will see some non-trainable parameters later in this chapter):

```
>>> model.summary()
Model: "sequential"
```

Layer (type)	Output Shape	Param #

¹⁴ You can also use `tf.keras.utils.plot_model()` to generate an image of your model.

```

=====
 flatten (Flatten)      (None, 784)          0
 dense (Dense)          (None, 300)          235500
 dense_1 (Dense)        (None, 100)          30100
 dense_2 (Dense)        (None, 10)           1010
 =====
 Total params: 266,610
 Trainable params: 266,610
 Non-trainable params: 0

```

Note that `Dense` layers often have a *lot* of parameters. For example, the first hidden layer has 784×300 connection weights, plus 300 bias terms, which adds up to 235,500 parameters! This gives the model quite a lot of flexibility to fit the training data, but it also means that the model runs the risk of overfitting, especially when you do not have a lot of training data. We will come back to this later.

All layers in a model must have a unique name (e.g., “`dense_2`”). You can set the layer names explicitly using the constructor’s `name` argument, but generally it’s simpler to let Keras name the layers automatically, as we just did: Keras takes the layer’s class name and converts it to snake case (e.g., a layer from the `MyCoolLayer` class is named “`my_cool_layer`” by default). Moreover, Keras ensures that the name is globally unique, even across models, by appending an index if needed, as in “`dense_2`”. But why, I hear you ask, does it bother making the names unique across models? Well, this makes it possible to merge models easily without getting name conflicts.



All global state managed by Keras is stored in a *Keras session*, which you can clear using `tf.keras.backend.clear_session()`. In particular, this resets the name counters.

You can easily get a model’s list of layers using the `layers` attribute, or use the `get_layer()` method to access a layer by name:

```

>>> model.layers
[<keras.layers.core.flatten.Flatten at 0x7fa1dea02250>,
 <keras.layers.core.dense.Dense at 0x7fa1c8f42520>,
 <keras.layers.core.dense.Dense at 0x7fa188be7ac0>,
 <keras.layers.core.dense.Dense at 0x7fa188be7fa0>]
>>> hidden1 = model.layers[1]
>>> hidden1.name
'dense'
>>> model.get_layer('dense') is hidden1
True

```

All the parameters of a layer can be accessed using its `get_weights()` and `set_weights()` methods. For a `Dense` layer, this includes both the connection weights and the bias terms:

```
>>> weights, biases = hidden1.get_weights()
>>> weights
array([[ 0.02448617, -0.00877795, -0.02189048, ...,  0.03859074, -0.06889391],
       [ 0.00476504, -0.03105379, -0.0586676 , ..., -0.02763776, -0.04165364],
       ...,
       [ 0.07061854, -0.06960931,  0.07038955, ...,  0.00034875,  0.02878492],
       [-0.06022581,  0.01577859, -0.02585464, ...,  0.00272203, -0.06793761]],
      dtype=float32)
>>> weights.shape
(784, 300)
>>> biases
array([0., 0., 0., 0., 0., 0., 0., 0., ..., 0., 0., 0.], dtype=float32)
>>> biases.shape
(300,)
```

Notice that the `Dense` layer initialized the connection weights randomly (which is needed to break symmetry, as we discussed earlier), and the biases were initialized to zeros, which is fine. If you ever want to use a different initialization method, you can set `kernel_initializer` (`kernel` is another name for the matrix of connection weights) or `bias_initializer` when creating the layer. We will discuss initializers further in [Chapter 11](#), but if you want the full list, see <https://keras.io/api/layers/initializers/>.



The shape of the weight matrix depends on the number of inputs, which is why we specified the `input_shape` when creating the model. If you do not specify the input shape, it's OK: Keras will simply wait until it knows the input shape before it actually builds the model parameters. This will happen either when you feed it some data (e.g., during training), or when you call its `build()` method. Until the model parameters are built, you will not be able to do certain things, such as display the model summary or save the model. So, if you know the input shape when creating the model, it is best to specify it.

Compiling the model

After a model is created, you must call its `compile()` method to specify the loss function and the optimizer to use. Optionally, you can specify a list of extra metrics to compute during training and evaluation:

```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```



Using `loss="sparse_categorical_crossentropy"` is equivalent to using `loss=tf.keras.losses.sparse_categorical_crossentropy`. Similarly, specifying `optimizer="sgd"` is equivalent to specifying `optimizer=tf.keras.optimizers.SGD()`, and `metrics=["accuracy"]` is equivalent to `metrics=[tf.keras.metrics.sparse_categorical_accuracy]` (when using this loss). We will use many other losses, optimizers, and metrics in this book; for the full lists, see <https://keras.io/api/losses/>, <https://keras.io/api/optimizers/>, and <https://keras.io/api/metrics/>.

This code requires some explanation. First, we use the "sparse_categorical_crossentropy" loss because we have sparse labels (i.e., for each instance, there is just a target class index, from 0 to 9 in this case), and the classes are exclusive. If instead we had one target probability per class for each instance (such as one-hot vectors, e.g. `[0., 0., 0., 1., 0., 0., 0., 0., 0.]` to represent class 3), then we would need to use the "categorical_crossentropy" loss instead. If we were doing binary classification or multilabel binary classification, then we would use the "sigmoid" activation function in the output layer instead of the "softmax" activation function, and we would use the "binary_crossentropy" loss.



If you want to convert sparse labels (i.e., class indices) to one-hot vector labels, use the `tf.keras.utils.to_categorical()` function. To go the other way round, use the `np.argmax()` function with `axis=1`.

Regarding the optimizer, "sgd" means that we will train the model using Stochastic Gradient Descent. In other words, Keras will perform the backpropagation algorithm described earlier (i.e., reverse-mode autodiff plus Gradient Descent). We will discuss more efficient optimizers in [Chapter 11](#). They improve Gradient Descent, not auto-diff.



When using the SGD optimizer, it is important to tune the learning rate. So, you will generally want to use `optimizer=tf.keras.optimizers.SGD(learning_rate=???)` to set the learning rate, rather than `optimizer="sgd"`, which defaults to a learning rate of 0.01.

Finally, since this is a classifier, it's useful to measure its accuracy during training and evaluation, which is why we set `metrics=["accuracy"]`.

Training and evaluating the model

Now the model is ready to be trained. For this we simply need to call its `fit()` method:

```
>>> history = model.fit(X_train, y_train, epochs=30,
...                      validation_data=(X_valid, y_valid))
...
Epoch 1/30
1719/1719 [=====] - 2s 989us/step
- loss: 0.7220 - sparse_categorical_accuracy: 0.7649
- val_loss: 0.4959 - val_sparse_categorical_accuracy: 0.8332
Epoch 2/30
1719/1719 [=====] - 2s 964us/step
- loss: 0.4825 - sparse_categorical_accuracy: 0.8332
- val_loss: 0.4567 - val_sparse_categorical_accuracy: 0.8384
[...]
Epoch 30/30
1719/1719 [=====] - 2s 963us/step
- loss: 0.2235 - sparse_categorical_accuracy: 0.9200
- val_loss: 0.3056 - val_sparse_categorical_accuracy: 0.8894
```

We pass it the input features (`X_train`) and the target classes (`y_train`), as well as the number of epochs to train (or else it would default to just 1, which would definitely not be enough to converge to a good solution). We also pass a validation set (this is optional). Keras will measure the loss and the extra metrics on this set at the end of each epoch, which is very useful to see how well the model really performs. If the performance on the training set is much better than on the validation set, your model is probably overfitting the training set, or there is a bug, such as a data mismatch between the training set and the validation set.



Shape errors are quite common, especially when getting started, so you should familiarize yourself with the error messages: try fitting a model with inputs and/or labels of the wrong shape, and see the errors you get. Similarly, try compiling the model with `loss="categorical_crossentropy"` instead of `loss="sparse_categorical_crossentropy"`, or remove the `Flatten` layer.

And that's it! The neural network is trained. At each epoch during training, Keras displays the number of mini-batches processed so far, on the left side of the progress bar. The batch size is 32 by default, and since the training set has 55,000 images, the model goes through 1,719 batches per epoch: 1,718 of size 32, and one of size 24. After the progress bar, you can see the mean training time per sample, and the loss and accuracy (or any other extra metrics you asked for) on both the training set and the validation set. Notice that the training loss went down, which is a good sign, and the validation accuracy reached 88.94% after 30 epochs. That's slightly below

the training accuracy, so there is a little bit of overfitting going on, but not a huge amount.



Instead of passing a validation set using the `validation_data` argument, you could set `validation_split` to the ratio of the training set that you want Keras to use for validation. For example, `validation_split=0.1` tells Keras to use the last 10% of the data (before shuffling) for validation.

If the training set was very skewed, with some classes being overrepresented and others underrepresented, it would be useful to set the `class_weight` argument when calling the `fit()` method, to give a larger weight to underrepresented classes and a lower weight to overrepresented classes. These weights would be used by Keras when computing the loss. If you need per-instance weights, set the `sample_weight` argument. If both `class_weight` and `sample_weight` are provided, then Keras multiplies them. Per-instance weights could be useful for example if some instances were labeled by experts while others were labeled using a crowdsourcing platform: you might want to give more weight to the former. You can also provide sample weights (but not class weights) for the validation set by adding them as a third item in the `validation_data` tuple.

The `fit()` method returns a `History` object containing the training parameters (`history.params`), the list of epochs it went through (`history.epoch`), and most importantly a dictionary (`history.history`) containing the loss and extra metrics it measured at the end of each epoch on the training set and on the validation set (if any). If you use this dictionary to create a Pandas DataFrame and call its `plot()` method, you get the learning curves shown in [Figure 10-11](#):

```
import matplotlib.pyplot as plt
import pandas as pd

pd.DataFrame(history.history).plot(
    figsize=(8, 5), xlim=[0, 29], ylim=[0, 1], grid=True, xlabel="Epoch",
    style=["r--", "r--.", "b-", "b-*"])
plt.show()
```

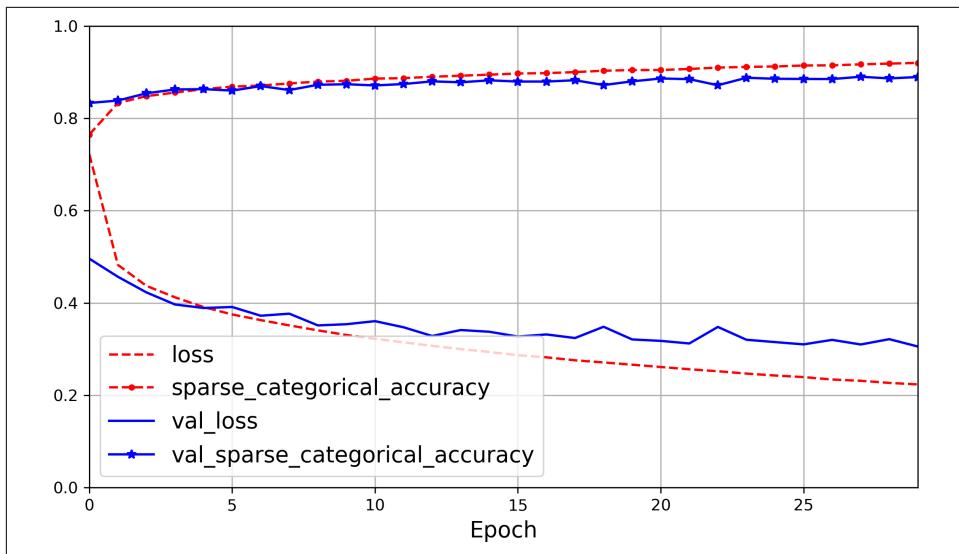


Figure 10-11. Learning curves: the mean training loss and accuracy measured over each epoch, and the mean validation loss and accuracy measured at the end of each epoch

You can see that both the training accuracy and the validation accuracy steadily increase during training, while the training loss and the validation loss decrease. Good! The validation curves are relatively close to each other at first, but they get further apart over time, which shows that there's a little bit of overfitting. In this particular case, the model looks like it performed better on the validation set than on the training set at the beginning of training. But that's not the case: indeed, the validation error is computed at the *end* of each epoch, while the training error is computed using a running mean *during* each epoch. So the training curve should be shifted by half an epoch to the left. If you do that, you will see that the training and validation curves overlap almost perfectly at the beginning of training.

The training set performance ends up beating the validation performance, as is generally the case when you train for long enough. You can tell that the model has not quite converged yet, as the validation loss is still going down, so you should probably continue training. It's as simple as calling the `fit()` method again, since Keras just continues training where it left off: you should be able to reach about 89.8% validation accuracy, while the training accuracy will continue to rise up to 100% (this is not always the case).

If you are not satisfied with the performance of your model, you should go back and tune the hyperparameters. The first one to check is the learning rate. If that doesn't help, try another optimizer (and always retune the learning rate after changing any hyperparameter). If the performance is still not great, then try tuning model hyperparameters such as the number of layers, the number of neurons per layer,

and the types of activation functions to use for each hidden layer. You can also try tuning other hyperparameters, such as the batch size (it can be set in the `fit()` method using the `batch_size` argument, which defaults to 32). We will get back to hyperparameter tuning at the end of this chapter. Once you are satisfied with your model's validation accuracy, you should evaluate it on the test set to estimate the generalization error before you deploy the model to production. You can easily do this using the `evaluate()` method (it also supports several other arguments, such as `batch_size` and `sample_weight`; please check the documentation for more details):

```
>>> model.evaluate(X_test, y_test)
313/313 [=====] - 0s 626us/step
  - loss: 0.3243 - sparse_categorical_accuracy: 0.8864
[0.32431697845458984, 0.8863999843597412]
```

As we saw in [Chapter 2](#), it is common to get slightly lower performance on the test set than on the validation set, because the hyperparameters are tuned on the validation set, not the test set (however, in this example, we did not do any hyperparameter tuning, so the lower accuracy is just bad luck). Remember to resist the temptation to tweak the hyperparameters on the test set, or else your estimate of the generalization error will be too optimistic.

Using the model to make predictions

Next, we can use the model's `predict()` method to make predictions on new instances. Since we don't have actual new instances, we will just use the first three instances of the test set:

```
>>> X_new = X_test[:3]
>>> y_proba = model.predict(X_new)
>>> y_proba.round(2)
array([[0. , 0. , 0. , 0. , 0. , 0.01, 0. , 0.02, 0. , 0.97],
       [0. , 0. , 0.99, 0. , 0.01, 0. , 0. , 0. , 0. , 0. ],
       [0. , 1. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ]],
      dtype=float32)
```

As you can see, for each instance the model estimates one probability per class, from class 0 to class 9. This is similar to the output of the `predict_proba()` method in Scikit-Learn classifiers. For example, for the first image it estimates that the probability of class 9 (ankle boot) is 96%, the probability of class 7 (sneaker) is 2%, the probability of class 5 (sandal) is 1%, and the probabilities of the other classes are negligible. In other words, it is highly confident that the first image is footwear, most likely ankle boots but possibly sneakers or sandals. If you only care about the class with the highest estimated probability (even if that probability is quite low), then you can use the `argmax()` to get the highest probability class index for each instance:

```
>>> import numpy as np
>>> y_pred = y_proba.argmax(axis=-1)
>>> y_pred
```

```

array([9, 2, 1])
>>> np.array(class_names)[y_pred]
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='|<U11')

```

Here, the classifier actually classified all three images correctly (these images are shown in [Figure 10-12](#)):

```

>>> y_new = y_test[:3]
>>> y_new
array([9, 2, 1], dtype=uint8)

```

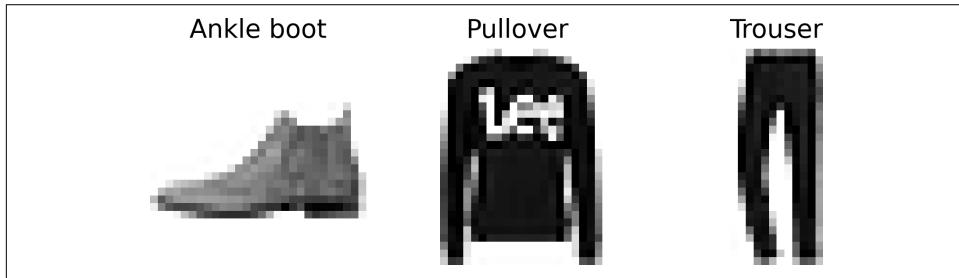


Figure 10-12. Correctly classified Fashion MNIST images

Now you know how to use the Sequential API to build, train and evaluate a classification MLP. But what about regression?

Building a Regression MLP Using the Sequential API

Let's switch back to the California housing problem and tackle it using the same MLP as earlier, with 3 hidden layers composed of 50 neurons each, but this time building it with Keras.

Using the Sequential API to build, train, evaluate, and use a regression MLP is quite similar to what we did for classification. The main differences in the following code example are the fact that the output layer has a single neuron (since we only want to predict a single value) and it uses no activation function, the loss function is the mean squared error, the metric is the RMSE, and we're using an Adam optimizer like Scikit-Learn's `MLPRegressor` did. Moreover, in this example we don't need a `Flatten` layer, and instead we're using a `Normalization` layer as the first layer: it does the same thing as Scikit-Learn's `StandardScaler`, but it must be fitted to the training data using its `adapt()` method *before* you call the model's `fit()` method. Keras has other preprocessing layers which will be covered in [Chapter 13](#).

```

tf.random.set_seed(42)
norm_layer = tf.keras.layers.Normalization(input_shape=X_train.shape[1:])
model = tf.keras.Sequential([
    norm_layer,
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(50, activation="relu"),

```

```

        tf.keras.layers.Dense(50, activation="relu"),
        tf.keras.layers.Dense(1)
    ])
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
model.compile(loss="mse", optimizer=optimizer, metrics=[ "RootMeanSquaredError"])
norm_layer.adapt(X_train)
history = model.fit(X_train, y_train, epochs=20,
                     validation_data=(X_valid, y_valid))
mse_test, rmse_test = model.evaluate(X_test, y_test)
X_new = X_test[:3]
y_pred = model.predict(X_new)

```



The Normalization layer learns the feature means and standard deviations in the training data when you call the `adapt()` method. Yet when you display the model's summary, these statistics are listed as non-trainable. This is because these parameters are not affected by Gradient Descent.

As you can see, the Sequential API is quite clean and straightforward. However, although Sequential models are extremely common, it is sometimes useful to build neural networks with more complex topologies, or with multiple inputs or outputs. For this purpose, Keras offers the Functional API.

Building Complex Models Using the Functional API

One example of a nonsequential neural network is a *Wide & Deep* neural network. This neural network architecture was introduced in a [2016 paper](#) by Heng-Tze Cheng et al.¹⁵ It connects all or part of the inputs directly to the output layer, as shown in [Figure 10-13](#). This architecture makes it possible for the neural network to learn both deep patterns (using the deep path) and simple rules (through the short path).¹⁶ In contrast, a regular MLP forces all the data to flow through the full stack of layers; thus, simple patterns in the data may end up being distorted by this sequence of transformations.

¹⁵ Heng-Tze Cheng et al., “Wide & Deep Learning for Recommender Systems,” *Proceedings of the First Workshop on Deep Learning for Recommender Systems* (2016): 7–10.

¹⁶ The short path can also be used to provide manually engineered features to the neural network.

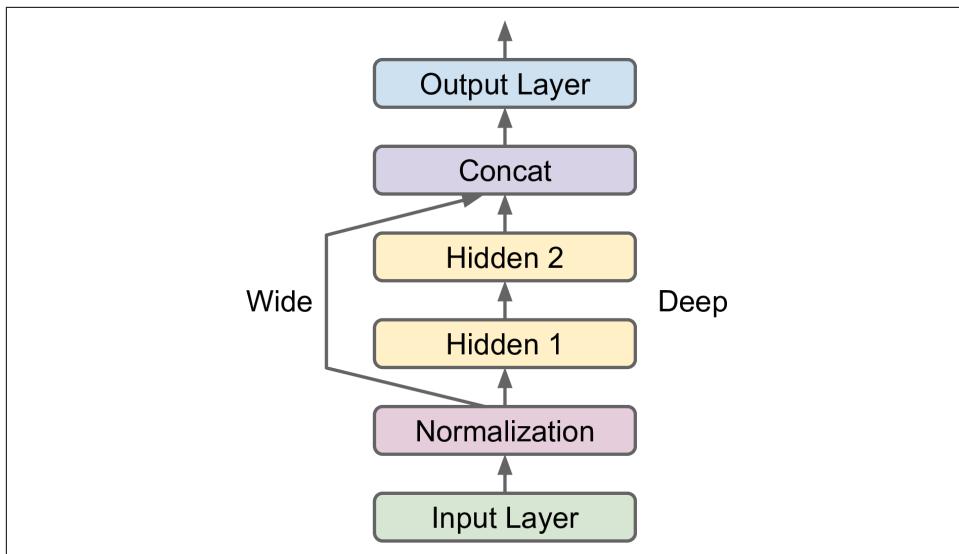


Figure 10-13. Wide & Deep neural network

Let's build such a neural network to tackle the California housing problem:

```

normalization_layer = tf.keras.layers.Normalization()
hidden_layer1 = tf.keras.layers.Dense(30, activation="relu")
hidden_layer2 = tf.keras.layers.Dense(30, activation="relu")
concat_layer = tf.keras.layers.concatenate()
output_layer = tf.keras.layers.Dense(1)

input_ = tf.keras.layers.Input(shape=X_train.shape[1:])
normalized = normalization_layer(input_)
hidden1 = hidden_layer1(normalized)
hidden2 = hidden_layer2(hidden1)
concat = concat_layer([input_, hidden2])
output = output_layer(concat)

model = tf.keras.Model(inputs=[input_], outputs=[output])
  
```

At a high level, the first five lines create all the layers we need to build the model, the next six lines use these layers just like functions to go from the input to the output, and the last line creates a Keras Model object by pointing to the input and the output. Let's go through this code in more detail:

- First, we create five layers: a `Normalization` layer to standardize the inputs, two `Dense` layers with 30 neurons each, using the ReLU activation function, a `Concatenate` layer, and one more `Dense` layer with a single neuron for the output layer, without any activation function.

- Next, we create an `Input` object (the variable name `input_` is used to avoid overshadowing Python's built-in `input()` function). This is a specification of the kind of input the model will get, including its `shape` and optionally its `dtype`, which defaults to 32-bit floats. A model may actually have multiple inputs, as we will see shortly.
- Then we use the `Normalization` layer just like a function, passing it the `Input` object. This is why this is called the Functional API. Note that we are just telling Keras how it should connect the layers together; no actual data is being processed yet, as the `Input` object is just a data specification. In other words, it's a symbolic input. The output of this call is also symbolic: `normalized` doesn't store any actual data, it's just used to construct the model.
- In the same way, we then pass `normalized` to `hidden_layer1`, which outputs `hidden1`, and we pass `hidden1` to `hidden_layer2`, which outputs `hidden2`.
- So far, we've connected the layers sequentially, but then we use the `concat_layer` to concatenate the input and the second hidden layer's output. Again, no actual data is concatenated yet: it's all symbolic, to build the model.
- Then we pass `concat` to the `output_layer`, which gives us the final `output`.
- Lastly, we create a Keras `Model`, specifying which inputs and outputs to use.

Once you have built this Keras model, everything is exactly like earlier, so there's no need to repeat it here: you must compile the model, adapt the `Normalization` layer, fit the model, evaluate it, and use it to make predictions.

But what if you want to send a subset of the features through the wide path and a different subset (possibly overlapping) through the deep path (see [Figure 10-14](#))? In this case, one solution is to use multiple inputs. For example, suppose we want to send five features through the wide path (features 0 to 4), and six features through the deep path (features 2 to 7):

```
input_wide = tf.keras.layers.Input(shape=[5]) # features 0 to 4
input_deep = tf.keras.layers.Input(shape=[6]) # features 2 to 7
norm_layer_wide = tf.keras.layers.Normalization()
norm_layer_deep = tf.keras.layers.Normalization()
norm_wide = norm_layer_wide(input_wide)
norm_deep = norm_layer_deep(input_deep)
hidden1 = tf.keras.layers.Dense(30, activation="relu")(norm_deep)
hidden2 = tf.keras.layers.Dense(30, activation="relu")(hidden1)
concat = tf.keras.layers.concatenate([norm_wide, hidden2])
output = tf.keras.layers.Dense(1)(concat)
model = tf.keras.Model(inputs=[input_wide, input_deep], outputs=[output])
```

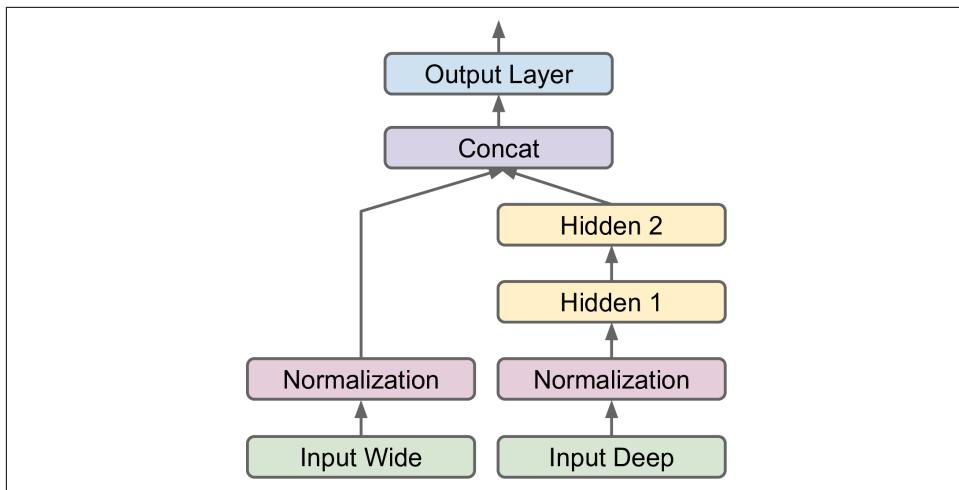


Figure 10-14. Handling multiple inputs

There are a few things to note in this code example, compared to the previous example:

- Each `Dense` layer is created and called on the same line. This is a common practice, as it makes the code more concise without losing clarity. However, we can't do this with the `Normalization` layer since we need a reference to the layer to be able to call its `adapt()` method before fitting the model.
- We used `tf.keras.layers.concatenate()`, which creates a `Concatenate` layer and calls it with the given inputs.
- We specified `inputs=[input_wide, input_deep]` when creating the model, since there are two inputs.

Now we can compile the model as usual, but when we call the `fit()` method, instead of passing a single input matrix `X_train`, we must pass a pair of matrices (`X_train_wide`, `X_train_deep`): one per input. The same is true for `X_valid`, and also for `X_test` and `X_new` when you call `evaluate()` or `predict()`:

```

optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
model.compile(loss="mse", optimizer=optimizer, metrics=["RootMeanSquaredError"])

X_train_wide, X_train_deep = X_train[:, :5], X_train[:, 2:]
X_valid_wide, X_valid_deep = X_valid[:, :5], X_valid[:, 2:]
X_test_wide, X_test_deep = X_test[:, :5], X_test[:, 2:]
X_new_wide, X_new_deep = X_test_wide[:3], X_test_deep[:3]

norm_layer_wide.adapt(X_train_wide)
norm_layer_deep.adapt(X_train_deep)
history = model.fit((X_train_wide, X_train_deep), y_train, epochs=20,
  
```

```
validation_data=((X_valid_wide, X_valid_deep), y_valid))  
mse_test = model.evaluate((X_test_wide, X_test_deep), y_test)  
y_pred = model.predict((X_new_wide, X_new_deep))
```



Instead of passing a tuple `(X_train_wide, X_train_deep)`, you can pass a dictionary `{"input_wide": X_train_wide, "input_deep": X_train_deep}`, assuming you set `name="input_wide"` and `name="input_deep"` when creating the inputs. This is highly recommended when there are many inputs, to clarify the code and avoid getting the order wrong.

There are also many use cases in which you may want to have multiple outputs:

- The task may demand it. For instance, you may want to locate and classify the main object in a picture. This is both a regression tasks and a classification task.
- Similarly, you may have multiple independent tasks based on the same data. Sure, you could train one neural network per task, but in many cases you will get better results on all tasks by training a single neural network with one output per task. This is because the neural network can learn features in the data that are useful across tasks. For example, you could perform *multitask classification* on pictures of faces, using one output to classify the person's facial expression (smiling, surprised, etc.) and another output to identify whether they are wearing glasses or not.
- Another use case is as a regularization technique (i.e., a training constraint whose objective is to reduce overfitting and thus improve the model's ability to generalize). For example, you may want to add an auxiliary output in a neural network architecture (see [Figure 10-15](#)) to ensure that the underlying part of the network learns something useful on its own, without relying on the rest of the network.

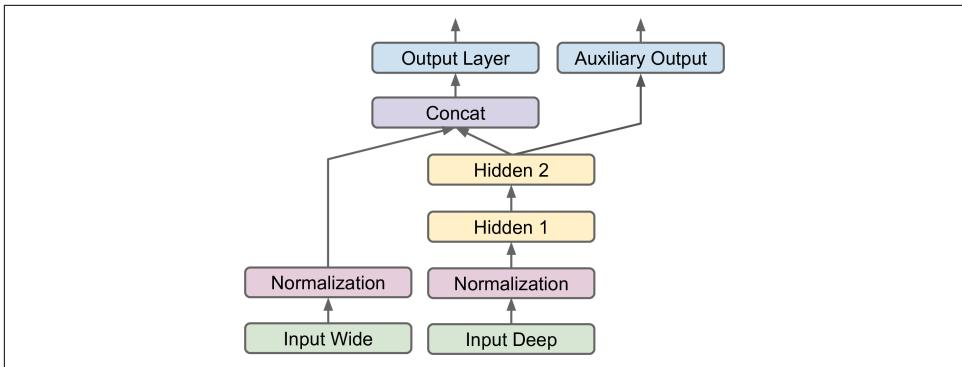


Figure 10-15. Handling multiple outputs, in this example to add an auxiliary output for regularization

Adding an extra output is quite easy: just connect it to the appropriate layer and add it to your model’s list of outputs. For example, the following code builds the network represented in Figure 10-15:

```
[...] # Same as above, up to the main output layer
output = tf.keras.layers.Dense(1)(concat)
aux_output = tf.keras.layers.Dense(1)(hidden2)
model = tf.keras.Model(inputs=[input_wide, input_deep],
                       outputs=[output, aux_output])
```

Each output will need its own loss function. Therefore, when we compile the model, we should pass a list of losses. If we pass a single loss, Keras will assume that the same loss must be used for all outputs. By default, Keras will compute all the losses and simply add them up to get the final loss used for training. Since we care much more about the main output than about the auxiliary output (as it is just used for regularization), we want to give the main output’s loss a much greater weight. Luckily, it is possible to set all the loss weights when compiling the model:

```
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
model.compile(loss={"mse", "mse"}, loss_weights=(0.9, 0.1), optimizer=optimizer,
               metrics=["RootMeanSquaredError"])
```



Instead of passing a tuple `loss=("mse", "mse")`, you can pass a dictionary `loss={"output": "mse", "aux_output": "mse"}`, assuming you created the output layers with `name="output"` and `name="aux_output"`. Just like for the inputs, this clarifies the code and avoids errors when there are several outputs. You can also pass a dictionary for `loss_weights`.

Now when you train the model, you need to provide labels for each output. In this example, the main output and the auxiliary output should try to predict the same

thing, so they should use the same labels. So instead of passing `y_train`, you need to pass `(y_train, y_train)`, or a dictionary `{"output": y_train, "aux_output": y_train}` if the outputs were named "output" and "aux_output". The same goes for `y_valid` and `y_test`:

```
norm_layer_wide.adapt(X_train_wide)
norm_layer_deep.adapt(X_train_deep)
history = model.fit(
    (X_train_wide, X_train_deep), (y_train, y_train), epochs=20,
    validation_data=((X_valid_wide, X_valid_deep), (y_valid, y_valid)))
)
```

When you evaluate the model, Keras returns the weighted sum of the losses, as well as all the individual losses and metrics:

```
eval_results = model.evaluate((X_test_wide, X_test_deep), (y_test, y_test))
weighted_sum_of_losses, main_loss, aux_loss, main_rmse, aux_rmse = eval_results
```



If you set `return_dict=True`, then `evaluate()` will return a dictionary instead of a big tuple.

Similarly, the `predict()` method will return predictions for each output:

```
y_pred_main, y_pred_aux = model.predict((X_new_wide, X_new_deep))
```

The `predict()` method returns a tuple, and it does not have a `return_dict` argument to get a dictionary instead, but you can create one using `model.output_names`:

```
y_pred_tuple = model.predict((X_new_wide, X_new_deep))
y_pred = dict(zip(model.output_names, y_pred_tuple))
```

As you can see, you can build all sorts of architectures with the Functional API. Let's look at one last way you can build Keras models.

Using the Subclassing API to Build Dynamic Models

Both the Sequential API and the Functional API are declarative: you start by declaring which layers you want to use and how they should be connected, and only then can you start feeding the model some data for training or inference. This has many advantages: the model can easily be saved, cloned, and shared; its structure can be displayed and analyzed; the framework can infer shapes and check types, so errors can be caught early (i.e., before any data ever goes through the model). It's also fairly straightforward to debug, since the whole model is a static graph of layers. But the flip side is just that: it's static. Some models involve loops, varying shapes, conditional branching, and other dynamic behaviors. For such cases, or simply if you prefer a more imperative programming style, the Subclassing API is for you.

You must subclass the `Model` class, create the layers you need in the constructor, and use them to perform the computations you want in the `call()` method. For example, creating an instance of the following `WideAndDeepModel` class gives us an equivalent model to the one we just built with the Functional API:

```
class WideAndDeepModel(tf.keras.Model):
    def __init__(self, units=30, activation="relu", **kwargs):
        super().__init__(**kwargs) # needed to support naming the model
        self.norm_layer_wide = tf.keras.layers.Normalization()
        self.norm_layer_deep = tf.keras.layers.Normalization()
        self.hidden1 = tf.keras.layers.Dense(units, activation=activation)
        self.hidden2 = tf.keras.layers.Dense(units, activation=activation)
        self.main_output = tf.keras.layers.Dense(1)
        self.aux_output = tf.keras.layers.Dense(1)

    def call(self, inputs):
        input_wide, input_deep = inputs
        norm_wide = self.norm_layer_wide(input_wide)
        norm_deep = self.norm_layer_deep(input_deep)
        hidden1 = self.hidden1(norm_deep)
        hidden2 = self.hidden2(hidden1)
        concat = tf.keras.layers.concatenate([norm_wide, hidden2])
        output = self.main_output(concat)
        aux_output = self.aux_output(hidden2)
        return output, aux_output

model = WideAndDeepModel(30, activation="relu", name="my_cool_model")
```

This example looks very much like the Functional API, except we separate the creation of the layers¹⁷ in the constructor from their usage in the `call()` method. We also do not need to create the `Input` objects: we just use the `input` argument to the `call()` method.

Now that you have a model instance, you can compile it, adapt its normalization layers (e.g., using `model.norm_layer_wide.adapt(...)` and `model.norm_layer_deep.adapt(...)`), fit it, evaluate it, and use it to make predictions, exactly like you did with the Functional API.

The big difference with this API is that you can do pretty much anything you want in the `call()` method: `for` loops, `if` statements, low-level TensorFlow operations—your imagination is the limit (see [Chapter 12](#))! This makes it a great API when experimenting with new ideas, especially for researchers. However, this extra flexibility does come at a cost: your model's architecture is hidden within the `call()` method, so Keras cannot easily inspect it, the model cannot be cloned using `tf.keras.mod`

¹⁷ Keras models have an `output` attribute, so we cannot use that name for the main output layer, which is why we renamed it to `main_output`.

`els.clone_model()`, and when you call the `summary()` method, you only get a list of layers, without any information on how they are connected to each other. Moreover, Keras cannot check types and shapes ahead of time, and it is easier to make mistakes. So unless you really need that extra flexibility, you should probably stick to the Sequential API or the Functional API.



Keras models can be used just like regular layers, so you can easily combine them to build complex architectures.

Now that you know how to build and train neural nets using Keras, you will want to save them!

Saving and Restoring a Model

Saving a trained Keras model is as simple as it gets:

```
model.save("my_keras_model", save_format="tf")
```

When you set `save_format="tf"`,¹⁸ Keras saves the model using TensorFlow's *SavedModel* format: this is a directory (with the given name) containing several files and subdirectories. In particular, the `saved_model.pb` file contains the model's architecture and logic in the form of a serialized computation graph, so you don't need to deploy the model's source code in order to use it in production, the `SavedModel` is sufficient (we will see how this works in [Chapter 12](#)). The `keras_metadata.pb` file contains extra information needed by Keras. The `variables` subdirectory contains all the parameter values (including the connection weights, the biases, the normalization statistics, and the optimizer's parameters), possibly split across multiple files if the model is very large. Lastly, the `assets` directory may contain extra files, for example data samples, feature names, class names, and so on. By default, the `assets` directory is empty. Since the optimizer is also saved, including its hyperparameters and any state it may have, after loading the model you can continue training if you want.



If you set `save_format="h5"` or use a file name ending with `.h5`, `.hdf5`, or `.keras`, then Keras will save the model to a single file using a Keras-specific format based on the HDF5 format. However, most TensorFlow deployment tools require the `SavedModel` format instead.

¹⁸ This is currently the default, but the Keras team is working on a new format that may become the default in upcoming versions, so I prefer to set the format explicitly to be future-proof.

You will typically have a script that trains a model and saves it, and one or more scripts (or web services) that load the model and use it to evaluate it or to make predictions. Loading the model is just as easy as saving it:

```
model = tf.keras.models.load_model("my_keras_model")
y_pred_main, y_pred_aux = model.predict((X_new_wide, X_new_deep))
```

You can also use `save_weights()` and `load_weights()` to save and load only the parameter values. This includes the connection weights, biases, preprocessing stats, optimizer state, etc. The parameter values are saved in one or more files such as `my_weights.data-00004-of-00052`, plus an index file like `my_weights.index`.

Saving just the weights is faster and uses less disk space than saving the whole model, so it's perfect to save quick checkpoints during training. If you're training a big model, and it takes hours or days, then you must save checkpoints regularly in case the computer crashes. But how can you tell the `fit()` method to save checkpoints? Use callbacks.

Using Callbacks

The `fit()` method accepts a `callbacks` argument that lets you specify a list of objects that Keras will call before and after training, before and after each epoch, and even before and after processing each batch. For example, the `ModelCheckpoint` callback saves checkpoints of your model at regular intervals during training, by default at the end of each epoch:

```
checkpoint_cb = tf.keras.callbacks.ModelCheckpoint("my_checkpoints",
                                                    save_weights_only=True)
history = model.fit([...], callbacks=[checkpoint_cb])
```

Moreover, if you use a validation set during training, you can set `save_best_only=True` when creating the `ModelCheckpoint`. In this case, it will only save your model when its performance on the validation set is the best so far. This way, you do not need to worry about training for too long and overfitting the training set: simply restore the last saved model after training, and this will be the best model on the validation set. This is one way to implement early stopping (introduced in [Chapter 4](#)), but it won't actually stop training.

Another way is to use the `EarlyStopping` callback. It will interrupt training when it measures no progress on the validation set for a number of epochs (defined by the `patience` argument), and if you set `restore_best_weights=True` it will roll back to the best model at the end of training. You can combine both callbacks to save checkpoints of your model in case your computer crashes, and interrupt training early when there is no more progress, to avoid wasting time and resources and to reduce overfitting:

```
early_stopping_cb = tf.keras.callbacks.EarlyStopping(patience=10,
                                                    restore_best_weights=True)
history = model.fit(..., callbacks=[checkpoint_cb, early_stopping_cb])
```

The number of epochs can be set to a large value since training will stop automatically when there is no more progress (just make sure the learning rate is not too small, or else it might keep making slow progress until the end). The `EarlyStopping` callback will store the weights of the best model in RAM, and it will restore them for you at the end of training.



There are many other callbacks available in the `tf.keras.callbacks` package.

If you need extra control, you can easily write your own custom callbacks. For example, the following custom callback will display the ratio between the validation loss and the training loss during training (e.g., to detect overfitting):

```
class PrintValTrainRatioCallback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs):
        ratio = logs["val_loss"] / logs["loss"]
        print(f"Epoch={epoch}, val/train={ratio:.2f}")
```

As you might expect, you can implement `on_train_begin()`, `on_train_end()`, `on_epoch_begin()`, `on_epoch_end()`, `on_batch_begin()`, and `on_batch_end()`. Callbacks can also be used during evaluation and predictions, should you ever need them (e.g., for debugging). For evaluation, you should implement `on_test_begin()`, `on_test_end()`, `on_test_batch_begin()`, or `on_test_batch_end()`, which are called by `evaluate()`. For prediction, you should implement `on_predict_begin()`, `on_predict_end()`, `on_predict_batch_begin()`, or `on_predict_batch_end()`, which are called by `predict()`.

Now let's take a look at one more tool you should definitely have in your toolbox when using Keras: TensorBoard.

Using TensorBoard for Visualization

TensorBoard is a great interactive visualization tool that you can use to view the learning curves during training, compare curves and metrics between multiple runs, visualize the computation graph, analyze training statistics, view images generated by your model, visualize complex multidimensional data projected down to 3D and automatically clustered for you, *profile* your network (i.e., measure its speed to identify bottlenecks), and more!

TensorBoard is installed automatically when you install TensorFlow. However, you will need a TensorBoard plugin to visualize profiling data. If you followed the installation instructions at <https://homl.info/install> to run everything locally, then you already have the plugin installed, but if you are using Colab, then you must run the following command:

```
%pip install -q -U tensorboard-plugin-profile
```

To use TensorBoard, you must modify your program so that it outputs the data you want to visualize to special binary log files called *event files*. Each binary data record is called a *summary*. The TensorBoard server will monitor the log directory, and it will automatically pick up the changes and update the visualizations: this allows you to visualize live data (with a short delay), such as the learning curves during training. In general, you want to point the TensorBoard server to a root log directory and configure your program so that it writes to a different subdirectory every time it runs. This way, the same TensorBoard server instance will allow you to visualize and compare data from multiple runs of your program, without getting everything mixed up.

Let's name the root log directory `my_logs`, and let's define a little function that generates the path of the log subdirectory based on the current date and time, so that it's different at every run:

```
from pathlib import Path
from time import strftime

def get_run_logdir(root_logdir="my_logs"):
    return Path(root_logdir) / strftime("run_%Y_%m_%d_%H_%M_%S")

run_logdir = get_run_logdir() # e.g., my_logs/run_2022_08_01_17_25_59
```

The good news is that Keras provides a convenient `TensorBoard()` callback which will take care of creating the log directory for you (along with its parent directories if needed), and it will create event files and write summaries to them during training. It will measure your model's training and validation loss and metrics (in this case, the MSE and RMSE), and it will also profile your neural network. It is straightforward to use:

```
tensorboard_cb = tf.keras.callbacks.TensorBoard(run_logdir,
                                                profile_batch=(100, 200))
history = model.fit(..., callbacks=[tensorboard_cb])
```

And that's all there is to it! In this example, it will profile the network between batches 100 and 200 during the first epoch. Why 100 and 200? Well, it often takes a few batches for the neural network to "warm up", so you don't want to profile too early. And profiling uses a bit of resources, so it's best not to do it for every batch.

Next, try changing the learning rate from 0.001 to 0.002, and run the code again, with a new log subdirectory. You will end up with a directory structure similar to this one:

```
my_logs
├── run_2022_08_01_17_25_59
│   ├── train
│   │   ├── events.out.tfevents.1659331561.my_host_name.42042.0.v2
│   │   ├── events.out.tfevents.1659331562.my_host_name.profile-empty
│   │   └── plugins
│   │       └── profile
│   │           └── 2022_08_01_17_26_02
│   │               ├── my_host_name.input_pipeline.pb
│   │               └── [...]
│   └── validation
│       └── events.out.tfevents.1659331562.my_host_name.42042.1.v2
└── run_2022_08_01_17_31_12
    └── [...]
```

There's one directory per run, each containing one subdirectory for training logs and one for validation logs. Both contain event files, and the training logs also include profiling traces.

Now that you have the event files ready, it's time to start the TensorBoard server. This can be done directly within Jupyter or Colab using the Jupyter extension for TensorBoard, which gets installed along with the TensorBoard library. This extension is preinstalled in Colab. The following code loads the Jupyter extension for TensorBoard, and the second line starts a TensorBoard server for the `my_logs` directory, and it connects to this server and displays the user interface directly inside of Jupyter. The server listens on the first available TCP port greater or equal to 6006 (or you can set the port you want using the `--port` option).

```
%load_ext tensorboard
%tensorboard --logdir=./my_logs
```



If you are running everything on your own machine, it's also possible to start TensorBoard by executing `tensorboard --logdir= ./my_logs` in a terminal. You must first activate the conda environment in which you installed TensorBoard, and go to the `hands-on-ml3` directory. Once the server is started, visit <http://localhost:6006/>.

Now you should see TensorBoard's user interface. Click the SCALARS tab to view the learning curves (see [Figure 10-16](#)). At the bottom left, select the logs you want to visualize (e.g., the training logs from the first and second run), and click the `epoch_loss` scalar. Notice that the training loss went down nicely during both runs, but the second run went down a bit faster thanks to the higher learning rate.

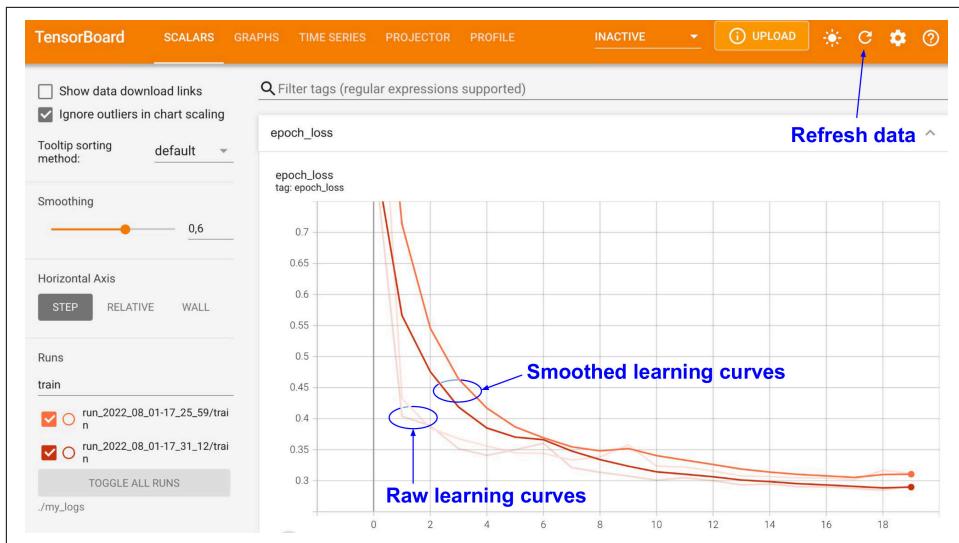


Figure 10-16. Visualizing learning curves with TensorBoard

You can also visualize the whole computation graph in the GRAPHS tab, the learned weights projected to 3D in the PROJECTOR tab, and the profiling traces in the PROFILE tab. The `TensorBoard()` callback has options to log extra data too (see the documentation for more details). You can click the refresh button (⟳) at the top right to make TensorBoard refresh data, and you can click the settings button (⚙) to activate auto-refresh and specify the refresh interval.

Additionally, TensorFlow offers a lower-level API in the `tf.summary` package. The following code creates a `SummaryWriter` using the `create_file_writer()` function, and it uses this writer as a Python context to log scalars, histograms, images, audio, and text, all of which can then be visualized using TensorBoard:

```
test_logdir = get_run_logdir()
writer = tf.summary.create_file_writer(str(test_logdir))
with writer.as_default():
    for step in range(1, 1000 + 1):
        tf.summary.scalar("my_scalar", np.sin(step / 10), step=step)

        data = (np.random.randn(100) + 2) * step / 100 # gets larger
        tf.summary.histogram("my_hist", data, buckets=50, step=step)

        images = np.random.rand(2, 32, 32, 3) * step / 1000 # gets brighter
        tf.summary.image("my_images", images, step=step)

        texts = ["The step is " + str(step), "Its square is " + str(step ** 2)]
        tf.summary.text("my_text", texts, step=step)

        sine_wave = tf.math.sin(tf.range(12000) / 48000 * 2 * np.pi * step)
```

```
audio = tf.reshape(tf.cast(sine_wave, tf.float32), [1, -1, 1])
tf.summary.audio("my_audio", audio, sample_rate=48000, step=step)
```

If you run this code and click the refresh button in TensorBoard, you will see several tabs appear: IMAGES, AUDIO, DISTRIBUTIONS, HISTOGRAMS and TEXT. Try clicking on the IMAGES tab, and use the slider above each image to view the images at different time steps. Similarly, go to the AUDIO tab and try listening to the audios at different time steps. As you can see, TensorBoard is a useful tool even beyond TensorFlow or Deep Learning.



You can share your results online by publishing them to <https://tensorboard.dev>. For this, just run `!tensorboard dev upload --logdir ./my_logs`. The first time, it will ask you to accept the terms and conditions and authenticate. Then your logs will be uploaded, and you will get a permanent link to view your results in a TensorBoard interface.

Let's summarize what you've learned so far in this chapter: you now know where neural nets came from, what an MLP is and how you can use it for classification and regression, how to use Keras's Sequential API to build MLPs, and how to use the Functional API or the Subclassing API to build more complex model architectures, including Wide & Deep models, as well as models with multiple inputs and outputs. You also learned how to save and restore a model and how to use callbacks for checkpointing, early stopping, and more. Finally, you learned how to use TensorBoard for visualization. You can already go ahead and use neural networks to tackle many problems! However, you may wonder how to choose the number of hidden layers, the number of neurons in the network, and all the other hyperparameters. Let's look at this now.

Fine-Tuning Neural Network Hyperparameters

The flexibility of neural networks is also one of their main drawbacks: there are many hyperparameters to tweak. Not only can you use any imaginable network architecture, but even in a basic MLP you can change the number of layers, the number of neurons and the type of activation function to use in each layer, the weight initialization logic, the type of optimizer to use, its learning rate, the batch size, and more. How do you know what combination of hyperparameters is the best for your task?

One option is to convert your Keras model to a Scikit-Learn estimator, and then use `GridSearchCV` or `RandomizedSearchCV` to fine-tune the hyperparameters, as you did in [Chapter 2](#). For this, you can use the `KerasRegressor` and `KerasClassifier` wrapper classes from the `SciKeras` library (see <https://github.com/adriangb/scikeras> for more details). However, there's a better way: you can use the `Keras Tuner` library,

which is a hyperparameter tuning library for Keras models. It offers several tuning strategies, it's highly customizable, and it has excellent integration with TensorBoard. Let's see how to use it.

If you followed the installation instructions at <https://homl.info/install> to run everything locally, then you already have Keras Tuner installed, but if you are using Colab, then you must run `%pip install -q -U keras-tuner`. Next, import `keras_tuner`, usually as `kt`, then write a function that builds, compiles, and returns a Keras model. The function must take a `kt.HyperParameters` object as argument, which it can use to define hyperparameters (integers, floats, strings, etc.) along with their range of possible values, and these hyperparameters may be used to build and compile the model. For example, the following function builds and compiles an MLP to classify Fashion MNIST images, using hyperparameters such as the number of hidden layers (`n_hidden`), the number of neurons per layer (`n_neurons`), the learning rate (`learning_rate`), and the type of optimizer to use (`optimizer`):

```
import keras_tuner as kt

def build_model(hp):
    n_hidden = hp.Int("n_hidden", min_value=0, max_value=8, default=2)
    n_neurons = hp.Int("n_neurons", min_value=16, max_value=256)
    learning_rate = hp.Float("learning_rate", min_value=1e-4, max_value=1e-2,
                             sampling="log")
    optimizer = hp.Choice("optimizer", values=["sgd", "adam"])
    if optimizer == "sgd":
        optimizer = tf.keras.optimizers.SGD(learning_rate=learning_rate)
    else:
        optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)

    model = tf.keras.Sequential()
    model.add(tf.keras.layers.Flatten())
    for _ in range(n_hidden):
        model.add(tf.keras.layers.Dense(n_neurons, activation="relu"))
    model.add(tf.keras.layers.Dense(10, activation="softmax"))
    model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
                  metrics=["accuracy"])
    return model
```

The first part of the function defines the hyperparameters. For example, `hp.Int("n_hidden", min_value=0, max_value=8, default=2)` checks whether a hyperparameter named "`n_hidden`" is already present in the `HyperParameters` object `hp`, and if so it returns its value. If not, then it registers a new integer hyperparameter named "`n_hidden`", whose possible values range from 0 to 8 (inclusive), and it returns the default value, which is 2 in this case (when `default` is not set, then `min_value` is returned). The "`n_neurons`" hyperparameter is registered in a similar way. The "`learning_rate`" hyperparameter is registered as a float ranging from 10^{-4} to 10^{-2} , and since `sampling="log"`, learning rates of all scales will be sampled equally.

Lastly, the `optimizer` hyperparameter is registered with two possible values: "sgd" or "adam" (the default value is the first one, which is "sgd" in this case). Depending on the value of `optimizer`, we create an SGD optimizer or an Adam optimizer with the given learning rate.

The second part of the function just builds the model using the hyperparameter values. It creates a Sequential model starting with a Flatten layer, followed by the requested number of hidden layers (as determined by the `n_hidden` hyperparameter) using the ReLU activation function, and an output layer with 10 neurons (one per class) and using the softmax activation function. Lastly, the function compiles the model and returns it.

Now if you want to do a basic random search, you can create a `kt.RandomSearch` tuner, passing the `build_model` function to the constructor, and call the tuner's `search()` method:

```
random_search_tuner = kt.RandomSearch(  
    build_model, objective="val_accuracy", max_trials=5, overwrite=True,  
    directory="my_fashion_mnist", project_name="my_rnd_search", seed=42)  
random_search_tuner.search(X_train, y_train, epochs=10,  
                           validation_data=(X_valid, y_valid))
```

The `RandomSearch` tuner first calls `build_model()` once with an empty `Hyperparameters` object, just to gather all the hyperparameters specifications. Then, in this example, it runs 5 trials, and for each trial it builds a model using hyperparameters sampled randomly within their respective ranges, then it trains that model for 10 epochs and saves it to a subdirectory of the `my_fashion_mnist/my_rnd_search` directory. Since `overwrite=True`, the `my_rnd_search` directory is deleted before training starts. If you run this code a second time but with `overwrite=False` and `max_trials=10`, the tuner will continue tuning where it left off, running 5 more trials: this means you don't have to run all the trials in one shot. Lastly, since `objective` is set to "val_accuracy", the tuner prefers models with a higher validation accuracy, so once the tuner has finished searching, you can get the best models like this:

```
top3_models = random_search_tuner.get_best_models(num_models=3)  
best_model = top3_models[0]
```

You can also call `get_best_hyperparameters()` to get the `kt.HyperParameters` of the best models:

```
>>> top3_params = random_search_tuner.get_best_hyperparameters(num_trials=3)  
>>> top3_params[0].values # best hyperparameter values  
{'n_hidden': 5,  
 'n_neurons': 70,  
 'learning_rate': 0.00041268008323824807,  
 'optimizer': 'adam'}
```

Each tuner is guided by a so-called *oracle*: before each trial, the tuner asks the oracle to tell it what the next trial should be. The `RandomSearch` tuner uses a `RandomSearchOracle`, which is pretty basic: it just picks the next trial randomly, as we saw earlier. Since the oracle keeps track of all the trials, you can ask it to give you the best one, and you can display a summary of that trial:

```
>>> best_trial = random_search_tuner.oracle.get_best_trials(num_trials=1)[0]
>>> best_trial.summary()
Trial summary
Hyperparameters:
n_hidden: 5
n_neurons: 70
learning_rate: 0.00041268008323824807
optimizer: adam
Score: 0.8736000061035156
```

This shows the best hyperparameters (like earlier), as well as the validation accuracy. You can also access all the metrics directly:

```
>>> best_trial.metrics.get_last_value("val_accuracy")
0.8736000061035156
```

If you are happy with the best model's performance, you may continue training it for a few epochs on the full training set (`X_train_full` and `y_train_full`), then evaluate it on the test set, and deploy it to production (see [Chapter 19](#)):

```
best_model.fit(X_train_full, y_train_full, epochs=10)
test_loss, test_accuracy = best_model.evaluate(X_test, y_test)
```

In some cases, you may want to fine-tune data preprocessing hyperparameters, or `model.fit()` arguments, such as the batch size. For this, you must use a slightly different technique: instead of writing a `build_model()` function, you must subclass the `kt.HyperModel` class and define two methods, `build()` and `fit()`. The `build()` method does the exact same thing as the `build_model()` function, and the `fit()` method takes a `HyperParameters` object and a compiled model as argument, as well as all the `model.fit()` arguments, and it must fit the model and return the `History` object. Crucially, the `fit()` method may use hyperparameters to decide how to preprocess the data, tweak the batch size, and more. For example, the following class builds the same model as before, with the same hyperparameters, but it also uses a boolean "normalize" hyperparameter to control whether or not to standardize the training data before fitting the model:

```
class MyClassificationHyperModel(kt.HyperModel):
    def build(self, hp):
        return build_model(hp)

    def fit(self, hp, model, X, y, **kwargs):
        if hp.Boolean("normalize"):
            norm_layer = tf.keras.layers.Normalization()
```

```
X = norm_layer(X)
return model.fit(X, y, **kwargs)
```

You can then pass an instance of this class to the tuner of your choice, instead of passing the `build_model` function. For example, let's build a `kt.Hyperband` tuner based on a `MyClassificationHyperModel` instance:

```
hyperband_tuner = kt.Hyperband(
    MyClassificationHyperModel(), objective="val_accuracy", seed=42,
    max_epochs=10, factor=3, hyperband_iterations=2,
    overwrite=True, directory="my_fashion_mnist", project_name="hyperband")
```

This tuner is similar to the `HalvingRandomSearchCV` class we discussed in [Chapter 2](#): it starts by training many different models for few epochs, then it eliminates the worst models and keeps only the top 1 / factor models (i.e., the top third in this case), repeating this selection process until a single model is left.¹⁹ The `max_epochs` argument controls the max number of epochs that the best model will be trained for. The whole process is repeated twice in this case (`hyperband_iterations=2`). The total number of training epochs across all models for each hyperband iteration is about $\text{max_epochs} * (\log(\text{max_epochs}) / \log(\text{factor}))^{** 2}$, so it's about 44 epochs in this example. The other arguments are the same as for `kt.RandomSearch`.

Let's run the Hyperband tuner now. We'll use the `TensorBoard` callback, this time pointing to the root log directory (the tuner will take care of using a different subdirectory for each trial), as well as an `EarlyStopping` callback:

```
root_logdir = Path(hyperband_tuner.project_dir) / "tensorboard"
tensorboard_cb = tf.keras.callbacks.TensorBoard(root_logdir)
early_stopping_cb = tf.keras.callbacks.EarlyStopping(patience=2)
hyperband_tuner.search(X_train, y_train, epochs=10,
                       validation_data=(X_valid, y_valid),
                       callbacks=[early_stopping_cb, tensorboard_cb])
```

Now if you open TensorBoard, pointing `--logdir` to the `my_fashion_mnist/hyperband/tensorboard` directory, you will see all the trial results, as they unfold. Make sure to visit the HPARAMS tab: it contains a summary of all the hyperparameter combinations that were tried, along with the corresponding metrics. Notice that there are three tabs inside the HPARAMS tab: a table view, a parallel coordinates view, and a scatter plot matrix view. In the lower part of the left panel, uncheck all metrics except for `validation.epoch_accuracy`: this will make the graphs clearer. In the parallel coordinates view, try selecting a range of high values on the `validation.epoch_accuracy` column: this will filter only the hyperparameter combinations that reached a good performance. Click on one of the hyperparameter combinations:

¹⁹ Hyperband is actually a bit more sophisticated than successive halving, see [the paper](#) by Lisha Li et al., "Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization," *Journal of Machine Learning Research* 18 (April 2018): 1–52.

the corresponding learning curves will appear at the bottom of the page. Take some time to go through each tab: they will help you understand the effect of each hyperparameter on performance, as well as the interactions between the hyperparameters.

Hyperband is smarter than pure random search in the way it allocates resources, but at its core it still explores the hyperparameter space randomly: it's fast, but coarse. However, Keras Tuner also includes a `kt.BayesianOptimization` tuner: this algorithm gradually learns which regions of the hyperparameter space are most promising by fitting a probabilistic model called a *Gaussian process*. This allows it to gradually zoom in on the best hyperparameters. The downside is that this algorithm has its own hyperparameters: `alpha` represents the level of noise you expect in the performance measures across trials (it defaults to 10^{-4}), and `beta` specifies how much you want the algorithm to explore, instead of simply exploiting the known good regions of hyperparameter space (it defaults to 2.6). Other than that, this tuner can be used just like the previous ones:

```
bayesian_opt_tuner = kt.BayesianOptimization(  
    MyClassificationHyperModel(), objective="val_accuracy", seed=42,  
    max_trials=10, alpha=1e-4, beta=2.6,  
    overwrite=True, directory="my_fashion_mnist", project_name="bayesian_opt")  
bayesian_opt_tuner.search([...])
```

Hyperparameter tuning is still an active area of research, and many other approaches are being explored. For example, check out DeepMind's excellent [2017 paper](#),²⁰ where the authors used an evolutionary algorithm to jointly optimize a population of models and their hyperparameters. Google has also used an evolutionary approach, not just to search for hyperparameters but also to explore all sorts of model architectures: it powers their *AutoML* service on *Google Vertex AI* (see [Chapter 19](#)). The term *AutoML* refers to any system that takes care of a large part of the ML workflow. Evolutionary algorithms have even been used successfully to train individual neural networks, replacing the ubiquitous Gradient Descent! For an example, see the [2017 post](#) by Uber where the authors introduce their *Deep Neuroevolution* technique.

But despite all this exciting progress and all these tools and services, it still helps to have an idea of what values are reasonable for each hyperparameter so that you can build a quick prototype and restrict the search space. The following sections provide guidelines for choosing the number of hidden layers and neurons in an MLP and for selecting good values for some of the main hyperparameters.

²⁰ Max Jaderberg et al., "Population Based Training of Neural Networks," arXiv preprint arXiv:1711.09846 (2017).

Number of Hidden Layers

For many problems, you can begin with a single hidden layer and get reasonable results. An MLP with just one hidden layer can theoretically model even the most complex functions, provided it has enough neurons. But for complex problems, deep networks have a much higher *parameter efficiency* than shallow ones: they can model complex functions using exponentially fewer neurons than shallow nets, allowing them to reach much better performance with the same amount of training data.

To understand why, suppose you are asked to draw a forest using some drawing software, but you are forbidden to copy and paste anything. It would take an enormous amount of time: you would have to draw each tree individually, branch by branch, leaf by leaf. If you could instead draw one leaf, copy and paste it to draw a branch, then copy and paste that branch to create a tree, and finally copy and paste this tree to make a forest, you would be finished in no time. Real-world data is often structured in such a hierarchical way, and deep neural networks automatically take advantage of this fact: lower hidden layers model low-level structures (e.g., line segments of various shapes and orientations), intermediate hidden layers combine these low-level structures to model intermediate-level structures (e.g., squares, circles), and the highest hidden layers and the output layer combine these intermediate structures to model high-level structures (e.g., faces).

Not only does this hierarchical architecture help DNNs converge faster to a good solution, but it also improves their ability to generalize to new datasets. For example, if you have already trained a model to recognize faces in pictures and you now want to train a new neural network to recognize hairstyles, you can kickstart the training by reusing the lower layers of the first network. Instead of randomly initializing the weights and biases of the first few layers of the new neural network, you can initialize them to the values of the weights and biases of the lower layers of the first network. This way the network will not have to learn from scratch all the low-level structures that occur in most pictures; it will only have to learn the higher-level structures (e.g., hairstyles). This is called *transfer learning*.

In summary, for many problems you can start with just one or two hidden layers and the neural network will work just fine. For instance, you can easily reach above 97% accuracy on the MNIST dataset using just one hidden layer with a few hundred neurons, and above 98% accuracy using two hidden layers with the same total number of neurons, in roughly the same amount of training time. For more complex problems, you can ramp up the number of hidden layers until you start overfitting the training set. Very complex tasks, such as large image classification or speech recognition, typically require networks with dozens of layers (or even hundreds, but not fully connected ones, as we will see in [Chapter 14](#)), and they need a huge amount of training data. You will rarely have to train such networks from scratch: it is much more common to reuse parts of a pretrained state-of-the-art network that performs

a similar task. Training will then be a lot faster and require much less data (we will discuss this in [Chapter 11](#)).

Number of Neurons per Hidden Layer

The number of neurons in the input and output layers is determined by the type of input and output your task requires. For example, the MNIST task requires $28 \times 28 = 784$ inputs and 10 output neurons.

As for the hidden layers, it used to be common to size them to form a pyramid, with fewer and fewer neurons at each layer—the rationale being that many low-level features can coalesce into far fewer high-level features. A typical neural network for MNIST might have 3 hidden layers, the first with 300 neurons, the second with 200, and the third with 100. However, this practice has been largely abandoned because it seems that using the same number of neurons in all hidden layers performs just as well in most cases, or even better; plus, there is only one hyperparameter to tune, instead of one per layer. That said, depending on the dataset, it can sometimes help to make the first hidden layer bigger than the others.

Just like the number of layers, you can try increasing the number of neurons gradually until the network starts overfitting. Alternatively, you can try building a model with a bit more layers and neurons than you actually need, then use early stopping and other regularization techniques to prevent it from overfitting too much. Vincent Vanhoucke, a scientist at Google, has dubbed this the “stretch pants” approach: instead of wasting time looking for pants that perfectly match your size, just use large stretch pants that will shrink down to the right size. With this approach, you avoid bottleneck layers that could ruin your model. Indeed, if a layer has too few neurons, it will not have enough representational power to preserve all the useful information from the inputs (e.g., a layer with two neurons can only output 2D data, so if it gets 3D data as input, some information will be lost). No matter how big and powerful the rest of the network is, that information will never be recovered.



In general you will get more bang for your buck by increasing the number of layers instead of the number of neurons per layer.

Learning Rate, Batch Size, and Other Hyperparameters

The number of hidden layers and neurons are not the only hyperparameters you can tweak in an MLP. Here are some of the most important ones, as well as tips on how to set them:

Learning rate

The learning rate is arguably the most important hyperparameter. In general, the optimal learning rate is about half of the maximum learning rate (i.e., the learning rate above which the training algorithm diverges, as we saw in [Chapter 4](#)). One way to find a good learning rate is to train the model for a few hundred iterations, starting with a very low learning rate (e.g., 10^{-5}) and gradually increasing it up to a very large value (e.g., 10). This is done by multiplying the learning rate by a constant factor at each iteration (e.g., by $(10 / 10^{-5})^{1 / 500}$ to go from 10^{-5} to 10 in 500 iterations). If you plot the loss as a function of the learning rate (using a log scale for the learning rate), you should see it dropping at first. But after a while, the learning rate will be too large, so the loss will shoot back up: the optimal learning rate will be a bit lower than the point at which the loss starts to climb (typically about 10 times lower than the turning point). You can then reinitialize your model and train it normally using this good learning rate. We will look at more learning rate techniques in [Chapter 11](#).

Optimizer

Choosing a better optimizer than plain old Mini-batch Gradient Descent (and tuning its hyperparameters) is also quite important. We will see several advanced optimizers in [Chapter 11](#).

Batch size

The batch size can have a significant impact on your model’s performance and training time. The main benefit of using large batch sizes is that hardware accelerators like GPUs can process them efficiently (see [Chapter 19](#)), so the training algorithm will see more instances per second. Therefore, many researchers and practitioners recommend using the largest batch size that can fit in GPU RAM. There’s a catch, though: in practice, large batch sizes often lead to training instabilities, especially at the beginning of training, and the resulting model may not generalize as well as a model trained with a small batch size. In April 2018, Yann LeCun even tweeted “Friends don’t let friends use mini-batches larger than 32,” citing a [2018 paper²¹](#) by Dominic Masters and Carlo Luschi which concluded that using small batches (from 2 to 32) was preferable because small batches led to better models in less training time. Other papers point in the opposite direction, however; in 2017, papers by [Elad Hoffer et al.²²](#) and [Priya Goyal et al.²³](#) showed that it was possible to use very large batch sizes (up to 8,192) using various techniques such as warming up the learning rate (i.e., starting training

²¹ Dominic Masters and Carlo Luschi, “Revisiting Small Batch Training for Deep Neural Networks,” arXiv preprint arXiv:1804.07612 (2018).

²² Elad Hoffer et al., “Train Longer, Generalize Better: Closing the Generalization Gap in Large Batch Training of Neural Networks,” *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017): 1729–1739.

with a small learning rate, then ramping it up, as we will see in [Chapter 11](#)). This led to a very short training time, without any generalization gap. So, one strategy is to try to use a large batch size, using learning rate warmup, and if training is unstable or the final performance is disappointing, then try using a small batch size instead.

Activation function

We discussed how to choose the activation function earlier in this chapter: in general, the ReLU activation function will be a good default for all hidden layers. For the output layer, it really depends on your task.

Number of iterations

In most cases, the number of training iterations does not actually need to be tweaked: just use early stopping instead.



The optimal learning rate depends on the other hyperparameters—especially the batch size—so if you modify any hyperparameter, make sure to update the learning rate as well.

For more best practices regarding tuning neural network hyperparameters, check out the excellent [2018 paper²⁴](#) by Leslie Smith.

This concludes our introduction to artificial neural networks and their implementation with Keras. In the next few chapters, we will discuss techniques to train very deep nets. We will also explore how to customize models using TensorFlow’s lower-level API and how to load and preprocess data efficiently using the Data API. And we will dive into other popular neural network architectures: convolutional neural networks for image processing, recurrent neural networks and transformers for sequential data and text, autoencoders for representation learning, and generative adversarial networks to model and generate data.²⁵

Exercises

1. The [TensorFlow Playground](#) is a handy neural network simulator built by the TensorFlow team. In this exercise, you will train several binary classifiers in just

²³ Priya Goyal et al., “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour,” arXiv preprint arXiv:1706.02677 (2017).

²⁴ Leslie N. Smith, “A Disciplined Approach to Neural Network Hyper-Parameters: Part 1—Learning Rate, Batch Size, Momentum, and Weight Decay,” arXiv preprint arXiv:1803.09820 (2018).

²⁵ A few extra ANN architectures are presented in the online notebook at <https://homl.info/extranns>.

a few clicks, and tweak the model’s architecture and its hyperparameters to gain some intuition on how neural networks work and what their hyperparameters do. Take some time to explore the following:

- a. The patterns learned by a neural net. Try training the default neural network by clicking the Run button (top left). Notice how it quickly finds a good solution for the classification task. The neurons in the first hidden layer have learned simple patterns, while the neurons in the second hidden layer have learned to combine the simple patterns of the first hidden layer into more complex patterns. In general, the more layers there are, the more complex the patterns can be.
- b. Activation functions. Try replacing the tanh activation function with a ReLU activation function, and train the network again. Notice that it finds a solution even faster, but this time the boundaries are linear. This is due to the shape of the ReLU function.
- c. The risk of local minima. Modify the network architecture to have just one hidden layer with three neurons. Train it multiple times (to reset the network weights, click the Reset button next to the Play button). Notice that the training time varies a lot, and sometimes it even gets stuck in a local minimum.
- d. What happens when neural nets are too small. Remove one neuron to keep just two. Notice that the neural network is now incapable of finding a good solution, even if you try multiple times. The model has too few parameters and systematically underfits the training set.
- e. What happens when neural nets are large enough. Set the number of neurons to eight, and train the network several times. Notice that it is now consistently fast and never gets stuck. This highlights an important finding in neural network theory: large neural networks rarely get stuck in local minima, and even when they do these local optima are often almost as good as the global optimum. However, they can still get stuck on long plateaus for a long time.
- f. The risk of vanishing gradients in deep networks. Select the spiral dataset (the bottom-right dataset under “DATA”), and change the network architecture to have four hidden layers with eight neurons each. Notice that training takes much longer and often gets stuck on plateaus for long periods of time. Also notice that the neurons in the highest layers (on the right) tend to evolve faster than the neurons in the lowest layers (on the left). This problem, called the “vanishing gradients” problem, can be alleviated with better weight initialization and other techniques, better optimizers (such as AdaGrad or Adam), or Batch Normalization (discussed in [Chapter 11](#)).
- g. Go further. Take an hour or so to play around with other parameters and get a feel for what they do, to build an intuitive understanding about neural networks.

2. Draw an ANN using the original artificial neurons (like the ones in [Figure 10-3](#)) that computes $A \oplus B$ (where \oplus represents the XOR operation). Hint: $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$.
3. Why is it generally preferable to use a Logistic Regression classifier rather than a classical Perceptron (i.e., a single layer of threshold logic units trained using the Perceptron training algorithm)? How can you tweak a Perceptron to make it equivalent to a Logistic Regression classifier?
4. Why was the sigmoid activation function a key ingredient in training the first MLPs?
5. Name three popular activation functions. Can you draw them?
6. Suppose you have an MLP composed of one input layer with 10 passthrough neurons, followed by one hidden layer with 50 artificial neurons, and finally one output layer with 3 artificial neurons. All artificial neurons use the ReLU activation function.
 - What is the shape of the input matrix \mathbf{X} ?
 - What are the shapes of the hidden layer's weight matrix \mathbf{W}_h and bias vector \mathbf{b}_h ?
 - What are the shapes of the output layer's weight matrix \mathbf{W}_o and bias vector \mathbf{b}_o ?
 - What is the shape of the network's output matrix \mathbf{Y} ?
 - Write the equation that computes the network's output matrix \mathbf{Y} as a function of \mathbf{X} , \mathbf{W}_h , \mathbf{b}_h , \mathbf{W}_o , and \mathbf{b}_o .
7. How many neurons do you need in the output layer if you want to classify email into spam or ham? What activation function should you use in the output layer? If instead you want to tackle MNIST, how many neurons do you need in the output layer, and which activation function should you use? What about for getting your network to predict housing prices, as in [Chapter 2](#)?
8. What is backpropagation and how does it work? What is the difference between backpropagation and reverse-mode autodiff?
9. Can you list all the hyperparameters you can tweak in a basic MLP? If the MLP overfits the training data, how could you tweak these hyperparameters to try to solve the problem?
10. Train a deep MLP on the MNIST dataset (you can load it using `tf.keras.datasets.mnist.load_data()`). See if you can get over 98% accuracy by manually tuning the hyperparameters. Try searching for the optimal learning rate by using the approach presented in this chapter (i.e., by growing the learning rate exponentially, plotting the loss, and finding the point where the loss shoots up). Next, try tuning the hyperparameters using Keras Tuner with all the bells and

whistles—save checkpoints, use early stopping, and plot learning curves using TensorBoard.

Solutions to these exercises are available at the end of this chapter’s notebook, at <https://homl.info/colab3>.

Training Deep Neural Networks

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 11th chapter of the final book. The GitHub repo is [https://git
hub.com/ageron/handsonml3](https://github.com/ageron/handsonml3).

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

In [Chapter 10](#) you built, trained, and fine-tuned your first artificial neural networks. But they were shallow nets, with just a few hidden layers. What if you need to tackle a complex problem, such as detecting hundreds of types of objects in high-resolution images? You may need to train a much deeper ANN, perhaps with 10 layers or many more, each containing hundreds of neurons, linked by hundreds of thousands of connections. Training a deep neural network isn’t a walk in the park. Here are some of the problems you could run into:

- You may be faced with the tricky *vanishing gradients* problem or the related *exploding gradients* problem. This is when the gradients grow smaller and smaller, or larger and larger, when flowing backward through the DNN during training. Both of these problems make lower layers very hard to train.
- You might not have enough training data for such a large network, or it might be too costly to label.
- Training may be extremely slow.

- A model with millions of parameters would severely risk overfitting the training set, especially if there are not enough training instances or if they are too noisy.

In this chapter we will go through each of these problems and present techniques to solve them. We will start by exploring the vanishing and exploding gradients problems and some of their most popular solutions. Next, we will look at transfer learning and unsupervised pretraining, which can help you tackle complex tasks even when you have little labeled data. Then we will discuss various optimizers that can speed up training large models tremendously. Finally, we will cover a few popular regularization techniques for large neural networks.

With these tools, you will be able to train very deep nets. Welcome to Deep Learning!

The Vanishing/Exploding Gradients Problems

As we discussed in [Chapter 10](#), the backpropagation algorithm’s second phase works by going from the output layer to the input layer, propagating the error gradient along the way. Once the algorithm has computed the gradient of the cost function with regard to each parameter in the network, it uses these gradients to update each parameter with a Gradient Descent step.

Unfortunately, gradients often get smaller and smaller as the algorithm progresses down to the lower layers. As a result, the Gradient Descent update leaves the lower layers’ connection weights virtually unchanged, and training never converges to a good solution. This is called the *vanishing gradients* problem. In some cases, the opposite can happen: the gradients can grow bigger and bigger until layers get insanely large weight updates and the algorithm diverges. This is the *exploding gradients* problem, which surfaces most often in recurrent neural networks (see [Chapter 15](#)). More generally, deep neural networks suffer from unstable gradients; different layers may learn at widely different speeds.

This unfortunate behavior was empirically observed long ago, and it was one of the reasons deep neural networks were mostly abandoned in the early 2000s. It wasn’t clear what caused the gradients to be so unstable when training a DNN, but some light was shed in a [2010 paper](#) by Xavier Glorot and Yoshua Bengio.¹ The authors found a few suspects, including the combination of the popular sigmoid (logistic) activation function and the weight initialization technique that was most popular at the time (i.e., a normal distribution with a mean of 0 and a standard deviation of 1). In short, they showed that with this activation function and this initialization scheme, the variance of the outputs of each layer is much greater than the variance of its

¹ Xavier Glorot and Yoshua Bengio, “Understanding the Difficulty of Training Deep Feedforward Neural Networks,” *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics* (2010): 249–256.

inputs. Going forward in the network, the variance keeps increasing after each layer until the activation function saturates at the top layers. This saturation is actually made worse by the fact that the sigmoid function has a mean of 0.5, not 0 (the hyperbolic tangent function has a mean of 0 and behaves slightly better than the sigmoid function in deep networks).

Looking at the sigmoid activation function (see [Figure 11-1](#)), you can see that when inputs become large (negative or positive), the function saturates at 0 or 1, with a derivative extremely close to 0 (i.e., the curve is flat at both extremes). Thus, when backpropagation kicks in it has virtually no gradient to propagate back through the network; and what little gradient exists keeps getting diluted as backpropagation progresses down through the top layers, so there is really nothing left for the lower layers.

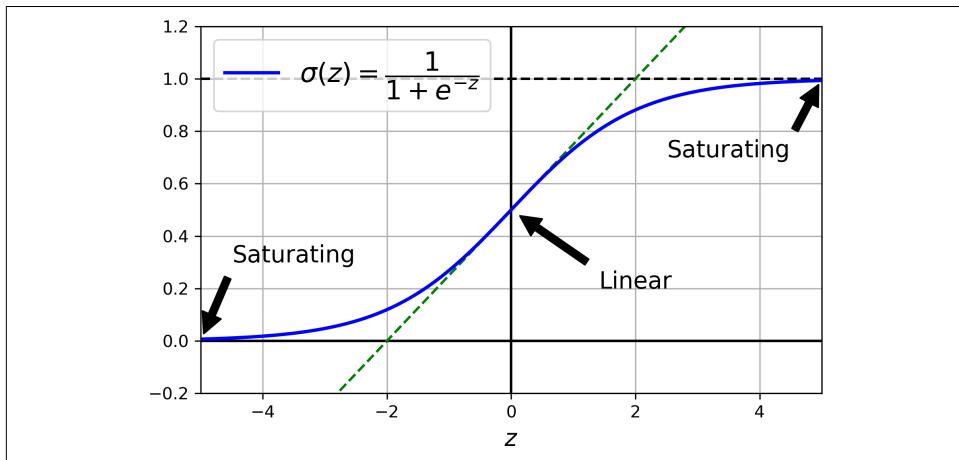


Figure 11-1. Sigmoid activation function saturation

Glorot and He Initialization

In their paper, Glorot and Bengio propose a way to significantly alleviate the unstable gradients problem. They point out that we need the signal to flow properly in both directions: in the forward direction when making predictions, and in the reverse direction when backpropagating gradients. We don't want the signal to die out, nor do we want it to explode and saturate. For the signal to flow properly, the authors argue that we need the variance of the outputs of each layer to be equal to the variance of its inputs,² and we need the gradients to have equal variance before and after flowing through a layer in the reverse direction (please check out the paper if

² Here's an analogy: if you set a microphone amplifier's knob too close to zero, people won't hear your voice, but if you set it too close to the max, your voice will be saturated and people won't understand what you are saying. Now imagine a chain of such amplifiers: they all need to be set properly in order for your voice to

you are interested in the mathematical details). It is actually not possible to guarantee both unless the layer has an equal number of inputs and outputs (these numbers are called the *fan-in* and *fan-out* of the layer), but Glorot and Bengio proposed a good compromise that has proven to work very well in practice: the connection weights of each layer must be initialized randomly as described in [Equation 11-1](#), where $\text{fan}_{\text{avg}} = (\text{fan}_{\text{in}} + \text{fan}_{\text{out}})/2$. This initialization strategy is called *Xavier initialization* or *Glorot initialization*, after the paper's first author.

Equation 11-1. Glorot initialization (when using the sigmoid activation function)

Normal distribution with mean 0 and variance $\sigma^2 = \frac{1}{\text{fan}_{\text{avg}}}$

Or a uniform distribution between $-r$ and $+r$, with $r = \sqrt{\frac{3}{\text{fan}_{\text{avg}}}}$

If you replace fan_{avg} with fan_{in} in [Equation 11-1](#), you get an initialization strategy that Yann LeCun proposed in the 1990s. He called it *LeCun initialization*. Genevieve Orr and Klaus-Robert Müller even recommended it in their 1998 book *Neural Networks: Tricks of the Trade* (Springer). LeCun initialization is equivalent to Glorot initialization when $\text{fan}_{\text{in}} = \text{fan}_{\text{out}}$. It took over a decade for researchers to realize how important this trick is. Using Glorot initialization can speed up training considerably, and it is one of the tricks that led to the success of Deep Learning.

Some papers³ have provided similar strategies for different activation functions. These strategies differ only by the scale of the variance and whether they use fan_{avg} or fan_{in} , as shown in [Table 11-1](#) (for the uniform distribution, just use $r = \sqrt{3\sigma^2}$). The initialization strategy for the ReLU activation function and its variants is called *He initialization* or *Kaiming initialization*, after the paper's first author. For SELU, use Yann LeCun's initialization method, preferably with a Normal distribution. We will cover all these activation functions shortly.

Table 11-1. Initialization parameters for each type of activation function

Initialization	Activation functions	σ^2 (Normal)
Xavier Glorot	None, tanh, sigmoid, softmax	$1 / \text{fan}_{\text{avg}}$
Kaiming He	ReLU, Leaky ReLU, ELU, GELU, Swish, Mish	$2 / \text{fan}_{\text{in}}$
Yann LeCun	SELU	$1 / \text{fan}_{\text{in}}$

come out loud and clear at the end of the chain. Your voice has to come out of each amplifier at the same amplitude as it came in.

³ E.g., Kaiming He et al., “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification,” *Proceedings of the 2015 IEEE International Conference on Computer Vision* (2015): 1026–1034.

By default, Keras uses Glorot initialization with a uniform distribution. When creating a layer, you can change this to He initialization by setting `kernel_initializer="he_uniform"` or `kernel_initializer="he_normal"` like this:

```
import tensorflow as tf

dense = tf.keras.layers.Dense(50, activation="relu",
                             kernel_initializer="he_normal")
```

Alternatively, you can obtain any of the initializations listed in [Table 11-1](#) and more using the `VarianceScaling` initializer. For example, if you want He initialization with a uniform distribution and based on fan_{avg} (rather than fan_{in}), you can use the following code:

```
he_avg_init = tf.keras.initializers.VarianceScaling(scale=2., mode="fan_avg",
                                                      distribution="uniform")
dense = tf.keras.layers.Dense(50, activation="sigmoid",
                             kernel_initializer=he_avg_init)
```

Better Activation Functions

One of the insights in the 2010 paper by Glorot and Bengio was that the problems with unstable gradients were in part due to a poor choice of activation function. Until then most people had assumed that if Mother Nature had chosen to use roughly sigmoid activation functions in biological neurons, they must be an excellent choice. But it turns out that other activation functions behave much better in deep neural networks—in particular, the ReLU activation function, mostly because it does not saturate for positive values, and also because it is very fast to compute.

Unfortunately, the ReLU activation function is not perfect. It suffers from a problem known as the *dying ReLUs*: during training, some neurons effectively “die,” meaning they stop outputting anything other than 0. In some cases, you may find that half of your network’s neurons are dead, especially if you used a large learning rate. A neuron dies when its weights get tweaked in such a way that the input of the ReLU function (i.e., the weighted sum of the neuron’s inputs plus its bias term) is negative for all instances in the training set. When this happens, it just keeps outputting zeros, and Gradient Descent does not affect it anymore because the gradient of the ReLU function is zero when its input is negative.⁴

To solve this problem, you may want to use a variant of the ReLU function, such as the *leaky ReLU*.

⁴ A dead neuron may come back to life if its inputs evolve over time and eventually return within a range where the ReLU activation function gets a positive input again. For example, this may happen if Gradient Descent tweaks the neurons in the layers below the dead neuron.

Leaky ReLU

The leaky ReLU activation function is defined as $\text{LeakyReLU}_\alpha(z) = \max(\alpha z, z)$ (see Figure 11-2). The hyperparameter α defines how much the function “leaks”: it is the slope of the function for $z < 0$. Having a slope for $z < 0$ ensures that leaky ReLUs never die; they can go into a long coma, but they have a chance to eventually wake up. A [2015 paper](#) by Bing Xu et al.⁵ compared several variants of the ReLU activation function, and one of its conclusions was that the leaky variants always outperformed the strict ReLU activation function. In fact, setting $\alpha = 0.2$ (a huge leak) seemed to result in better performance than $\alpha = 0.01$ (a small leak). The paper also evaluated the *randomized leaky ReLU* (RReLU), where α is picked randomly in a given range during training and is fixed to an average value during testing. RReLU also performed fairly well and seemed to act as a regularizer, reducing the risk of overfitting the training set. Finally, the paper evaluated the *parametric leaky ReLU* (PReLU), where α is authorized to be learned during training: instead of being a hyperparameter, it becomes a parameter that can be modified by backpropagation like any other parameter. PReLU was reported to strongly outperform ReLU on large image datasets, but on smaller datasets it runs the risk of overfitting the training set.

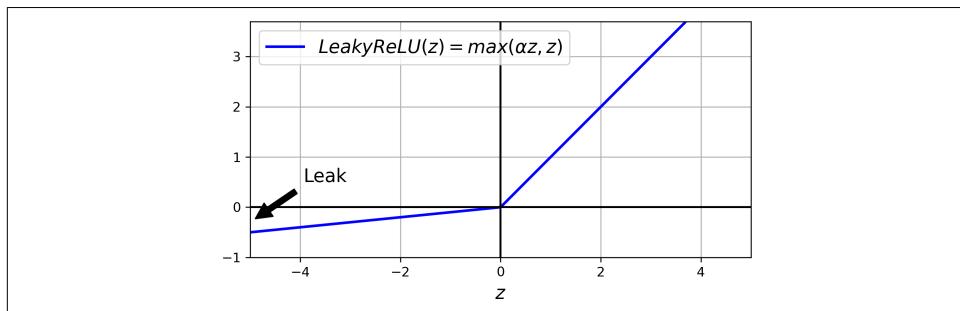


Figure 11-2. Leaky ReLU: like ReLU, but with a small slope for negative values

Keras includes the classes `LeakyReLU` and `PReLU` in the `tf.keras.layers` package. Just like for other ReLU variants, you should use He initialization, for example:

```
leaky_relu = tf.keras.layers.LeakyReLU(alpha=0.2) # defaults to alpha=0.3
dense = tf.keras.layers.Dense(50, activation=leaky_relu,
                             kernel_initializer="he_normal")
```

If you prefer, you can also use `LeakyReLU` as a separate layer in your model, it makes no difference for training and predictions:

```
model = tf.keras.models.Sequential([
    [...] # more layers
```

⁵ Bing Xu et al., “Empirical Evaluation of Rectified Activations in Convolutional Network,” arXiv preprint arXiv:1505.00853 (2015).

```

    tf.keras.layers.Dense(50, kernel_initializer="he_normal"), # no activation
    tf.keras.layers.LeakyReLU(alpha=0.2), # activation as a separate layer
    [...] # more layers
])

```

For PReLU, replace LeakyReLU with PReLU. There is currently no official implementation of RReLU in Keras, but you can fairly easily implement your own (to learn how to do that, see the exercises at the end of [Chapter 12](#)).

ReLU, leaky ReLU, and PReLU all suffer from the fact that they are not smooth functions: their derivatives abruptly change (at $z = 0$). As we saw in [Chapter 4](#) when we discussed Lasso, this sort of discontinuity can make Gradient Descent bounce around the optimum, and slow down convergence. So now we will look at smooth variants of the ReLU activation function, starting with ELU and SELU.

ELU and SELU

In a [2016 paper](#) by Djork-Arné Clevert et al.⁶, a new activation function was proposed, called the *exponential linear unit* (ELU), that outperformed all the ReLU variants in the authors' experiments: training time was reduced, and the neural network performed better on the test set. [Equation 11-2](#) shows this activation function's definition.

Equation 11-2. ELU activation function

$$\text{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

The ELU activation function looks a lot like the ReLU function (see [Figure 11-3](#)), with a few major differences:

- It takes on negative values when $z < 0$, which allows the unit to have an average output closer to 0 and helps alleviate the vanishing gradients problem. The hyperparameter α defines the opposite of the value that the ELU function approaches when z is a large negative number. It is usually set to 1, but you can tweak it like any other hyperparameter.
- It has a nonzero gradient for $z < 0$, which avoids the dead neurons problem.
- If α is equal to 1 then the function is smooth everywhere, including around $z = 0$, which helps speed up Gradient Descent since it does not bounce as much to the left and right of $z = 0$.

⁶ Djork-Arné Clevert et al., “Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs),” *Proceedings of the International Conference on Learning Representations* (2016).

Using ELU with Keras is as easy as setting `activation="elu"`, and like with other ReLU variants, you should use He initialization. The main drawback of the ELU activation function is that it is slower to compute than the ReLU function and its variants (due to the use of the exponential function). Its faster convergence rate during training may compensate for that slow computation, but still, at test time an ELU network will be a bit slower than a ReLU network.

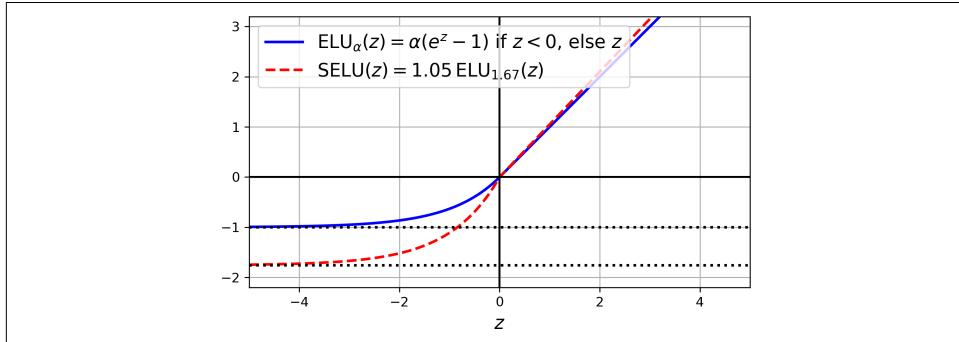


Figure 11-3. ELU and SELU activation functions

A year after ELU, a [2017 paper](#) by Günter Klambauer et al.⁷ introduced the Scaled ELU (SELU) activation function: as its name suggests, it is a scaled variant of the ELU activation function (about 1.05 times ELU, using $\alpha \approx 1.67$). The authors showed that if you build a neural network composed exclusively of a stack of dense layers (i.e., an MLP), and if all hidden layers use the SELU activation function, then the network will *self-normalize*: the output of each layer will tend to preserve a mean of 0 and a standard deviation of 1 during training, which solves the vanishing/exploding gradients problem. As a result, the SELU activation function may outperform other activation functions for MLPs, especially deep ones. To use it with Keras, just set `activation="selu"`. There are, however, a few conditions for self-normalization to happen (see the paper for the mathematical justification):

- The input features must be standardized: mean 0 and standard deviation 1.
- Every hidden layer's weights must be initialized using LeCun normal initialization. In Keras, this means setting `kernel_initializer="lecun_normal"`.
- The self-normalizing property is only guaranteed with plain MLPs. If you try to use SELU in other architectures, like recurrent networks (see [Chapter 15](#)) or networks with *skip connections* (i.e., connections that skip layers, such as in Wide & Deep nets), it will probably not outperform ELU.

⁷ Günter Klambauer et al., “Self-Normalizing Neural Networks,” *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017): 972–981.

- You cannot use regularization techniques like ℓ_1 or ℓ_2 regularization, max-norm, batch-norm or regular dropout (these are discussed later in this chapter).

These are significant constraints, so despite its promises, SELU did not gain a lot of traction. Moreover, three more activation functions seem to outperform it quite consistently on most tasks: GELU, Swish and Mish.

GELU, Swish and Mish

GELU was introduced in a [2016 paper](#) by Dan Hendrycks and Kevin Gimpel.⁸ Once again, you can think of it as a smooth variant of the ReLU activation function. Its definition is given in [Equation 11-3](#), where Φ is the standard Gaussian cumulative distribution function (CDF): $\Phi(z)$ corresponds to the probability that a value sampled randomly from a Normal distribution of mean 0 and variance 1 is lower than z .

Equation 11-3. GELU activation function

$$\text{GELU}(z) = z\Phi(z)$$

As you can see in [Figure 11-4](#), GELU resembles ReLU: it approaches 0 when its input z is very negative, and it approaches z when z is very positive. However, whereas all activation functions we discussed so far were both convex and monotonic,⁹ the GELU activation function is neither: from left to right, it starts by going straight, then it wiggles down, reaches a low point around -0.17 (near $z \approx -0.75$), and finally it bounces up and ends up going straight towards the top-right. This fairly complex shape and the fact that it has a curvature at every point may explain why it works so well, especially for complex tasks: Gradient Descent may find it easier to fit complex patterns. In practice, it often outperforms every other activation function discussed so far. However, it is a bit more computationally intensive, and the performance boost it provides is not always sufficient to justify the extra cost. That said, it is possible to show that it is approximately equal to $z\sigma(1.702 z)$, where σ is the sigmoid function: using this approximation also works very well, and it has the advantage of being much faster to compute. In fact, the GELU paper also introduced the SiLU activation function, which is equal to $z\sigma(z)$, but it was outperformed by GELU in their tests.

⁸ Dan Hendrycks and Kevin Gimpel, “Gaussian error linear units (GELUs),” arXiv preprint arXiv:1606.08415 (2016).

⁹ A function is convex if the line segment between any two points on the curve never lies below the curve. A monotonic function only increases, or only decreases.

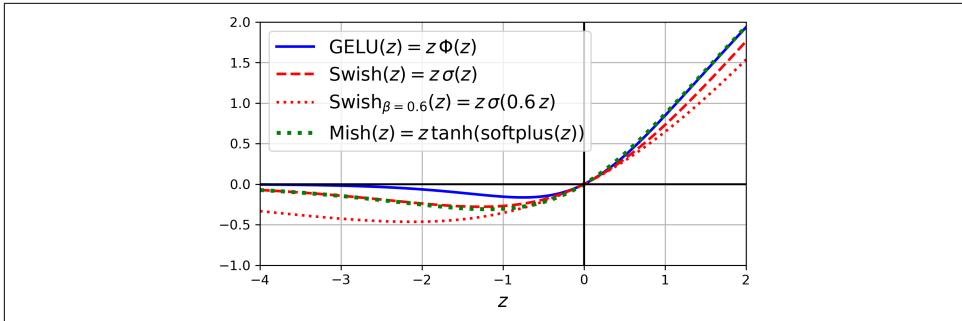


Figure 11-4. GELU, Swish, parametrized Swish, and Mish activation functions

Interestingly, a [2017 paper](#) by Prajit Ramachandran et al. rediscovered the SiLU function by automatically searching for good activation functions.¹⁰ They named it *Swish*, and the name caught on. In their paper, Swish outperformed every other function, including GELU. The authors later generalized Swish by adding an extra hyperparameter β to scale the sigmoid function's input. The generalized Swish function is $\text{Swish}_\beta(z) = z\sigma(\beta z)$, so GELU is approximately equal to the generalized Swish function using $\beta = 1.702$. You can tune β like any other hyperparameter. Alternatively, it's also possible to make β trainable and let Gradient Descent optimize it: much like PReLU, this can make your model more powerful, but it also runs the risk of overfitting the data.

Another quite similar activation function is *Mish*, which was introduced in a [2019 paper](#) by Diganta Misra.¹¹ It is defined as $\text{mish}(z) = z\tanh(\text{softplus}(z))$, where $\text{softplus}(z) = \log(1 + \exp(z))$. Just like GELU and Swish, it is a smooth, non-convex, and non-monotonic variant of ReLU, and once again the author ran many experiments and found that Mish generally outperformed other activation functions, even Swish and GELU by a tiny margin. [Figure 11-4](#) shows GELU, Swish (both with the default $\beta = 1$ and with $\beta = 0.6$), and lastly Mish. As you can see, Mish overlaps almost perfectly with Swish when z is negative, and almost perfectly with GELU when z is positive.

Keras supports GELU and Swish out of the box, just use `activation="gelu"` or `activation="swish"`. However, it does not support Mish or the generalized Swish activation function yet (but see [Chapter 12](#) to see how to implement your own activation functions and layers).

¹⁰ Prajit Ramachandran et al., “Searching for Activation Functions,” arXiv preprint arXiv:1710.05941 (2017).

¹¹ Diganta Misra, “Mish: A self regularized non-monotonic activation function.” arXiv preprint arXiv:1908.08681 (2019).



So, which activation function should you use for the hidden layers of your deep neural networks? ReLU remains a good default for simple tasks: it's often just as good as the more sophisticated activation functions, plus it's very fast to compute, and many libraries and hardware accelerators provide ReLU-specific optimizations. However, Swish is probably a better default for more complex tasks, and you can even try parametrized Swish with a learnable β parameter for the most complex tasks. Mish may give you slightly better results, but it requires a bit more compute. If you care a lot about runtime latency, then you may prefer leaky ReLU, or parametrized leaky ReLU for more complex tasks. For deep MLPs, give SELU a try, but make sure to respect the constraints listed earlier. If you have spare time and computing power, you can use cross-validation to evaluate other activation functions as well.

That's all for activation functions! Now, let's look at a completely different way to solve the unstable gradients problem: Batch Normalization.

Batch Normalization

Although using He initialization along with ReLU (or any of its variants) can significantly reduce the danger of the vanishing/exploding gradients problems at the beginning of training, it doesn't guarantee that they won't come back during training.

In a [2015 paper](#),¹² Sergey Ioffe and Christian Szegedy proposed a technique called *Batch Normalization* (BN) that addresses these problems. The technique consists of adding an operation in the model just before or after the activation function of each hidden layer. This operation simply zero-centers and normalizes each input, then scales and shifts the result using two new parameter vectors per layer: one for scaling, the other for shifting. In other words, the operation lets the model learn the optimal scale and mean of each of the layer's inputs. In many cases, if you add a BN layer as the very first layer of your neural network, you do not need to standardize your training set (i.e., no need for `StandardScaler` or `Normalization`); the BN layer will do it for you (well, approximately, since it only looks at one batch at a time, and it can also rescale and shift each input feature).

In order to zero-center and normalize the inputs, the algorithm needs to estimate each input's mean and standard deviation. It does so by evaluating the mean and

¹² Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," *Proceedings of the 32nd International Conference on Machine Learning* (2015): 448–456.

standard deviation of the input over the current mini-batch (hence the name “Batch Normalization”). The whole operation is summarized step by step in [Equation 11-4](#).

Equation 11-4. Batch Normalization algorithm

1. $\mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$
2. $\sigma_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} (\mathbf{x}^{(i)} - \mu_B)^2$
3. $\hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$
4. $\mathbf{z}^{(i)} = \gamma \otimes \hat{\mathbf{x}}^{(i)} + \beta$

In this algorithm:

- μ_B is the vector of input means, evaluated over the whole mini-batch B (it contains one mean per input).
- m_B is the number of instances in the mini-batch.
- σ_B is the vector of input standard deviations, also evaluated over the whole mini-batch (it contains one standard deviation per input).
- $\hat{\mathbf{x}}^{(i)}$ is the vector of zero-centered and normalized inputs for instance i .
- ε is a tiny number that avoids division by zero and ensures the gradients don't grow too large (typically 10^{-5}). This is called a *smoothing term*.
- γ is the output scale parameter vector for the layer (it contains one scale parameter per input).
- \otimes represents element-wise multiplication (each input is multiplied by its corresponding output scale parameter).
- β is the output shift (offset) parameter vector for the layer (it contains one offset parameter per input). Each input is offset by its corresponding shift parameter.
- $\mathbf{z}^{(i)}$ is the output of the BN operation. It is a rescaled and shifted version of the inputs.

So during training, BN standardizes its inputs, then rescales and offsets them. Good! What about at test time? Well, it's not that simple. Indeed, we may need to make predictions for individual instances rather than for batches of instances: in this case, we will have no way to compute each input's mean and standard deviation. Moreover, even if we do have a batch of instances, it may be too small, or the instances may not be independent and identically distributed, so computing statistics over the batch

instances would be unreliable. One solution could be to wait until the end of training, then run the whole training set through the neural network and compute the mean and standard deviation of each input of the BN layer. These “final” input means and standard deviations could then be used instead of the batch input means and standard deviations when making predictions. However, most implementations of Batch Normalization estimate these final statistics during training by using a moving average of the layer’s input means and standard deviations. This is what Keras does automatically when you use the `BatchNormalization` layer. To sum up, four parameter vectors are learned in each batch-normalized layer: γ (the output scale vector) and β (the output offset vector) are learned through regular backpropagation, and μ (the final input mean vector) and σ (the final input standard deviation vector) are estimated using an exponential moving average. Note that μ and σ are estimated during training, but they are used only after training (to replace the batch input means and standard deviations in [Equation 11-4](#)).

Ioffe and Szegedy demonstrated that Batch Normalization considerably improved all the deep neural networks they experimented with, leading to a huge improvement in the ImageNet classification task (ImageNet is a large database of images classified into many classes, commonly used to evaluate computer vision systems). The vanishing gradients problem was strongly reduced, to the point that they could use saturating activation functions such as the tanh and even the sigmoid activation function. The networks were also much less sensitive to the weight initialization. The authors were able to use much larger learning rates, significantly speeding up the learning process. Specifically, they note that:

Applied to a state-of-the-art image classification model, Batch Normalization achieves the same accuracy with 14 times fewer training steps, and beats the original model by a significant margin. [...] Using an ensemble of batch-normalized networks, we improve upon the best published result on ImageNet classification: reaching 4.9% top-5 validation error (and 4.8% test error), exceeding the accuracy of human raters.

Finally, like a gift that keeps on giving, Batch Normalization acts like a regularizer, reducing the need for other regularization techniques (such as dropout, described later in this chapter).

Batch Normalization does, however, add some complexity to the model (although it can remove the need for normalizing the input data, as we discussed earlier). Moreover, there is a runtime penalty: the neural network makes slower predictions due to the extra computations required at each layer. Fortunately, it’s often possible to fuse the BN layer with the previous layer, after training, thereby avoiding the runtime penalty. This is done by updating the previous layer’s weights and biases so that it directly produces outputs of the appropriate scale and offset. For example, if the previous layer computes $XW + b$, then the BN layer will compute $\gamma \otimes (XW + b - \mu) / \sigma + \beta$ (ignoring the smoothing term ϵ in the denominator). If we define $W' = \gamma \otimes W / \sigma$ and $b' = \gamma \otimes (b - \mu) / \sigma + \beta$, the equation simplifies to $XW' + b'$. So if we

replace the previous layer's weights and biases (\mathbf{W} and \mathbf{b}) with the updated weights and biases (\mathbf{W}' and \mathbf{b}'), we can get rid of the BN layer (TFLite's converter does this automatically; see [Chapter 19](#)).



You may find that training is rather slow, because each epoch takes much more time when you use Batch Normalization. This is usually counterbalanced by the fact that convergence is much faster with BN, so it will take fewer epochs to reach the same performance. All in all, *wall time* will usually be shorter (this is the time measured by the clock on your wall).

Implementing Batch Normalization with Keras

As with most things with Keras, implementing Batch Normalization is straightforward and intuitive. Just add a `BatchNormalization` layer before or after each hidden layer's activation function. You may also add a BN layer as the first layer in your model, but a plain `Normalization` layer generally performs just as well in this location (its only drawback is that you must first call its `adapt()` method). For example, this model applies BN after every hidden layer and as the first layer in the model (after flattening the input images):

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(300, activation="relu",
                         kernel_initializer="he_normal"),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(100, activation="relu",
                         kernel_initializer="he_normal"),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(10, activation="softmax")
])
```

That's all! In this tiny example with just two hidden layers, Batch Normalization is unlikely to have a large impact, but for deeper networks it can make a tremendous difference.

Let's display the model summary:

```
>>> model.summary()
Model: "sequential"
```

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
batch_normalization (BatchNo	(None, 784)	3136
dense (Dense)	(None, 300)	235500

batch_normalization_1 (Batch (None, 300)	1200
dense_1 (Dense) (None, 100)	30100
batch_normalization_2 (Batch (None, 100)	400
dense_2 (Dense) (None, 10)	1010
Total params:	271,346
Trainable params:	268,978
Non-trainable params:	2,368

As you can see, each BN layer adds four parameters per input: γ , β , μ , and σ (for example, the first BN layer adds 3,136 parameters, which is 4×784). The last two parameters, μ and σ , are the moving averages; they are not affected by backpropagation, so Keras calls them “non-trainable”¹³ (if you count the total number of BN parameters, $3,136 + 1,200 + 400$, and divide by 2, you get 2,368, which is the total number of non-trainable parameters in this model).

Let’s look at the parameters of the first BN layer. Two are trainable (by backpropagation), and two are not:

```
>>> [(var.name, var.trainable) for var in model.layers[1].variables]
[('batch_normalization/gamma:0', True),
 ('batch_normalization/beta:0', True),
 ('batch_normalization/moving_mean:0', False),
 ('batch_normalization/moving_variance:0', False)]
```

The authors of the BN paper argued in favor of adding the BN layers before the activation functions, rather than after (as we just did). There is some debate about this, as which is preferable seems to depend on the task—you can experiment with this too to see which option works best on your dataset. To add the BN layers before the activation functions, you must remove the activation function from the hidden layers and add them as separate layers after the BN layers. Moreover, since a Batch Normalization layer includes one offset parameter per input, you can remove the bias term from the previous layer by passing `use_bias=False` when creating it. And lastly, you can usually drop the first BN layer to avoid sandwiching the first hidden layer between two BN layers. The updated code looks like this:

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(300, kernel_initializer="he_normal", use_bias=False),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Activation("relu"),
```

¹³ However, they are estimated during training, based on the training data, so arguably they *are* trainable. In Keras, “non-trainable” really means “untouched by backpropagation.”

```

    tf.keras.layers.Dense(100, kernel_initializer="he_normal", use_bias=False),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Activation("relu"),
    tf.keras.layers.Dense(10, activation="softmax")
])

```

The `BatchNormalization` class has quite a few hyperparameters you can tweak. The defaults will usually be fine, but you may occasionally need to tweak the `momentum`. This hyperparameter is used by the `BatchNormalization` layer when it updates the exponential moving averages; given a new value \mathbf{v} (i.e., a new vector of input means or standard deviations computed over the current batch), the layer updates the running average $\hat{\mathbf{v}}$ using the following equation:

$$\hat{\mathbf{v}} \leftarrow \hat{\mathbf{v}} \times \text{momentum} + \mathbf{v} \times (1 - \text{momentum})$$

A good momentum value is typically close to 1; for example, 0.9, 0.99, or 0.999. You want more 9s for larger datasets and for smaller mini-batches.

Another important hyperparameter is `axis`: it determines which axis should be normalized. It defaults to `-1`, meaning that by default it will normalize the last axis (using the means and standard deviations computed across the *other* axes). When the input batch is 2D (i.e., the batch shape is `[batch size, features]`), this means that each input feature will be normalized based on the mean and standard deviation computed across all the instances in the batch. For example, the first BN layer in the previous code example will independently normalize (and rescale and shift) each of the 784 input features. If we move the first BN layer before the `Flatten` layer, then the input batches will be 3D, with shape `[batch size, height, width]`; therefore, the BN layer will compute 28 means and 28 standard deviations (1 per column of pixels, computed across all instances in the batch and across all rows in the column), and it will normalize all pixels in a given column using the same mean and standard deviation. There will also be just 28 scale parameters and 28 shift parameters. If instead you still want to treat each of the 784 pixels independently, then you should set `axis=[1, 2]`.

Batch Normalization has become one of the most-used layers in deep neural networks, especially deep convolutional neural networks (Chapter 14), to the point that it is often omitted in the architecture diagrams: it is assumed that BN is added after every layer. Now let's look at one last technique to stabilize gradients during training: Gradient Clipping.

Gradient Clipping

Another technique to mitigate the exploding gradients problem is to clip the gradients during backpropagation so that they never exceed some threshold. This is called

Gradient Clipping.¹⁴ This technique is generally used in recurrent neural networks, since Batch Normalization is tricky to use in recurrent neural networks (RNNs), as we will see in [Chapter 15](#).

In Keras, implementing Gradient Clipping is just a matter of setting the `clipvalue` or `clipnorm` argument when creating an optimizer, like this:

```
optimizer = tf.keras.optimizers.SGD(clipvalue=1.0)
model.compile(..., optimizer=optimizer)
```

This optimizer will clip every component of the gradient vector to a value between -1.0 and 1.0 . This means that all the partial derivatives of the loss (with regard to each and every trainable parameter) will be clipped between -1.0 and 1.0 . The threshold is a hyperparameter you can tune. Note that it may change the orientation of the gradient vector. For instance, if the original gradient vector is $[0.9, 100.0]$, it points mostly in the direction of the second axis; but once you clip it by value, you get $[0.9, 1.0]$, which points roughly in the diagonal between the two axes. In practice, this approach works well. If you want to ensure that Gradient Clipping does not change the direction of the gradient vector, you should clip by norm by setting `clipnorm` instead of `clipvalue`. This will clip the whole gradient if its ℓ_2 norm is greater than the threshold you picked. For example, if you set `clipnorm=1.0`, then the vector $[0.9, 100.0]$ will be clipped to $[0.00899964, 0.9999595]$, preserving its orientation but almost eliminating the first component. If you observe that the gradients explode during training (you can track the size of the gradients using TensorBoard), you may want to try clipping by value or clipping by norm, with different thresholds, and see which option performs best on the validation set.

Reusing Pretrained Layers

It is generally not a good idea to train a very large DNN from scratch without first trying to find an existing neural network that accomplishes a similar task to the one you are trying to tackle (we will discuss how to find them in [Chapter 14](#)). If you find such a neural network, then you can generally reuse most of its layers, except for the top ones. This technique is called *transfer learning*. It will not only speed up training considerably, but also require significantly less training data.

Suppose you have access to a DNN that was trained to classify pictures into 100 different categories, including animals, plants, vehicles, and everyday objects. You now want to train a DNN to classify specific types of vehicles. These tasks are very similar, even partly overlapping, so you should try to reuse parts of the first network (see [Figure 11-5](#)).

¹⁴ Razvan Pascanu et al., “On the Difficulty of Training Recurrent Neural Networks,” *Proceedings of the 30th International Conference on Machine Learning* (2013): 1310–1318.

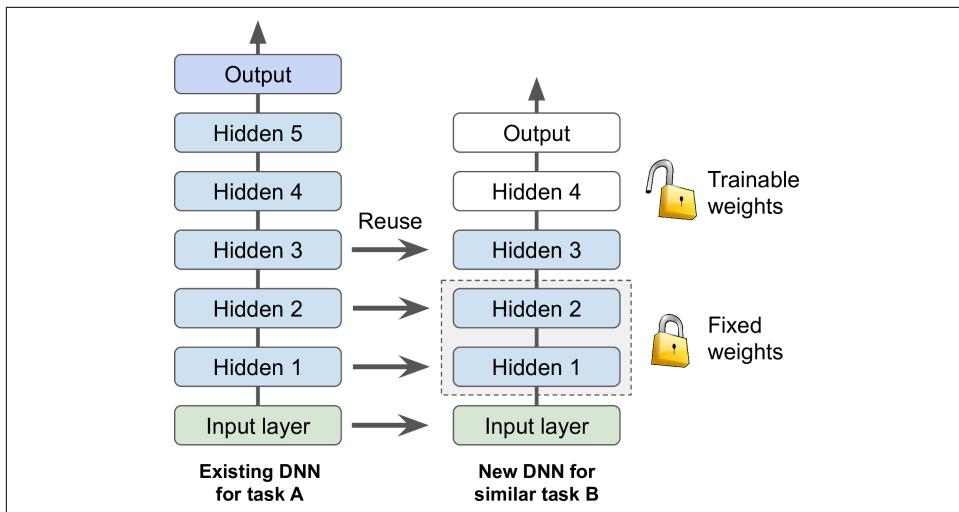


Figure 11-5. Reusing pretrained layers



If the input pictures of your new task don't have the same size as the ones used in the original task, you will usually have to add a preprocessing step to resize them to the size expected by the original model. More generally, transfer learning will work best when the inputs have similar low-level features.

The output layer of the original model should usually be replaced because it is most likely not useful at all for the new task, and it generally does not even have the right number of outputs.

Similarly, the upper hidden layers of the original model are less likely to be as useful as the lower layers, since the high-level features that are most useful for the new task may differ significantly from the ones that were most useful for the original task. You want to find the right number of layers to reuse.



The more similar the tasks are, the more layers you want to reuse (starting with the lower layers). For very similar tasks, try to keep all the hidden layers and just replace the output layer.

Try freezing all the reused layers first (i.e., make their weights non-trainable so that Gradient Descent won't modify them and they will remain fixed), then train your model and see how it performs. Then try unfreezing one or two of the top hidden layers to let backpropagation tweak them and see if performance improves. The more training data you have, the more layers you can unfreeze. It is also useful to reduce

the learning rate when you unfreeze reused layers: this will avoid wrecking their fine-tuned weights.

If you still cannot get good performance, and you have little training data, try dropping the top hidden layer(s) and freezing all the remaining hidden layers again. You can iterate until you find the right number of layers to reuse. If you have plenty of training data, you may try replacing the top hidden layers instead of dropping them, and even adding more hidden layers.

Transfer Learning with Keras

Let's look at an example. Suppose the Fashion MNIST dataset only contained eight classes—for example, all the classes except for sandal and shirt. Someone built and trained a Keras model on that set and got reasonably good performance (>90% accuracy). Let's call this model A. You now want to tackle a different task: you have images of T-shirts and pullovers, and you want to train a binary classifier: positive for T-shirts (and tops), negative for sandals. Your dataset is quite small; you only have 200 labeled images. When you train a new model for this task (let's call it model B) with the same architecture as model A, you get 91.85% test accuracy. While drinking your morning coffee, you realize that your task is quite similar to task A, so perhaps transfer learning can help? Let's find out!

First, you need to load model A and create a new model based on that model's layers. Let's reuse all the layers except for the output layer:

```
[...] # Assuming model A was already trained and saved to "my_model_A"
model_A = tf.keras.models.load_model("my_model_A")
model_B_on_A = tf.keras.Sequential(model_A.layers[:-1])
model_B_on_A.add(tf.keras.layers.Dense(1, activation="sigmoid"))
```

Note that `model_A` and `model_B_on_A` now share some layers. When you train `model_B_on_A`, it will also affect `model_A`. If you want to avoid that, you need to `clone` `model_A` before you reuse its layers. To do this, you clone model A's architecture with `clone_model()`, then copy its weights:

```
model_A_clone = tf.keras.models.clone_model(model_A)
model_A_clone.set_weights(model_A.get_weights())
```



`tf.keras.models.clone_model()` only clones the architecture, not the weights. If you don't copy them manually using `set_weights()`, they will be initialized randomly when the cloned model is first used.

Now you could train `model_B_on_A` for task B, but since the new output layer was initialized randomly it will make large errors (at least during the first few epochs), so there will be large error gradients that may wreck the reused weights. To avoid this,

one approach is to freeze the reused layers during the first few epochs, giving the new layer some time to learn reasonable weights. To do this, set every layer's `trainable` attribute to `False` and compile the model:

```
for layer in model_B_on_A.layers[:-1]:
    layer.trainable = False

optimizer = tf.keras.optimizers.SGD(learning_rate=0.001)
model_B_on_A.compile(loss="binary_crossentropy", optimizer=optimizer,
                      metrics=["accuracy"])
```



You must always compile your model after you freeze or unfreeze layers.

Now you can train the model for a few epochs, then unfreeze the reused layers (which requires compiling the model again) and continue training to fine-tune the reused layers for task B. After unfreezing the reused layers, it is usually a good idea to reduce the learning rate, once again to avoid damaging the reused weights.

```
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=4,
                            validation_data=(X_valid_B, y_valid_B))

for layer in model_B_on_A.layers[:-1]:
    layer.trainable = True

optimizer = tf.keras.optimizers.SGD(learning_rate=0.001)
model_B_on_A.compile(loss="binary_crossentropy", optimizer=optimizer,
                      metrics=["accuracy"])
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=16,
                            validation_data=(X_valid_B, y_valid_B))
```

So, what's the final verdict? Well, this model's test accuracy is 93.85%, up exactly two percentage points from 91.85%! This means that transfer learning reduced the error rate by almost 25%!

```
>>> model_B_on_A.evaluate(X_test_B, y_test_B)
[0.2546142041683197, 0.9384999871253967]
```

Are you convinced? You shouldn't be: I cheated! I tried many configurations until I found one that demonstrated a strong improvement. If you try to change the classes or the random seed, you will see that the improvement generally drops, or even vanishes or reverses. What I did is called "torturing the data until it confesses." When a paper just looks too positive, you should be suspicious: perhaps the flashy new technique does not actually help much (in fact, it may even degrade performance), but the authors tried many variants and reported only the best results (which may be due to sheer luck), without mentioning how many failures they encountered on the

way. Most of the time, this is not malicious at all, but it is part of the reason so many results in science can never be reproduced.

Why did I cheat? It turns out that transfer learning does not work very well with small dense networks, presumably because small networks learn few patterns, and dense networks learn very specific patterns, which are unlikely to be useful in other tasks. Transfer learning works best with deep convolutional neural networks, which tend to learn feature detectors that are much more general (especially in the lower layers). We will revisit transfer learning in [Chapter 14](#), using the techniques we just discussed (and this time there will be no cheating, I promise!).

Unsupervised Pretraining

Suppose you want to tackle a complex task for which you don't have much labeled training data, but unfortunately you cannot find a model trained on a similar task. Don't lose hope! First, you should try to gather more labeled training data, but if you can't, you may still be able to perform *unsupervised pretraining* (see [Figure 11-6](#)). Indeed, it is often cheap to gather unlabeled training examples, but expensive to label them. If you can gather plenty of unlabeled training data, you can try to use it to train an unsupervised model, such as an autoencoder or a generative adversarial network (see [Chapter 17](#)). Then you can reuse the lower layers of the autoencoder or the lower layers of the GAN's discriminator, add the output layer for your task on top, and fine-tune the final network using supervised learning (i.e., with the labeled training examples).

It is this technique that Geoffrey Hinton and his team used in 2006 and which led to the revival of neural networks and the success of Deep Learning. Until 2010, unsupervised pretraining—typically with restricted Boltzmann machines (RBMs; see the notebook at <https://homl.info/extr-anns>)—was the norm for deep nets, and only after the vanishing gradients problem was alleviated did it become much more common to train DNNs purely using supervised learning. Unsupervised pretraining (today typically using autoencoders or GANs rather than RBMs) is still a good option when you have a complex task to solve, no similar model you can reuse, and little labeled training data but plenty of unlabeled training data.

Note that in the early days of Deep Learning it was difficult to train deep models, so people would use a technique called *greedy layer-wise pretraining* (depicted in [Figure 11-6](#)). They would first train an unsupervised model with a single layer, typically an RBM, then they would freeze that layer and add another one on top of it, then train the model again (effectively just training the new layer), then freeze the new layer and add another layer on top of it, train the model again, and so on. Nowadays, things are much simpler: people generally train the full unsupervised model in one shot and use autoencoders or GANs rather than RBMs.

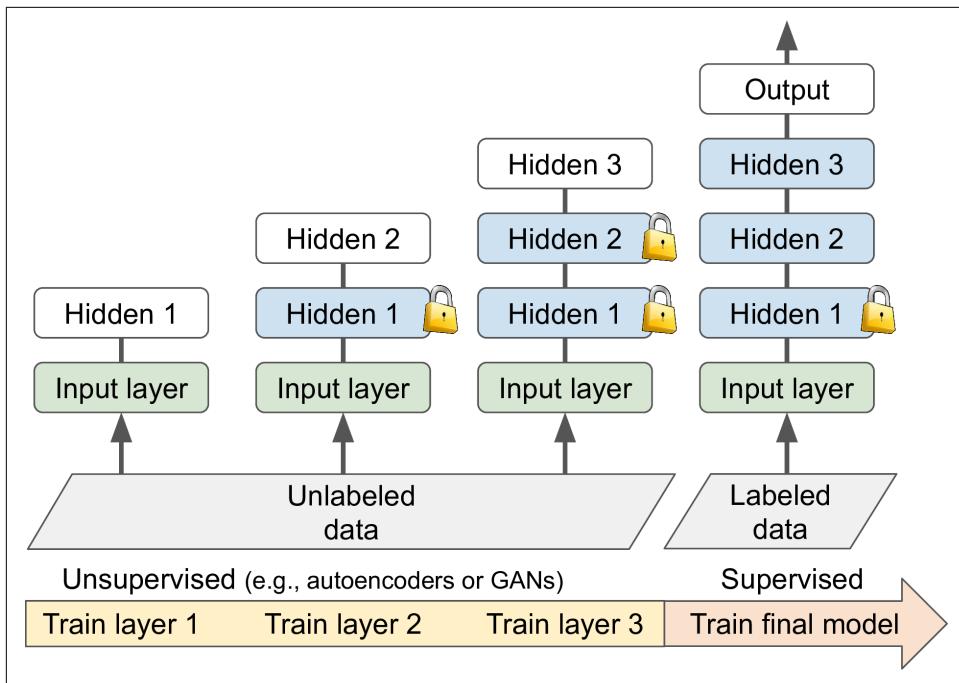


Figure 11-6. In unsupervised training, a model is trained on all data, including the unlabeled data, using an unsupervised learning technique, then it is fine-tuned for the final task on just the labeled data using a supervised learning technique; the unsupervised part may train one layer at a time as shown here, or it may train the full model directly

Pretraining on an Auxiliary Task

If you do not have much labeled training data, one last option is to train a first neural network on an auxiliary task for which you can easily obtain or generate labeled training data, then reuse the lower layers of that network for your actual task. The first neural network's lower layers will learn feature detectors that will likely be reusable by the second neural network.

For example, if you want to build a system to recognize faces, you may only have a few pictures of each individual—clearly not enough to train a good classifier. Gathering hundreds of pictures of each person would not be practical. You could, however, gather a lot of pictures of random people on the web and train a first neural network to detect whether or not two different pictures feature the same person. Such a network would learn good feature detectors for faces, so reusing its lower layers would allow you to train a good face classifier that uses little training data.

For *natural language processing* (NLP) applications, you can download a corpus of millions of text documents and automatically generate labeled data from it. For

example, you could randomly mask out some words and train a model to predict what the missing words are (e.g., it should predict that the missing word in the sentence “What ___ you saying?” is probably “are” or “were”). If you can train a model to reach good performance on this task, then it will already know quite a lot about language, and you can certainly reuse it for your actual task and fine-tune it on your labeled data (we will discuss more pretraining tasks in [Chapter 15](#)).



Self-supervised learning is when you automatically generate the labels from the data itself, as in the text-masking example, then you train a model on the resulting “labeled” dataset using supervised learning techniques.

Faster Optimizers

Training a very large deep neural network can be painfully slow. So far we have seen four ways to speed up training (and reach a better solution): applying a good initialization strategy for the connection weights, using a good activation function, using Batch Normalization, and reusing parts of a pretrained network (possibly built on an auxiliary task or using unsupervised learning). Another huge speed boost comes from using a faster optimizer than the regular Gradient Descent optimizer. In this section we will present the most popular optimizers: momentum optimization, Nesterov Accelerated Gradient, AdaGrad, RMSProp, and finally Adam optimization and its variants.

Momentum Optimization

Imagine a bowling ball rolling down a gentle slope on a smooth surface: it will start out slowly, but it will quickly pick up momentum until it eventually reaches terminal velocity (if there is some friction or air resistance). This is the core idea of *momentum optimization*, proposed by Boris Polyak in 1964.¹⁵ In contrast, regular Gradient Descent will take small steps when the slope is gentle, and big steps when the slope is steep, but it will never pick up speed. As a result, regular Gradient Descent is generally much slower to reach the minimum than momentum optimization.

Recall that Gradient Descent updates the weights θ by directly subtracting the gradient of the cost function $J(\theta)$ with regard to the weights ($\nabla_{\theta}J(\theta)$) multiplied by the learning rate η . The equation is: $\theta \leftarrow \theta - \eta \nabla_{\theta}J(\theta)$. It does not care about what the earlier gradients were. If the local gradient is tiny, it goes very slowly.

¹⁵ Boris T. Polyak, “Some Methods of Speeding Up the Convergence of Iteration Methods,” *USSR Computational Mathematics and Mathematical Physics* 4, no. 5 (1964): 1–17.

Momentum optimization cares a great deal about what previous gradients were: at each iteration, it subtracts the local gradient from the *momentum vector* \mathbf{m} (multiplied by the learning rate η), and it updates the weights by adding this momentum vector (see [Equation 11-5](#)). In other words, the gradient is used as an acceleration, not as a speed. To simulate some sort of friction mechanism and prevent the momentum from growing too large, the algorithm introduces a new hyperparameter β , called the *momentum*, which must be set between 0 (high friction) and 1 (no friction). A typical momentum value is 0.9.

Equation 11-5. Momentum algorithm

1. $\mathbf{m} \leftarrow \beta\mathbf{m} - \eta\nabla_{\theta}J(\theta)$
2. $\theta \leftarrow \theta + \mathbf{m}$

You can verify that if the gradient remains constant, the terminal velocity (i.e., the maximum size of the weight updates) is equal to that gradient multiplied by the learning rate η multiplied by $1 / (1 - \beta)$ (ignoring the sign). For example, if $\beta = 0.9$, then the terminal velocity is equal to 10 times the gradient times the learning rate, so momentum optimization ends up going 10 times faster than Gradient Descent! This allows momentum optimization to escape from plateaus much faster than Gradient Descent. We saw in [Chapter 4](#) that when the inputs have very different scales, the cost function will look like an elongated bowl (see [Figure 4-7](#)). Gradient Descent goes down the steep slope quite fast, but then it takes a very long time to go down the valley. In contrast, momentum optimization will roll down the valley faster and faster until it reaches the bottom (the optimum). In deep neural networks that don't use Batch Normalization, the upper layers will often end up having inputs with very different scales, so using momentum optimization helps a lot. It can also help roll past local optima.



Due to the momentum, the optimizer may overshoot a bit, then come back, overshoot again, and oscillate like this many times before stabilizing at the minimum. This is one of the reasons it's good to have a bit of friction in the system: it gets rid of these oscillations and thus speeds up convergence.

Implementing momentum optimization in Keras is a no-brainer: just use the SGD optimizer and set its `momentum` hyperparameter, then lie back and profit!

```
optimizer = tf.keras.optimizers.SGD(learning_rate=0.001, momentum=0.9)
```

The one drawback of momentum optimization is that it adds yet another hyperparameter to tune. However, the momentum value of 0.9 usually works well in practice and almost always goes faster than regular Gradient Descent.

Nesterov Accelerated Gradient

One small variant to momentum optimization, proposed by [Yurii Nesterov in 1983](#),¹⁶ is almost always faster than regular momentum optimization. The *Nesterov Accelerated Gradient* (NAG) method, also known as *Nesterov momentum optimization*, measures the gradient of the cost function not at the local position θ but slightly ahead in the direction of the momentum, at $\theta + \beta\mathbf{m}$ (see [Equation 11-6](#)).

Equation 11-6. Nesterov Accelerated Gradient algorithm

1. $\mathbf{m} \leftarrow \beta\mathbf{m} - \eta\nabla_{\theta}J(\theta + \beta\mathbf{m})$
2. $\theta \leftarrow \theta + \mathbf{m}$

This small tweak works because in general the momentum vector will be pointing in the right direction (i.e., toward the optimum), so it will be slightly more accurate to use the gradient measured a bit farther in that direction rather than the gradient at the original position, as you can see in [Figure 11-7](#) (where ∇_1 represents the gradient of the cost function measured at the starting point θ , and ∇_2 represents the gradient at the point located at $\theta + \beta\mathbf{m}$).

As you can see, the Nesterov update ends up closer to the optimum. After a while, these small improvements add up and NAG ends up being significantly faster than regular momentum optimization. Moreover, note that when the momentum pushes the weights across a valley, ∇_1 continues to push farther across the valley, while ∇_2 pushes back toward the bottom of the valley. This helps reduce oscillations and thus NAG converges faster.

NAG is generally faster than regular momentum optimization. To use it, simply set `nesterov=True` when creating the SGD optimizer:

```
optimizer = tf.keras.optimizers.SGD(learning_rate=0.001, momentum=0.9,  
                                   nesterov=True)
```

¹⁶ Yurii Nesterov, “A Method for Unconstrained Convex Minimization Problem with the Rate of Convergence $O(1/k^2)$,” *Doklady AN USSR* 269 (1983): 543–547.

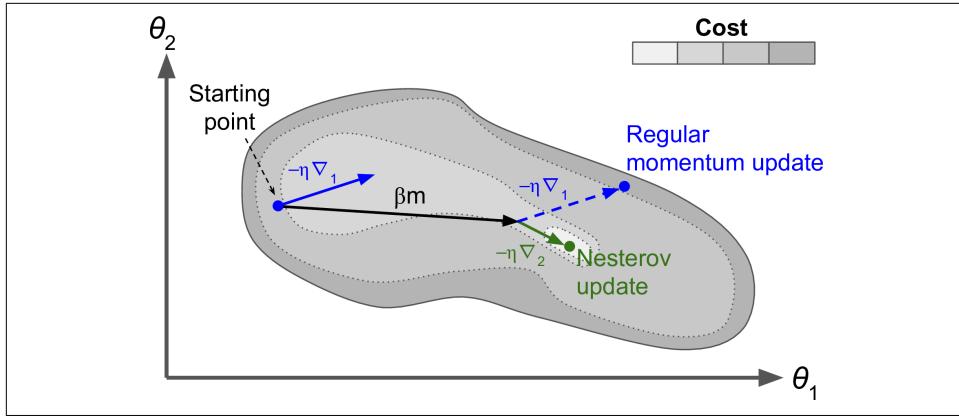


Figure 11-7. Regular versus Nesterov momentum optimization: the former applies the gradients computed before the momentum step, while the latter applies the gradients computed after

AdaGrad

Consider the elongated bowl problem again: Gradient Descent starts by quickly going down the steepest slope, which does not point straight toward the global optimum, then it very slowly goes down to the bottom of the valley. It would be nice if the algorithm could correct its direction earlier to point a bit more toward the global optimum. The *AdaGrad* algorithm¹⁷ achieves this correction by scaling down the gradient vector along the steepest dimensions (see Equation 11-7).

Equation 11-7. AdaGrad algorithm

1. $s \leftarrow s + \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$
2. $\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{s + \varepsilon}$

The first step accumulates the square of the gradients into the vector s (recall that the \otimes symbol represents the element-wise multiplication). This vectorized form is equivalent to computing $s_i \leftarrow s_i + (\partial J(\theta) / \partial \theta_i)^2$ for each element s_i of the vector s ; in other words, each s_i accumulates the squares of the partial derivative of the cost function with regard to parameter θ_i . If the cost function is steep along the i^{th} dimension, then s_i will get larger and larger at each iteration.

The second step is almost identical to Gradient Descent, but with one big difference: the gradient vector is scaled down by a factor of $\sqrt{s + \varepsilon}$ (the \oslash symbol represents

¹⁷ John Duchi et al., “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization,” *Journal of Machine Learning Research* 12 (2011): 2121–2159.

the element-wise division, and ε is a smoothing term to avoid division by zero, typically set to 10^{-10}). This vectorized form is equivalent to simultaneously computing $\theta_i \leftarrow \theta_i - \eta \frac{\partial J(\theta)}{\partial \theta_i} / \sqrt{s_i + \varepsilon}$ for all parameters θ_i .

In short, this algorithm decays the learning rate, but it does so faster for steep dimensions than for dimensions with gentler slopes. This is called an *adaptive learning rate*. It helps point the resulting updates more directly toward the global optimum (see [Figure 11-8](#)). One additional benefit is that it requires much less tuning of the learning rate hyperparameter η .

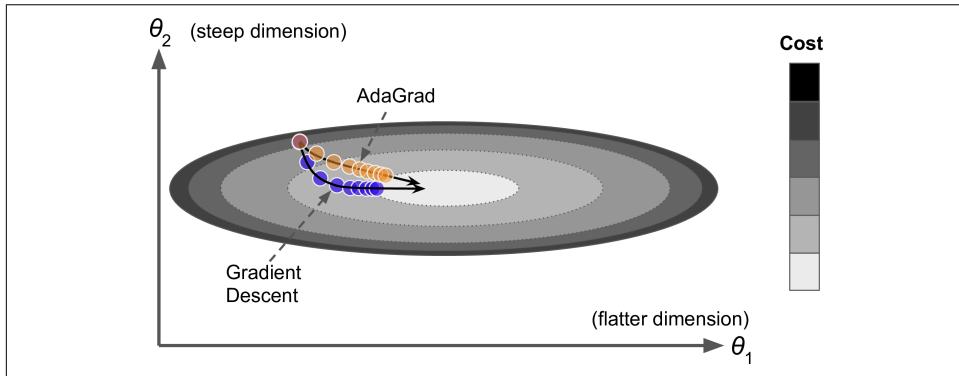


Figure 11-8. AdaGrad versus Gradient Descent: the former can correct its direction earlier to point to the optimum

AdaGrad frequently performs well for simple quadratic problems, but it often stops too early when training neural networks. The learning rate gets scaled down so much that the algorithm ends up stopping entirely before reaching the global optimum. So even though Keras has an `Adagrad` optimizer, you should not use it to train deep neural networks (it may be efficient for simpler tasks such as Linear Regression, though). Still, understanding AdaGrad is helpful to grasp the other adaptive learning rate optimizers.

RMSProp

As we've seen, AdaGrad runs the risk of slowing down a bit too fast and never converging to the global optimum. The *RMSProp* algorithm¹⁸ fixes this by accumulating only the gradients from the most recent iterations, as opposed to all the gradients

¹⁸ This algorithm was created by Geoffrey Hinton and Tijmen Tieleman in 2012 and presented by Geoffrey Hinton in his Coursera class on neural networks (slides: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides.pdf; video: <https://www.youtube.com/watch?v=QXvKdWkqzUw>). Amusingly, since the authors did not write a paper to describe the algorithm, researchers often cite "slide 29 in lecture 6e" in their papers.

since the beginning of training. It does so by using exponential decay in the first step (see [Equation 11-8](#)).

Equation 11-8. RMSProp algorithm

1. $s \leftarrow \rho s + (1 - \rho) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$
2. $\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{s + \varepsilon}$

The decay rate ρ is typically set to 0.9.¹⁹ Yes, it is once again a new hyperparameter, but this default value often works well, so you may not need to tune it at all.

As you might expect, Keras has an RMSprop optimizer:

```
optimizer = tf.keras.optimizers.RMSprop(learning_rate=0.001, rho=0.9)
```

Except on very simple problems, this optimizer almost always performs much better than AdaGrad. In fact, it was the preferred optimization algorithm of many researchers until Adam optimization came around.

Adam Optimization

[Adam](#),²⁰ which stands for *adaptive moment estimation*, combines the ideas of momentum optimization and RMSProp: just like momentum optimization, it keeps track of an exponentially decaying average of past gradients; and just like RMSProp, it keeps track of an exponentially decaying average of past squared gradients (see [Equation 11-9](#)). These are estimations of the mean and (uncentered) variance of the gradients.

¹⁹ ρ is the greek letter rho.

²⁰ Diederik P. Kingma and Jimmy Ba, “Adam: A Method for Stochastic Optimization,” arXiv preprint arXiv:1412.6980 (2014).

The mean is often called the *first moment* while the variance is often called the *second moment*, hence the name of the algorithm.

Equation 11-9. Adam algorithm

1. $\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1 - \beta_1) \nabla_{\theta} J(\theta)$
2. $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$
3. $\widehat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1 - \beta_1^t}$
4. $\widehat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \beta_2^t}$
5. $\theta \leftarrow \theta + \eta \widehat{\mathbf{m}} \oslash \sqrt{\widehat{\mathbf{s}}} + \varepsilon$

In this equation, t represents the iteration number (starting at 1).

If you just look at steps 1, 2, and 5, you will notice Adam's close similarity to both momentum optimization and RMSProp: β_1 corresponds to β in momentum optimization, and β_2 corresponds to α in RMSProp. *The only difference is that step 1 computes an exponentially decaying average rather than an exponentially decaying sum, but these are actually equivalent except for a constant factor (the decaying average is just $1 - \beta_1$ times the decaying sum).* Steps 3 and 4 are somewhat of a technical detail: since \mathbf{m} and \mathbf{s} are initialized at 0, they will be biased toward 0 at the beginning of training, so these two steps will help boost \mathbf{m} and \mathbf{s} at the beginning of training.

The momentum decay hyperparameter β_1 is typically initialized to 0.9, while the scaling decay hyperparameter β_2 is often initialized to 0.999. As earlier, the smoothing term ε is usually initialized to a tiny number such as 10^{-7} . These are the default values for the `Adam` class. Here is how to create an Adam optimizer using Keras:

```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9,
                                    beta_2=0.999)
```

Since Adam is an adaptive learning rate algorithm, like AdaGrad and RMSProp, it requires less tuning of the learning rate hyperparameter η . You can often use the default value $\eta = 0.001$, making Adam even easier to use than Gradient Descent.



If you are starting to feel overwhelmed by all these different techniques and are wondering how to choose the right ones for your task, don't worry: some practical guidelines are provided at the end of this chapter.

Finally, three variants of Adam are worth mentioning: AdaMax, Nadam, and AdamW.

AdaMax

The Adam paper also introduced AdaMax. Notice that in step 2 of [Equation 11-9](#), Adam accumulates the squares of the gradients in s (with a greater weight for more recent gradients). In step 5, if we ignore ϵ and steps 3 and 4 (which are technical details anyway), Adam scales down the parameter updates by the square root of s . In short, Adam scales down the parameter updates by the ℓ_2 norm of the time-decayed gradients (recall that the ℓ_2 norm is the square root of the sum of squares).

AdaMax replaces the ℓ_2 norm with the ℓ_∞ norm (a fancy way of saying the max). Specifically, it replaces step 2 in [Equation 11-9](#) with $s \leftarrow \max(\beta_2 s, \text{abs}(\nabla_{\theta} J(\theta)))$, it drops step 4, and in step 5 it scales down the gradient updates by a factor of s , which is the max of the absolute value of the time-decayed gradients.

In practice, this can make AdaMax more stable than Adam, but it really depends on the dataset, and in general Adam performs better. So, this is just one more optimizer you can try if you experience problems with Adam on some task.

Nadam

Nadam optimization is Adam optimization plus the Nesterov trick, so it will often converge slightly faster than Adam. In [his report introducing this technique](#),²¹ the researcher Timothy Dozat compares many different optimizers on various tasks and finds that Nadam generally outperforms Adam but is sometimes outperformed by RMSProp.

AdamW

[AdamW](#)²² is a variant of Adam that integrates a regularization technique called *weight decay*. Weight decay reduces the size of the model's weights at each training iteration by multiplying them by a decay factor such as 0.99. This may remind you of ℓ_2 regularization (introduced in [Chapter 4](#)), which also aims to keep the weights small, and indeed it can be shown mathematically that ℓ_2 regularization is equivalent to weight decay when using SGD. However, when using Adam or its variants, ℓ_2 regularization and weight decay are *not* equivalent: in practice, combining Adam with ℓ_2 regularization results in models that often don't generalize as well as those produced by SGD. AdamW fixes this issue by properly combining Adam with weight decay.

²¹ Timothy Dozat, "Incorporating Nesterov Momentum into Adam" (2016).

²² Ilya Loshchilov, and Frank Hutter. "Decoupled weight decay regularization." arXiv preprint arXiv:1711.05101 (2017).



Adaptive optimization methods (including RMSProp, Adam, AdaMax, Nadam, and AdamW optimization) are often great, converging fast to a good solution. However, a [2017 paper²³](#) by Ashia C. Wilson et al. showed that they can lead to solutions that generalize poorly on some datasets. So when you are disappointed by your model's performance, try using Nesterov Accelerated Gradient instead: your dataset may just be allergic to adaptive gradients. Also check out the latest research, because it's moving fast.

To use Nadam or AdaMax in Keras, just replace `tf.keras.optimizers.Adam` with `tf.keras.optimizers.Nadam` or `tf.keras.optimizers.Adamax`. AdamW is not included in Keras yet (as of TF 2.9.0), but it is being added so it will likely be available by the time you read this (please check the documentation). In the meantime, AdamW is available in the TensorFlow Add-Ons library. If you followed the installation instructions at <https://hml.info/install> to run everything locally, then you already have this library installed, but if you are using Google Colab, then you need to run `%pip install -q -U tensorflow_addons`. After that, you can use `tfa.optimizers.AdamW` optimizer just like the `tf.keras.optimizers.Adam` optimizer, with an extra `weight_decay` hyperparameter (which needs to be tuned):

```
import tensorflow_addons as tfa

optimizer = tfa.optimizers.AdamW(weight_decay=1e-5, learning_rate=0.001,
                                 beta_1=0.9, beta_2=0.999)
```

All the optimization techniques discussed so far only rely on the *first-order partial derivatives (Jacobians)*. The optimization literature also contains amazing algorithms based on the *second-order partial derivatives* (the *Hessians*, which are the partial derivatives of the Jacobians). Unfortunately, these algorithms are very hard to apply to deep neural networks because there are n^2 Hessians per output (where n is the number of parameters), as opposed to just n Jacobians per output. Since DNNs typically have tens of thousands of parameters or more, the second-order optimization algorithms often don't even fit in memory, and even when they do, computing the Hessians is just too slow.

Training Sparse Models

All the optimization algorithms we just discussed produce dense models, meaning that most parameters will be nonzero. If you need a blazingly fast model at runtime, or if you need it to take up less memory, you may prefer to end up with a sparse model instead.

²³ Ashia C. Wilson et al., “The Marginal Value of Adaptive Gradient Methods in Machine Learning,” *Advances in Neural Information Processing Systems* 30 (2017): 4148–4158.

One way to achieve this is to train the model as usual, then get rid of the tiny weights (set them to zero). However, this will typically not lead to a very sparse model, and it may degrade the model’s performance.

A better option is to apply strong ℓ_1 regularization during training (we will see how later in this chapter), as it pushes the optimizer to zero out as many weights as it can (as discussed in “[Lasso Regression](#)” on page 161 in Chapter 4).

If these techniques remain insufficient, check out the [TensorFlow Model Optimization Toolkit \(TF-MOT\)](#), which provides a pruning API capable of iteratively removing connections during training based on their magnitude.

Table 11-2 compares all the optimizers we’ve discussed so far (* is bad, ** is average, and *** is good).

Table 11-2. Optimizer comparison

Class	Convergence speed	Convergence quality
SGD	*	***
SGD(momentum=...)	**	***
SGD(momentum=..., nesterov=True)	**	***
Adagrad	***	* (stops too early)
RMSprop	***	** or ***
Adam	***	** or ***
AdaMax	***	** or ***
Nadam	***	** or ***
AdamW	***	** or ***

Learning Rate Scheduling

Finding a good learning rate is very important. If you set it much too high, training may diverge (as we discussed in “[Gradient Descent](#)” on page 140). If you set it too low, training will eventually converge to the optimum, but it will take a very long time. If you set it slightly too high, it will make progress very quickly at first, but it will end up dancing around the optimum, never really settling down. If you have a limited computing budget, you may have to interrupt training before it has converged properly, yielding a suboptimal solution (see [Figure 11-9](#)).

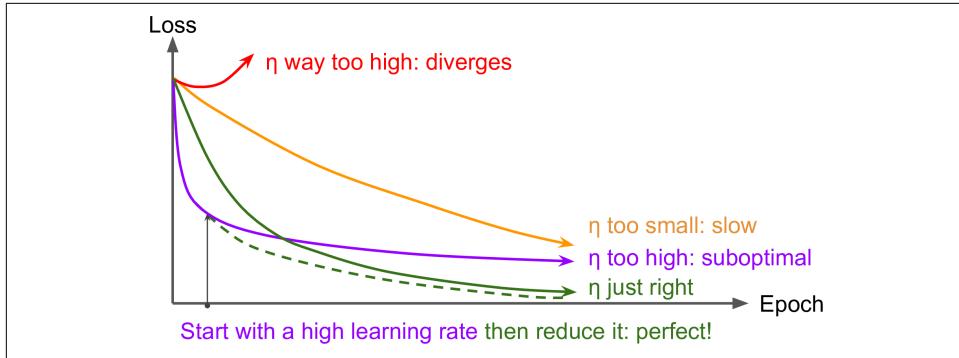


Figure 11-9. Learning curves for various learning rates η

As we discussed in [Chapter 10](#), you can find a good learning rate by training the model for a few hundred iterations, exponentially increasing the learning rate from a very small value to a very large value, and then looking at the learning curve and picking a learning rate slightly lower than the one at which the learning curve starts shooting back up. You can then reinitialize your model and train it with that learning rate.

But you can do better than a constant learning rate: if you start with a large learning rate and then reduce it once training stops making fast progress, you can reach a good solution faster than with the optimal constant learning rate. There are many different strategies to reduce the learning rate during training. It can also be beneficial to start with a low learning rate, increase it, then drop it again. These strategies are called *learning schedules* (we briefly introduced this concept in [Chapter 4](#)). These are the most commonly used learning schedules:

Power scheduling

Set the learning rate to a function of the iteration number t : $\eta(t) = \eta_0 / (1 + t/s)^c$. The initial learning rate η_0 , the power c (typically set to 1), and the steps s are hyperparameters. The learning rate drops at each step. After s steps, it is down to $\eta_0 / 2$. After s more steps, it is down to $\eta_0 / 3$, then it goes down to $\eta_0 / 4$, then $\eta_0 / 5$, and so on. As you can see, this schedule first drops quickly, then more and more slowly. Of course, power scheduling requires tuning η_0 and s (and possibly c).

Exponential scheduling

Set the learning rate to $\eta(t) = \eta_0 0.1^{t/s}$. The learning rate will gradually drop by a factor of 10 every s steps. While power scheduling reduces the learning rate more and more slowly, exponential scheduling keeps slashing it by a factor of 10 every s steps.

Piecewise constant scheduling

Use a constant learning rate for a number of epochs (e.g., $\eta_0 = 0.1$ for 5 epochs), then a smaller learning rate for another number of epochs (e.g., $\eta_1 = 0.001$ for 50 epochs), and so on. Although this solution can work very well, it requires fiddling around to figure out the right sequence of learning rates and how long to use each of them.

Performance scheduling

Measure the validation error every N steps (just like for early stopping), and reduce the learning rate by a factor of λ when the error stops dropping.

1cycle scheduling

1cycle was introduced in a [2018 paper](#) by Leslie Smith.²⁴ Contrary to the other approaches, it starts by increasing the initial learning rate η_0 , growing linearly up to η_1 halfway through training. Then it decreases the learning rate linearly down to η_0 again during the second half of training, finishing the last few epochs by dropping the rate down by several orders of magnitude (still linearly). The maximum learning rate η_1 is chosen using the same approach we used to find the optimal learning rate, and the initial learning rate η_0 is usually 10 times lower. When using a momentum, we start with a high momentum first (e.g., 0.95), then drop it down to a lower momentum during the first half of training (e.g., down to 0.85, linearly), and then bring it back up to the maximum value (e.g., 0.95) during the second half of training, finishing the last few epochs with that maximum value. Smith did many experiments showing that this approach was often able to speed up training considerably and reach better performance. For example, on the popular CIFAR10 image dataset, this approach reached 91.9% validation accuracy in just 100 epochs, instead of 90.3% accuracy in 800 epochs through a standard approach (with the same neural network architecture). This feat was dubbed *super-convergence*.

A [2013 paper](#) by Andrew Senior et al.²⁵ compared the performance of some of the most popular learning schedules when using momentum optimization to train deep neural networks for speech recognition. The authors concluded that, in this setting, both performance scheduling and exponential scheduling performed well. They favored exponential scheduling because it was easy to tune and it converged slightly faster to the optimal solution. They also mentioned that it was easier to implement than performance scheduling, but in Keras both options are easy. That said, the *1cycle* approach seems to perform even better.

²⁴ Leslie N. Smith, “A Disciplined Approach to Neural Network Hyper-Parameters: Part 1—Learning Rate, Batch Size, Momentum, and Weight Decay,” arXiv preprint arXiv:1803.09820 (2018).

²⁵ Andrew Senior et al., “An Empirical Study of Learning Rates in Deep Neural Networks for Speech Recognition,” *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing* (2013): 6724–6728.

Implementing power scheduling in Keras is the easiest option: just set the `decay` hyperparameter when creating an optimizer:

```
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01, decay=1e-4)
```

The `decay` is the inverse of s (the number of steps it takes to divide the learning rate by one more unit), and Keras assumes that c is equal to 1.

Exponential scheduling and piecewise scheduling are quite simple too. You first need to define a function that takes the current epoch and returns the learning rate. For example, let's implement exponential scheduling:

```
def exponential_decay_fn(epoch):
    return 0.01 * 0.1 ** (epoch / 20)
```

If you do not want to hardcode η_0 and s , you can create a function that returns a configured function:

```
def exponential_decay(lr0, s):
    def exponential_decay_fn(epoch):
        return lr0 * 0.1 ** (epoch / s)
    return exponential_decay_fn

exponential_decay_fn = exponential_decay(lr0=0.01, s=20)
```

Next, create a `LearningRateScheduler` callback, giving it the schedule function, and pass this callback to the `fit()` method:

```
lr_scheduler = tf.keras.callbacks.LearningRateScheduler(exponential_decay_fn)
history = model.fit(X_train, y_train, [...], callbacks=[lr_scheduler])
```

The `LearningRateScheduler` will update the optimizer's `learning_rate` attribute at the beginning of each epoch. Updating the learning rate once per epoch is usually enough, but if you want it to be updated more often, for example at every step, you can always write your own callback (see the “Exponential Scheduling” section of the notebook for an example). Updating the learning rate at every step may help if there are many steps per epoch. Alternatively, you can use the `tf.keras.optimizers.schedules` approach, described shortly.



After training, `history.history["lr"]` gives you access to the list of learning rates used during training.

The `schedule` function can optionally take the current learning rate as a second argument. For example, the following `schedule` function multiplies the previous learning rate by $0.1^{1/20}$, which results in the same exponential decay (except the decay now starts at the beginning of epoch 0 instead of 1):

```
def exponential_decay_fn(epoch, lr):
    return lr * 0.1 ** (1 / 20)
```

This implementation relies on the optimizer's initial learning rate (contrary to the previous implementation), so make sure to set it appropriately.

When you save a model, the optimizer and its learning rate get saved along with it. This means that with this new schedule function, you could just load a trained model and continue training where it left off, no problem. Things are not so simple if your schedule function uses the epoch argument, however: the epoch does not get saved, and it gets reset to 0 every time you call the `fit()` method. If you were to continue training a model where it left off, this could lead to a very large learning rate, which would likely damage your model's weights. One solution is to manually set the `fit()` method's `initial_epoch` argument so the epoch starts at the right value.

For piecewise constant scheduling, you can use a schedule function like the following one (as earlier, you can define a more general function if you want; see the “Piecewise Constant Scheduling” section of the notebook for an example), then create a `LearningRateScheduler` callback with this function and pass it to the `fit()` method, just like we did for exponential scheduling:

```
def piecewise_constant_fn(epoch):
    if epoch < 5:
        return 0.01
    elif epoch < 15:
        return 0.005
    else:
        return 0.001
```

For performance scheduling, use the `ReduceLROnPlateau` callback. For example, if you pass the following callback to the `fit()` method, it will multiply the learning rate by 0.5 whenever the best validation loss does not improve for five consecutive epochs (other options are available; please check the documentation for more details):

```
lr_scheduler = tf.keras.callbacks.ReduceLROnPlateau(factor=0.5, patience=5)
history = model.fit(X_train, y_train, [...], callbacks=[lr_scheduler])
```

Lastly, Keras offers an alternative way to implement learning rate scheduling: you can define a scheduled learning rate using one of the classes available in `tf.keras.optimizers.schedules`, then pass it to any optimizer. This approach updates the learning rate at each step rather than at each epoch. For example, here is how to implement the same exponential schedule as the `exponential_decay_fn()` function we defined earlier:

```
import math

batch_size = 32
n_epochs = 25
n_steps = n_epochs * math.ceil(len(X_train) / batch_size)
```

```
scheduled_learning_rate = tf.keras.optimizers.schedules.ExponentialDecay(  
    initial_learning_rate=0.01, decay_steps=n_steps, decay_rate=0.1)  
optimizer = tf.keras.optimizers.SGD(learning_rate=scheduled_learning_rate)
```

This is nice and simple, plus when you save the model, the learning rate and its schedule (including its state) get saved as well.

As for 1cycle, Keras does not support it, but it's possible to implement it in less than 30 lines of code by creating a custom callback that modifies the learning rate at each iteration. To update the optimizer's learning rate from within the callback's `on_batch_begin()` method, you need to call `tf.keras.backend.set_value(self.model.optimizer.learning_rate, new_learning_rate)`. See the “1Cycle scheduling” section of the notebook for an example.

To sum up, exponential decay, performance scheduling, and 1cycle can considerably speed up convergence, so give them a try!

Avoiding Overfitting Through Regularization

With four parameters I can fit an elephant and with five I can make him wiggle his trunk.

—John von Neumann, cited by Enrico Fermi in *Nature* 427

With thousands of parameters, you can fit the whole zoo. Deep neural networks typically have tens of thousands of parameters, sometimes even millions. This gives them an incredible amount of freedom and means they can fit a huge variety of complex datasets. But this great flexibility also makes the network prone to overfitting the training set. Regularization is often needed to prevent this.

We already implemented one of the best regularization techniques in [Chapter 10](#): early stopping. Moreover, even though Batch Normalization was designed to solve the unstable gradients problems, it also acts like a pretty good regularizer. In this section we will examine other popular regularization techniques for neural networks: ℓ_1 and ℓ_2 regularization, dropout, and max-norm regularization.

ℓ_1 and ℓ_2 Regularization

Just like you did in [Chapter 4](#) for simple linear models, you can use ℓ_2 regularization to constrain a neural network's connection weights, and/or ℓ_1 regularization if you want a sparse model (with many weights equal to 0). Here is how to apply ℓ_2 regularization to a Keras layer's connection weights, using a regularization factor of 0.01:

```
layer = tf.keras.layers.Dense(100, activation="relu",  
    kernel_initializer="he_normal",  
    kernel_regularizer=tf.keras.regularizers.l2(0.01))
```

The `l2()` function returns a regularizer that will be called at each step during training to compute the regularization loss. This is then added to the final loss. As you might expect, you can just use `tf.keras.regularizers.l1()` if you want ℓ_1 regularization; if you want both ℓ_1 and ℓ_2 regularization, use `tf.keras.regularizers.l1_l2()` (specifying both regularization factors).

Since you will typically want to apply the same regularizer to all layers in your network, as well as using the same activation function and the same initialization strategy in all hidden layers, you may find yourself repeating the same arguments. This makes the code ugly and error-prone. To avoid this, you can try refactoring your code to use loops. Another option is to use Python's `functools.partial()` function, which lets you create a thin wrapper for any callable, with some default argument values:

```
from functools import partial

RegularizedDense = partial(tf.keras.layers.Dense,
                          activation="relu",
                          kernel_initializer="he_normal",
                          kernel_regularizer=tf.keras.regularizers.l2(0.01))

model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    RegularizedDense(100),
    RegularizedDense(100),
    RegularizedDense(10, activation="softmax")
])
```



As we saw earlier, ℓ_2 regularization is fine when using SGD, momentum optimization and Nesterov momentum optimization, but not with Adam and its variants. If you want to use Adam with weight decay, then do not use ℓ_2 regularization: use AdamW instead.

Dropout

Dropout is one of the most popular regularization techniques for deep neural networks. It was proposed in a paper²⁶ by Geoffrey Hinton in 2012 and further detailed in a 2014 paper²⁷ by Nitish Srivastava et al., and it has proven to be highly successful: many state-of-the-art neural networks use dropout, as it gives them a 1–2% accuracy boost. This may not sound like a lot, but when a model already has 95% accuracy,

²⁶ Geoffrey E. Hinton et al., “Improving Neural Networks by Preventing Co-Adaptation of Feature Detectors,” arXiv preprint arXiv:1207.0580 (2012).

²⁷ Nitish Srivastava et al., “Dropout: A Simple Way to Prevent Neural Networks from Overfitting,” *Journal of Machine Learning Research* 15 (2014): 1929–1958.

getting a 2% accuracy boost means dropping the error rate by almost 40% (going from 5% error to roughly 3%).

It is a fairly simple algorithm: at every training step, every neuron (including the input neurons, but always excluding the output neurons) has a probability p of being temporarily “dropped out,” meaning it will be entirely ignored during this training step, but it may be active during the next step (see [Figure 11-10](#)). The hyperparameter p is called the *dropout rate*, and it is typically set between 10% and 50%: closer to 20–30% in recurrent neural nets (see [Chapter 15](#)), and closer to 40–50% in convolutional neural networks (see [Chapter 14](#)). After training, neurons don’t get dropped anymore. And that’s all (except for a technical detail we will discuss momentarily).

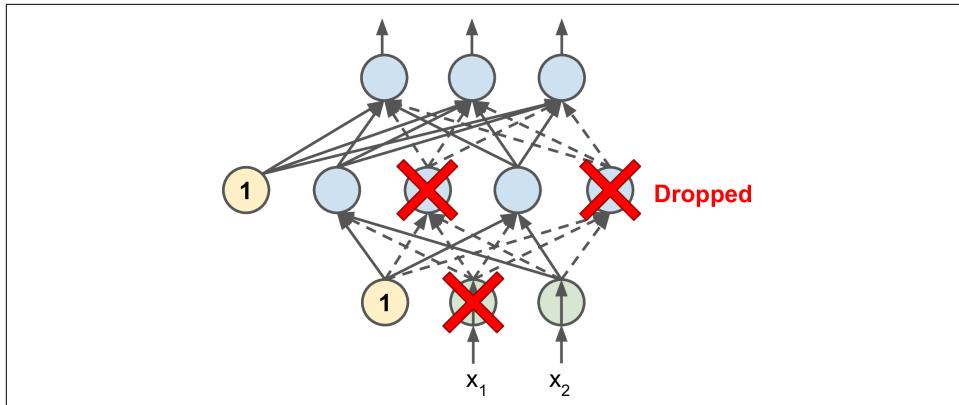


Figure 11-10. With dropout regularization, at each training iteration a random subset of all neurons in one or more layers—except the output layer—are “dropped out”; these neurons output 0 at this iteration (represented by the dashed arrows)

It’s surprising at first that this destructive technique works at all. Would a company perform better if its employees were told to toss a coin every morning to decide whether or not to go to work? Well, who knows; perhaps it would! The company would be forced to adapt its organization; it could not rely on any single person to work the coffee machine or perform any other critical tasks, so this expertise would have to be spread across several people. Employees would have to learn to cooperate with many of their coworkers, not just a handful of them. The company would become much more resilient. If one person quit, it wouldn’t make much of a difference. It’s unclear whether this idea would actually work for companies, but it certainly does for neural networks. Neurons trained with dropout cannot co-adapt with their neighboring neurons; they have to be as useful as possible on their own. They also cannot rely excessively on just a few input neurons; they must pay attention to each of their input neurons. They end up being less sensitive to slight changes in the inputs. In the end, you get a more robust network that generalizes better.

Another way to understand the power of dropout is to realize that a unique neural network is generated at each training step. Since each neuron can be either present or absent, there are a total of 2^N possible networks (where N is the total number of droppable neurons). This is such a huge number that it is virtually impossible for the same neural network to be sampled twice. Once you have run 10,000 training steps, you have essentially trained 10,000 different neural networks, each with just one training instance. These neural networks are obviously not independent because they share many of their weights, but they are nevertheless all different. The resulting neural network can be seen as an averaging ensemble of all these smaller neural networks.



In practice, you can usually apply dropout only to the neurons in the top one to three layers (excluding the output layer).

There is one small but important technical detail. Suppose $p = 75\%$: on average only 25% of all neurons are active at each step during training. This means that after training, a neuron would be connected to four times as many input neurons as it would be during training. To compensate for this fact, we need to multiply each neuron's input connection weights by four during training. If we don't, the neural network will not perform well as it will see different data during and after training. More generally, we need to divide the connection weights by the *keep probability* ($1 - p$) during training.

To implement dropout using Keras, you can use the `tf.keras.layers.Dropout` layer. During training, it randomly drops some inputs (setting them to 0) and divides the remaining inputs by the keep probability. After training, it does nothing at all; it just passes the inputs to the next layer. The following code applies dropout regularization before every dense layer, using a dropout rate of 0.2:

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dropout(rate=0.2),
    tf.keras.layers.Dense(100, activation="relu",
                         kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(rate=0.2),
    tf.keras.layers.Dense(100, activation="relu",
                         kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(rate=0.2),
    tf.keras.layers.Dense(10, activation="softmax")
])
[...] # compile and train the model
```



Since dropout is only active during training, comparing the training loss and the validation loss can be misleading. In particular, a model may be overfitting the training set and yet have similar training and validation losses. So make sure to evaluate the training loss without dropout (e.g., after training).

If you observe that the model is overfitting, you can increase the dropout rate. Conversely, you should try decreasing the dropout rate if the model underfits the training set. It can also help to increase the dropout rate for large layers, and reduce it for small ones. Moreover, many state-of-the-art architectures only use dropout after the last hidden layer, so you may want to try this if full dropout is too strong.

Dropout does tend to significantly slow down convergence, but it often results in a better model when tuned properly. So, it is generally well worth the extra time and effort, especially for large models.



If you want to regularize a self-normalizing network based on the SELU activation function (as discussed earlier), you should use *alpha dropout*: this is a variant of dropout that preserves the mean and standard deviation of its inputs. It was introduced in the same paper as SELU, as regular dropout would break self-normalization.

Monte Carlo (MC) Dropout

In 2016, a [paper²⁸](#) by Yarin Gal and Zoubin Ghahramani added a few more good reasons to use dropout:

- First, the paper established a profound connection between dropout networks (i.e., neural networks containing Dropout layers) and approximate Bayesian inference,²⁹ giving dropout a solid mathematical justification.
- Second, the authors introduced a powerful technique called *MC Dropout*, which can boost the performance of any trained dropout model without having to retrain it or even modify it at all. It also provides a much better measure of the model's uncertainty, and it can be implemented in just a few lines of code.

If this all sounds like some “one weird trick” clickbait, then take a look at the following code. It is the full implementation of *MC Dropout*, boosting the dropout model we trained earlier without retraining it:

²⁸ Yarin Gal and Zoubin Ghahramani, “Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning,” *Proceedings of the 33rd International Conference on Machine Learning* (2016): 1050–1059.

²⁹ Specifically, they show that training a dropout network is mathematically equivalent to approximate Bayesian inference in a specific type of probabilistic model called a *Deep Gaussian Process*.

```
import numpy as np

y_probas = np.stack([model(X_test, training=True)
                     for sample in range(100)])
y_proba = y_probas.mean(axis=0)
```

Note that `model(X)` is similar to `model.predict(X)` except it returns a tensor rather than a NumPy array, and it supports the `training` argument. In this code example, setting `training=True` ensures that the Dropout layer remains active, so all predictions will be a bit different. We just make 100 predictions over the test set, and we compute their average. More specifically, each call to the model returns a matrix with one row per instance and one column per class. Because there are 10,000 instances in the test set and 10 classes, this is a matrix of shape [10000, 10]. We stack 100 such matrices, so `y_probas` is a 3D array of shape [100, 10000, 10]. Once we average over the first dimension (`axis=0`), we get `y_proba`, an array of shape [10000, 10], like we would get with a single prediction. That's all! Averaging over multiple predictions with dropout turned on gives us a Monte Carlo estimate that is generally more reliable than the result of a single prediction with dropout turned off. For example, let's look at the model's prediction for the first instance in the Fashion MNIST test set, with dropout turned off:

```
>>> model.predict(X_test[:1]).round(3)
array([[0.    , 0.    , 0.    , 0.    , 0.    , 0.024, 0.    , 0.132, 0.    ,
       0.844]], dtype=float32)
```

The model is fairly confident (84.4%) that this image belongs to class 9 (ankle boot). Compare this with the MC dropout prediction:

```
>>> y_proba[0].round(3)
array([0.    , 0.    , 0.    , 0.    , 0.    , 0.067, 0.    , 0.209, 0.001,
       0.723], dtype=float32)
```

The model still seems to prefer class 9, but its confidence dropped down to 72.3%, and the estimated probabilities for classes 5 (sandal) and 7 (sneaker) have increased, which makes sense given they're also footwear.

MC Dropout tends to improve the reliability of the model's probability estimates. This means that it's less likely to be confident but wrong, which can be dangerous: just imagine a self-driving car confidently ignoring a stop sign. It's also useful to know exactly which other classes are most likely. And you can also take a look at the **standard deviation of the probability estimates**:

```
>>> y_std = y_probas.std(axis=0)
>>> y_std[0].round(3)
array([0.    , 0.    , 0.    , 0.001, 0.    , 0.096, 0.    , 0.162, 0.001,
       0.183], dtype=float32)
```

Apparently there's quite a lot of variance in the probability estimates for class 9: the standard deviation is 0.183, which should be compared to the estimated probability

of 0.723: if you were building a risk-sensitive system (e.g., a medical or financial system), you would probably treat such an uncertain prediction with extreme caution. You would definitely not treat it like an 84.4% confident prediction. Moreover, the model's accuracy got a (very) small boost from 87.0% to 87.2%:

```
>>> y_pred = y_proba.argmax(axis=1)
>>> accuracy = (y_pred == y_test).sum() / len(y_test)
>>> accuracy
0.8717
```



The number of Monte Carlo samples you use (100 in this example) is a hyperparameter you can tweak. The higher it is, the more accurate the predictions and their uncertainty estimates will be. However, if you double it, inference time will also be doubled. Moreover, above a certain number of samples, you will notice little improvement. So your job is to find the right trade-off between latency and accuracy, depending on your application.

If your model contains other layers that behave in a special way during training (such as `BatchNormalization` layers), then you should not force training mode like we just did. Instead, you should replace the `Dropout` layers with the following `MCDropout` class:³⁰

```
class MCDropout(tf.keras.layers.Dropout):
    def call(self, inputs, training=False):
        return super().call(inputs, training=True)
```

Here, we just subclass the `Dropout` layer and override the `call()` method to force its `training` argument to `True` (see [Chapter 12](#)). Similarly, you could define an `MCAlphaDropout` class by subclassing `AlphaDropout` instead. If you are creating a model from scratch, it's just a matter of using `MCDropout` rather than `Dropout`. But if you have a model that was already trained using `Dropout`, you need to create a new model that's identical to the existing model except with `Dropout` instead of `MCDropout`, then copy the existing model's weights to your new model.

In short, MC Dropout is a great technique that boosts dropout models and provides better uncertainty estimates. And of course, since it is just regular dropout during training, it also acts like a regularizer.

³⁰ This `MCDropout` class will work with all Keras APIs, including the Sequential API. If you only care about the Functional API or the Subclassing API, you do not have to create an `MCDropout` class; you can create a regular `Dropout` layer and call it with `training=True`.

Max-Norm Regularization

Another popular regularization technique for neural networks is called *max-norm regularization*: for each neuron, it constrains the weights \mathbf{w} of the incoming connections such that $\|\mathbf{w}\|_2 \leq r$, where r is the max-norm hyperparameter and $\|\cdot\|_2$ is the ℓ_2 norm.

Max-norm regularization does not add a regularization loss term to the overall loss function. Instead, it is typically implemented by computing $\|\mathbf{w}\|_2$ after each training step and rescaling \mathbf{w} if needed ($\mathbf{w} \leftarrow \mathbf{w} / \|\mathbf{w}\|_2$).

Reducing r increases the amount of regularization and helps reduce overfitting. Max-norm regularization can also help alleviate the unstable gradients problems (if you are not using Batch Normalization).

To implement max-norm regularization in Keras, set the `kernel_constraint` argument of each hidden layer to a `max_norm()` constraint with the appropriate max value, like this:

```
dense = tf.keras.layers.Dense(  
    100, activation="relu", kernel_initializer="he_normal",  
    kernel_constraint=tf.keras.constraints.max_norm(1.))
```

After each training iteration, the model's `fit()` method will call the object returned by `max_norm()`, passing it the layer's weights and getting rescaled weights in return, which then replace the layer's weights. As you'll see in [Chapter 12](#), you can define your own custom constraint function if necessary and use it as the `kernel_constraint`. You can also constrain the bias terms by setting the `bias_constraint` argument.

The `max_norm()` function has an `axis` argument that defaults to `0`. A `Dense` layer usually has weights of shape `[number of inputs, number of neurons]`, so using `axis=0` means that the max-norm constraint will apply independently to each neuron's weight vector. If you want to use max-norm with convolutional layers (see [Chapter 14](#)), make sure to set the `max_norm()` constraint's `axis` argument appropriately (usually `axis=[0, 1, 2]`).

Summary and Practical Guidelines

In this chapter we have covered a wide range of techniques, and you may be wondering which ones you should use. This depends on the task, and there is no clear consensus yet, but I have found the configuration in [Table 11-3](#) to work fine in most cases, without requiring much hyperparameter tuning. That said, please do not consider these defaults as hard rules!

Table 11-3. Default DNN configuration

Hyperparameter	Default value
Kernel initializer	He initialization
Activation function	ReLU if shallow; Swish if deep
Normalization	None if shallow; Batch Norm if deep
Regularization	Early stopping; Weight decay if needed
Optimizer	Nesterov Accelerated Gradients or AdamW
Learning rate schedule	Performance scheduling or 1cycle

If the network is a simple stack of dense layers, then it can self-normalize, and you should use the configuration in [Table 11-4](#) instead.

Table 11-4. DNN configuration for a self-normalizing net

Hyperparameter	Default value
Kernel initializer	LeCun initialization
Activation function	SELU
Normalization	None (self-normalization)
Regularization	Alpha dropout if needed
Optimizer	Nesterov Accelerated Gradients
Learning rate schedule	Performance scheduling or 1cycle

Don't forget to normalize the input features! You should also try to reuse parts of a pretrained neural network if you can find one that solves a similar problem, or use unsupervised pretraining if you have a lot of unlabeled data, or use pretraining on an auxiliary task if you have a lot of labeled data for a similar task.

While the previous guidelines should cover most cases, here are some exceptions:

- If you need a sparse model, you can use ℓ_1 regularization (and optionally zero out the tiny weights after training). If you need an even sparser model, you can use the TensorFlow Model Optimization Toolkit. This will break self-normalization, so you should use the default configuration in this case.
- If you need a low-latency model (one that performs lightning-fast predictions), you may need to use fewer layers, use a fast activation function such as ReLU or leaky ReLU, and fold the Batch Normalization layers into the previous layers after training. Having a sparse model will also help. Finally, you may want to reduce the float precision from 32 bits to 16 or even 8 bits (see [“Deploying a Model to a Mobile or Embedded Device” on page 753](#)). Again, check out TF-MOT.

- If you are building a risk-sensitive application, or inference latency is not very important in your application, you can use MC Dropout to boost performance and get more reliable probability estimates, along with uncertainty estimates.

With these guidelines, you are now ready to train very deep nets! I hope you are now convinced that you can go quite a long way using just the convenient Keras API. There may come a time, however, when you need to have even more control; for example, to write a custom loss function or to tweak the training algorithm. For such cases you will need to use TensorFlow's lower-level API, as you will see in the next chapter.

Exercises

1. What is the problem that Glorot initialization and He initialization aim to fix?
2. Is it OK to initialize all the weights to the same value as long as that value is selected randomly using He initialization?
3. Is it OK to initialize the bias terms to 0?
4. In which cases would you want to use each of the activation functions we discussed in this chapter?
5. What may happen if you set the `momentum` hyperparameter too close to 1 (e.g., 0.99999) when using an SGD optimizer?
6. Name three ways you can produce a sparse model.
7. Does dropout slow down training? Does it slow down inference (i.e., making predictions on new instances)? What about MC Dropout?
8. Practice training a deep neural network on the CIFAR10 image dataset:
 - a. Build a DNN with 20 hidden layers of 100 neurons each (that's too many, but it's the point of this exercise). Use He initialization and the Swish activation function.
 - b. Using Nadam optimization and early stopping, train the network on the CIFAR10 dataset. You can load it with `tf.keras.datasets.cifar10.load_data()`. The dataset is composed of 60,000 32×32 -pixel color images (50,000 for training, 10,000 for testing) with 10 classes, so you'll need a softmax output layer with 10 neurons. Remember to search for the right learning rate each time you change the model's architecture or hyperparameters.
 - c. Now try adding Batch Normalization and compare the learning curves: is it converging faster than before? Does it produce a better model? How does it affect training speed?
 - d. Try replacing Batch Normalization with SELU, and make the necessary adjustments to ensure the network self-normalizes (i.e., standardize the input features).

- tures, use LeCun normal initialization, make sure the DNN contains only a sequence of dense layers, etc.).
- e. Try regularizing the model with alpha dropout. Then, without retraining your model, see if you can achieve better accuracy using MC Dropout.
 - f. Retrain your model using 1cycle scheduling and see if it improves training speed and model accuracy.

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab3>.

Custom Models and Training with TensorFlow

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 12th chapter of the final book. The GitHub repo is <https://github.com/ageron/handsonml3>.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

Up until now, we’ve used only TensorFlow’s high-level API, Keras, but it already got us pretty far: we built various neural network architectures, including regression and classification nets, Wide & Deep nets, and self-normalizing nets, using all sorts of techniques, such as Batch Normalization, dropout, and learning rate schedules. In fact, 95% of the use cases you will encounter will not require anything other than Keras (and `tf.data`; see Chapter 13). But now it’s time to dive deeper into TensorFlow and take a look at its lower-level [Python API](#). This will be useful when you need extra control to write custom loss functions, custom metrics, layers, models, initializers, regularizers, weight constraints, and more. You may even need to fully control the training loop itself, for example to apply special transformations or constraints to the gradients (beyond just clipping them) or to use multiple optimizers for different parts of the network. We will cover all these cases in this chapter, and we will also look at how you can boost your custom models and training algorithms using TensorFlow’s automatic graph generation feature. But first, let’s take a quick tour of TensorFlow.

A Quick Tour of TensorFlow

As you know, TensorFlow is a powerful library for numerical computation, particularly well suited and fine-tuned for large-scale Machine Learning (but you could use it for anything else that requires heavy computations). It was developed by the Google Brain team and it powers many of Google's large-scale services, such as Google Cloud Speech, Google Photos, and Google Search. It was open sourced in November 2015, and it is now the most widely used Deep Learning library in the industry:¹ countless projects use TensorFlow for all sorts of Machine Learning tasks, such as image classification, natural language processing, recommender systems, and time series forecasting.

So what does TensorFlow offer? Here's a summary:

- Its core is very similar to NumPy, but with GPU support.
- It supports distributed computing (across multiple devices and servers).
- It includes a kind of just-in-time (JIT) compiler that allows it to optimize computations for speed and memory usage. It works by extracting the *computation graph* from a Python function, then optimizing it (e.g., by pruning unused nodes), and finally running it efficiently (e.g., by automatically running independent operations in parallel).
- Computation graphs can be exported to a portable format, so you can train a TensorFlow model in one environment (e.g., using Python on Linux) and run it in another (e.g., using Java on an Android device).
- It implements reverse-mode autodiff (see [Chapter 10](#) and [Appendix B](#)) and provides some excellent optimizers, such as RMSProp and Nadam (see [Chapter 11](#)), so you can easily minimize all sorts of loss functions.

TensorFlow offers many more features built on top of these core features: the most important is of course Keras,² but it also has data loading and preprocessing ops (`tf.data`, `tf.io`, etc.), image processing ops (`tf.image`), signal processing ops (`tf.signal`), and more (see [Figure 12-1](#) for an overview of TensorFlow's Python API).

¹ However, Facebook's PyTorch library is currently more popular in Academia: more papers cite PyTorch than TensorFlow or Keras. Moreover, Google's JAX library is gaining momentum, especially in academia.

² TensorFlow includes another Deep Learning API called the *Estimators API*, but it is now deprecated.



We will cover many of the packages and functions of the TensorFlow API, but it's impossible to cover them all, so you should really take some time to browse through the API; you will find that it is quite rich and well documented.

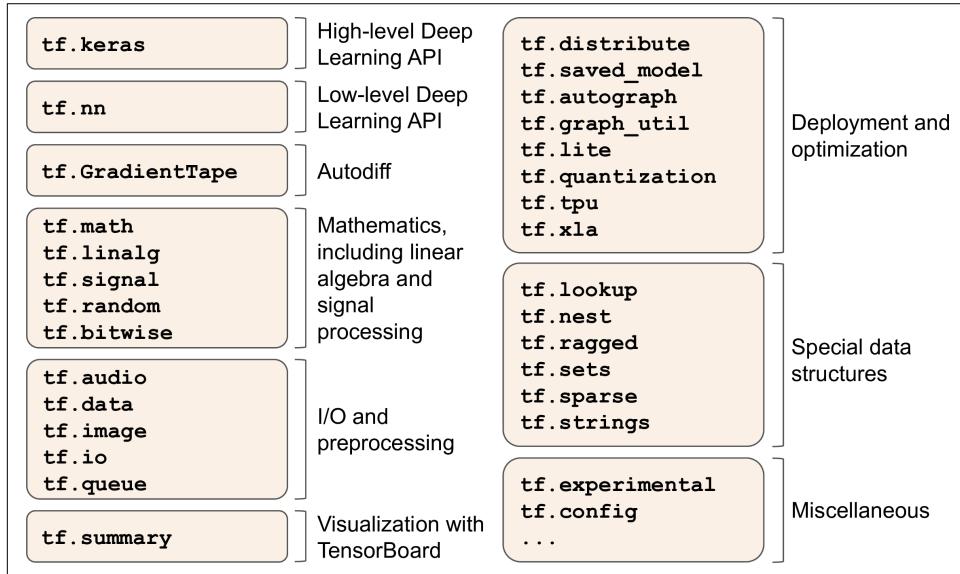


Figure 12-1. TensorFlow's Python API

At the lowest level, each TensorFlow operation (*op* for short) is implemented using highly efficient C++ code.³ Many operations have multiple implementations called *kernels*: each kernel is dedicated to a specific device type, such as CPUs, GPUs, or even TPUs (*tensor processing units*). As you may know, GPUs can dramatically speed up computations by splitting them into many smaller chunks and running them in parallel across many GPU threads. TPUs are even faster: they are custom ASIC chips built specifically for Deep Learning operations⁴ (we will discuss how to use TensorFlow with GPUs or TPUs in Chapter 19).

TensorFlow's architecture is shown in Figure 12-2. Most of the time your code will use the high-level APIs (especially Keras and `tf.data`); but when you need more flexibility, you will use the lower-level Python API, handling tensors directly. In any case, TensorFlow's execution engine will take care of running the operations efficiently, even across multiple devices and machines if you tell it to.

³ If you ever need to (but you probably won't), you can write your own operations using the C++ API.

⁴ To learn more about TPUs and how they work, check out <https://hml.info/tpus>.

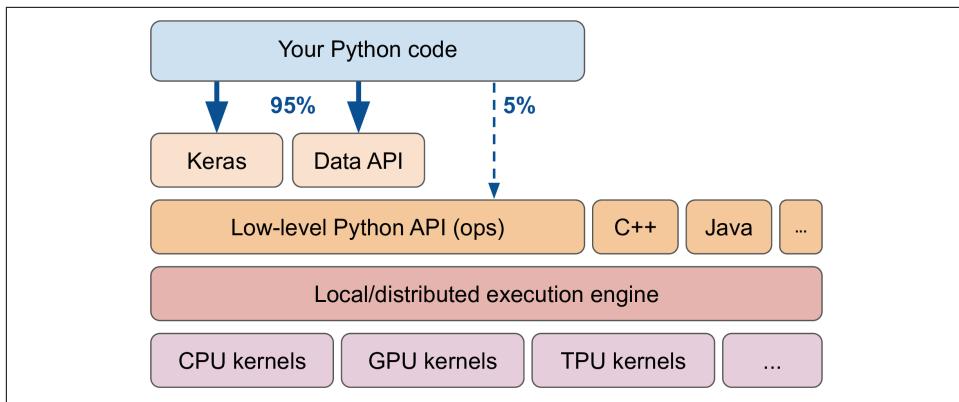


Figure 12-2. TensorFlow's architecture

TensorFlow runs not only on Windows, Linux, and macOS, but also on mobile devices (using *TensorFlow Lite*), including both iOS and Android (see Chapter 19). Note that APIs for other languages are also available, if you do not want to use the Python API: there are C++, Java, and Swift APIs. There is even a JavaScript implementation called *TensorFlow.js* that makes it possible to run your models directly in your browser.

There's more to TensorFlow than the library. TensorFlow is at the center of an extensive ecosystem of libraries. First, there's TensorBoard for visualization (see Chapter 10). Next, there's **TensorFlow Extended (TFX)**, which is a set of libraries built by Google to productionize TensorFlow projects: it includes tools for data validation, preprocessing, model analysis, and serving (with TF Serving; see Chapter 19). Google's *TensorFlow Hub* provides a way to easily download and reuse pretrained neural networks. You can also get many neural network architectures, some of them pretrained, in TensorFlow's **model garden**. Check out the **TensorFlow Resources** and <https://github.com/jtoy/awesome-tensorflow> for more TensorFlow-based projects. You will find hundreds of TensorFlow projects on GitHub, so it is often easy to find existing code for whatever you are trying to do.



More and more ML papers are released along with their implementations, and sometimes even with pretrained models. Check out <https://paperswithcode.com/> to easily find them.

Last but not least, TensorFlow has a dedicated team of passionate and helpful developers, as well as a large community contributing to improving it. To ask technical questions, you should use <https://stackoverflow.com/> and tag your question with

tensorflow and *python*. You can file bugs and feature requests through [GitHub](#). For general discussions, join the [TensorFlow Forum](#).

OK, it's time to start coding!

Using TensorFlow like NumPy

TensorFlow's API revolves around *tensors*, which flow from operation to operation—hence the name *TensorFlow*. A tensor is very similar to a NumPy `ndarray`: it is usually a multidimensional array, but it can also hold a scalar (a simple value, such as 42). These tensors will be important when we create custom cost functions, custom metrics, custom layers, and more, so let's see how to create and manipulate them.

Tensors and Operations

You can create a tensor with `tf.constant()`. For example, here is a tensor representing a matrix with two rows and three columns of floats:

```
>>> import tensorflow as tf
>>> t = tf.constant([[1., 2., 3.], [4., 5., 6.]]) # matrix
>>> t
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>
```

Just like an `ndarray`, a `tf.Tensor` has a shape and a data type (`dtype`):

```
>>> t.shape
TensorShape([2, 3])
>>> t.dtype
tf.float32
```

Indexing works much like in NumPy:

```
>>> t[:, 1:]
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[2., 3.],
       [5., 6.]], dtype=float32)>
>>> t[..., 1, tf.newaxis]
<tf.Tensor: shape=(2, 1), dtype=float32, numpy=
array([[2.],
       [5.]], dtype=float32)>
```

Most importantly, all sorts of tensor operations are available:

```
>>> t + 10
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[11., 12., 13.],
       [14., 15., 16.]], dtype=float32)>
>>> tf.square(t)
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
```

```
array([[ 1.,  4.,  9.],
       [16., 25., 36.]], dtype=float32)>
>>> t @ tf.transpose(t)
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[14., 32.],
       [32., 77.]], dtype=float32)>
```

Note that writing `t + 10` is equivalent to calling `tf.add(t, 10)` (indeed, Python calls the magic method `t.__add__(10)`, which just calls `tf.add(t, 10)`). Other operators like `-` and `*` are also supported. The `@` operator was added in Python 3.5, for matrix multiplication: it is equivalent to calling the `tf.matmul()` function.



Many functions and classes have aliases. For example, `tf.add()` and `tf.math.add()` are the same function. This allows TensorFlow to have concise names for the most common operations⁵ while preserving well-organized packages.

A tensor can also hold a scalar value. In this case, the shape is empty:

```
>>> tf.constant(42)
<tf.Tensor: shape=(), dtype=int32, numpy=42>
```

You will find all the basic math operations you need (`tf.add()`, `tf.multiply()`, `tf.square()`, `tf.exp()`, `tf.sqrt()`, etc.) and most operations that you can find in NumPy (e.g., `tf.reshape()`, `tf.squeeze()`, `tf.tile()`). Some functions have a different name than in NumPy; for instance, `tf.reduce_mean()`, `tf.reduce_sum()`, `tf.reduce_max()`, and `tf.math.log()` are the equivalent of `np.mean()`, `np.sum()`, `np.max()` and `np.log()`. When the name differs, there is usually a good reason for it. For example, in TensorFlow you must write `tf.transpose(t)`; you cannot just write `t.T` like in NumPy. The reason is that the `tf.transpose()` function does not do exactly the same thing as NumPy's `T` attribute: in TensorFlow, a new tensor is created with its own copy of the transposed data, while in NumPy, `t.T` is just a transposed view on the same data. Similarly, the `tf.reduce_sum()` operation is named this way because its GPU kernel (i.e., GPU implementation) uses a reduce algorithm that does not guarantee the order in which the elements are added: because 32-bit floats have limited precision, the result may change ever so slightly every time you call this operation. The same is true of `tf.reduce_mean()` (but of course `tf.reduce_max()` is deterministic).

⁵ A notable exception is `tf.math.log()`, which is commonly used but doesn't have a `tf.log()` alias, as it might be confused with logging.



The Keras API has its own low-level API, located in `tf.keras.backend`. This package is usually imported as `K`, for conciseness. It used to include functions like `K.square()`, `K.exp()`, and `K.sqrt()`, which you may run across in existing code: this was useful to write portable code back when Keras supported multiple backends, but now that Keras is TensorFlow-only, you should call TensorFlow's low-level API directly (e.g., `tf.square()` instead of `K.square()`). Technically `K.square()` and its friends are still there for backward compatibility, but the documentation of the `tf.keras.backend` package only lists a handful of utility functions, such as `clear_session()` (discussed in [Chapter 10](#)).

Tensors and NumPy

Tensors play nice with NumPy: you can create a tensor from a NumPy array, and vice versa. You can even apply TensorFlow operations to NumPy arrays and NumPy operations to tensors:

```
>>> import numpy as np
>>> a = np.array([2., 4., 5.])
>>> tf.constant(a)
<tf.Tensor: id=111, shape=(3,), dtype=float64, numpy=array([2., 4., 5.])>
>>> t.numpy() # or np.array(t)
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)
>>> tf.square(a)
<tf.Tensor: id=116, shape=(3,), dtype=float64, numpy=array([4., 16., 25.])>
>>> np.square(t)
array([[ 1.,   4.,   9.],
       [16.,  25.,  36.]], dtype=float32)
```



Notice that NumPy uses 64-bit precision by default, while TensorFlow uses 32-bit. This is because 32-bit precision is generally more than enough for neural networks, plus it runs faster and uses less RAM. So when you create a tensor from a NumPy array, make sure to set `dtype=tf.float32`.

Type Conversions

Type conversions can significantly hurt performance, and they can easily go unnoticed when they are done automatically. To avoid this, TensorFlow does not perform any type conversions automatically: it just raises an exception if you try to execute an operation on tensors with incompatible types. For example, you cannot add a float tensor and an integer tensor, and you cannot even add a 32-bit float and a 64-bit float:

```
>>> tf.constant(2.) + tf.constant(40)
[...] InvalidArgumentError: [...] expected to be a float tensor [...]
```

```
>>> tf.constant(2.) + tf.constant(40., dtype=tf.float64)
[...] InvalidArgumentError: [...] expected to be a float tensor [...]
```

This may be a bit annoying at first, but remember that it's for a good cause! And of course you can use `tf.cast()` when you really need to convert types:

```
>>> t2 = tf.constant(40., dtype=tf.float64)
>>> tf.constant(2.0) + tf.cast(t2, tf.float32)
<tf.Tensor: id=136, shape=(), dtype=float32, numpy=42.0>
```

Variables

The `tf.Tensor` values we've seen so far are immutable: you cannot modify them. This means that we cannot use regular tensors to implement weights in a neural network, since they need to be tweaked by backpropagation. Plus, other parameters may also need to change over time (e.g., a momentum optimizer keeps track of past gradients). What we need is a `tf.Variable`:

```
>>> v = tf.Variable([[1., 2., 3.], [4., 5., 6.]])
>>> v
<tf.Variable 'Variable:0' shape=(2, 3) dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>
```

A `tf.Variable` acts much like a `tf.Tensor`: you can perform the same operations with it, it plays nicely with NumPy as well, and it is just as picky with types. But it can also be modified in place using the `assign()` method (or `assign_add()` or `assign_sub()`, which increment or decrement the variable by the given value). You can also modify individual cells (or slices), by using the cell's (or slice's) `assign()` method or by using the `scatter_update()` or `scatter_nd_update()` methods:

```
v.assign(2 * v)           # v now equals [[2., 4., 6.], [8., 10., 12.]]
v[0, 1].assign(42)         # v now equals [[2., 42., 6.], [8., 10., 12.]]
v[:, 2].assign([0., 1.])   # v now equals [[2., 42., 0.], [8., 10., 1.]]
v.scatter_nd_update(
    indices=[[0, 0], [1, 2]], updates=[100., 200.])
```

Direct assignment will not work:

```
>>> v[1] = [7., 8., 9.]
[...] TypeError: 'ResourceVariable' object does not support item assignment
```



In practice you will rarely have to create variables manually, since Keras provides an `add_weight()` method that will take care of it for you, as we will see. Moreover, model parameters will generally be updated directly by the optimizers, so you will rarely need to update variables manually.

Other Data Structures

TensorFlow supports several other data structures, including the following (please see the “Other Data Structures” section in the notebook or [Appendix C](#) for more details):

Sparse tensors (tf.SparseTensor)

Efficiently represent tensors containing mostly zeros. The `tf.sparse` package contains operations for sparse tensors.

Tensor arrays (tf.TensorArray)

Are lists of tensors. They have a fixed length by default but can optionally be made extensible. All tensors they contain must have the same shape and data type.

Ragged tensors (tf.RaggedTensor)

Represent lists of tensors, all of the same rank and data type, but with varying sizes. The dimensions along which the tensor sizes vary are called the *ragged dimensions*. The `tf.ragged` package contains operations for ragged tensors.

String tensors

Are regular tensors of type `tf.string`. These represent byte strings, not Unicode strings, so if you create a string tensor using a Unicode string (e.g., a regular Python 3 string like "café"), then it will get encoded to UTF-8 automatically (e.g., `b"caf\xc3\xa9"`). Alternatively, you can represent Unicode strings using tensors of type `tf.int32`, where each item represents a Unicode code point (e.g., `[99, 97, 102, 233]`). The `tf.strings` package (with an `s`) contains ops for byte strings and Unicode strings (and to convert one into the other). It’s important to note that a `tf.string` is atomic, meaning that its length does not appear in the tensor’s shape. Once you convert it to a Unicode tensor (i.e., a tensor of type `tf.int32` holding Unicode code points), the length appears in the shape.

Sets

Are represented as regular tensors (or sparse tensors). For example, `tf.constant([[1, 2], [3, 4]])` represents the two sets $\{1, 2\}$ and $\{3, 4\}$. More generally, each set is represented by a vector in the tensor’s last axis. You can manipulate sets using operations from the `tf.sets` package.

Queues

Store tensors across multiple steps. TensorFlow offers various kinds of queues: basic First In, First Out (FIFO) queues (`FIFOQueue`), queues that can prioritize some items (`PriorityQueue`), shuffle their items (`RandomShuffleQueue`), and batch items of different shapes by padding (`PaddingFIFOQueue`). These classes are all in the `tf.queue` package.

With tensors, operations, variables, and various data structures at your disposal, you are now ready to customize your models and training algorithms!

Customizing Models and Training Algorithms

Let's start by creating a custom loss function, which is a straightforward and common use case.

Custom Loss Functions

Suppose you want to train a regression model, but your training set is a bit noisy. Of course, you start by trying to clean up your dataset by removing or fixing the outliers, but that turns out to be insufficient; the dataset is still noisy. Which loss function should you use? The mean squared error might penalize large errors too much and cause your model to be imprecise. The mean absolute error would not penalize outliers as much, but training might take a while to converge, and the trained model might not be very precise. This is probably a good time to use the Huber loss (introduced in [Chapter 10](#)) instead of the good old MSE. The Huber loss is available in Keras (just use an instance of the `tf.keras.losses.Huber` class). But let's pretend it's not there. To implement it, just create a function that takes the labels and the model's predictions as arguments, and uses TensorFlow operations to compute a tensor containing all the losses (one per sample):

```
def huber_fn(y_true, y_pred):
    error = y_true - y_pred
    is_small_error = tf.abs(error) < 1
    squared_loss = tf.square(error) / 2
    linear_loss = tf.abs(error) - 0.5
    return tf.where(is_small_error, squared_loss, linear_loss)
```



For better performance, you should use a vectorized implementation, as in this example. Moreover, if you want to benefit from TensorFlow's graph optimization features, you should use only TensorFlow operations.

It is also possible to return the mean loss instead of the individual sample losses, but this is not recommended as it makes it impossible to use class weights or sample weights when you need them (see [Chapter 10](#)).

Now you can use this Huber loss function when you compile the Keras model, then train your model as usual:

```
model.compile(loss=huber_fn, optimizer="nadam")
model.fit(X_train, y_train, [...])
```

And that's it! For each batch during training, Keras will call the `huber_fn()` function to compute the loss, then it will use reverse-mode autodiff to compute the gradients of the loss with regards to all the model parameters, and finally it will perform a Gradient Descent step (in this example using a Nadam optimizer). Moreover, it will keep track of the total loss since the beginning of the epoch, and it will display the mean loss.

But what happens to this custom loss when you save the model?

Saving and Loading Models That Contain Custom Components

Saving a model containing a custom loss function works fine, but when you load it, you'll need to provide a dictionary that maps the function name to the actual function. More generally, when you load a model containing custom objects, you need to map the names to the objects:

```
model = tf.keras.models.load_model("my_model_with_a_custom_loss",
                                    custom_objects={"huber_fn": huber_fn})
```



If you decorate the `huber_fn()` function with `@keras.utils.register_keras_serializable()`, it will automatically be available to the `load_model()` function: no need to include it in the `custom_objects` dictionary.

With the current implementation, any error between -1 and 1 is considered “small.” But what if you want a different threshold? One solution is to create a function that creates a configured loss function:

```
def create_huber(threshold=1.0):
    def huber_fn(y_true, y_pred):
        error = y_true - y_pred
        is_small_error = tf.abs(error) < threshold
        squared_loss = tf.square(error) / 2
        linear_loss = threshold * tf.abs(error) - threshold ** 2 / 2
        return tf.where(is_small_error, squared_loss, linear_loss)
    return huber_fn

model.compile(loss=create_huber(2.0), optimizer="nadam")
```

Unfortunately, when you save the model, the `threshold` will not be saved. This means that you will have to specify the `threshold` value when loading the model (note that the name to use is `"huber_fn"`, which is the name of the function you gave Keras, not the name of the function that created it):

```
model = tf.keras.models.load_model(
    "my_model_with_a_custom_loss_threshold_2",
    custom_objects={"huber_fn": create_huber(2.0)})
)
```

You can solve this by creating a subclass of the `tf.keras.losses.Loss` class, and then implementing its `get_config()` method:

```
class HuberLoss(tf.keras.losses.Loss):
    def __init__(self, threshold=1.0, **kwargs):
        self.threshold = threshold
        super().__init__(**kwargs)

    def call(self, y_true, y_pred):
        error = y_true - y_pred
        is_small_error = tf.abs(error) < self.threshold
        squared_loss = tf.square(error) / 2
        linear_loss = self.threshold * tf.abs(error) - self.threshold**2 / 2
        return tf.where(is_small_error, squared_loss, linear_loss)

    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "threshold": self.threshold}
```

Let's walk through this code:

- The constructor accepts `**kwargs` and passes them to the parent constructor, which handles standard hyperparameters: the name of the loss and the reduction algorithm to use to aggregate the individual instance losses. By default, it is "AUTO", which is equivalent to "SUM_OVER_BATCH_SIZE": the loss will be the sum of the instance losses, weighted by the sample weights, if any, and divided by the batch size (not by the sum of weights, so this is *not* the weighted mean).⁶ Other possible values are "SUM" and "NONE".
- The `call()` method takes the labels and predictions, computes all the instance losses, and returns them.
- The `get_config()` method returns a dictionary mapping each hyperparameter name to its value. It first calls the parent class's `get_config()` method, then adds the new hyperparameters to this dictionary.⁷

You can then use any instance of this class when you compile the model:

```
model.compile(loss=HuberLoss(2.), optimizer="nadam")
```

When you save the model, the threshold will be saved along with it; and when you load the model, you just need to map the class name to the class itself:

⁶ It would not be a good idea to use a weighted mean: if you did, then two instances with the same weight but in different batches would have a different impact on training, depending on the total weight of each batch.

⁷ The `{**x, [...]}` syntax was added in Python 3.5, to merge all the key/value pairs from dictionary `x` into another dictionary. Since Python 3.9, you can use the nicer `x | y` syntax instead (where `x` and `y` are two dictionaries).

```
model = tf.keras.models.load_model("my_model_with_a_custom_loss_class",
                                    custom_objects={"HuberLoss": HuberLoss})
```

When you save a model, Keras calls the loss instance's `get_config()` method and saves the config in the `SavedModel`. When you load the model, it calls the `from_config()` class method on the `HuberLoss` class: this method is implemented by the base class (`Loss`) and creates an instance of the class, passing `**config` to the constructor.

That's it for losses! As we will see now, custom activation functions, initializers, regularizers, and constraints are not much different.

Custom Activation Functions, Initializers, Regularizers, and Constraints

Most Keras functionalities, such as losses, regularizers, constraints, initializers, metrics, activation functions, layers, and even full models, can be customized in very much the same way. Most of the time, you will just need to write a simple function with the appropriate inputs and outputs. Here are examples of a custom activation function (equivalent to `tf.keras.activations.softplus()` or `tf.nn.softplus()`), a custom Glorot initializer (equivalent to `tf.keras.initializers.glorot_normal()`), a custom ℓ_1 regularizer (equivalent to `tf.keras.regularizers.l1(0.01)`), and a custom constraint that ensures weights are all positive (equivalent to `tf.keras.constraints.nonneg()` or `tf.nn.relu()`):

```
def my_softplus(z):
    return tf.math.log(1.0 + tf.exp(z))

def my_glorot_initializer(shape, dtype=tf.float32):
    stddev = tf.sqrt(2. / (shape[0] + shape[1]))
    return tf.random.normal(shape, stddev=stddev, dtype=dtype)

def my_l1_regularizer(weights):
    return tf.reduce_sum(tf.abs(0.01 * weights))

def my_positive_weights(weights): # return value is just tf.nn.relu(weights)
    return tf.where(weights < 0., tf.zeros_like(weights), weights)
```

As you can see, the arguments depend on the type of custom function. These custom functions can then be used normally; for example:

```
layer = tf.keras.layers.Dense(1, activation=my_softplus,
                             kernel_initializer=my_glorot_initializer,
                             kernel_regularizer=my_l1_regularizer,
                             kernel_constraint=my_positive_weights)
```

The activation function will be applied to the output of this `Dense` layer, and its result will be passed on to the next layer. The layer's weights will be initialized using the value returned by the initializer. At each training step the weights will be passed to the regularization function to compute the regularization loss, which will be added to

the main loss to get the final loss used for training. Finally, the constraint function will be called after each training step, and the layer's weights will be replaced by the constrained weights.

If a function has hyperparameters that need to be saved along with the model, then you will want to subclass the appropriate class, such as `tf.keras.regularizers.Regularizer`, `tf.keras.constraints.Constraint`, `tf.keras.initializers.Initializer`, or `tf.keras.layers.Layer` (for any layer, including activation functions). Much like we did for the custom loss, here is a simple class for ℓ_1 regularization that saves its `factor` hyperparameter (this time we do not need to call the parent constructor or the `get_config()` method, as they are not defined by the parent class):

```
class MyL1Regularizer(tf.keras.regularizers.Regularizer):
    def __init__(self, factor):
        self.factor = factor

    def __call__(self, weights):
        return tf.reduce_sum(tf.abs(self.factor * weights))

    def get_config(self):
        return {"factor": self.factor}
```

Note that you must implement the `call()` method for losses, layers (including activation functions), and models, or the `__call__()` method for regularizers, initializers, and constraints. For metrics, things are a bit different, as we will see now.

Custom Metrics

Losses and metrics are conceptually not the same thing: losses (e.g., cross entropy) are used by Gradient Descent to *train* a model, so they must be differentiable (at least at the points where they are evaluated), and their gradients should not be 0 everywhere. Plus, it's OK if they are not easily interpretable by humans. In contrast, metrics (e.g., accuracy) are used to *evaluate* a model: they must be more easily interpretable, and they can be non-differentiable or have 0 gradients everywhere.

That said, in most cases, defining a custom metric function is exactly the same as defining a custom loss function. In fact, we could even use the Huber loss function we created earlier as a metric;⁸ it would work just fine (and persistence would also work the same way, in this case only saving the name of the function, "huber_fn", not the threshold):

```
model.compile(loss="mse", optimizer="nadam", metrics=[create_huber(2.0)])
```

⁸ However, the Huber loss is seldom used as a metric—the MAE or MSE are generally preferred.

For each batch during training, Keras will compute this metric and keep track of its mean since the beginning of the epoch. Most of the time, this is exactly what you want. But not always! Consider a binary classifier's precision, for example. As we saw in [Chapter 3](#), precision is the number of true positives divided by the number of positive predictions (including both true positives and false positives). Suppose the model made five positive predictions in the first batch, four of which were correct: that's 80% precision. Then suppose the model made three positive predictions in the second batch, but they were all incorrect: that's 0% precision for the second batch. If you just compute the mean of these two precisions, you get 40%. But wait a second—that's *not* the model's precision over these two batches! Indeed, there were a total of four true positives (4 + 0) out of eight positive predictions (5 + 3), so the overall precision is 50%, not 40%. What we need is an object that can keep track of the number of true positives and the number of false positives and that can compute the precision based on these numbers when requested. This is precisely what the `tf.keras.metrics.Precision` class does:

```
>>> precision = tf.keras.metrics.Precision()
>>> precision([0, 1, 1, 1, 0, 1, 0, 1], [1, 1, 0, 1, 0, 1, 0, 1])
<tf.Tensor: shape=(), dtype=float32, numpy=0.8>
>>> precision([0, 1, 0, 0, 1, 0, 1, 1], [1, 0, 1, 1, 0, 0, 0, 0])
<tf.Tensor: shape=(), dtype=float32, numpy=0.5>
```

In this example, we created a `Precision` object, then we used it like a function, passing it the labels and predictions for the first batch, then for the second batch (you can optionally pass sample weights as well, if you want). We used the same number of true and false positives as in the example we just discussed. After the first batch, it returns a precision of 80%; then after the second batch, it returns 50% (which is the overall precision so far, not the second batch's precision). This is called a *streaming metric* (or *stateful metric*), as it is gradually updated, batch after batch.

At any point, we can call the `result()` method to get the current value of the metric. We can also look at its variables (tracking the number of true and false positives) by using the `variables` attribute, and we can reset these variables using the `reset_states()` method:

```
>>> precision.result()
<tf.Tensor: shape=(), dtype=float32, numpy=0.5>
>>> precision.variables
[<tf.Variable 'true_positives:0' [...], numpy=array([4.], dtype=float32)>,
 <tf.Variable 'false_positives:0' [...], numpy=array([4.], dtype=float32)>]
>>> precision.reset_states() # both variables get reset to 0.0
```

If you need to define your own custom streaming metric, create a subclass of the `tf.keras.metrics.Metric` class. Here is a basic example that keeps track of the total Huber loss and the number of instances seen so far. When asked for the result, it returns the ratio, which is just the mean Huber loss:

```

class HuberMetric(tf.keras.metrics.Metric):
    def __init__(self, threshold=1.0, **kwargs):
        super().__init__(**kwargs) # handles base args (e.g., dtype)
        self.threshold = threshold
        self.huber_fn = create_huber(threshold)
        self.total = self.add_weight("total", initializer="zeros")
        self.count = self.add_weight("count", initializer="zeros")

    def update_state(self, y_true, y_pred, sample_weight=None):
        sample_metrics = self.huber_fn(y_true, y_pred)
        self.total.assign_add(tf.reduce_sum(sample_metrics))
        self.count.assign_add(tf.cast(tf.size(y_true), tf.float32))

    def result(self):
        return self.total / self.count

    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "threshold": self.threshold}

```

Let's walk through this code:⁹

- The constructor uses the `add_weight()` method to create the variables needed to keep track of the metric's state over multiple batches—in this case, the sum of all Huber losses (`total`) and the number of instances seen so far (`count`). You could just create variables manually if you preferred. Keras tracks any `tf.Variable` that is set as an attribute (and more generally, any “trackable” object, such as layers or models).
- The `update_state()` method is called when you use an instance of this class as a function (as we did with the `Precision` object). It updates the variables, given the labels and predictions for one batch (and sample weights, but in this case we ignore them).
- The `result()` method computes and returns the final result, in this case the mean Huber metric over all instances. When you use the metric as a function, the `update_state()` method gets called first, then the `result()` method is called, and its output is returned.
- We also implement the `get_config()` method to ensure the `threshold` gets saved along with the model.
- The default implementation of the `reset_states()` method resets all variables to 0.0 (but you can override it if needed).

⁹ This class is for illustration purposes only. A simpler and better implementation would just subclass the `tf.keras.metrics.Mean` class; see the “Streaming metrics” section of the notebook for an example.



Keras will take care of variable persistence seamlessly; no action is required.

When you define a metric using a simple function, Keras automatically calls it for each batch, and it keeps track of the mean during each epoch, just like we did manually. So the only benefit of our `HuberMetric` class is that the threshold will be saved. But of course, some metrics, like precision, cannot simply be averaged over batches: in those cases, there's no other option than to implement a streaming metric.

Now that we have built a streaming metric, building a custom layer will seem like a walk in the park!

Custom Layers

You may occasionally want to build an architecture that contains an exotic layer for which TensorFlow does not provide a default implementation. Or you may simply want to build a very repetitive architecture, in which a particular block of layers is repeated many times, and it would be convenient to treat each block as a single layer. For such cases, you'll want to build a custom layer.

First, some layers have no weights, such as `tf.keras.layers.Flatten` or `tf.keras.layers.ReLU`. If you want to create a custom layer without any weights, the simplest option is to write a function and wrap it in a `tf.keras.layers.Lambda` layer. For example, the following layer will apply the exponential function to its inputs:

```
exponential_layer = tf.keras.layers.Lambda(lambda x: tf.exp(x))
```

This custom layer can then be used like any other layer, using the Sequential API, the Functional API, or the Subclassing API. You can also use it as an activation function, or you could use `activation=tf.exp`. The exponential layer is sometimes used in the output layer of a regression model when the values to predict have very different scales (e.g., 0.001, 10., 1,000.). In fact, the exponential function is one of the standard activation functions in Keras, so you can just use `activation="exponential"`.

As you might guess, to build a custom stateful layer (i.e., a layer with weights), you need to create a subclass of the `tf.keras.layers.Layer` class. For example, the following class implements a simplified version of the `Dense` layer:

```
class MyDense(tf.keras.layers.Layer):
    def __init__(self, units, activation=None, **kwargs):
        super().__init__(**kwargs)
        self.units = units
        self.activation = tf.keras.activations.get(activation)

    def build(self, batch_input_shape):
```

```

    self.kernel = self.add_weight(
        name="kernel", shape=[batch_input_shape[-1], self.units],
        initializer="glorot_normal")
    self.bias = self.add_weight(
        name="bias", shape=[self.units], initializer="zeros")

def call(self, X):
    return self.activation(X @ self.kernel + self.bias)

def get_config(self):
    base_config = super().get_config()
    return {**base_config, "units": self.units,
            "activation": tf.keras.activations.serialize(self.activation)}

```

Let's walk through this code:

- The constructor takes all the hyperparameters as arguments (in this example, `units` and `activation`), and importantly it also takes a `**kwargs` argument. It calls the parent constructor, passing it the `kwargs`: this takes care of standard arguments such as `input_shape`, `trainable`, and `name`. Then it saves the hyperparameters as attributes, converting the `activation` argument to the appropriate activation function using the `tf.keras.activations.get()` function (it accepts functions, standard strings like "`relu`" or "`swish`", or simply `None`).
- The `build()` method's role is to create the layer's variables by calling the `add_weight()` method for each weight. The `build()` method is called the first time the layer is used. At that point, Keras will know the shape of this layer's inputs, and it will pass it to the `build()` method,¹⁰ which is often necessary to create some of the weights. For example, we need to know the number of neurons in the previous layer in order to create the connection weights matrix (i.e., the "`kernel`"): this corresponds to the size of the last dimension of the inputs. At the end of the `build()` method (and only at the end), you must call the parent's `build()` method: this tells Keras that the layer is built (it just sets `self.built = True`).
- The `call()` method performs the desired operations. In this case, we compute the matrix multiplication of the inputs `X` and the layer's kernel, we add the bias vector, and we apply the activation function to the result, and this gives us the output of the layer.

¹⁰ The Keras API calls this argument `input_shape`, but since it also includes the batch dimension, I prefer to call it `batch_input_shape`.

- The `get_config()` method is just like in the previous custom classes. Note that we save the activation function's full configuration by calling `tf.keras.activations.serialize()`.

You can now use a `MyDense` layer just like any other layer!



Keras automatically infers the output shape, except when the layer is dynamic (as we will see shortly). In this (rare) case, you need to implement the `compute_output_shape()` method, which must return a `TensorShape` object.

To create a layer with multiple inputs (e.g., `Concatenate`), the argument to the `call()` method should be a tuple containing all the inputs. To create a layer with multiple outputs, the `call()` method should return the list of outputs. For example, the following toy layer takes two inputs and returns three outputs:

```
class MyMultiLayer(tf.keras.layers.Layer):
    def call(self, X):
        X1, X2 = X
        return X1 + X2, X1 * X2, X1 / X2
```

This layer may now be used like any other layer, but of course only using the Functional and Subclassing APIs, not the Sequential API (which only accepts layers with one input and one output).

If your layer needs to have a different behavior during training and during testing (e.g., if it uses `Dropout` or `BatchNormalization` layers), then you must add a `training` argument to the `call()` method and use this argument to decide what to do. For example, let's create a layer that adds Gaussian noise during training (for regularization) but does nothing during testing (Keras has a layer that does the same thing, `tf.keras.layers.GaussianNoise`):

```
class MyGaussianNoise(tf.keras.layers.Layer):
    def __init__(self, stddev, **kwargs):
        super().__init__(**kwargs)
        self.stddev = stddev

    def call(self, X, training=False):
        if training:
            noise = tf.random.normal(tf.shape(X), stddev=self.stddev)
            return X + noise
        else:
            return X
```

With that, you can now build any custom layer you need! Now let's create custom models.

Custom Models

We already looked at creating custom model classes in [Chapter 10](#), when we discussed the Subclassing API.¹¹ It's straightforward: subclass the `tf.keras.Model` class, create layers and variables in the constructor, and implement the `call()` method to do whatever you want the model to do. For example, suppose you want to build the model represented in [Figure 12-3](#).

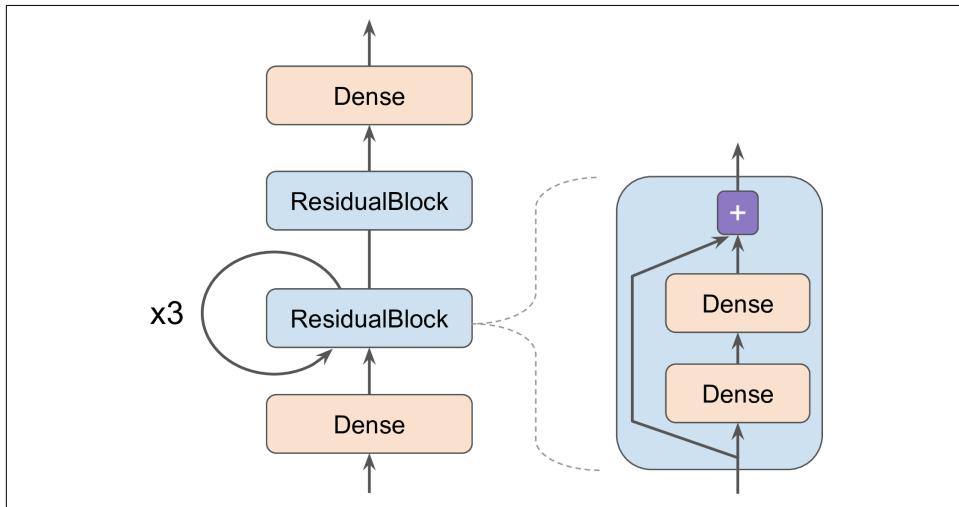


Figure 12-3. Custom model example: an arbitrary model with a custom `ResidualBlock` layer containing a skip connection

The inputs go through a first dense layer, then through a *residual block* composed of two dense layers and an addition operation (as we will see in [Chapter 14](#), a residual block adds its inputs to its outputs), then through this same residual block three more times, then through a second residual block, and the final result goes through a dense output layer. Don't worry if this model does not make much sense; it's just an example to illustrate the fact that you can easily build any kind of model you want, even one that contains loops and skip connections. To implement this model, it is best to first create a `ResidualBlock` layer, since we are going to create a couple of identical blocks (and we might want to reuse it in another model):

```
class ResidualBlock(tf.keras.layers.Layer):
    def __init__(self, n_layers, n_neurons, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [tf.keras.layers.Dense(n_neurons, activation="relu",
                                            kernel_initializer="he_normal")]
```

¹¹ The name “Subclassing API” in Keras usually refers only to the creation of custom models by subclassing, although many other things can be created by subclassing, as we saw in this chapter.

```

        for _ in range(n_layers)]
```

`def call(self, inputs):
 Z = inputs
 for layer in self.hidden:
 Z = layer(Z)
 return inputs + Z`

This layer is a bit special since it contains other layers. This is handled transparently by Keras: it automatically detects that the `hidden` attribute contains trackable objects (layers in this case), so their variables are automatically added to this layer's list of variables. The rest of this class is self-explanatory. Next, let's use the Subclassing API to define the model itself:

```

class ResidualRegressor(tf.keras.Model):
    def __init__(self, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.hidden1 = tf.keras.layers.Dense(30, activation="relu",
                                            kernel_initializer="he_normal")
        self.block1 = ResidualBlock(2, 30)
        self.block2 = ResidualBlock(2, 30)
        self.out = tf.keras.layers.Dense(output_dim)

    def call(self, inputs):
        Z = self.hidden1(inputs)
        for _ in range(1 + 3):
            Z = self.block1(Z)
        Z = self.block2(Z)
        return self.out(Z)
```

We create the layers in the constructor and use them in the `call()` method. This model can then be used like any other model (compile it, fit it, evaluate it, and use it to make predictions). If you also want to be able to save the model using the `save()` method and load it using the `tf.keras.models.load_model()` function, you must implement the `get_config()` method (as we did earlier) in both the `ResidualBlock` class and the `ResidualRegressor` class. Alternatively, you can save and load the weights using the `save_weights()` and `load_weights()` methods.

The `Model` class is a subclass of the `Layer` class, so models can be defined and used exactly like layers. But a model has some extra functionalities, including of course its `compile()`, `fit()`, `evaluate()`, and `predict()` methods (and a few variants), plus the `get_layers()` method (which can return any of the model's layers by name or by index) and the `save()` method (and support for `tf.keras.models.load_model()` and `tf.keras.models.clone_model()`).



If models provide more functionality than layers, why not just define every layer as a model? Well, technically you could, but it is usually cleaner to distinguish the internal components of your model (i.e., layers or reusable blocks of layers) from the model itself (i.e., the object you will train). The former should subclass the `Layer` class, while the latter should subclass the `Model` class.

With that, you can naturally and concisely build almost any model that you find in a paper, using the Sequential API, the Functional API, the Subclassing API, or even a mix of these. “Almost” any model? Yes, there are still a few things that we need to look at: first, how to define losses or metrics based on model internals, and second, how to build a custom training loop.

Losses and Metrics Based on Model Internals

The custom losses and metrics we defined earlier were all based on the labels and the predictions (and optionally sample weights). There will be times when you want to define losses based on other parts of your model, such as the weights or activations of its hidden layers. This may be useful for regularization purposes or to monitor some internal aspect of your model.

To define a custom loss based on model internals, compute it based on any part of the model you want, then pass the result to the `add_loss()` method. For example, let’s build a custom regression MLP model composed of a stack of five hidden layers plus an output layer. This custom model will also have an auxiliary output on top of the upper hidden layer. The loss associated to this auxiliary output will be called the *reconstruction loss* (see [Chapter 17](#)): it is the mean squared difference between the reconstruction and the inputs. By adding this reconstruction loss to the main loss, we will encourage the model to preserve as much information as possible through the hidden layers—even information that is not directly useful for the regression task itself. In practice, this loss sometimes improves generalization (it is a regularization loss). It is also possible to add a custom metric using the model’s `add_metric()` method. Here is the code for this custom model with a custom reconstruction loss and a corresponding metric:

```
class ReconstructingRegressor(tf.keras.Model):
    def __init__(self, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [tf.keras.layers.Dense(30, activation="relu",
                                            kernel_initializer="he_normal")]
        for _ in range(5)]
        self.out = tf.keras.layers.Dense(output_dim)
        self.reconstruction_mean = tf.keras.metrics.Mean(
            name="reconstruction_error")

    def build(self, batch_input_shape):
```

```

n_inputs = batch_input_shape[-1]
self.reconstruct = tf.keras.layers.Dense(n_inputs)

def call(self, inputs, training=False):
    Z = inputs
    for layer in self.hidden:
        Z = layer(Z)
    reconstruction = self.reconstruct(Z)
    recon_loss = tf.reduce_mean(tf.square(reconstruction - inputs))
    self.add_loss(0.05 * recon_loss)
    if training:
        result = self.reconstruction_mean(recon_loss)
        self.add_metric(result)
    return self.out(Z)

```

Let's go through this code:

- The constructor creates the DNN with five dense hidden layers and one dense output layer. We also create a `Mean` streaming metric to keep track of the reconstruction error during training.
- The `build()` method creates an extra dense layer which will be used to reconstruct the inputs of the model. It must be created here because its number of units must be equal to the number of inputs, and this number is unknown before the `build()` method is called.¹²
- The `call()` method processes the inputs through all five hidden layers, then passes the result through the reconstruction layer, which produces the reconstruction.
- Then the `call()` method computes the reconstruction loss (the mean squared difference between the reconstruction and the inputs), and adds it to the model's list of losses using the `add_loss()` method.¹³ Notice that we scale down the reconstruction loss by multiplying it by 0.05 (this is a hyperparameter you can tune). This ensures that the reconstruction loss does not dominate the main loss.
- Next, during training only, the `call()` method updates the reconstruction metric, and adds it to the model so it can be displayed. This code example can actually be simplified by calling `self.add_metric(recon_loss)` instead: Keras will automatically track the mean for you (no need for `self.reconstruction_mean`).

¹² Due to TensorFlow issue #46858, the call to `super().build()` may fail in this case, unless the issue was fixed by the time you read this. If not, you need to replace this line with `self.built = True`.

¹³ You can also call `add_loss()` on any layer inside the model, as the model recursively gathers losses from all of its layers.

- Finally, the `call()` method passes the output of the hidden layers to the output layer and returns its output.

Both the total loss and the reconstruction loss will go down during training:

```
Epoch 1/5
363/363 [=====] - 1s 820us/step - loss: 0.7640 - reconstruction_error: 1.2728
Epoch 2/5
363/363 [=====] - 0s 809us/step - loss: 0.4584 - reconstruction_error: 0.6340
[...]
```

In most cases, everything we have discussed so far will be sufficient to implement whatever model you want to build, even with complex architectures, losses, and metrics. However, for some architectures such as GANs (see [Chapter 17](#)), you will have to customize the training loop itself. Before we get there, we must look at how to compute gradients automatically in TensorFlow.

Computing Gradients Using Autodiff

To understand how to use autodiff (see [Chapter 10](#) and [Appendix B](#)) to compute gradients automatically, let's consider a simple toy function:

```
def f(w1, w2):
    return 3 * w1 ** 2 + 2 * w1 * w2
```

If you know calculus, you can analytically find that the partial derivative of this function with regard to w_1 is $6 * w_1 + 2 * w_2$. You can also find that its partial derivative with regard to w_2 is $2 * w_1$. For example, at the point $(w_1, w_2) = (5, 3)$, these partial derivatives are equal to 36 and 10, respectively, so the gradient vector at this point is (36, 10). But if this were a neural network, the function would be much more complex, typically with tens of thousands of parameters, and finding the partial derivatives analytically by hand would be a virtually impossible task. One solution could be to compute an approximation of each partial derivative by measuring how much the function's output changes when you tweak the corresponding parameter by a tiny amount:

```
>>> w1, w2 = 5, 3
>>> eps = 1e-6
>>> (f(w1 + eps, w2) - f(w1, w2)) / eps
36.000003007075065
>>> (f(w1, w2 + eps) - f(w1, w2)) / eps
10.00000003174137
```

Looks about right! This works rather well and is easy to implement, but it is just an approximation, and importantly you need to call `f()` at least once per parameter (not twice, since we could compute `f(w1, w2)` just once). Having to call `f()` at least once per parameter makes this approach intractable for large neural networks. So instead, we should use reverse-mode autodiff. TensorFlow makes this pretty simple:

```
w1, w2 = tf.Variable(5.), tf.Variable(3.)
with tf.GradientTape() as tape:
    z = f(w1, w2)

gradients = tape.gradient(z, [w1, w2])
```

We first define two variables `w1` and `w2`, then we create a `tf.GradientTape` context that will automatically record every operation that involves a variable, and finally we ask this tape to compute the gradients of the result `z` with regard to both variables `[w1, w2]`. Let's take a look at the gradients that TensorFlow computed:

```
>>> gradients
[<tf.Tensor: shape=(), dtype=float32, numpy=36.0>,
 <tf.Tensor: shape=(), dtype=float32, numpy=10.0>]
```

Perfect! Not only is the result accurate (the precision is only limited by the floating-point errors), but the `gradient()` method only goes through the recorded computations once (in reverse order), no matter how many variables there are, so it is incredibly efficient. It's like magic!



To save memory, only put the strict minimum inside the `tf.GradientTape()` block. Alternatively, pause recording by creating a `with tape.stop_recording()` block inside the `tf.GradientTape()` block.

The tape is automatically erased immediately after you call its `gradient()` method, so you will get an exception if you try to call `gradient()` twice:

```
with tf.GradientTape() as tape:
    z = f(w1, w2)

dz_dw1 = tape.gradient(z, w1) # returns tensor 36.0
dz_dw2 = tape.gradient(z, w2) # raises a RuntimeError!
```

If you need to call `gradient()` more than once, you must make the tape persistent and delete it each time you are done with it to free resources:¹⁴

```
with tf.GradientTape(persistent=True) as tape:
    z = f(w1, w2)

dz_dw1 = tape.gradient(z, w1) # returns tensor 36.0
dz_dw2 = tape.gradient(z, w2) # returns tensor 10.0, works fine now!
del tape
```

¹⁴ If the tape goes out of scope, for example when the function that used it returns, Python's garbage collector will delete it for you.

By default, the tape will only track operations involving variables, so if you try to compute the gradient of z with regard to anything other than a variable, the result will be `None`:

```
c1, c2 = tf.constant(5.), tf.constant(3.)
with tf.GradientTape() as tape:
    z = f(c1, c2)

gradients = tape.gradient(z, [c1, c2]) # returns [None, None]
```

However, you can force the tape to watch any tensors you like, to record every operation that involves them. You can then compute gradients with regard to these tensors, as if they were variables:

```
with tf.GradientTape() as tape:
    tape.watch(c1)
    tape.watch(c2)
    z = f(c1, c2)

gradients = tape.gradient(z, [c1, c2]) # returns [tensor 36., tensor 10.]
```

This can be useful in some cases, like if you want to implement a regularization loss that penalizes activations that vary a lot when the inputs vary little: the loss will be based on the gradient of the activations with regard to the inputs. Since the inputs are not variables, you would need to tell the tape to watch them.

Most of the time a gradient tape is used to compute the gradients of a single value (usually the loss) with regard to a set of values (usually the model parameters). This is where reverse-mode autodiff shines, as it just needs to do one forward pass and one reverse pass to get all the gradients at once. If you try to compute the gradients of a vector, for example a vector containing multiple losses, then TensorFlow will compute the gradients of the vector's sum. So if you ever need to get the individual gradients (e.g., the gradients of each loss with regard to the model parameters), you must call the tape's `jacobian()` method: it will perform reverse-mode autodiff once for each loss in the vector (all in parallel by default). It is even possible to compute second-order partial derivatives (the Hessians, i.e., the partial derivatives of the partial derivatives), but this is rarely needed in practice (see the “Computing Gradients Using Autodiff” section of the notebook for an example).

In some cases you may want to stop gradients from backpropagating through some part of your neural network. To do this, you must use the `tf.stop_gradient()` function. The function returns its inputs during the forward pass (like `tf.identity()`), but it does not let gradients through during backpropagation (it acts like a constant):

```
def f(w1, w2):
    return 3 * w1 ** 2 + tf.stop_gradient(2 * w1 * w2)

with tf.GradientTape() as tape:
    z = f(w1, w2) # the forward pass is not affected by stop_gradient()
```

```
gradients = tape.gradient(z, [w1, w2]) # returns [tensor 30., None]
```

Finally, you may occasionally run into some numerical issues when computing gradients. For example, if you compute the gradients of the square root function at $x = 10^{-50}$, the result will be infinite. In reality, the slope at that point is not infinite, but it's more than 32-bit floats can handle:

```
>>> x = tf.Variable(1e-50)
>>> with tf.GradientTape() as tape:
...     z = tf.sqrt(x)
...
>>> tape.gradient(z, [x])
[<tf.Tensor: shape=(), dtype=float32, numpy=inf>]
```

To solve this, it's often a good idea to add a tiny value to x (such as 10^{-6}) when computing its square root.

The exponential function is also a frequent source of headaches, as it grows extremely fast. For example, the way `my_softplus()` was defined earlier is not numerically stable. If you compute `my_softplus(100.0)`, you will get infinity rather than the correct result (about 100). But it's possible to rewrite the function to make it numerically stable: the softplus function is defined as $\log(1 + \exp(z))$, which is also equal to $\log(1 + \exp(-|z|)) + \max(z, 0)$ (see the notebook for the mathematical proof) and the advantage of this second form is that the exponential term cannot explode. So here's a better implementation of the `my_softplus()` function:

```
def my_softplus(z):
    return tf.math.log(1 + tf.exp(-tf.abs(z))) + tf.maximum(0., z)
```

In some rare cases, a numerically stable function may still have numerically unstable gradients. In such cases, you will have to tell TensorFlow which equation to use for the gradients, rather than letting it use autodiff. For this, you must use the `@tf.custom_gradient` decorator when defining the function, and return both the function's usual result plus a function that computes the gradients. For example, let's update the `my_softplus()` function to also return a numerically stable gradients function:

```
@tf.custom_gradient
def my_softplus(z):
    def my_softplus_gradients(grads): # grads = backprop'ed from upper layers
        return grads * (1 - 1 / (1 + tf.exp(z))) # stable grads of softplus

    result = tf.math.log(1 + tf.exp(-tf.abs(z))) + tf.maximum(0., z)
    return result, my_softplus_gradients
```

If you know differential calculus (see the tutorial notebook on this topic), you can find that the derivative of $\log(1 + \exp(z))$ is $\exp(z) / (1 + \exp(z))$. But this form is not stable: for large values of z , it ends up computing infinity divided by infinity, which return NaN. However, with a bit of algebraic manipulation, you can show that it's also

equal to $1 - 1 / (1 + \exp(z))$, which is stable. The `my_softplus_gradients()` function uses this equation to compute the gradients. Note that this function will receive as input the gradients that were backpropagated so far, down to the `my_softplus()` function; and according to the chain rule, we must multiply them with this function's gradients.

Now when we compute the gradients of the `my_softplus()` function, we get the proper result, even for large input values.

Congratulations! You can now compute the gradients of any function (provided it is differentiable at the point where you compute it), even blocking backpropagation when needed, and write your own gradient functions! This is probably more flexibility than you will ever need, even if you build your own custom training loops, as we will see now.

Custom Training Loops

In some cases, the `fit()` method may not be flexible enough for what you need to do. For example, the [Wide & Deep paper](#) we discussed in [Chapter 10](#) uses two different optimizers: one for the wide path and the other for the deep path. Since the `fit()` method only uses one optimizer (the one that we specify when compiling the model), implementing this paper requires writing your own custom loop.

You may also like to write custom training loops simply to feel more confident that they do precisely what you intend them to do (perhaps you are unsure about some details of the `fit()` method). It can sometimes feel safer to make everything explicit. However, remember that writing a custom training loop will make your code longer, more error-prone, and harder to maintain.



Unless you're learning or you really need the extra flexibility, you should prefer using the `fit()` method rather than implementing your own training loop, especially if you work in a team.

First, let's build a simple model. No need to compile it, since we will handle the training loop manually:

```
l2_reg = tf.keras.regularizers.l2(0.05)
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(30, activation="relu", kernel_initializer="he_normal",
                         kernel_regularizer=l2_reg),
    tf.keras.layers.Dense(1, kernel_regularizer=l2_reg)
])
```

Next, let's create a tiny function that will randomly sample a batch of instances from the training set (in [Chapter 13](#) we will discuss the Data API, which offers a much better alternative):

```
def random_batch(X, y, batch_size=32):
    idx = np.random.randint(len(X), size=batch_size)
    return X[idx], y[idx]
```

Let's also define a function that will display the training status, including the number of steps, the total number of steps, the mean loss since the start of the epoch (i.e., we will use the `Mean` metric to compute it), and other metrics:

```
def print_status_bar(step, total, loss, metrics=None):
    metrics = " - ".join([f"{m.name}: {m.result():.4f}" for m in [loss] + (metrics or [])])
    end = "" if step < total else "\n"
    print(f"\r{step}/{total} - {metrics}, end=end)
```

This code is self-explanatory, unless you are unfamiliar with Python string formatting: `{m.result():.4f}` will format the metric's result as a float with four digits after the decimal point; moreover, using `\r` (carriage return) along with `end=""` ensures that the status bar always gets printed on the same line.

With that, let's get down to business! First, we need to define some hyperparameters and choose the optimizer, the loss function, and the metrics (just the MAE in this example):

```
n_epochs = 5
batch_size = 32
n_steps = len(X_train) // batch_size
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)
loss_fn = tf.keras.losses.mean_squared_error
mean_loss = tf.keras.metrics.Mean(name="mean_loss")
metrics = [tf.keras.metrics.MeanAbsoluteError()]
```

And now we are ready to build the custom loop!

```
for epoch in range(1, n_epochs + 1):
    print("Epoch {}/{}".format(epoch, n_epochs))
    for step in range(1, n_steps + 1):
        X_batch, y_batch = random_batch(X_train_scaled, y_train)
        with tf.GradientTape() as tape:
            y_pred = model(X_batch, training=True)
            main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
            loss = tf.add_n([main_loss] + model.losses)

            gradients = tape.gradient(loss, model.trainable_variables)
            optimizer.apply_gradients(zip(gradients, model.trainable_variables))
            mean_loss(loss)
            for metric in metrics:
                metric(y_batch, y_pred)
```

```
print_status_bar(step, n_steps, mean_loss, metrics)

for metric in [mean_loss] + metrics:
    metric.reset_states()
```

There's a lot going on in this code, so let's walk through it:

- We create two nested loops: one for the epochs, the other for the batches within an epoch.
- Then we sample a random batch from the training set.
- Inside the `tf.GradientTape()` block, we make a prediction for one batch, using the model as a function, and we compute the loss: it is equal to the main loss plus the other losses (in this model, there is one regularization loss per layer). Since the `mean_squared_error()` function returns one loss per instance, we compute the mean over the batch using `tf.reduce_mean()` (if you wanted to apply different weights to each instance, this is where you would do it). The regularization losses are already reduced to a single scalar each, so we just need to sum them (using `tf.add_n()`, which sums multiple tensors of the same shape and data type).
- Next, we ask the tape to compute the gradients of the loss with regard to each trainable variable—*not* all variables!—and we apply them to the optimizer to perform a Gradient Descent step.
- Then we update the mean loss and the metrics (over the current epoch), and we display the status bar.
- At the end of each epoch, we reset the states of the mean loss and the metrics.

If you want to apply Gradient Clipping (see [Chapter 11](#)), just set the optimizer's `clip_norm` or `clipvalue` hyperparameter. If you want to apply any other transformation to the gradients, simply do so before calling the `apply_gradients()` method. And if you want to add weight constraints to your model (e.g., by setting `kernel_constraint` or `bias_constraint` when creating a layer), you should update the training loop to apply these constraints just after `apply_gradients()`, like so:

```
for variable in model.variables:
    if variable.constraint is not None:
        variable.assign(variable.constraint(variable))
```



Don't forget to set `training=True` when calling the model in the training loop, especially if your model behaves differently during training and testing (e.g., if it uses BatchNormalization or Dropout). If it's a custom model, make sure to propagate the `training` argument to the layers that your model calls.

As you can see, there are quite a lot of things you need to get right, and it's easy to make a mistake. But on the bright side, you get full control.

Now that you know how to customize any part of your models¹⁵ and training algorithms, let's see how you can use TensorFlow's automatic graph generation feature: it can speed up your custom code considerably, and it will also make it portable to any platform supported by TensorFlow (see [Chapter 19](#)).

TensorFlow Functions and Graphs

Back in TensorFlow 1, graphs were unavoidable (as were the complexities that came with them) because they were a central part of TensorFlow's API. Since TensorFlow 2 (released in 2019), graphs are still there, but not as central, and they're much (much!) simpler to use. To show just how simple, let's start with a trivial function that computes the cube of its input:

```
def cube(x):
    return x ** 3
```

We can obviously call this function with a Python value, such as an int or a float, or we can call it with a tensor:

```
>>> cube(2)
8
>>> cube(tf.constant(2.0))
<tf.Tensor: shape=(), dtype=float32, numpy=8.0>
```

Now, let's use `tf.function()` to convert this Python function to a *TensorFlow Function*:

```
>>> tf_cube = tf.function(cube)
>>> tf_cube
<tensorflow.python.eager.def_function.Function at 0x7fbfe0c54d50>
```

This TF Function can then be used exactly like the original Python function, and it will return the same result (but always as tensors):

```
>>> tf_cube(2)
<tf.Tensor: shape=(), dtype=int32, numpy=8>
>>> tf_cube(tf.constant(2.0))
<tf.Tensor: shape=(), dtype=float32, numpy=8.0>
```

Under the hood, `tf.function()` analyzed the computations performed by the `cube()` function and generated an equivalent computation graph! As you can see, it was rather painless (we will see how this works shortly). Alternatively, we could have used `tf.function` as a decorator; this is actually more common:

¹⁵ With the exception of optimizers, as very few people ever customize these; see the "Custom Optimizers" section in the notebook for an example.

```
@tf.function  
def tf_cube(x):  
    return x ** 3
```

The original Python function is still available via the TF Function's `python_function` attribute, in case you ever need it:

```
>>> tf_cube.python_function(2)  
8
```

TensorFlow optimizes the computation graph, pruning unused nodes, simplifying expressions (e.g., $1 + 2$ would get replaced with 3), and more. Once the optimized graph is ready, the TF Function efficiently executes the operations in the graph, in the appropriate order (and in parallel when it can). As a result, a TF Function will usually run much faster than the original Python function, especially if it performs complex computations.¹⁶ Most of the time you will not really need to know more than that: when you want to boost a Python function, just transform it into a TF Function. That's all!

Moreover, if you set `jit_compile=True` when calling `tf.function()`, then TensorFlow will use *Accelerated Linear Algebra* (XLA) to compile dedicated kernels for your graph, often fusing multiple operations. For example, if your TF function calls `tf.reduce_sum(a * b + c)`, then without XLA the function would first need to compute `a * b` and store the result in a temporary variable, then add `c` to that variable, and lastly call `tf.reduce_sum()` on the result. With XLA, the whole computation gets compiled into a single kernel, which will compute `tf.reduce_sum(a * b + c)` in one shot, without using any large temporary variable. Not only will this be much faster, it will also use dramatically less RAM.

When you write a custom loss function, a custom metric, a custom layer, or any other custom function and you use it in a Keras model (as we did throughout this chapter), Keras automatically converts your function into a TF Function—no need to use `tf.function()`. So most of the time, the magic is 100% transparent. And if you want Keras to use XLA, you just need to set `jit_compile=True` when calling the `compile()` method. Easy!



You can tell Keras *not* to convert your Python functions to TF Functions by setting `dynamic=True` when creating a custom layer or a custom model. Alternatively, you can set `run_eagerly=True` when calling the model's `compile()` method.

¹⁶ However, in this trivial example, the computation graph is so small that there is nothing at all to optimize, so `tf_cube()` actually runs much slower than `cube()`.

By default, a TF Function generates a new graph for every unique set of input shapes and data types and caches it for subsequent calls. For example, if you call `tf_cube(tf.constant(10))`, a graph will be generated for int32 tensors of shape `[]`. Then if you call `tf_cube(tf.constant(20))`, the same graph will be reused. But if you then call `tf_cube(tf.constant([10, 20]))`, a new graph will be generated for int32 tensors of shape `[2]`. This is how TF Functions handle polymorphism (i.e., varying argument types and shapes). However, this is only true for tensor arguments: if you pass numerical Python values to a TF Function, a new graph will be generated for every distinct value: for example, calling `tf_cube(10)` and `tf_cube(20)` will generate two graphs.



If you call a TF Function many times with different numerical Python values, then many graphs will be generated, slowing down your program and using up a lot of RAM (you must delete the TF Function to release it). Python values should be reserved for arguments that will have few unique values, such as hyperparameters like the number of neurons per layer. This allows TensorFlow to better optimize each variant of your model.

AutoGraph and Tracing

So how does TensorFlow generate graphs? It starts by analyzing the Python function's source code to capture all the control flow statements, such as `for` loops, `while` loops, and `if` statements, as well as `break`, `continue`, and `return` statements. This first step is called *AutoGraph*. The reason TensorFlow has to analyze the source code is that Python does not provide any other way to capture control flow statements: it offers magic methods like `__add__()` and `__mul__()` to capture operators like `+` and `*`, but there are no `__while__()` or `__if__()` magic methods. After analyzing the function's code, AutoGraph outputs an upgraded version of that function in which all the control flow statements are replaced by the appropriate TensorFlow operations, such as `tf.while_loop()` for loops and `tf.cond()` for `if` statements. For example, in [Figure 12-4](#), AutoGraph analyzes the source code of the `sum_squares()` Python function, and it generates the `tf_sum_squares()` function. In this function, the `for` loop is replaced by the definition of the `loop_body()` function (containing the body of the original `for` loop), followed by a call to the `for_stmt()` function. This call will build the appropriate `tf.while_loop()` operation in the computation graph.

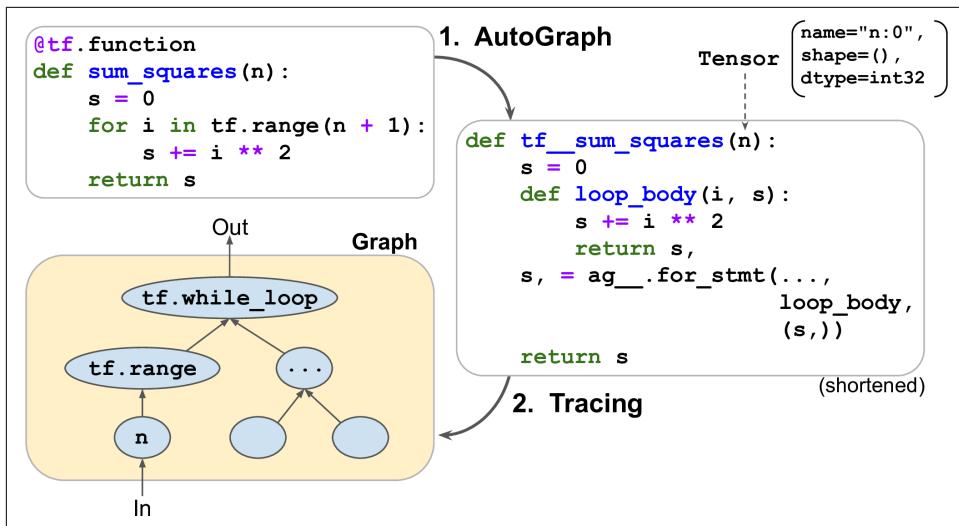


Figure 12-4. How TensorFlow generates graphs using AutoGraph and tracing

Next, TensorFlow calls this “upgraded” function, but instead of passing the argument, it passes a *symbolic tensor*—a tensor without any actual value, only a name, a data type, and a shape. For example, if you call `sum_squares(tf.constant(10))`, then the `tf_sum_squares()` function will be called with a symbolic tensor of type `int32` and shape `[]`. The function will run in *graph mode*, meaning that each TensorFlow operation will add a node in the graph to represent itself and its output tensor(s) (as opposed to the regular mode, called *eager execution*, or *eager mode*). In graph mode, TF operations do not perform any computations. Graph mode was the default mode in TensorFlow 1. In Figure 12-4, you can see the `tf_sum_squares()` function being called with a symbolic tensor as its argument (in this case, an `int32` tensor of shape `[]`) and the final graph being generated during tracing. The nodes represent operations, and the arrows represent tensors (both the generated function and the graph are simplified).



To view the generated function’s source code, you can call `tf.autograph.to_code(sum_squares.python_function)`. The code is not meant to be pretty, but it can sometimes help for debugging.

TF Function Rules

Most of the time, converting a Python function that performs TensorFlow operations into a TF Function is trivial: decorate it with `@tf.function` or let Keras take care of it for you. However, there are a few rules to respect:

- If you call any external library, including NumPy or even the standard library, this call will run only during tracing; it will not be part of the graph. Indeed, a TensorFlow graph can only include TensorFlow constructs (tensors, operations, variables, datasets, and so on). So, make sure you use `tf.reduce_sum()` instead of `np.sum()`, `tf.sort()` instead of the built-in `sorted()` function, and so on (unless you really want the code to run only during tracing). This has a few additional implications:
 - If you define a TF Function `f(x)` that just returns `np.random.rand()`, a random number will only be generated when the function is traced, so `f(tf.constant(2.))` and `f(tf.constant(3.))` will return the same random number, but `f(tf.constant([2., 3.]))` will return a different one. If you replace `np.random.rand()` with `tf.random.uniform([])`, then a new random number will be generated upon every call, since the operation will be part of the graph.
 - If your non-TensorFlow code has side effects (such as logging something or updating a Python counter), then you should not expect those side effects to occur every time you call the TF Function, as they will only occur when the function is traced.
 - You can wrap arbitrary Python code in a `tf.py_function()` operation, but doing so will hinder performance, as TensorFlow will not be able to do any graph optimization on this code. It will also reduce portability, as the graph will only run on platforms where Python is available (and where the right libraries are installed).
- You can call other Python functions or TF Functions, but they should follow the same rules, as TensorFlow will capture their operations in the computation graph. Note that these other functions do not need to be decorated with `@tf.function`.
- If the function creates a TensorFlow variable (or any other stateful TensorFlow object, such as a dataset or a queue), it must do so upon the very first call, and only then, or else you will get an exception. It is usually preferable to create variables outside of the TF Function (e.g., in the `build()` method of a custom layer). If you want to assign a new value to the variable, make sure you call its `assign()` method, instead of using the `=` operator.
- The source code of your Python function should be available to TensorFlow. If the source code is unavailable (for example, if you define your function in the Python shell, which does not give access to the source code, or if you deploy only the compiled `*.pyc` Python files to production), then the graph generation process will fail or have limited functionality.

- TensorFlow will only capture for loops that iterate over a tensor or a `tf.data.Dataset` (see [Chapter 13](#)). So make sure you use `for i in tf.range(x)` rather than `for i in range(x)`, or else the loop will not be captured in the graph. Instead, it will run during tracing. (This may be what you want if the for loop is meant to build the graph, for example to create each layer in a neural network.)
- As always, for performance reasons, you should prefer a vectorized implementation whenever you can, rather than using loops.

It's time to sum up! In this chapter we started with a brief overview of TensorFlow, then we looked at TensorFlow's low-level API, including tensors, operations, variables, and special data structures. We then used these tools to customize almost every component in `tf.keras`. Finally, we looked at how TF Functions can boost performance, how graphs are generated using AutoGraph and tracing, and what rules to follow when you write TF Functions (if you would like to open the black box a bit further and explore the generated graphs, you will find technical details in [Appendix D](#)).

In the next chapter, we will look at how to efficiently load and preprocess data with TensorFlow.

Exercises

1. How would you describe TensorFlow in a short sentence? What are its main features? Can you name other popular Deep Learning libraries?
2. Is TensorFlow a drop-in replacement for NumPy? What are the main differences between the two?
3. Do you get the same result with `tf.range(10)` and `tf.constant(np.arange(10))`?
4. Can you name six other data structures available in TensorFlow, beyond regular tensors?
5. A custom loss function can be defined by writing a function or by subclassing the `tf.keras.losses.Loss` class. When would you use each option?
6. Similarly, a custom metric can be defined in a function or a subclass of `tf.keras.metrics.Metric`. When would you use each option?
7. When should you create a custom layer versus a custom model?
8. What are some use cases that require writing your own custom training loop?
9. Can custom Keras components contain arbitrary Python code, or must they be convertible to TF Functions?

10. What are the main rules to respect if you want a function to be convertible to a TF Function?
11. When would you need to create a dynamic Keras model? How do you do that? Why not make all your models dynamic?
12. Implement a custom layer that performs *Layer Normalization* (we will use this type of layer in [Chapter 15](#)):
 - a. The `build()` method should define two trainable weights α and β , both of shape `input_shape[-1:]` and data type `tf.float32`. α should be initialized with 1s, and β with 0s.
 - b. The `call()` method should compute the mean μ and standard deviation σ of each instance's features. For this, you can use `tf.nn.moments(inputs, axes=-1, keepdims=True)`, which returns the mean μ and the variance σ^2 of all instances (compute the square root of the variance to get the standard deviation). Then the function should compute and return $\alpha \otimes (X - \mu) / (\sigma + \epsilon) + \beta$, where \otimes represents itemwise multiplication (*) and ϵ is a smoothing term (small constant to avoid division by zero, e.g., 0.001).
 - c. Ensure that your custom layer produces the same (or very nearly the same) output as the `tf.keras.layers.LayerNormalization` layer.
13. Train a model using a custom training loop to tackle the Fashion MNIST dataset (see [Chapter 10](#)).
 - a. Display the epoch, iteration, mean training loss, and mean accuracy over each epoch (updated at each iteration), as well as the validation loss and accuracy at the end of each epoch.
 - b. Try using a different optimizer with a different learning rate for the upper layers and the lower layers.

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab3>.

Loading and Preprocessing Data with TensorFlow

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 13th chapter of the final book. The GitHub repo is <https://github.com/ageron/handsonml3>.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

In [Chapter 2](#), we saw that loading and preprocessing data is an important part of any Machine Learning project. We used Pandas to load and explore the (modified) California housing dataset—which was stored in a CSV file—and we applied Scikit-Learn’s transformers for preprocessing. These tools are quite convenient, and you will probably be using them often, especially when exploring and experimenting with data.

However, when training TensorFlow models on large datasets, you may prefer to use TensorFlow’s own data loading and preprocessing API, called `tf.data`. Indeed, it is capable of loading and preprocessing data extremely efficiently, reading from multiple files in parallel using multithreading and queuing, shuffling and batching samples, and more. Plus, it can do all of this on the fly—it loads and preprocesses the next batch of data across multiple CPU cores, while your GPUs or TPUs are busy training the current batch of data.

The `tf.data` API lets you handle datasets that don't fit in memory, and it allows you to make full use of your hardware resources, thereby speeding up training. Off the shelf, the `tf.data` API can read from text files (such as CSV files), binary files with fixed-size records, and binary files that use TensorFlow's TFRecord format, which supports records of varying sizes.

TFRecord is a flexible and efficient binary format usually containing protocol buffers (an open source binary format). The `tf.data` API also has support for reading from SQL databases. Moreover, many open source extensions are available to read from all sorts of data sources, such as Google's BigQuery service (see <https://tensorflow.org/io>).

Keras also comes with powerful yet easy-to-use preprocessing layers that can be embedded in your models: this way, when you deploy a model to production, it will be able to ingest raw data directly, without having to add any additional preprocessing code. This eliminates the risk of mismatch between the preprocessing code used during training and the preprocessing code used in production, which would likely cause *training/serving skew*. And if you deploy your model in multiple apps coded in different programming languages, you won't have to reimplement the same preprocessing code multiple times, which also reduces the risk of mismatch.

As we will see, both APIs can be used jointly, for example to benefit from the efficient data loading offered by `tf.data`, and the convenience of the Keras preprocessing layers.

In this chapter, we will first cover the `tf.data` API and the TFRecord format. Then we will explore the Keras preprocessing layers, and how to use them with the `tf.data` API. Lastly, we will take a quick look at a few related libraries that you may find useful for loading and preprocessing data, such as TensorFlow Datasets and TensorFlow Hub. So, let's get started!

The `tf.data` API

The whole `tf.data` API revolves around the concept of a `tf.data.Dataset`: this represents a sequence of data items. Usually you will use datasets that gradually read data from disk, but for simplicity let's create a dataset from a simple data tensor using `tf.data.Dataset.from_tensor_slices()`:

```
>>> import tensorflow as tf
>>> X = tf.range(10) # any data tensor
>>> dataset = tf.data.Dataset.from_tensor_slices(X)
>>> dataset
<TensorSliceDataset shapes: (), types: tf.int32>
```

The `from_tensor_slices()` function takes a tensor and creates a `tf.data.Dataset` whose elements are all the slices of `X` along the first dimension, so this dataset contains 10 items: tensors 0, 1, 2, ..., 9. In this case we would have obtained the same

dataset if we had used `tf.data.Dataset.range(10)` (except the elements would be 64-bit integers instead of 32-bit integers).

You can simply iterate over a dataset's items like this:

```
>>> for item in dataset:  
...     print(item)  
...  
tf.Tensor(0, shape=(), dtype=int32)  
tf.Tensor(1, shape=(), dtype=int32)  
[...]  
tf.Tensor(9, shape=(), dtype=int32)
```



The `tf.data` API is a streaming API: you can very efficiently iterate through a dataset's items, but the API is not designed for indexing or slicing.

A dataset may also contain tuples of tensors, or dictionaries of name/tensor pairs, or even nested tuples and dictionaries of tensors. When slicing a tuple, a dictionary, or a nested structure, the dataset will only slice the tensors it contains, while preserving the tuple/dictionary structure. For example:

```
>>> X_nested = {"a": ([1, 2, 3], [4, 5, 6]), "b": [7, 8, 9]}  
>>> dataset = tf.data.Dataset.from_tensor_slices(X_nested)  
>>> for item in dataset:  
...     print(item)  
...  
{'a': (<tf.Tensor: [...]=1>, <tf.Tensor: [...]=4>), 'b': <tf.Tensor: [...]=7>}  
{'a': (<tf.Tensor: [...]=2>, <tf.Tensor: [...]=5>), 'b': <tf.Tensor: [...]=8>}  
{'a': (<tf.Tensor: [...]=3>, <tf.Tensor: [...]=6>), 'b': <tf.Tensor: [...]=9>}
```

Chaining Transformations

Once you have a dataset, you can apply all sorts of transformations to it by calling its transformation methods. Each method returns a new dataset, so you can chain transformations like this (this chain is illustrated in [Figure 13-1](#)):

```
>>> dataset = tf.data.Dataset.from_tensor_slices(tf.range(10))  
>>> dataset = dataset.repeat(3).batch(7)  
>>> for item in dataset:  
...     print(item)  
...  
tf.Tensor([0 1 2 3 4 5 6], shape=(7,), dtype=int32)  
tf.Tensor([7 8 9 0 1 2 3], shape=(7,), dtype=int32)  
tf.Tensor([4 5 6 7 8 9 0], shape=(7,), dtype=int32)  
tf.Tensor([1 2 3 4 5 6 7], shape=(7,), dtype=int32)  
tf.Tensor([8 9], shape=(2,), dtype=int32)
```

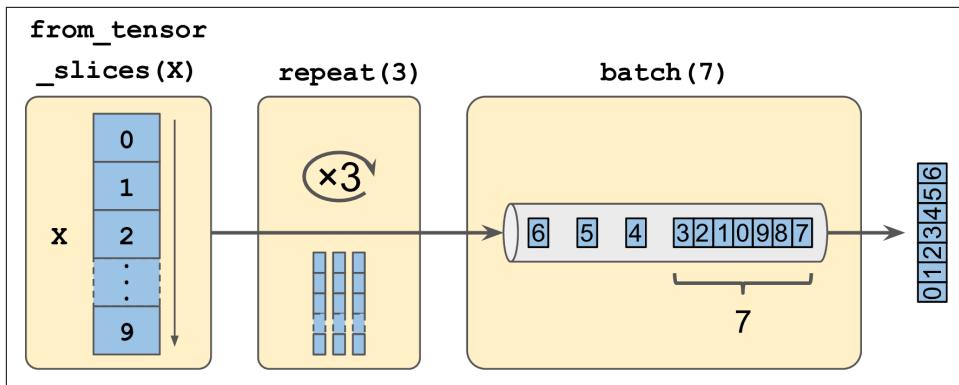


Figure 13-1. Chaining dataset transformations

In this example, we first call the `repeat()` method on the original dataset, and it returns a new dataset that will repeat the items of the original dataset three times. Of course, this will not copy all the data in memory three times! If you call this method with no arguments, the new dataset will repeat the source dataset forever, so the code that iterates over the dataset will have to decide when to stop.

Then we call the `batch()` method on this new dataset, and again this creates a new dataset. This one will group the items of the previous dataset in batches of seven items.

Finally, we iterate over the items of this final dataset. The `batch()` method had to output a final batch of size two instead of seven, but you can call `batch()` with `drop_remainder=True` if you want it to drop this final batch, such that all batches have the exact same size.



The dataset methods do *not* modify datasets—they create new ones. So make sure to keep a reference to these new datasets (e.g., with `dataset = ...`), or else nothing will happen.

You can also transform the items by calling the `map()` method. For example, this creates a new dataset with all batches multiplied by two:

```
>>> dataset = dataset.map(lambda x: x * 2) # x is a batch
>>> for item in dataset:
...     print(item)
...
tf.Tensor([ 0  2  4  6  8 10 12], shape=(7,), dtype=int32)
tf.Tensor([14 16 18  0  2  4  6], shape=(7,), dtype=int32)
[...]
```

This `map()` method is the one you will call to apply any preprocessing to your data. Sometimes this will include computations that can be quite intensive, such as reshaping or rotating an image, so you will usually want to spawn multiple threads to speed things up. This can be done by setting the `num_parallel_calls` argument to the number of threads to run, or to `tf.data.AUTOTUNE`. Note that the function you pass to the `map()` method must be convertible to a TF Function (see [Chapter 12](#)).

It is also possible to simply filter the dataset using the `filter()` method. For example, this code creates a dataset that only contains the batches whose sum is greater than 50:

```
>>> dataset = dataset.filter(lambda x: tf.reduce_sum(x) > 50)
>>> for item in dataset:
...     print(item)
...
tf.Tensor([14 16 18  0  2  4  6], shape=(7,), dtype=int32)
tf.Tensor([ 8 10 12 14 16 18  0], shape=(7,), dtype=int32)
tf.Tensor([-2  4  6  8 10 12 14], shape=(7,), dtype=int32)
```

You will often want to look at just a few items from a dataset. You can use the `take()` method for that:

```
>>> for item in dataset.take(2):
...     print(item)
...
tf.Tensor([14 16 18  0  2  4  6], shape=(7,), dtype=int32)
tf.Tensor([ 8 10 12 14 16 18  0], shape=(7,), dtype=int32)
```

Shuffling the Data

As we discussed in [Chapter 4](#), Gradient Descent works best when the instances in the training set are independent and identically distributed (IID). A simple way to ensure this is to shuffle the instances, using the `shuffle()` method. It will create a new dataset that will start by filling up a buffer with the first items of the source dataset. Then, whenever it is asked for an item, it will pull one out randomly from the buffer and replace it with a fresh one from the source dataset, until it has iterated entirely through the source dataset. At this point it will continue to pull out items randomly from the buffer until it is empty. You must specify the buffer size, and it is important to make it large enough, or else shuffling will not be very effective.¹ Just don't exceed the amount of RAM you have, and even if you have plenty of it, there's no need to go beyond the dataset's size. You can provide a random seed if you want the same random order every time you run your program. For example, the following code

¹ Imagine a sorted deck of cards on your left: suppose you just take the top three cards and shuffle them, then pick one randomly and put it to your right, keeping the other two in your hands. Take another card on your left, shuffle the three cards in your hands and pick one of them randomly, and put it on your right. When you are done going through all the cards like this, you will have a deck of cards on your right: do you think it will be perfectly shuffled?

creates and displays a dataset containing the integers 0 to 9, repeated twice, shuffled using a buffer of size 4 and a random seed of 42, and batched with a batch size of 7:

```
>>> dataset = tf.data.Dataset.range(10).repeat(2)
>>> dataset = dataset.shuffle(buffer_size=4, seed=42).batch(7)
>>> for item in dataset:
...     print(item)
...
tf.Tensor([3 0 1 6 2 5 7], shape=(7,), dtype=int64)
tf.Tensor([8 4 1 9 4 2 3], shape=(7,), dtype=int64)
tf.Tensor([7 5 0 8 9 6], shape=(6,), dtype=int64)
```



If you call `repeat()` on a shuffled dataset, by default it will generate a new order at every iteration. This is generally a good idea, but if you prefer to reuse the same order at each iteration (e.g., for tests or debugging), you can set `reshuffle_each_iteration=False` when calling `shuffle()`.

For a large dataset that does not fit in memory, this simple shuffling-buffer approach may not be sufficient, since the buffer will be small compared to the dataset. One solution is to shuffle the source data itself (for example, on Linux you can shuffle text files using the `shuf` command). This will definitely improve shuffling a lot! Even if the source data is shuffled, you will usually want to shuffle it some more, or else the same order will be repeated at each epoch, and the model may end up being biased (e.g., due to some spurious patterns present by chance in the source data's order). To shuffle the instances some more, a common approach is to split the source data into multiple files, then read them in a random order during training. However, instances located in the same file will still end up close to each other. To avoid this you can pick multiple files randomly and read them simultaneously, interleaving their records. Then on top of that you can add a shuffling buffer using the `shuffle()` method. If all this sounds like a lot of work, don't worry: the `tf.data` API makes all this possible in just a few lines of code. Let's go over how you can do this.

Interleaving lines from multiple files

First, suppose you've loaded the California housing dataset, shuffled it (unless it was already shuffled), and split it into a training set, a validation set, and a test set. Then you split each set into many CSV files that each look like this (each row contains eight input features plus the target median house value):

```
MedInc,HouseAge,AveRooms,AveBedrms,Popul...,AveOccup,Lat...,Long...,MedianHouseValue
3.5214,15.0,3.050,1.107,1447.0,1.606,37.63,-122.43,1.442
5.3275,5.0,6.490,0.991,3464.0,3.443,33.69,-117.39,1.687
3.1,29.0,7.542,1.592,1328.0,2.251,38.44,-122.98,1.621
[...]
```

Let's also suppose `train_filepaths` contains the list of training file paths (and you also have `valid_filepaths` and `test_filepaths`):

```
>>> train_filepaths
['datasets/housing/my_train_00.csv', 'datasets/housing/my_train_01.csv', ...]
```

Alternatively, you could use file patterns; for example, `train_filepaths = "datasets/housing/my_train_*.csv"`. Now let's create a dataset containing only these file paths:

```
filepath_dataset = tf.data.Dataset.list_files(train_filepaths, seed=42)
```

By default, the `list_files()` function returns a dataset that shuffles the file paths. In general this is a good thing, but you can set `shuffle=False` if you do not want that for some reason.

Next, you can call the `interleave()` method to read from five files at a time and interleave their lines. You can also skip the first line of each file—which is the header row—using the `skip()` method:

```
n_readers = 5
dataset = filepath_dataset.interleave(
    lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
    cycle_length=n_readers)
```

The `interleave()` method will create a dataset that will pull five file paths from the `filepath_dataset`, and for each one it will call the function you gave it (a `lambda` in this example) to create a new dataset (in this case a `TextLineDataset`). To be clear, at this stage there will be seven datasets in all: the `filepath` dataset, the `interleave` dataset, and the five `TextLineDatasets` created internally by the `interleave` dataset. When we iterate over the `interleave` dataset, it will cycle through these five `TextLineDatasets`, reading one line at a time from each until all datasets are out of items. Then it will fetch the next five file paths from the `filepath_dataset` and interleave them the same way, and so on until it runs out of file paths. For interleaving to work best, it is preferable to have files of identical length; otherwise the end of the longest file will not be interleaved.

By default, `interleave()` does not use parallelism; it just reads one line at a time from each file, sequentially. If you want it to actually read files in parallel, you can set the `interleave()` method's `num_parallel_calls` argument to the number of threads you want (recall that the `map()` method also has this argument). You can even set it to `tf.data.AUTOTUNE` to make TensorFlow choose the right number of threads dynamically based on the available CPU. Let's look at what the dataset contains now:

```
>>> for line in dataset.take(5):
...     print(line)
...
tf.Tensor(b'4.5909,16.0,[...],33.63,-117.71,2.418', shape=(), dtype=string)
```

```
tf.Tensor(b'2.4792,24.0,[...],34.18,-118.38,2.0', shape=(), dtype=string)
tf.Tensor(b'4.2708,45.0,[...],37.48,-122.19,2.67', shape=(), dtype=string)
tf.Tensor(b'2.1856,41.0,[...],32.76,-117.12,1.205', shape=(), dtype=string)
tf.Tensor(b'4.1812,52.0,[...],33.73,-118.31,3.215', shape=(), dtype=string)
```

These are the first rows (ignoring the header row) of five CSV files, chosen randomly. Looks good!



It's possible to pass a list of file paths to the `TextLineDataset` constructor: it will go through each file in order, line by line. If you also set the `num_parallel_reads` argument to a number greater than one, then the dataset will read that number of files in parallel and interleave their lines (without having to call the `interleave()` method). However, it will *not* shuffle the files, nor will it skip the header lines.

Preprocessing the Data

Now that we have a housing dataset that returns each instance as a tensor containing a byte string, we need to do a bit of preprocessing, including parsing the strings and scaling the data. Let's implement a couple custom functions that will perform this preprocessing:

```
X_mean, X_std = [...] # mean and scale of each feature in the training set
n_inputs = 8

def parse_csv_line(line):
    defs = [0.] * n_inputs + [tf.constant([], dtype=tf.float32)]
    fields = tf.io.decode_csv(line, record_defaults=defs)
    return tf.stack(fields[:-1]), tf.stack(fields[-1:])

def preprocess(line):
    x, y = parse_csv_line(line)
    return (x - X_mean) / X_std, y
```

Let's walk through this code:

- First, the code assumes that we have precomputed the mean and standard deviation of each feature in the training set. `X_mean` and `X_std` are just 1D tensors (or NumPy arrays) containing eight floats, one per input feature. This can be done using a Scikit-Learn `StandardScaler` on a large enough random sample of the dataset. Later in this chapter, we will use a Keras preprocessing layer instead.
- The `parse_csv_line()` function takes one CSV line and parses it. To help with that, it uses the `tf.io.decode_csv()` function, which takes two arguments: the first is the line to parse, and the second is an array containing the default value for each column in the CSV file. This array (`defs`) tells TensorFlow not only the default value for each column, but also the number of columns and their types.

In this example, we tell it that all feature columns are floats and that missing values should default to 0, but we provide an empty array of type `tf.float32` as the default value for the last column (the target): the array tells TensorFlow that this column contains floats, but that there is no default value, so it will raise an exception if it encounters a missing value.

- The `tf.io.decode_csv()` function returns a list of scalar tensors (one per column), but we need to return a 1D tensor array. So we call `tf.stack()` on all tensors except for the last one (the target): this will stack these tensors into a 1D array. We then do the same for the target value: this makes it a 1D tensor array with a single value, rather than a scalar tensor. The `tf.io.decode_csv()` function is done, so it returns the input features and the target.
- Finally, the custom `preprocess()` function just calls the `parse_csv_line()` function, scales the input features by subtracting the feature means and then dividing by the feature standard deviations, and it returns a tuple containing the scaled features and the target.

Let's test this preprocessing function:

```
>>> preprocess(b'4.2083,44.0,5.3232,0.9171,846.0,2.3370,37.47,-122.2,2.782')
(<tf.Tensor: shape=(8,), dtype=float32, numpy=
 array([ 0.16579159,  1.216324   , -0.05204564, -0.39215982, -0.5277444 ,
        -0.2633488 ,  0.8543046 , -1.3072058 ], dtype=float32)>,
 <tf.Tensor: shape=(1,), dtype=float32, numpy=array([2.782], dtype=float32)>)
```

Looks good! The `preprocess()` function can convert an instance from a byte string to a nice scaled tensor, with its corresponding label. We can now use the dataset's `map()` method to apply the `preprocess()` function to each sample in the dataset.

Putting Everything Together

To make the code more reusable, let's put together everything we have discussed so far into another helper function: it will create and return a dataset that will efficiently load California housing data from multiple CSV files, preprocess it, shuffle it, and batch it (see Figure 13-2):

```
def csv_reader_dataset(filepaths, n_readers=5, n_read_threads=None,
                      n_parse_threads=5, shuffle_buffer_size=10_000, seed=42,
                      batch_size=32):
    dataset = tf.data.Dataset.list_files(filepaths, seed=seed)
    dataset = dataset.interleave(
        lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
        cycle_length=n_readers, num_parallel_calls=n_read_threads)
    dataset = dataset.map(preprocess, num_parallel_calls=n_parse_threads)
    dataset = dataset.shuffle(shuffle_buffer_size, seed=seed)
    return dataset.batch(batch_size).prefetch(1)
```

Note that we use the `prefetch()` method on the very last line. This is important for performance, as we will see now.

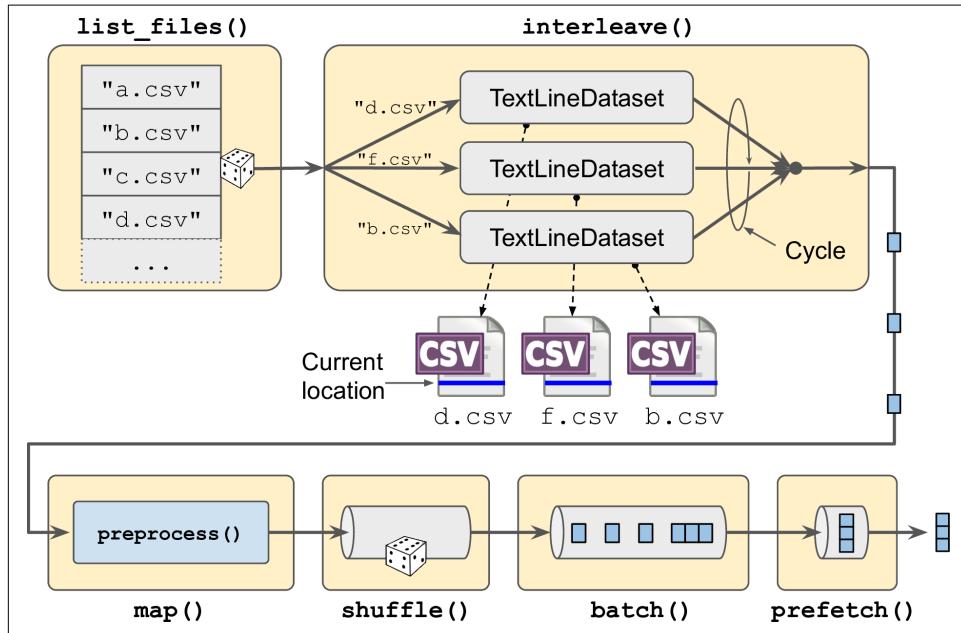


Figure 13-2. Loading and preprocessing data from multiple CSV files

Prefetching

By calling `prefetch(1)` at the end of the custom `csv_reader_dataset()` function, we are creating a dataset that will do its best to always be one batch ahead.² In other words, while our training algorithm is working on one batch, the dataset will already be working in parallel on getting the next batch ready (e.g., reading the data from disk and preprocessing it). This can improve performance dramatically, as is illustrated in Figure 13-3.

If we also ensure that loading and preprocessing are multithreaded (by setting `num_parallel_calls` when calling `interleave()` and `map()`), we can exploit multiple CPU cores and hopefully make preparing one batch of data shorter than running a

² In general, just prefetching one batch is fine, but in some cases you may need to prefetch a few more.

Alternatively, you can let TensorFlow decide automatically by passing `tf.data.AUTOTUNE` to `prefetch()`.

training step on the GPU: this way the GPU will be almost 100% utilized (except for the data transfer time from the CPU to the GPU³), and training will run much faster.

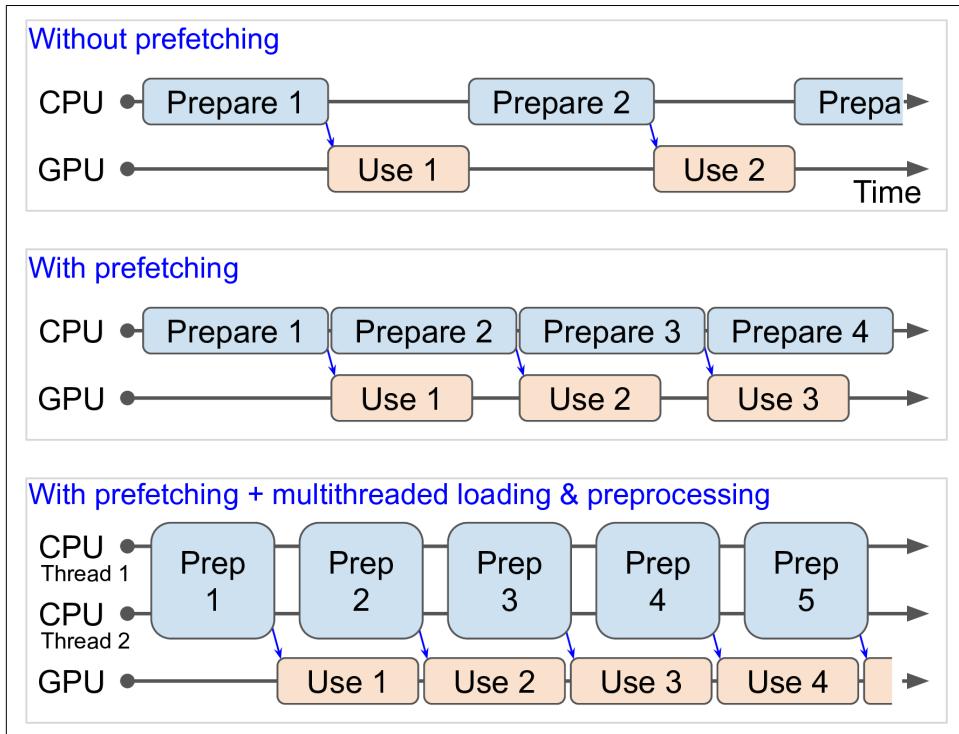


Figure 13-3. With prefetching, the CPU and the GPU work in parallel: as the GPU works on one batch, the CPU works on the next



If you plan to purchase a GPU card, its processing power and its memory size are of course very important (in particular, a large amount of RAM is crucial for large computer vision or natural language processing models). Just as important for good performance is the GPU's *memory bandwidth*; this is the number of gigabytes of data it can get into or out of its RAM per second.

If the dataset is small enough to fit in memory, you can significantly speed up training by using the dataset's `cache()` method to cache its content to RAM. You should

³ But check out the experimental `tf.data.experimental.prefetch_to_device()` function, which can prefetch data directly to the GPU. Any TensorFlow function or class with `experimental` in its name may change without warning in future versions. If an experimental function fails, try removing the word `experimental`: it may have been moved to the core API. If not, then please check the notebook, as I will ensure it contains up-to-date code.

generally do this after loading and preprocessing the data, but before shuffling, repeating, batching, and prefetching. This way, each instance will only be read and preprocessed once (instead of once per epoch), but the data will still be shuffled differently at each epoch, and the next batch will still be prepared in advance.

You have now learned how to build efficient input pipelines to load and preprocess data from multiple text files. We have discussed the most common dataset methods, but there are a few more you may want to look at, such as `concatenate()`, `zip()`, `window()`, `reduce()`, `shard()`, `flat_map()`, `apply()`, `unbatch()`, and `padded_batch()`. There are also a few more class methods, such as `from_generator()` and `from_tensors()`, which create a new dataset from a Python generator or a list of tensors, respectively. Please check the API documentation for more details. Also note that there are experimental features available in `tf.data.experimental`, many of which will likely make it to the core API in future releases (e.g., check out the `CsvDataset` class, as well as the `make_csv_dataset()` method, which takes care of inferring the type of each column).

Using the Dataset with Keras

Now we can use the custom `csv_reader_dataset()` function you wrote earlier to create a dataset for the training set, and for the validation set and the test set. The training set will be shuffled at each epoch (note that the validation set and the test set will also be shuffled, even though we don't really need that).

```
train_set = csv_reader_dataset(train_filepaths)
valid_set = csv_reader_dataset(valid_filepaths)
test_set = csv_reader_dataset(test_filepaths)
```

And now you can simply build and train a Keras model using these datasets. When you call the model's `fit()` method, you can pass `train_set` instead of `X_train`, `y_train`, and pass `validation_data=valid_set` instead of `validation_data=(X_valid, y_valid)`. The `fit()` method will take care of repeating the training dataset once per epoch, using a different random order at each epoch.

```
model = tf.keras.Sequential([...])
model.compile(loss="mse", optimizer="sgd")
model.fit(train_set, validation_data=valid_set, epochs=5)
```

Similarly, you can pass a dataset to the `evaluate()` and `predict()` methods:

```
test_mse = model.evaluate(test_set)
new_set = test_set.take(3) # pretend we have 3 new samples
y_pred = model.predict(new_set) # or you could just pass a NumPy array
```

Unlike the other sets, the `new_set` will usually not contain labels. If it does, as is the case here, Keras will ignore them. Note that in all these cases, you can still use NumPy

arrays instead of datasets if you prefer (but of course they need to have been loaded and preprocessed first).

If you want to build your own custom training loop (as in [Chapter 12](#)), you can just iterate over the training set, very naturally:

```
n_epochs = 5
for epoch in range(n_epochs):
    for X_batch, y_batch in train_set:
        [...] # perform one Gradient Descent step
```

In fact, it is even possible to create a TF Function (see [Chapter 12](#)) that trains the model for a whole epoch. This can really speed up training:

```
@tf.function
def train_one_epoch(model, optimizer, loss_fn, train_set):
    for X_batch, y_batch in train_set:
        with tf.GradientTape() as tape:
            y_pred = model(X_batch)
            main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
            loss = tf.add_n([main_loss] + model.losses)
            gradients = tape.gradient(loss, model.trainable_variables)
            optimizer.apply_gradients(zip(gradients, model.trainable_variables))

optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)
loss_fn = tf.keras.losses.mean_squared_error
for epoch in range(n_epochs):
    print("\rEpoch {} / {}".format(epoch + 1, n_epochs), end="")
    train_one_epoch(model, optimizer, loss_fn, train_set)
```

In Keras, the `steps_per_execution` argument of the `compile()` method lets you define the number of batches that the `fit()` method will process during each call to the `tf.function` it uses for training. The default is just 1, so if you set it to 50 you will often see a significant performance improvement. However, the `on_batch_*`() methods of Keras callbacks will only be called every 50 batches.

Congratulations, you now know how to build powerful input pipelines using the `tf.data` API! However, so far we have used CSV files, which are common, simple, and convenient but not really efficient, and do not support large or complex data structures (such as images or audio) very well. So let's see how to use TFRecords instead.



If you are happy with CSV files (or whatever other format you are using), you do not *have* to use TFRecords. As the saying goes, if it ain't broke, don't fix it! TFRecords are useful when the bottleneck during training is loading and parsing the data.

The TFRecord Format

The TFRecord format is TensorFlow's preferred format for storing large amounts of data and reading it efficiently. It is a very simple binary format that just contains a sequence of binary records of varying sizes (each record is comprised of a length, a CRC checksum to check that the length was not corrupted, then the actual data, and finally a CRC checksum for the data). You can easily create a TFRecord file using the `tf.io.TFRecordWriter` class:

```
with tf.io.TFRecordWriter("my_data.tfrecord") as f:  
    f.write(b"This is the first record")  
    f.write(b"And this is the second record")
```

And you can then use a `tf.data.TFRecordDataset` to read one or more TFRecord files:

```
filepaths = ["my_data.tfrecord"]  
dataset = tf.data.TFRecordDataset(filepaths)  
for item in dataset:  
    print(item)
```

This will output:

```
tf.Tensor(b'This is the first record', shape=(), dtype=string)  
tf.Tensor(b'And this is the second record', shape=(), dtype=string)
```



By default, a `TFRecordDataset` will read files one by one, but you can make it read multiple files in parallel and interleave their records by passing the constructor a list of file paths and setting `num_parallel_reads` to a number greater than one. Alternatively, you could obtain the same result by using `list_files()` and `interleave()` as we did earlier to read multiple CSV files.

Compressed TFRecord Files

It can sometimes be useful to compress your TFRecord files, especially if they need to be loaded via a network connection. You can create a compressed TFRecord file by setting the `options` argument:

```
options = tf.io.TFRecordOptions(compression_type="GZIP")  
with tf.io.TFRecordWriter("my_compressed.tfrecord", options) as f:  
    f.write(b"Compress, compress, compress!")
```

When reading a compressed TFRecord file, you need to specify the compression type:

```
dataset = tf.data.TFRecordDataset(["my_compressed.tfrecord"],  
                                 compression_type="GZIP")
```

A Brief Introduction to Protocol Buffers

Even though each record can use any binary format you want, TFRecord files usually contain serialized protocol buffers (also called *protobufs*). This is a portable, extensible, and efficient binary format developed at Google back in 2001 and made open source in 2008; protobufs are now widely used, in particular in [gRPC](#), Google's remote procedure call system. They are defined using a simple language that looks like this:

```
syntax = "proto3";
message Person {
    string name = 1;
    int32 id = 2;
    repeated string email = 3;
}
```

This protobuf definition says we are using version 3 of the protobuf format, and it specifies that each `Person` object⁴ may (optionally) have a `name` of type `string`, an `id` of type `int32`, and zero or more `email` fields, each of type `string`. The numbers 1, 2, and 3 are the field identifiers: they will be used in each record's binary representation. Once you have a definition in a `.proto` file, you can compile it. This requires `protoc`, the protobuf compiler, to generate access classes in Python (or some other language). Note that the protobuf definitions you will generally use in TensorFlow have already been compiled for you, and their Python classes are part of the TensorFlow library, so you will not need to use `protoc`. All you need to know is how to *use* protobuf access classes in Python. To illustrate the basics, let's look at a simple example that uses the access classes generated for the `Person` protobuf (the code is explained in the comments):

```
>>> from person_pb2 import Person # import the generated access class
>>> person = Person(name="Al", id=123, email=["a@b.com"]) # create a Person
>>> print(person) # display the Person
name: "Al"
id: 123
email: "a@b.com"
>>> person.name # read a field
'Al'
>>> person.name = "Alice" # modify a field
>>> person.email[0] # repeated fields can be accessed like arrays
'a@b.com'
>>> person.email.append("c@d.com") # add an email address
>>> serialized = person.SerializeToString() # serialize person to a byte string
>>> serialized
b'\n\x05Alice\x10{\x1a\x07a@b.com\x1a\x07c@d.com'
>>> person2 = Person() # create a new Person
>>> person2.ParseFromString(serialized) # parse the byte string (27 bytes long)
```

⁴ Since protobuf objects are meant to be serialized and transmitted, they are called *messages*.

```
27  
>>> person == person2 # now they are equal  
True
```

In short, we import the `Person` class generated by `protoc`, we create an instance and play with it, visualizing it and reading and writing some fields, then we serialize it using the `SerializeToString()` method. This is the binary data that is ready to be saved or transmitted over the network. When reading or receiving this binary data, we can parse it using the `ParseFromString()` method, and we get a copy of the object that was serialized.⁵

We could save the serialized `Person` object to a `TFRecord` file, then we could load and parse it: everything would work fine. However, `ParseFromString()` is not a TensorFlow operations, so you couldn't use it in a preprocessing function in a `tf.data` pipeline (except by wrapping it in a `tf.py_function()` operation, which would make the code slower and less portable, as we saw in [Chapter 12](#)). However, you could use the `tf.io.decode_proto()` function, which can parse any protobuf you want, provided you give it the protobuf definition (see the notebook for an example). That said, in practice you will generally want to use instead the predefined protobufs for which TensorFlow provides dedicated parsing operations. Let's look at these predefined protobufs now.

TensorFlow Protobufs

The main protobuf typically used in a `TFRecord` file is the `Example` protobuf, which represents one instance in a dataset. It contains a list of named features, where each feature can either be a list of byte strings, a list of floats, or a list of integers. Here is the protobuf definition (from TensorFlow's source code):

```
syntax = "proto3";  
message BytesList { repeated bytes value = 1; }  
message FloatList { repeated float value = 1 [packed = true]; }  
message Int64List { repeated int64 value = 1 [packed = true]; }  
message Feature {  
    oneof kind {  
        BytesList bytes_list = 1;  
        FloatList float_list = 2;  
        Int64List int64_list = 3;  
    }  
};  
message Features { map<string, Feature> feature = 1; };  
message Example { Features features = 1; };
```

⁵ This chapter contains the bare minimum you need to know about protobufs to use `TFRecords`. To learn more about protobufs, please visit <https://homl.info/protobuf>.

The definitions of `BytesList`, `FloatList`, and `Int64List` are straightforward enough. Note that `[packed = true]` is used for repeated numerical fields, for a more efficient encoding. A `Feature` contains either a `BytesList`, a `FloatList`, or an `Int64List`. A `Features` (with an `s`) contains a dictionary that maps a feature name to the corresponding feature value. And finally, an `Example` contains only a `Features` object.



Why was `Example` even defined, since it contains no more than a `Features` object? Well, TensorFlow's developers may one day decide to add more fields to it. As long as the new `Example` definition still contains the `features` field, with the same ID, it will be backward compatible. This extensibility is one of the great features of protobufs.

Here is how you could create a `tf.train.Example` representing the same person as earlier:

```
from tensorflow.train import BytesList, FloatList, Int64List
from tensorflow.train import Feature, Features, Example

person_example = Example(
    features=Features(
        feature={
            "name": Feature(bytes_list=BytesList(value=[b"Alice"])),
            "id": Feature(int64_list=Int64List(value=[123])),
            "emails": Feature(bytes_list=BytesList(value=[b"a@b.com",
                b"c@d.com"]))
        })
)
```

The code is a bit verbose and repetitive, but you could easily wrap it inside a small helper function. Now that we have an `Example` protobuf, we can serialize it by calling its `SerializeToString()` method, then write the resulting data to a TFRecord file. Let's write it five times to pretend we have several contacts:

```
with tf.io.TFRecordWriter("my_contacts.tfrecord") as f:
    for _ in range(5):
        f.write(person_example.SerializeToString())
```

Normally you would write much more than five `Examples`! Typically, you would create a conversion script that reads from your current format (say, CSV files), creates an `Example` protobuf for each instance, serializes them, and saves them to several TFRecord files, ideally shuffling them in the process. This requires a bit of work, so once again make sure it is really necessary (perhaps your pipeline works fine with CSV files).

Now that we have a nice TFRecord file containing several serialized `Examples`, let's try to load it.

Loading and Parsing Examples

To load the serialized `Example` protobufs, we will use a `tf.data.TFRecordDataset` once again, and we will parse each `Example` using `tf.io.parse_single_example()`. It requires at least two arguments: a string scalar tensor containing the serialized data, and a description of each feature. The description is a dictionary that maps each feature name to either a `tf.io.FixedLenFeature` descriptor indicating the feature's shape, type, and default value, or a `tf.io.VarLenFeature` descriptor indicating only the type if the length of the feature's list may vary (such as for the "emails" feature).

The following code defines a description dictionary, then it creates a `TFRecordDataset`, and applies a custom preprocessing function to it, to parse each serialized `Example` protobuf that this dataset contains:

```
feature_description = {
    "name": tf.io.FixedLenFeature([], tf.string, default_value=""),
    "id": tf.io.FixedLenFeature([], tf.int64, default_value=0),
    "emails": tf.io.VarLenFeature(tf.string),
}

def parse(serialized_example):
    return tf.io.parse_single_example(serialized_example, feature_description)

dataset = tf.data.TFRecordDataset(["my_contacts.tfrecord"]).map(parse)
for parsed_example in dataset:
    print(parsed_example)
```

The fixed-length features are parsed as regular tensors, but the variable-length features are parsed as sparse tensors. You can convert a sparse tensor to a dense tensor using `tf.sparse.to_dense()`, but in this case it is simpler to just access its values:

```
>>> tf.sparse.to_dense(parsed_example["emails"], default_value=b"")
<tf.Tensor: [...] dtype=string, numpy=array([b'a@b.com', b'c@d.com'], [...])>
>>> parsed_example["emails"].values
<tf.Tensor: [...] dtype=string, numpy=array([b'a@b.com', b'c@d.com'], [...])>
```

Instead of parsing examples one by one using `tf.io.parse_single_example()`, you may want to parse them batch by batch using `tf.io.parse_example()`:

```
def parse(serialized_examples):
    return tf.io.parse_example(serialized_examples, feature_description)

dataset = tf.data.TFRecordDataset(["my_contacts.tfrecord"]).batch(2).map(parse)
for parsed_examples in dataset:
    print(parsed_examples) # two examples at a time
```

Lastly, a `BytesList` can contain any binary data you want, including any serialized object. For example, you can use `tf.io.encode_jpeg()` to encode an image using the JPEG format and put this binary data in a `BytesList`. Later, when your code reads the TFRecord, it will start by parsing the `Example`, then it will need to call

`tf.io.decode_jpeg()` to parse the data and get the original image (or you can use `tf.io.decode_image()`, which can decode any BMP, GIF, JPEG, or PNG image). You can also store any tensor you want in a `BytesList` by serializing the tensor using `tf.io.serialize_tensor()` then putting the resulting byte string in a `BytesList` feature. Later, when you parse the TFRecord, you can parse this data using `tf.io.parse_tensor()`. See the notebook for examples of storing images and tensors in a TFRecord file.

As you can see, the `Example` protobuf is quite flexible, so it will probably be sufficient for most use cases. However, it may be a bit cumbersome to use when you are dealing with lists of lists. For example, suppose you want to classify text documents. Each document may be represented as a list of sentences, where each sentence is represented as a list of words. And perhaps each document also has a list of comments, where each comment is represented as a list of words. There may be some contextual data too, such as the document's author, title, and publication date. TensorFlow's `SequenceExample` protobuf is designed for such use cases.

Handling Lists of Lists Using the `SequenceExample` Protobuf

Here is the definition of the `SequenceExample` protobuf:

```
message FeatureList { repeated Feature feature = 1; };
message FeatureLists { map<string, FeatureList> feature_list = 1; };
message SequenceExample {
    Features context = 1;
    FeatureLists feature_lists = 2;
};
```

A `SequenceExample` contains a `Features` object for the contextual data and a `FeatureLists` object that contains one or more named `FeatureList` objects (e.g., a `FeatureList` named "content" and another named "comments"). Each `FeatureList` contains a list of `Feature` objects, each of which may be a list of byte strings, a list of 64-bit integers, or a list of floats (in this example, each `Feature` would represent a sentence or a comment, perhaps in the form of a list of word identifiers). Building a `SequenceExample`, serializing it, and parsing it is similar to building, serializing, and parsing an `Example`, but you must use `tf.io.parse_single_sequence_example()` to parse a single `SequenceExample` or `tf.io.parse_sequence_example()` to parse a batch. Both functions return a tuple containing the context features (as a dictionary) and the feature lists (also as a dictionary). If the feature lists contain sequences of varying sizes (as in the preceding example), you may want to convert them to ragged tensors, using `tf.RaggedTensor.from_sparse()` (see the notebook for the full code):

```
parsed_context, parsed_feature_lists = tf.io.parse_single_sequence_example(
    serialized_sequence_example, context_feature_descriptions,
    sequence_feature_descriptions)
parsed_content = tf.RaggedTensor.from_sparse(parsed_feature_lists["content"])
```

Now that you know how to efficiently store, load, parse, and preprocess the data using the `tf.data` API, TFRecords, and protobufs, it's time to turn our attention to the Keras preprocessing layers.

Keras Preprocessing Layers

Preparing your data for a neural network typically requires normalizing the numerical features, encoding the categorical features and text, cropping and resizing images, and more. There are several options for this:

- The preprocessing can be done ahead of time when preparing your training data files, using any tools you like, such as NumPy, Pandas, or Scikit-Learn. You will need to apply the exact same preprocessing steps in production, to ensure your production model receives preprocessed inputs similar to the ones it was trained on.
- Alternatively, you can preprocess your data on the fly while loading it with `tf.data`, by applying a preprocessing function to every element of a dataset using that dataset's `map()` method, as we did earlier in this chapter. Again, you will need to apply the same preprocessing steps in production.
- One last approach is to include preprocessing layers directly inside your model so it can preprocess all the input data on the fly during training, then use the same preprocessing layers in production. The rest of this chapter will look at this last approach.

Keras offers many preprocessing layers that you can include in your models: they can be applied to numerical features, categorical features, images, and text. We'll go over the numerical and categorical features in the next sections, as well as basic text preprocessing, and we will cover image preprocessing in [Chapter 14](#), and more advanced text preprocessing in [Chapter 16](#).

The Normalization Layer

As we saw in [Chapter 10](#), Keras provides a `Normalization` layer that you can use to standardize the input features. You can either specify the mean and variance of each feature when creating the layer, or—more simply—you can pass the training set to the layer's `adapt()` method before fitting the model, so the layer can measure the feature means and variances on its own before training:

```
norm_layer = tf.keras.layers.Normalization()
model = tf.keras.models.Sequential([
    norm_layer,
    tf.keras.layers.Dense(1)
])
model.compile(loss="mse", optimizer=tf.keras.optimizers.SGD(learning_rate=2e-3))
```

```
norm_layer.adapt(X_train) # computes the mean and variance of every feature  
model.fit(X_train, y_train, validation_data=(X_valid, y_valid), epochs=5)
```



The data sample passed to the `adapt()` method must be large enough to be representative of your dataset, but it does not have to be the full training set: for the Normalization layer, a few hundred instances randomly sampled from the training set will generally be sufficient to get a good estimate of the feature means and variances.

Since we included the Normalization layer inside the model, we can now deploy this model to production without having to worry about normalization again: the model will just handle it (see [Figure 13-4](#)). Fantastic! This approach completely eliminates the risk of *preprocessing mismatch*: this happens when people try to maintain different preprocessing code for training and for production, but they update one and forget to update the other. The production model ends up receiving data preprocessed in a way it doesn't expect. If they're lucky, they get a clear bug. If not, the model's accuracy just silently degrades.

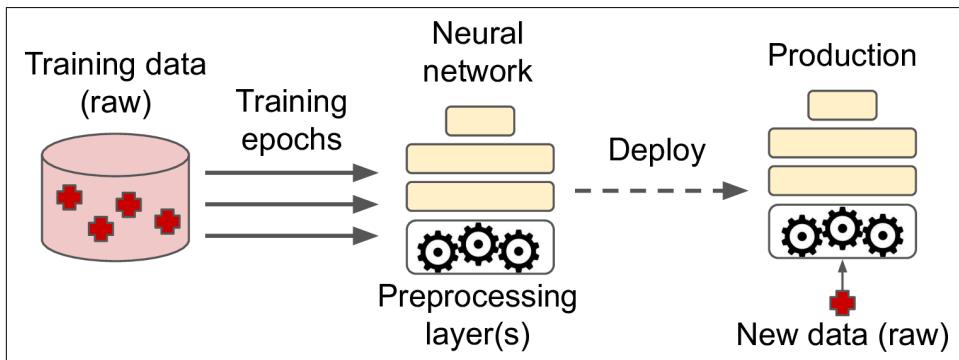


Figure 13-4. Including preprocessing layers inside a model

Including the preprocessing layer directly in the model is nice and straightforward, but it will slow down training (only very slightly in the case of the Normalization layer): indeed, since preprocessing is performed on the fly during training, it happens once per epoch. We can do better by normalizing the whole training set just once before training. To do this, you can use the Normalization layer in a standalone fashion (much like a Scikit-Learn `StandardScaler`):

```
norm_layer = tf.keras.layers.Normalization()  
norm_layer.adapt(X_train)  
X_train_scaled = norm_layer(X_train)  
X_valid_scaled = norm_layer(X_valid)
```

Now we can train a model on the scaled data, but this time without any Normalization layer inside the model:

```

model = tf.keras.models.Sequential([tf.keras.layers.Dense(1)])
model.compile(loss="mse", optimizer=tf.keras.optimizers.SGD(learning_rate=2e-3))
model.fit(X_train_scaled, y_train, epochs=5,
           validation_data=(X_valid_scaled, y_valid))

```

Good! This should speed up training a bit. But now the model won't preprocess its inputs when we deploy it to production. To fix this, we just need to create a new model that wraps both the adapted Normalization layer and the model we just trained. We can then deploy this final model to production, and it will take care of both preprocessing its inputs and making predictions (see [Figure 13-5](#)):

```

final_model = tf.keras.Sequential([norm_layer, model])
X_new = X_test[:3] # pretend we have a few new instances (unscaled)
y_pred = final_model(X_new) # preprocesses the data and makes predictions

```

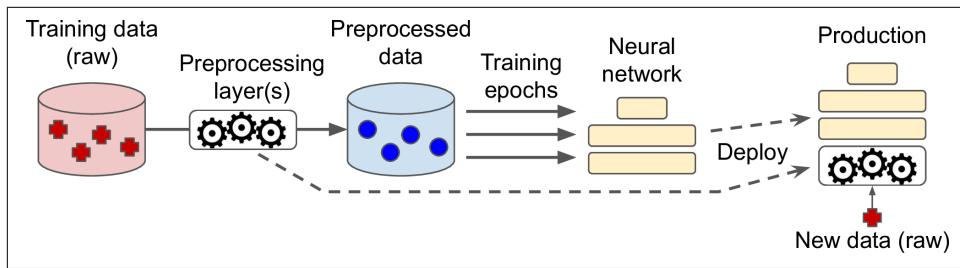


Figure 13-5. Preprocessing the data just once before training using preprocessing layers, then deploying these layers inside the final model

Now we have the best of both worlds: training is fast because we only preprocess the data once before training begins, and the final model can preprocess its inputs on the fly without any risk of preprocessing mismatch.

Moreover, the Keras preprocessing layers play nicely with the `tf.data` API. For example, it's possible to pass a `tf.data.Dataset` to a preprocessing layer's `adapt()` method. It's also possible to apply a Keras preprocessing layer to a `tf.data.Dataset` using the dataset's `map()` method. For example, here's how you could apply an adapted Normalization layer to the input features of each batch in a dataset:

```
dataset = dataset.map(lambda X, y: (norm_layer(X), y))
```

Lastly, if you ever need more features than the Keras preprocessing layers provide, you can always write your own Keras layer, just like we discussed in [Chapter 12](#). For example, if the Normalization layer didn't exist, you could get a similar result using the following custom layer:

```

import numpy as np

class MyNormalization(tf.keras.layers.Layer):
    def adapt(self, X):
        self.mean_ = np.mean(X, axis=0, keepdims=True)

```

```

    self.std_ = np.std(X, axis=0, keepdims=True)

    def call(self, inputs):
        eps = tf.keras.backend.epsilon() # a small smoothing term
        return (inputs - self.mean_) / (self.std_ + eps)

```

Next, let's look at another Keras preprocessing layer for numerical features: the `Discretization` layer.

The Discretization Layer

The `Discretization` layer's goal is to transform a numerical feature into a categorical feature by mapping value ranges (called bins) to categories. This is sometimes useful for features with multimodal distributions, or with features that have a highly non-linear relationship with the target. For example, the following code maps a numerical age feature to three categories: less than 18, 18 to 50 (not included), and 50 or over.

```

>>> age = tf.constant([[10.], [93.], [57.], [18.], [37.], [5.]])
>>> discretize_layer = tf.keras.layers.Discretization(bin_boundaries=[18., 50.])
>>> age_categories = discretize_layer(age)
>>> age_categories
<tf.Tensor: shape=(6, 1), dtype=int64, numpy=array([[0],[2],[2],[1],[1],[0]])>

```

In this example, we provided the desired bin boundaries. If you prefer, you can instead provide the number of bins you want, then call the layer's `adapt()` method to let it find the appropriate bin boundaries, based on the value percentiles. For example, if we set `num_bins=3`, then the bin boundaries will be located at the values just below the 33rd and 66th percentiles (in this example, at the values 10 and 37).

```

>>> discretize_layer = tf.keras.layers.Discretization(num_bins=3)
>>> discretize_layer.adapt(age)
>>> age_categories = discretize_layer(age)
>>> age_categories
<tf.Tensor: shape=(6, 1), dtype=int64, numpy=array([[1],[2],[2],[1],[2],[0]])>

```

Category identifiers such as these should generally not be passed directly to a neural network, as their values cannot be meaningfully compared. Instead, they should be encoded, for example, using one-hot encoding. Let's look at how to do this now.

The CategoryEncoding Layer

When there are only a few categories (e.g., less than a dozen or two), then one-hot encoding is often a good option (as discussed in [Chapter 2](#)). To do this, Keras provides the `CategoryEncoding` layer. For example, let's one-hot encode the `age_categories` feature we have just created:

```

>>> onehot_layer = tf.keras.layers.CategoryEncoding(num_tokens=3)
>>> onehot_layer(age_categories)
<tf.Tensor: shape=(6, 3), dtype=float32, numpy=

```

```
array([[0., 1., 0.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 1., 0.],
       [0., 0., 1.],
       [1., 0., 0.]], dtype=float32)>
```

If you try to encode more than one categorical feature at a time (which only makes sense if they all use the same categories), the `CategoryEncoding` class will perform *multi-hot encoding* by default: the output tensor will contain a 1 for each category present in *any* input feature. For example:

```
>>> two_age_categories = np.array([[1, 0], [2, 2], [2, 0]])
>>> onehot_layer(two_age_categories)
<tf.Tensor: shape=(3, 3), dtype=float32, numpy=
array([[1., 1., 0.],
       [0., 0., 1.],
       [1., 0., 1.]], dtype=float32)>
```

If you believe it's useful to know how many times each category occurred, you can set `output_mode="count"` when creating the `CategoryEncoding` layer, in which case the output tensor will contain the number of occurrences of each category. In the preceding example, the output would be the same except for the second row, which would become [0., 0., 2.].

Note that both multi-hot encoding and count encoding lose information, since it's not possible to know which feature each active category came from. For example, both [0, 1] and [1, 0] are encoded as [1., 1., 0.]. If you want to avoid this, then you need to one-hot encode each feature separately and concatenate the outputs. This way, [0, 1] would get encoded as [1., 0., 0., 0., 1., 0.] and [1, 0] would get encoded as [0., 1., 0., 1., 0., 0.]. You can get the same result by tweaking the category identifiers so they don't overlap, for example:

```
>>> onehot_layer = tf.keras.layers.CategoryEncoding(num_tokens=3 + 3)
>>> onehot_layer(two_age_categories + [0, 3]) # adds 3 to the second feature
<tf.Tensor: shape=(3, 6), dtype=float32, numpy=
array([[0., 1., 0., 1., 0., 0.],
       [0., 0., 1., 0., 0., 1.],
       [0., 0., 1., 1., 0., 0.]], dtype=float32)>
```

In this output, the first three columns correspond to the first feature, and the last three correspond to the second feature. This allows the model to distinguish the two features. However, it also increases the number of features fed to the model, and thereby requires more model parameters. It's hard to know in advance whether a single multi-hot encoding or a per-feature one-hot encoding will work best: it depends on the task, and you may need to test both options.

Now you can encode categorical integer features using one-hot or multi-hot encoding. But what about categorical text features? For this, you can use the `StringLookup` layer.

The `StringLookup` Layer

You can also use the Keras `StringLookup` layer to one-hot encode a `cities` feature:

```
>>> cities = ["Auckland", "Paris", "Paris", "San Francisco"]
>>> str_lookup_layer = tf.keras.layers.StringLookup()
>>> str_lookup_layer.adapt(cities)
>>> str_lookup_layer([["Paris"], ["Auckland"], ["Auckland"], ["Montreal"]])
<tf.Tensor: shape=(4, 1), dtype=int64, numpy=array([[1], [3], [3], [0]])>
```

We first create a `StringLookup` layer, then we adapt it to the data: it finds that there are three distinct categories. Then we use the layer to encode a few cities: they are encoded as integers by default. Unknown categories get mapped to 0, as is the case for “Montreal” in this example. The known categories are numbered starting at 1, from the most frequent categories to the least frequent.

Conveniently, if you set `output_mode="one_hot"` when creating the `StringLookup` layer, it will output a one-hot vector for each category, instead of an integer:

```
>>> str_lookup_layer = tf.keras.layers.StringLookup(output_mode="one_hot")
>>> str_lookup_layer.adapt(cities)
>>> str_lookup_layer([["Paris"], ["Auckland"], ["Auckland"], ["Montreal"]])
<tf.Tensor: shape=(4, 4), dtype=float32, numpy=
array([[0., 1., 0., 0.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [1., 0., 0., 0.]]), dtype=float32>
```



Keras also includes an `IntegerLookup` layer which acts much like the `StringLookup` layer but takes integers as input, rather than strings.

If the training set is very large, it may be convenient to adapt the layer to just a random subset of the training set. In this case, the layer’s `adapt()` method may miss some of the rarer categories. By default, it would then map them all to category 0, making them indistinguishable by the model. To reduce this risk (while still adapting the layer only on a subset of the training set), you can set `num_oov_indices` to an integer greater than 1. This is the number of out-of-vocabulary (oov) buckets to use: each unknown category will get mapped pseudo-randomly to one of the oov buckets, using a hash function modulo the number of oov buckets. This will allow the model to distinguish at least some of the rare categories. For example:

```
>>> str_lookup_layer = tf.keras.layers.StringLookup(num_oov_indices=5)
>>> str_lookup_layer.adapt(cities)
>>> str_lookup_layer([["Paris"], ["Auckland"], ["Foo"], ["Bar"], ["Baz"]])
<tf.Tensor: shape=(4, 1), dtype=int64, numpy=array([[5], [7], [4], [3], [4]])>
```

Since there are 5 oov buckets, the first known category's id is now 5 ("Paris"). But "Foo", "Bar", and "Baz" are unknown, so they each get mapped to one of the oov buckets. "Bar" gets its own dedicated bucket (with id 3), but sadly "Foo" and "Baz" happen to be mapped to the same bucket (with id 4), so they remain indistinguishable by the model. This is called a *hashing collision*. The only way to reduce the risk of collision is to increase the number of oov buckets. However, this will also increase the total number of categories, which will require more RAM and extra model parameters once the categories are one-hot encoded. So don't increase that number too much.

This idea of mapping categories pseudo-randomly to buckets is called the *hashing trick*. Keras provides a dedicated layer which does just that: the `Hashing` layer.

The Hashing Layer

For each category, the Keras `Hashing` layer computes a hash, modulo the number of buckets (or "bins"). The mapping is entirely pseudo-random, but stable across runs and platforms (i.e., the same category will always be mapped to the same integer, as long as the number of bins is unchanged). For example, let's use the `Hashing` layer to encode a few cities:

```
>>> hashing_layer = tf.keras.layers.Hashing(num_bins=10)
>>> hashing_layer([["Paris"], ["Tokyo"], ["Auckland"], ["Montreal"]])
<tf.Tensor: shape=(4, 1), dtype=int64, numpy=array([[0], [1], [9], [1]])>
```

The benefit of this layer is that it does not need to be adapted at all, which may sometimes be useful, especially in an out-of-core setting (when the dataset is too large to fit in memory). However, we once again get a hashing collision: "Tokyo" and "Montreal" are mapped to the same id, making them indistinguishable by the model. So it's usually preferable to stick to the `StringLookup` layer.

Let's now look at another way to encode categories: trainable embeddings.

Encoding Categorical Features Using Embeddings

An embedding is a dense representation of some higher-dimensional data, such as a category, or a word in a vocabulary. If there are 50,000 possible categories, then one-hot encoding would produce a 50,000-dimensional sparse vector (i.e., containing mostly zeroes). In contrast, an embedding would be a comparatively small dense vector, for example with just 100 dimensions.

In Deep Learning, embeddings are usually initialized randomly, and they are then trained by Gradient Descent, along with the other model parameters. For example the "NEAR BAY" category in the California housing dataset could be represented initially by a random vector such as [0.131, 0.890], while the "NEAR OCEAN" category might be represented by another random vector such as [0.631, 0.791]. In this example, we use 2D embeddings, but the number of dimensions is a hyperparameter you can tweak.

Since these embeddings are trainable, they will gradually improve during training; and as they represent fairly similar categories in this case, Gradient Descent will certainly end up pushing them closer together, while it will tend to move them away from the "INLAND" category's embedding (see [Figure 13-6](#)). Indeed, the better the representation, the easier it will be for the neural network to make accurate predictions, so training tends to make embeddings useful representations of the categories. This is called *representation learning* (we will see other types of representation learning in [Chapter 17](#)).

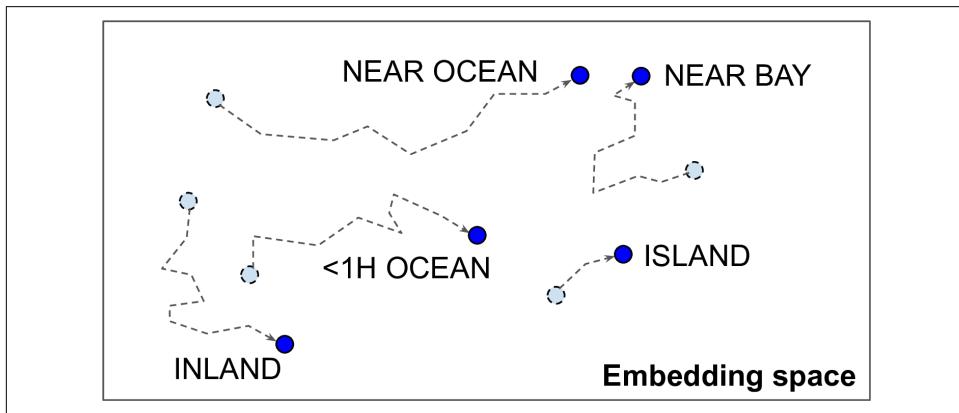


Figure 13-6. Embeddings will gradually improve during training

Word Embeddings

Not only will embeddings generally be useful representations for the task at hand, but quite often these same embeddings can be reused successfully for other tasks. The most common example of this is *word embeddings* (i.e., embeddings of individual words): when you are working on a natural language processing task, you are often better off reusing pretrained word embeddings than training your own.

The idea of using vectors to represent words dates back to the 1960s, and many sophisticated techniques have been used to generate useful vectors, including using neural networks. But things really took off in 2013, when Tomáš Mikolov and other

Google researchers published a [paper](#)⁶ describing an efficient technique to learn word embeddings using neural networks, significantly outperforming previous attempts. This allowed them to learn embeddings on a very large corpus of text: they trained a neural network to predict the words near any given word, and obtained astounding word embeddings. For example, synonyms had very close embeddings, and semantically related words such as France, Spain, and Italy ended up clustered together.

It's not just about proximity, though: word embeddings were also organized along meaningful axes in the embedding space. Here is a famous example: if you compute King – Man + Woman (adding and subtracting the embedding vectors of these words), then the result will be very close to the embedding of the word Queen (see [Figure 13-7](#)). In other words, the word embeddings encode the concept of gender! Similarly, you can compute Madrid – Spain + France, and the result is close to Paris, which seems to show that the notion of capital city was also encoded in the embeddings.

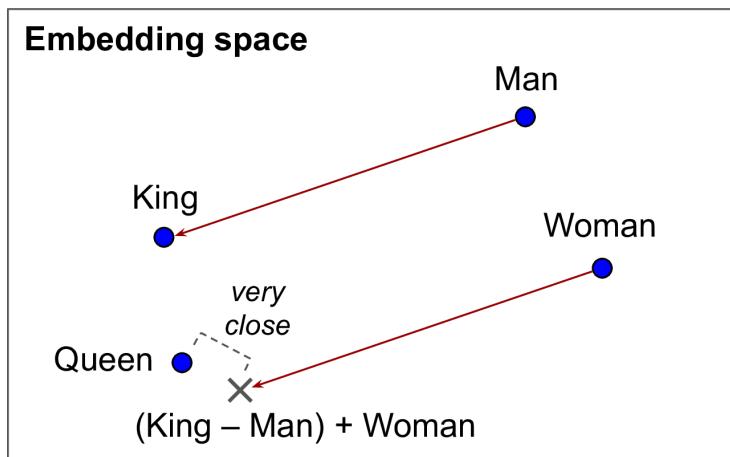


Figure 13-7. Word embeddings of similar words tend to be close, and some axes seem to encode meaningful concepts

Unfortunately, word embeddings sometimes capture our worst biases. For example, although they correctly learn that Man is to King as Woman is to Queen, they also seem to learn that Man is to Doctor as Woman is to Nurse: quite a sexist bias! To be fair, this particular example is probably exaggerated, as was pointed out in a [2019 paper](#)⁷ by Malvina Nissim et al. Nevertheless, ensuring fairness in Deep Learning algorithms is an important and active research topic.

⁶ Tomas Mikolov et al., “Distributed Representations of Words and Phrases and Their Compositionality,” *Proceedings of the 26th International Conference on Neural Information Processing Systems 2* (2013): 3111–3119.

Keras provides an `Embedding` layer, which wraps an *embedding matrix*: this matrix has one row per category, and one column per embedding dimension. By default, it is initialized randomly. To convert a category id to an embedding, the `Embedding` layer just looks up and returns the row that corresponds to that category. That's all there is to it! For example, let's initialize an `Embedding` layer with 5 rows, and 2D embeddings, and let's use it to encode some categories:

```
>>> tf.random.set_seed(42)
>>> embedding_layer = tf.keras.layers.Embedding(input_dim=5, output_dim=2)
>>> embedding_layer(np.array([2, 4, 2]))
<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[-0.04663396,  0.01846724],
       [-0.02736737, -0.02768031],
       [-0.04663396,  0.01846724]], dtype=float32)>
```

As you can see, category 2 gets encoded (twice) as the 2D vector $[-0.04663396, 0.01846724]$, while category 4 gets encoded as $[-0.02736737, -0.02768031]$. Since the layer is not trained yet, these encodings are just random.



An `Embedding` layer is initialized randomly, so it does not make sense to use it outside of a model as a standalone preprocessing layer, unless you initialize it with pretrained weights.

If you want to embed a categorical text attribute, you can simply chain a `StringLookup` layer and an `Embedding` layer, like this:

```
>>> tf.random.set_seed(42)
>>> ocean_prox = ["<1H OCEAN", "INLAND", "NEAR OCEAN", "NEAR BAY", "ISLAND"]
>>> str_lookup_layer = tf.keras.layers.StringLookup()
>>> str_lookup_layer.adapt(ocean_prox)
>>> lookup_and_embed = tf.keras.Sequential([
...     str_lookup_layer,
...     tf.keras.layers.Embedding(input_dim=str_lookup_layer.vocabulary_size(),
...                               output_dim=2)
... ])
...
>>> lookup_and_embed(np.array([["<1H OCEAN"], ["ISLAND"], ["<1H OCEAN"]]))
<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[-0.01896119,  0.02223358],
       [ 0.02401174,  0.03724445],
       [-0.01896119,  0.02223358]], dtype=float32)>
```

⁷ Malvina Nissim et al., “Fair Is Better Than Sensational: Man Is to Doctor as Woman Is to Doctor,” arXiv preprint arXiv:1905.09866 (2019).

Note that the number of rows in the embedding matrix needs to be equal to the vocabulary size: that's the total number of categories, including the known categories plus the oov buckets (just one by default). The `vocabulary_size()` method of the `StringLookup` class conveniently returns this number.



In this example we used 2D embeddings, but as a rule of thumb embeddings typically have 10 to 300 dimensions, depending on the task, the vocabulary size, and the size of your training set. You will have to tune this hyperparameter.

Putting everything together, we can now create a Keras model that can process a categorical text feature along with regular numerical features and learn an embedding for each category (as well as for each oov bucket):

```
X_train_num, X_train_cat, y_train = [...] # load the training set
X_valid_num, X_valid_cat, y_valid = [...] # and the validation set

num_input = tf.keras.layers.Input(shape=[8], name="num")
cat_input = tf.keras.layers.Input(shape=[], dtype=tf.string, name="cat")
cat_embeddings = lookup_and_embed(cat_input)
encoded_inputs = tf.keras.layers.concatenate([num_input, cat_embeddings])
outputs = tf.keras.layers.Dense(1)(encoded_inputs)
model = tf.keras.models.Model(inputs=[num_input, cat_input], outputs=[outputs])
model.compile(loss="mse", optimizer="sgd")
history = model.fit((X_train_num, X_train_cat), y_train, epochs=5,
                     validation_data=((X_valid_num, X_valid_cat), y_valid))
```

This model takes two inputs: `num_input` which contains eight numerical features per instance, plus `cat_input` which contains a single categorical text input per instance. The model uses the `lookup_and_embed` model we created earlier to encode each ocean-proximity category as the corresponding trainable embedding. Next, it concatenates the numerical inputs and the embeddings using the `concatenate()` function to produce the complete encoded inputs, which are ready to be fed to a neural network. We could add any kind of neural network at this point, but for simplicity we just add a single dense output layer, and then we create the Keras Model with the inputs and output we've just defined. Next we compile the model, and train it, passing both the numerical and categorical inputs.

As we saw in [Chapter 10](#), since the `Input` layers are named "num" and "cat", we could also have passed the training data to the `fit()` method using a dictionary instead of a tuple: `{"num": X_train_num, "cat": X_train_cat}`. Alternatively, we could have passed a `tf.data.Dataset` containing batches, each represented as `((X_batch_num, X_batch_cat), y_batch)` or as `({"num": X_batch_num, "cat": X_batch_cat}, y_batch)`. And of course the same goes for the validation data.



One-hot encoding followed by a `Dense` layer (with no activation function and no biases) is equivalent to an `Embedding` layer. However, the `Embedding` layer uses way fewer computations as it avoids many multiplications by zero—the performance difference becomes clear when the size of the embedding matrix grows. The `Dense` layer’s weight matrix plays the role of the embedding matrix. For example, using one-hot vectors of size 20 and a `Dense` layer with 10 units is equivalent to using an `Embedding` layer with `input_dim=20` and `output_dim=10`. As a result, it would be wasteful to use more embedding dimensions than the number of units in the layer that follows the `Embedding` layer.

OK, now that you have learned how to encode categorical features, it’s time to turn our attention to text preprocessing.

Text Preprocessing

Keras provides a `TextVectorization` layer for basic text preprocessing. Much like the `StringLookup` layer, you must either pass it a vocabulary upon creation, or let it learn the vocabulary from some training data using the `adapt()` method. Let’s look at an example:

```
>>> train_data = ["To be", "!(to be)", "That's the question", "Be, be, be."]
>>> text_vec_layer = tf.keras.layers.TextVectorization()
>>> text_vec_layer.adapt(train_data)
>>> text_vec_layer(["Be good!", "Question: be or be?"])
<tf.Tensor: shape=(2, 4), dtype=int64, numpy=
array([[2, 1, 0, 0],
       [6, 2, 1, 2]])>
```

The two sentences “Be good!” and “Question: be or be?” were encoded as `[2, 1, 0, 0]` and `[6, 2, 1, 2]`, respectively. The vocabulary was learned from the 4 sentences in the training data: “be” = 2, “to” = 3, etc. To construct the vocabulary, the `adapt()` method first converted the training sentences to lowercase and removed punctuation, which is why “Be”, “be,” and “be?” are all encoded as “be” = 2. Next, the sentences were split at the whitespaces, and the resulting words were sorted by descending frequency, producing the final vocabulary. When encoding sentences, unknown words get encoded as 1s. Lastly, since the first sentence is shorter than the second, it was padded with 0s.



The `TextVectorization` layer has many options. For example, you can preserve the case and punctuation if you want, by setting `standardize=None`, or you can pass any standardization function you please as the `standardize` argument. You can prevent splitting by setting `split=None`, or you can even pass your own splitting function instead. You can set the `output_sequence_length` argument to ensure that the output sequences all get cropped or padded to the desired length. Or, you can set `ragged=True` to get a ragged tensor instead of a regular tensor. Please check out the documentation for more options.

The word ids must be encoded, typically using an `Embedding` layer: we will do this in [Chapter 16](#). Alternatively, you can set the `TextVectorization` layer's `output_mode` argument to "`multi_hot`" or "`count`" to get the corresponding encodings. However, simply counting words is usually not ideal: indeed, words like "to" or "the" are so frequent that they hardly matter at all. Conversely, rarer words such as "basketball" are much more informative. So rather than setting `output_mode` to "`multi_hot`" or "`count`", it is usually preferable to set it to "`tf_idf`", which stands for *Term-Frequency × Inverse-Document-Frequency* (TF-IDF). This is similar to the count encoding, but words that occur frequently in the training data are downweighted, and conversely, rare words are upweighted. For example:

```
>>> text_vec_layer = tf.keras.layers.TextVectorization(output_mode="tf_idf")
>>> text_vec_layer.adapt(train_data)
>>> text_vec_layer(["Be good!", "Question: be or be?"])
<tf.Tensor: shape=(2, 6), dtype=float32, numpy=
array([[0.96725637, 0.6931472 , 0. , 0. , 0. , 0.        ],
       [0.96725637, 1.3862944 , 0. , 0. , 0. , 1.0986123 ]], dtype=float32)>
```

There are many TF-IDF variants, but the way the `TextVectorization` layer implements it is by multiplying each word count by a weight equal to $\log(1 + d / (f + 1))$ where d is the total number of sentences (a.k.a., documents) in the training data, and f counts how many of these training sentences contain the given word. For example, there are $d = 4$ sentences in the training data, and the word "be" appears in $f = 3$ of these. Since the word "be" occurs twice in the sentence "Question: be or be?", it gets encoded as $2 \times \log(1 + 4 / (1 + 3)) \approx 1.3862944$. The word "question" only appears once, but since it is a less common word, its encoding is almost as high: $1 \times \log(1 + 4 / (1 + 1)) \approx 1.0986123$. Note that the average weight is used for unknown words.

This approach to text encoding is straightforward to use and it can give fairly good results for basic natural language processing tasks, but it has several important limitations: it only works with languages that separate words with spaces, it doesn't distinguish between homonyms (e.g., "to bear" versus "teddy bear"), it gives no hint to your model that words like "evolution" and "evolutionary" are related, etc. And if

you use multi-hot, count or TF-IDF encoding, then the order of the words is lost. So what are the other options?

One option is to use the [TensorFlow Text library](#), which provides more advanced text preprocessing features than the `TextVectorization` layer. For example, it includes several subword tokenizers capable of splitting the text into tokens smaller than words, which makes it possible for the model to more easily detect that “evolution” and “evolutionary” have something in common (more on subword tokenization in [Chapter 16](#)).

Yet another option is to use pretrained language model components. Let’s look at this now.

Using Pretrained Language Model Components

The [TensorFlow Hub library](#) makes it easy to reuse pretrained model components in your own models, for text, image, audio, or more. These model components are called *modules*. Simply browse the [TF Hub repository](#), find the one you need, and copy the code example into your project, and the module will be automatically downloaded and bundled into a Keras layer that you can directly include in your model. Modules typically contain both preprocessing code and pretrained weights, and they generally require no extra training (but of course, the rest of your model will certainly require training).

For example, some powerful pretrained language models are available. The most powerful are quite large (several Gigabytes), so for a quick example let’s use the `nnlm-en-dim50` module, version 2, which is a fairly basic module that takes raw text as input and outputs 50-dimensional sentence embeddings. Let’s import TensorFlow Hub and use it to load the module, then use that module to encode two sentences to vectors:⁸

```
>>> import tensorflow_hub as hub
>>> hub_layer = hub.KerasLayer("https://tfhub.dev/google/nnlm-en-dim50/2")
>>> sentence_embeddings = hub_layer(tf.constant(["To be", "Not to be"]))
>>> sentence_embeddings.numpy().round(2)
array([[-0.25,  0.28,  0.01,  0.1 , [...],  0.05,  0.31],
       [-0.2 ,  0.2 , -0.08,  0.02, [...], -0.04,  0.15]], dtype=float32)
```

The `hub.KerasLayer` layer downloads the module from the given URL. This particular module is a *sentence encoder*: it takes strings as input and encodes each one as a single vector (in this case, a 50-dimensional vector). Internally, it parses the string (splitting words on spaces) and embeds each word using an embedding matrix that was pretrained on a huge corpus: the Google News 7B corpus (seven billion words

⁸ TensorFlow Hub is not bundled with TensorFlow, but if you are running on Colab or if you followed the installation instructions at <https://hml.info/install>, then it’s already installed.

long!). Then it computes the mean of all the word embeddings, and the result is the sentence embedding.⁹

You just need to include this `hub_layer` in your model, and you're ready to go. Note that this particular language model was trained on the English language, but many other languages are available, as well as multilingual models.

Last but not least, the excellent open source [Transformers library by Hugging Face](#) also makes it easy to include powerful language model components inside your own models. You can browse the [HuggingFace Hub](#), choose the model you want, and use the provided code examples to get started. It used to contain only language models, but it has now expanded to image models and more.

We will come back to natural language processing in more depth in [Chapter 16](#). Let's now look at Keras's image preprocessing layers.

Image Preprocessing Layers

The Keras preprocessing API includes three image preprocessing layers:

- `tf.keras.layers.Resizing` resizes the input images to the desired size. For example `Resizing(height=100, width=200)` resizes the image to 100×200 , possibly distorting the image. If you set `crop_to_aspect_ratio=True`, then the image will be cropped to the target image ratio, to avoid distortion.
- `tf.keras.layers.Rescaling` rescales the pixel values, for example `Rescaling(scale=2/255, offset=-1)` scales the values from $0 \rightarrow 255$ to $-1 \rightarrow 1$.
- `tf.keras.layers.CenterCrop` crops the image, keeping only a center patch of the desired height and width.

For example, let's load a couple of sample images and center-crop them. For this, we will use Scikit-Learn's `load_sample_images()` function, which loads two color images: one of a Chinese temple, and the other of a flower (this requires the Pillow library, which should already be installed if you are using Colab or if you followed the installation instructions).

```
from sklearn.datasets import load_sample_images

images = load_sample_images()["images"]
crop_image_layer = tf.keras.layers.CenterCrop(height=100, width=100)
cropped_images = crop_image_layer(images)
```

⁹ To be precise, the sentence embedding is equal to the mean word embedding multiplied by the square root of the number of words in the sentence. This compensates for the fact that the mean of n random vectors gets shorter as n grows.

Keras also includes several layers for data augmentation, such as `RandomCrop`, `RandomFlip`, `RandomTranslation`, `RandomRotation`, `RandomZoom`, `RandomHeight`, `RandomWidth`, and `RandomContrast`. These layers are only active during training, and they randomly apply some transformation to the input images (their names are self-explanatory). Data augmentation will artificially increase the size of the training set, which often leads to improved performance, as long as the transformed images look like realistic (non-augmented) images. We'll cover image processing more closely in the next chapter.



Under the hood, the Keras preprocessing layers are based on TensorFlow's low-level API. For example, the `Normalization` layer uses `tf.nn.moments()` to compute both the mean and variance, the `Discretization` layer uses `tf.raw_ops.Bucketize()`, `CategoricalEncoding` uses `tf.math.bincount()`, `IntegerLookup` and `StringLookup` use the `tf.lookup` package, `Hashing` and `TextVectorization` use several ops from the `tf.strings` package, `Embedding` uses `tf.nn.embedding_lookup()`, and the image preprocessing layers use the ops from the `tf.image` package. If the Keras preprocessing API isn't sufficient for your needs, you may occasionally need to use TensorFlow's low-level API directly.

Now let's look at another way to load data easily and efficiently in TensorFlow.

The TensorFlow Datasets (TFDS) Project

The [TensorFlow Datasets](#) project makes it very easy to load common datasets, from small ones like MNIST or Fashion MNIST to huge datasets like ImageNet (you will need quite a bit of disk space!). The list includes image datasets, text datasets (including translation datasets), audio and video datasets, time series, and much more. You can visit <https://hml.info/tfds> to view the full list, along with a description of each dataset. You can also check out [Know Your Data](#), which is a tool to explore and understand many of the datasets provided by TFDS.

TFDS is not bundled with TensorFlow, but if you are running on Colab or if you followed the installation instructions at <https://hml.info/install>, then it's already installed. You can then import `tensorflow_datasets`, usually as `tfds`, then call the `tfds.load()` function, which will download the data you want (unless it was already downloaded earlier) and return the data as a dictionary of datasets (typically one for training and one for testing, but this depends on the dataset you choose). For example, let's download MNIST:

```
import tensorflow_datasets as tfds
```

```
datasets = tfds.load(name="mnist")
mnist_train, mnist_test = datasets["train"], datasets["test"]
```

You can then apply any transformation you want (typically shuffling, batching, and prefetching), and you're ready to train your model. Here is a simple example:

```
for batch in mnist_train.shuffle(10_000, seed=42).batch(32).prefetch(1):
    images = batch["image"]
    labels = batch["label"]
    # [...] do something with the images and labels
```



The `load()` function can shuffle the files it downloads: just set `shuffle_files=True`. However, this may be insufficient, so it's best to shuffle the training data some more.

Note that each item in the dataset is a dictionary containing both the features and the labels. But Keras expects each item to be a tuple containing two elements (again, the features and the labels). You could transform the dataset using the `map()` method, like this:

```
mnist_train = mnist_train.shuffle(buffer_size=10_000, seed=42).batch(32)
mnist_train = mnist_train.map(lambda items: (items["image"], items["label"]))
mnist_train = mnist_train.prefetch(1)
```

But it's simpler to ask the `load()` function to do this for you by setting `as_supervised=True` (obviously this works only for labeled datasets).

Lastly, TFDS provides a convenient way to split the data using the `split` argument. For example, if you want to use the first 90% of the training set for training, and the remaining 10% for validation, and the whole test set for testing, then you can set `split=["train[:90%]", "train[90%:]", "test"]`. The `load()` function will return all three sets. Here is a complete example, loading and splitting the MNIST dataset using TFDS, then using these sets to train and evaluate a simple Keras model:

```
train_set, valid_set, test_set = tfds.load(
    name="mnist",
    split=["train[:90%]", "train[90%:]", "test"],
    as_supervised=True
)
train_set = train_set.shuffle(buffer_size=10_000, seed=42).batch(32).prefetch(1)
valid_set = valid_set.batch(32).cache()
test_set = test_set.batch(32).cache()
tf.random.set_seed(42)
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(10, activation="softmax")
])
model.compile(loss="sparse_categorical_crossentropy", optimizer="nadam",
```

```
metrics=["accuracy"])
history = model.fit(train_set, validation_data=valid_set, epochs=5)
test_loss, test_accuracy = model.evaluate(test_set)
```

Congratulations, you reached the end of this quite technical chapter! You may feel that it is a bit far from the abstract beauty of neural networks, but the fact is Deep Learning often involves large amounts of data, and knowing how to load, parse, and preprocess it efficiently is a crucial skill to have. In the next chapter, we will look at convolutional neural networks, which are among the most successful neural net architectures for image processing and many other applications.

Exercises

1. Why would you want to use the `tf.data` API?
2. What are the benefits of splitting a large dataset into multiple files?
3. During training, how can you tell that your input pipeline is the bottleneck? What can you do to fix it?
4. Can you save any binary data to a TFRecord file, or only serialized protocol buffers?
5. Why would you go through the hassle of converting all your data to the `Example` protobuf format? Why not use your own protobuf definition?
6. When using TFRecords, when would you want to activate compression? Why not do it systematically?
7. Data can be preprocessed directly when writing the data files, or within the `tf.data` pipeline, or in preprocessing layers within your model. Can you list a few pros and cons of each option?
8. Name a few common ways you can encode categorical text features. What about text?
9. Load the Fashion MNIST dataset (introduced in [Chapter 10](#)); split it into a training set, a validation set, and a test set; shuffle the training set; and save each dataset to multiple TFRecord files. Each record should be a serialized `Example` protobuf with two features: the serialized image (use `tf.io.serialize_tensor()` to serialize each image), and the label.¹⁰ Then use `tf.data` to create an efficient dataset for each set. Finally, use a Keras model to train these datasets, including a preprocessing layer to standardize each input feature. Try to make the input pipeline as efficient as possible, using TensorBoard to visualize profiling data.

¹⁰ For large images, you could use `tf.io.encode_jpeg()` instead. This would save a lot of space, but it would lose a bit of image quality.

10. In this exercise you will download a dataset, split it, create a `tf.data.Dataset` to load it and preprocess it efficiently, then build and train a binary classification model containing an `Embedding` layer:
 - a. Download the [Large Movie Review Dataset](#), which contains 50,000 movies reviews from the [Internet Movie Database](#). The data is organized in two directories, `train` and `test`, each containing a `pos` subdirectory with 12,500 positive reviews and a `neg` subdirectory with 12,500 negative reviews. Each review is stored in a separate text file. There are other files and folders (including preprocessed bag-of-words), but we will ignore them in this exercise.
 - b. Split the test set into a validation set (15,000) and a test set (10,000).
 - c. Use `tf.data` to create an efficient dataset for each set.
 - d. Create a binary classification model, using a `TextVectorization` layer to preprocess each review.
 - e. Add an `Embedding` layer and compute the mean embedding for each review, multiplied by the square root of the number of words (see [Chapter 16](#)). This rescaled mean embedding can then be passed to the rest of your model.
 - f. Train the model and see what accuracy you get. Try to optimize your pipelines to make training as fast as possible.
 - g. Use TFDS to load the same dataset more easily: `tfds.load("imdb_reviews")`.

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab3>.

Deep Computer Vision Using Convolutional Neural Networks

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 14th chapter of the final book. The GitHub repo is <https://github.com/ageron/handsonml3>.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

Although IBM’s Deep Blue supercomputer beat the chess world champion Garry Kasparov back in 1996, it wasn’t until fairly recently that computers were able to reliably perform seemingly trivial tasks such as detecting a puppy in a picture or recognizing spoken words. Why are these tasks so effortless to us humans? The answer lies in the fact that perception largely takes place outside the realm of our consciousness, within specialized visual, auditory, and other sensory modules in our brains. By the time sensory information reaches our consciousness, it is already adorned with high-level features; for example, when you look at a picture of a cute puppy, you cannot choose *not* to see the puppy, *not* to notice its cuteness. Nor can you explain *how* you recognize a cute puppy; it’s just obvious to you. Thus, we cannot trust our subjective experience: perception is not trivial at all, and to understand it we must look at how our sensory modules work.

Convolutional neural networks (CNNs) emerged from the study of the brain's visual cortex, and they have been used in computer image recognition since the 1980s. Over the last ten years, thanks to the increase in computational power, the amount of available training data, and the tricks presented in [Chapter 11](#) for training deep nets, CNNs have managed to achieve superhuman performance on some complex visual tasks. They power image search services, self-driving cars, automatic video classification systems, and more. Moreover, CNNs are not restricted to visual perception: they are also successful at many other tasks, such as voice recognition and natural language processing. However, we will focus on visual applications for now.

In this chapter we will explore where CNNs came from, what their building blocks look like, and how to implement them using Keras. Then we will discuss some of the best CNN architectures, as well as other visual tasks, including object detection (classifying multiple objects in an image and placing bounding boxes around them) and semantic segmentation (classifying each pixel according to the class of the object it belongs to).

The Architecture of the Visual Cortex

David H. Hubel and Torsten Wiesel performed a series of experiments on cats in [1958¹](#) and [1959²](#) (and a [few years later on monkeys³](#)), giving crucial insights into the structure of the visual cortex (the authors received the Nobel Prize in Physiology or Medicine in 1981 for their work). In particular, they showed that many neurons in the visual cortex have a small *local receptive field*, meaning they react only to visual stimuli located in a limited region of the visual field (see [Figure 14-1](#), in which the local receptive fields of five neurons are represented by dashed circles). The receptive fields of different neurons may overlap, and together they tile the whole visual field.

Moreover, the authors showed that some neurons react only to images of horizontal lines, while others react only to lines with different orientations (two neurons may have the same receptive field but react to different line orientations). They also noticed that some neurons have larger receptive fields, and they react to more complex patterns that are combinations of the lower-level patterns. These observations led to the idea that the higher-level neurons are based on the outputs of neighboring lower-level neurons (in [Figure 14-1](#), notice that each neuron is connected only to

¹ David H. Hubel, "Single Unit Activity in Striate Cortex of Unrestrained Cats," *The Journal of Physiology* 147 (1959): 226–238.

² David H. Hubel and Torsten N. Wiesel, "Receptive Fields of Single Neurons in the Cat's Striate Cortex," *The Journal of Physiology* 148 (1959): 574–591.

³ David H. Hubel and Torsten N. Wiesel, "Receptive Fields and Functional Architecture of Monkey Striate Cortex," *The Journal of Physiology* 195 (1968): 215–243.

nearby neurons from the previous layer). This powerful architecture is able to detect all sorts of complex patterns in any area of the visual field.

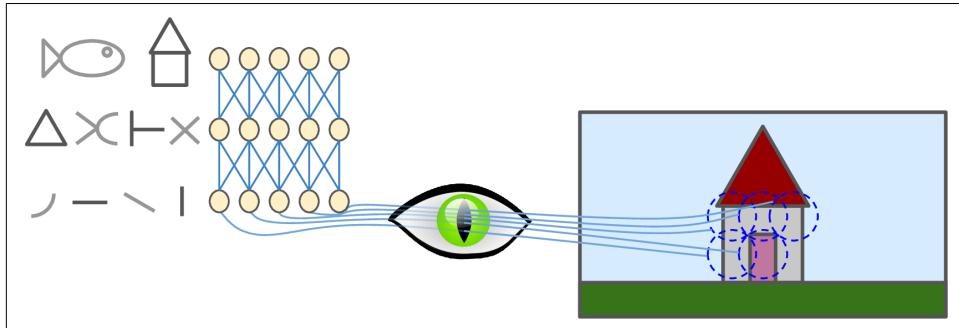


Figure 14-1. Biological neurons in the visual cortex respond to specific patterns in small regions of the visual field called receptive fields; as the visual signal makes its way through consecutive brain modules, neurons respond to more complex patterns in larger receptive fields.

These studies of the visual cortex inspired the [neocognitron](#),⁴ introduced in 1980, which gradually evolved into what we now call *convolutional neural networks*. An important milestone was a [1998 paper](#)⁵ by Yann LeCun et al. that introduced the famous *LeNet-5* architecture, which became widely used by banks to recognize handwritten digits on checks. This architecture has some building blocks that you already know, such as fully connected layers and sigmoid activation functions, but it also introduces two new building blocks: *convolutional layers* and *pooling layers*. Let's look at them now.



Why not simply use a deep neural network with fully connected layers for image recognition tasks? Unfortunately, although this works fine for small images (e.g., MNIST), it breaks down for larger images because of the huge number of parameters it requires. For example, a 100×100 -pixel image has 10,000 pixels, and if the first layer has just 1,000 neurons (which already severely restricts the amount of information transmitted to the next layer), this means a total of 10 million connections. And that's just the first layer. CNNs solve this problem using partially connected layers and weight sharing.

⁴ Kunihiko Fukushima, “Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position,” *Biological Cybernetics* 36 (1980): 193–202.

⁵ Yann LeCun et al., “Gradient-Based Learning Applied to Document Recognition,” *Proceedings of the IEEE* 86, no. 11 (1998): 2278–2324.

Convolutional Layers

The most important building block of a CNN is the *convolutional layer*:⁶ neurons in the first convolutional layer are not connected to every single pixel in the input image (like they were in the layers discussed in previous chapters), but only to pixels in their receptive fields (see [Figure 14-2](#)). In turn, each neuron in the second convolutional layer is connected only to neurons located within a small rectangle in the first layer. This architecture allows the network to concentrate on small low-level features in the first hidden layer, then assemble them into larger higher-level features in the next hidden layer, and so on. This hierarchical structure is common in real-world images, which is one of the reasons why CNNs work so well for image recognition.

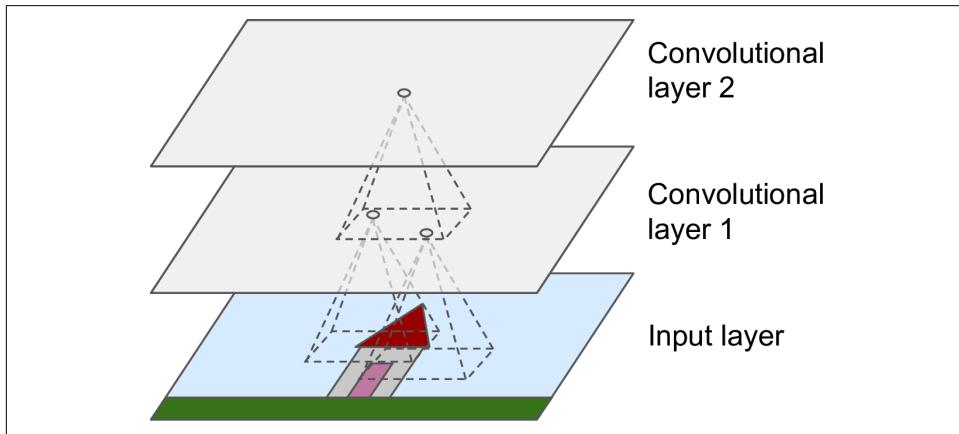


Figure 14-2. CNN layers with rectangular local receptive fields



All the multilayer neural networks we've looked at so far had layers composed of a long line of neurons, and we had to flatten input images to 1D before feeding them to the neural network. In a CNN each layer is represented in 2D, which makes it easier to match neurons with their corresponding inputs.

A neuron located in row i , column j of a given layer is connected to the outputs of the neurons in the previous layer located in rows i to $i + f_h - 1$, columns j to $j + f_w - 1$, where f_h and f_w are the height and width of the receptive field (see [Figure 14-3](#)). In order for a layer to have the same height and width as the previous layer, it is

6 A convolution is a mathematical operation that slides one function over another and measures the integral of their pointwise multiplication. It has deep connections with the Fourier transform and the Laplace transform and is heavily used in signal processing. Convolutional layers actually use cross-correlations, which are very similar to convolutions (see <https://homl.info/76> for more details).

common to add zeros around the inputs, as shown in the diagram. This is called *zero padding*.

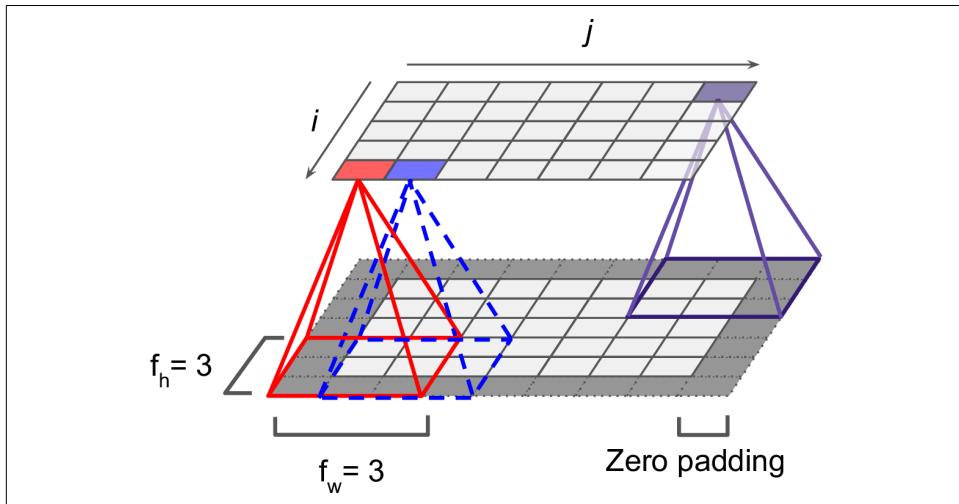


Figure 14-3. Connections between layers and zero padding

It is also possible to connect a large input layer to a much smaller layer by spacing out the receptive fields, as shown in [Figure 14-4](#). This dramatically reduces the model's computational complexity. The horizontal or vertical step size from one receptive field to the next is called the *stride*. In the diagram, a 5×7 input layer (plus zero padding) is connected to a 3×4 layer, using 3×3 receptive fields and a stride of 2 (in this example the stride is the same in both directions, but it does not have to be so). A neuron located in row i , column j in the upper layer is connected to the outputs of the neurons in the previous layer located in rows $i \times s_h$ to $i \times s_h + f_h - 1$, columns $j \times s_w$ to $j \times s_w + f_w - 1$, where s_h and s_w are the vertical and horizontal strides.

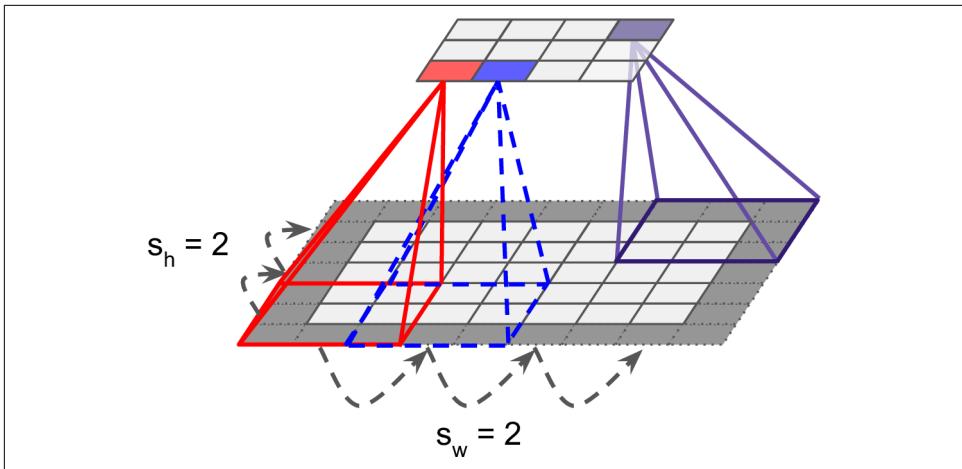


Figure 14-4. Reducing dimensionality using a stride of 2

Filters

A neuron's weights can be represented as a small image the size of the receptive field. For example, Figure 14-5 shows two possible sets of weights, called *filters* (or *convolution kernels*, or just *kernels*). The first one is represented as a black square with a vertical white line in the middle (it is a 7×7 matrix full of 0s except for the central column, which is full of 1s); neurons using these weights will ignore everything in their receptive field except for the central vertical line (since all inputs will get multiplied by 0, except for the ones located in the central vertical line). The second filter is a black square with a horizontal white line in the middle. Neurons using these weights will ignore everything in their receptive field except for the central horizontal line.

Now if all neurons in a layer use the same vertical line filter (and the same bias term), and you feed the network the input image shown in Figure 14-5 (the bottom image), the layer will output the top-left image. Notice that the vertical white lines get enhanced while the rest gets blurred. Similarly, the upper-right image is what you get if all neurons use the same horizontal line filter; notice that the horizontal white lines get enhanced while the rest is blurred out. Thus, a layer full of neurons using the same filter outputs a *feature map*, which highlights the areas in an image that activate the filter the most. But don't worry, you won't have to define the filters manually: instead, during training the convolutional layer will automatically learn the most useful filters for its task, and the layers above will learn to combine them into more complex patterns.

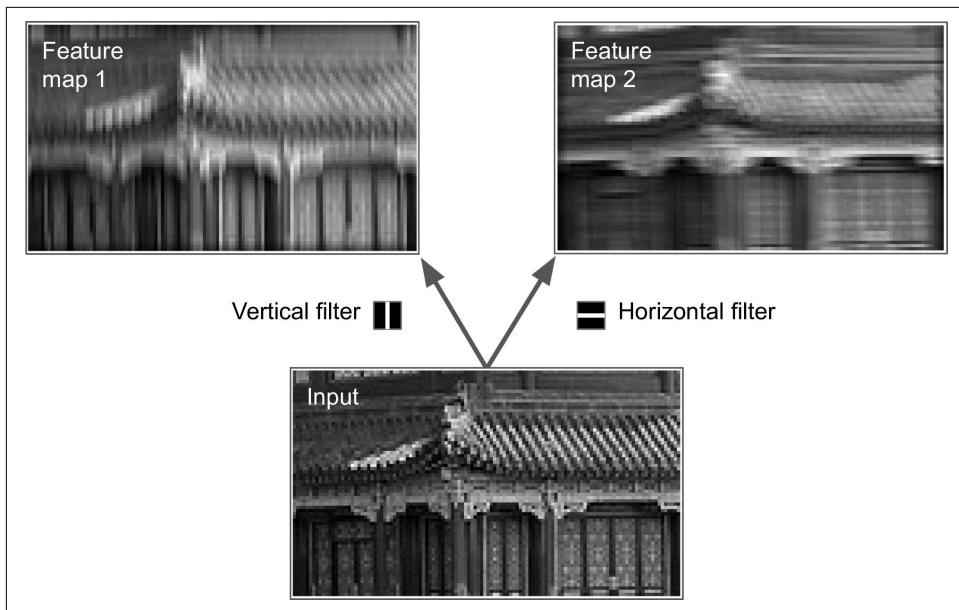


Figure 14-5. Applying two different filters to get two feature maps

Stacking Multiple Feature Maps

Up to now, for simplicity, I have represented the output of each convolutional layer as a 2D layer, but in reality a convolutional layer has multiple filters (you decide how many) and outputs one feature map per filter, so it is more accurately represented in 3D (see [Figure 14-6](#)). It has one neuron per pixel in each feature map, and all neurons within a given feature map share the same parameters (i.e., the same kernel and bias term). Neurons in different feature maps use different parameters. A neuron's receptive field is the same as described earlier, but it extends across all the feature maps of the previous layer. In short, a convolutional layer simultaneously applies multiple trainable filters to its inputs, making it capable of detecting multiple features anywhere in its inputs.



The fact that all neurons in a feature map share the same parameters dramatically reduces the number of parameters in the model. Once the CNN has learned to recognize a pattern in one location, it can recognize it in any other location. In contrast, once a fully-connected neural network has learned to recognize a pattern in one location, it can only recognize it in that particular location.

Input images are also composed of multiple sublayers: one per *color channel*. There are typically three: red, green, and blue (RGB). Grayscale images have just one

channel, but some images may have much more—for example, satellite images that capture extra light frequencies (such as infrared).

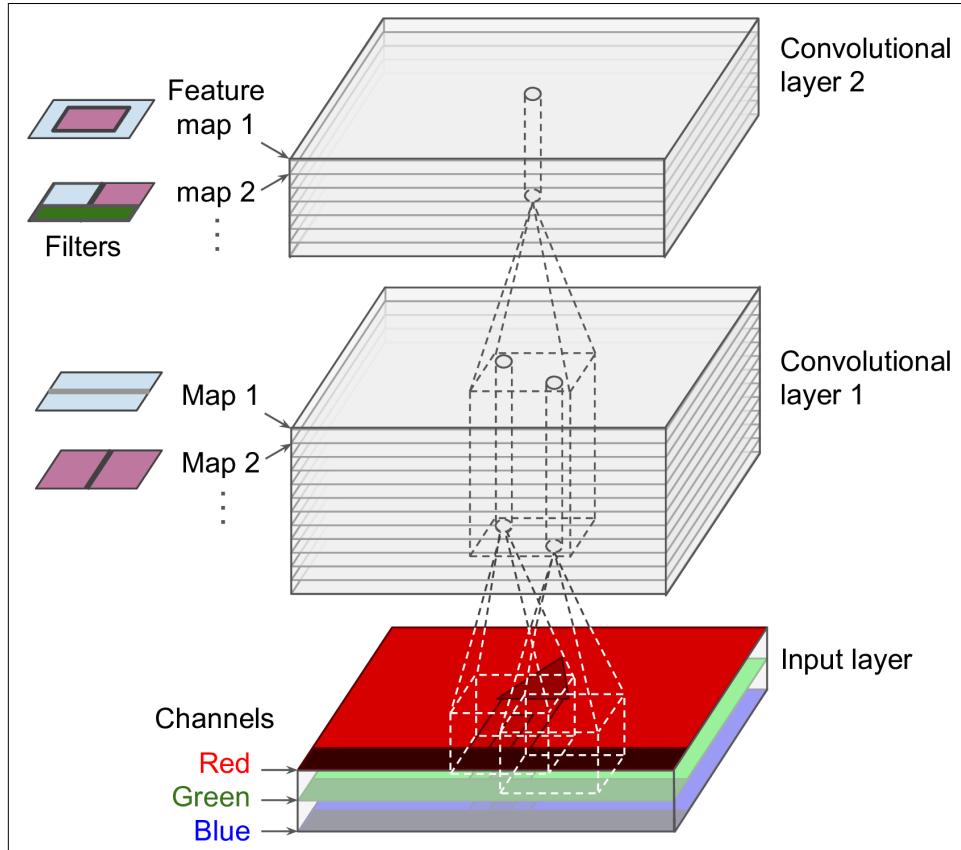


Figure 14-6. Two convolutional layers with multiple filters each (kernels), processing a color image with three color channels. Each convolutional layer outputs one feature map per filter.

Specifically, a neuron located in row i , column j of the feature map k in a given convolutional layer l is connected to the outputs of the neurons in the previous layer $l - 1$, located in rows $i \times s_h$ to $i \times s_h + f_h - 1$ and columns $j \times s_w$ to $j \times s_w + f_w - 1$, across all feature maps (in layer $l - 1$). Note that, within a layer, all neurons located in the same row i and column j but in different feature maps are connected to the outputs of the exact same neurons in the previous layer.

Equation 14-1 summarizes the preceding explanations in one big mathematical equation: it shows how to compute the output of a given neuron in a convolutional layer.

It is a bit ugly due to all the different indices, but all it does is calculate the weighted sum of all the inputs, plus the bias term.

Equation 14-1. Computing the output of a neuron in a convolutional layer

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} \times w_{u,v,k',k} \quad \text{with} \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}$$

In this equation:

- $z_{i,j,k}$ is the output of the neuron located in row i , column j in feature map k of the convolutional layer (layer l).
- As explained earlier, s_h and s_w are the vertical and horizontal strides, f_h and f_w are the height and width of the receptive field, and $f_{n'}$ is the number of feature maps in the previous layer (layer $l - 1$).
- $x_{i',j',k'}$ is the output of the neuron located in layer $l - 1$, row i' , column j' , feature map k' (or channel k' if the previous layer is the input layer).
- b_k is the bias term for feature map k (in layer l). You can think of it as a knob that tweaks the overall brightness of the feature map k .
- $w_{u,v,k',k}$ is the connection weight between any neuron in feature map k of the layer l and its input located at row u , column v (relative to the neuron's receptive field), and feature map k' .

Let's see how to create and use a convolutional layer using Keras.

Implementing Convolutional Layers With Keras

First, let's load and preprocess a couple of sample images, using Scikit-Learn's `load_sample_image()` function, and Keras's `CenterCrop` and `Rescaling` layers (all of which were introduced in [Chapter 13](#)):

```
from sklearn.datasets import load_sample_images
import tensorflow as tf

images = load_sample_images()["images"]
images = tf.keras.layers.CenterCrop(height=70, width=120)(images)
images = tf.keras.layers.Rescaling(scale=1 / 255)(images)
```

Let's look at the shape of the `images` tensor:

```
>>> images.shape
TensorShape([2, 70, 120, 3])
```

Yikes, it's a 4D tensor, we haven't seen this before! What do all these dimensions mean? Well, there are two sample images, which explains the first dimension. Then

each image is 70×120 , since that's the size we specified when creating the Center Crop layer (the original images were 427×640). This explains the second and third dimensions. And lastly, each pixel holds one value per color channel, and there are three of them—red, green, and blue—which explains the last dimension.

Now let's create a 2D convolutional layer and feed it these images to see what comes out. For this, Keras provides a Convolution2D layer, alias Conv2D. Under the hood, this layer relies on TensorFlow's `tf.nn.conv2d()` operation. Let's create a convolutional layer with 32 filters, each of size 7×7 (using `kernel_size=7`, which is equivalent to using `kernel_size=(7, 7)`), and let's apply this layer to our small batch of two images:

```
conv_layer = tf.keras.layers.Conv2D(filters=32, kernel_size=7)
fmaps = conv_layer(images)
```



When we talk about a 2D convolutional layer, “2D” refers to the number of *spatial* dimensions (height and width), but as you can see, the layer takes 4D inputs: as we saw, the two additional dimensions are the batch size (first dimension) and the channels (last dimension).

Now let's look at the output's shape:

```
>>> fmaps.shape
TensorShape([2, 64, 114, 32])
```

The output shape is similar to the input shape, with two main differences: first, there are 32 channels instead of 3. This is because we set `filters=32`, so we get 32 output feature maps: instead of the intensity of red, green, and blue, at each location, we now have the intensity of each feature at each location. Second, the height and width have both shrunk by 6 pixels. This is due to the fact that the Conv2D layer does not use any zero-padding by default, which means that we lose a few pixels on the sides of the output feature maps, depending on the size of the filters. In this case, since the kernel size is 7, we lose 6 pixels horizontally and 6 pixels vertically (i.e., 3 pixels on each side).



The default option is surprisingly named `padding="valid"`, which actually means no zero-padding at all! This name comes from the fact that in this case every neuron's receptive field lies strictly within *valid* positions inside the input (it does not go out of bounds). It's not a Keras naming quirk: everyone uses this odd nomenclature.

If instead we set `padding="same"`, then the inputs are padded with enough zeros on all sides to ensure that the output feature maps end up with the *same* size as the inputs (hence the name of this option):

```
>>> conv_layer = tf.keras.layers.Conv2D(filters=32, kernel_size=7,
...                                         padding="same")
...
...
>>> fmaps = conv_layer(images)
>>> fmaps.shape
TensorShape([2, 70, 120, 32])
```

These two padding options are illustrated in [Figure 14-7](#). For simplicity, only the horizontal dimension is shown here, but of course the same logic applies to the vertical dimension as well.

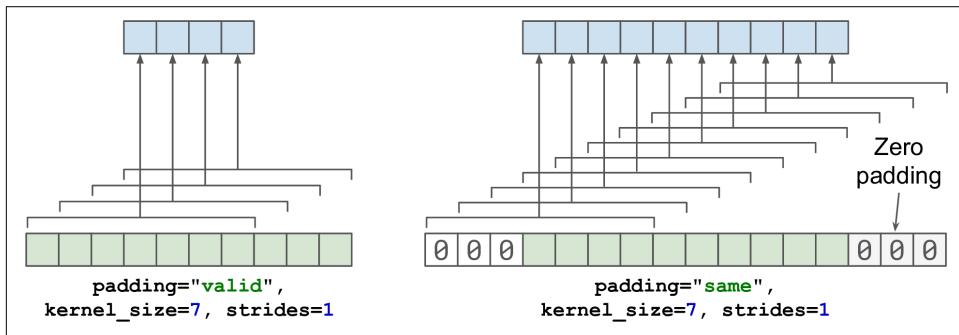


Figure 14-7. The two padding options, when `strides=1`

If the stride is greater than 1 (in any direction), then the output size will not be equal to the input size, even if `padding="same"`. For example, if you set `strides=2` (or equivalently `strides=(2, 2)`), then the output feature maps will be 35×60 : halved both vertically and horizontally. [Figure 14-8](#) shows what happens when `strides=2`, with both padding options.

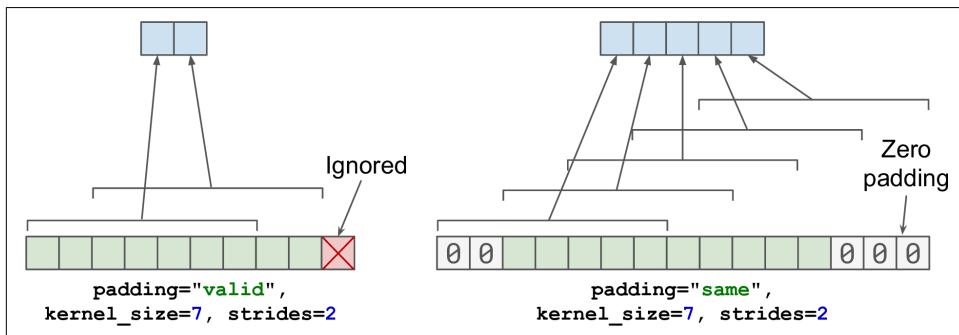


Figure 14-8. With `strides` greater than 1, the output is much smaller even when using "same" padding (and "valid" padding may ignore some inputs)

If you are curious, this is how the output size is computed:

- With `padding="valid"`, if the width of the input is i_h , then the output width is equal to $(i_h - f_h + s_h) / s_h$, rounded down. Recall that f_h is the kernel width, and s_h is the horizontal stride. Any remainder in the division corresponds to ignored columns on the right side of the input image. The same logic can be used to compute the output height, and any ignored rows at the bottom of the image.
- With `padding="same"`, the output width is equal to i_h / s_h , rounded up. To make this possible, the appropriate number of zero columns are padded to the left and right of the input image (an equal number if possible, or just one more on the right side). Assuming the output width is o_w , then the number of padded zero columns is $(o_w - 1) \times s_h + f_h - i_h$. Again, the same logic can be used to compute the output height, and the number of padded rows.

Now let's look at the layer's weights (which were noted $w_{u, v, k', k}$ and b_k in [Equation 14-1](#)). Just like a Dense layer, a Conv2D layer holds all the layer's weights, including the kernels and biases. The kernels are initialized randomly, while the biases are initialized to zero. These weights are accessible as TF variables via the `weights` attribute, or as NumPy arrays via the `get_weights()` method:

```
>>> kernels, biases = conv_layer.get_weights()
>>> kernels.shape
(7, 7, 3, 32)
>>> biases.shape
(32,)
```

The `kernels` array is 4-dimensional, and its shape is [*kernel_height*, *kernel_width*, *input_channels*, *output_channels*]. The `biases` array is 1D, with shape [*output_channels*]. The number of output channels is equal to the number of output feature maps, which is also equal to the number of filters.

Most importantly, note that the height and width of the input images do not appear in the kernel's shape: this is because all the neurons in the output feature maps share the same weights, as explained earlier. This means that you can feed images of any size to this layer, as long as they are at least as large as the kernels, and if they have the right number of channels (3 in this case).

Lastly, you will generally want to specify an activation function (such as ReLU) when creating a Conv2D layer, and also specify the corresponding kernel initializer (such as He initialization). This is for the same reason as for Dense layers: a convolutional layer performs a linear operation, so if you stacked multiple convolutional layers without any activation functions, they would all be equivalent to a single convolutional layer, so they wouldn't be able to learn anything really complex.

As you can see, convolutional layers have quite a few hyperparameters: `filters`, `kernel_size`, `padding`, `strides`, `activation`, `kernel_initializer`, etc. As always, you can use cross-validation to find the right hyperparameter values, but this is very time-consuming. We will discuss common CNN architectures later in this chapter, to give you some idea of which hyperparameter values work best in practice.

Memory Requirements

Another challenge with CNNs is that the convolutional layers require a huge amount of RAM. This is especially true during training, because the reverse pass of backpropagation requires all the intermediate values computed during the forward pass.

For example, consider a convolutional layer with 200 5×5 filters, with stride 1 and “same” padding. If the input is a 150×100 RGB image (three channels), then the number of parameters is $(5 \times 5 \times 3 + 1) \times 200 = 15,200$ (the + 1 corresponds to the bias terms), which is fairly small compared to a fully connected layer.⁷ However, each of the 200 feature maps contains 150×100 neurons, and each of these neurons needs to compute a weighted sum of its $5 \times 5 \times 3 = 75$ inputs: that’s a total of 225 million float multiplications. Not as bad as a fully connected layer, but still quite computationally intensive. Moreover, if the feature maps are represented using 32-bit floats, then the convolutional layer’s output will occupy $200 \times 150 \times 100 \times 32 = 96$ million bits (12 MB) of RAM.⁸ And that’s just for one instance—if a training batch contains 100 instances, then this layer will use up 1.2 GB of RAM!

During inference (i.e., when making a prediction for a new instance) the RAM occupied by one layer can be released as soon as the next layer has been computed, so you only need as much RAM as required by two consecutive layers. But during training everything computed during the forward pass needs to be preserved for the reverse pass, so the amount of RAM needed is (at least) the total amount of RAM required by all layers.



If training crashes because of an out-of-memory error, you can try reducing the mini-batch size. Alternatively, you can try reducing dimensionality using a stride, or removing a few layers. Or you can try using 16-bit floats instead of 32-bit floats. Or you could distribute the CNN across multiple devices (we will see how to do this in [Chapter 19](#)).

⁷ To produce the same size outputs, a fully connected layer would need $200 \times 150 \times 100$ neurons, each connected to all $150 \times 100 \times 3$ inputs. It would have $200 \times 150 \times 100 \times (150 \times 100 \times 3 + 1) \approx 135$ billion parameters!

⁸ In the international system of units (SI), 1 MB = 1,000 KB = 1,000 × 1,000 bytes = 1,000 × 1,000 × 8 bits. And 1 MiB = 1,024 kB = 1,024 × 1,024 bytes. So 12 MB ≈ 11.44 MiB.

Now let's look at the second common building block of CNNs: the *pooling layer*.

Pooling Layers

Once you understand how convolutional layers work, the pooling layers are quite easy to grasp. Their goal is to *subsample* (i.e., shrink) the input image in order to reduce the computational load, the memory usage, and the number of parameters (thereby limiting the risk of overfitting).

Just like in convolutional layers, each neuron in a pooling layer is connected to the outputs of a limited number of neurons in the previous layer, located within a small rectangular receptive field. You must define its size, the stride, and the padding type, just like before. However, a pooling neuron has no weights; all it does is aggregate the inputs using an aggregation function such as the max or mean. [Figure 14-9](#) shows a *max pooling layer*, which is the most common type of pooling layer. In this example, we use a 2×2 pooling kernel,⁹ with a stride of 2 and no padding. Only the max input value in each receptive field makes it to the next layer, while the other inputs are dropped. For example, in the lower-left receptive field in [Figure 14-9](#), the input values are 1, 5, 3, 2, so only the max value, 5, is propagated to the next layer. Because of the stride of 2, the output image has half the height and half the width of the input image (rounded down since we use no padding).

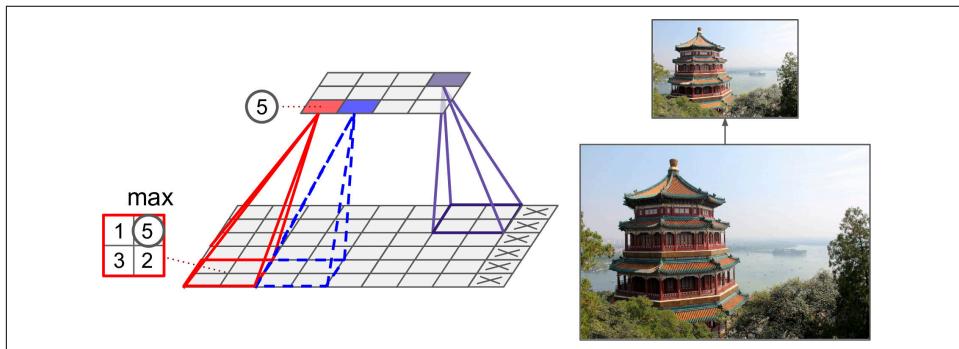


Figure 14-9. Max pooling layer (2×2 pooling kernel, stride 2, no padding)



A pooling layer typically works on every input channel independently, so the output depth (i.e., the number of channels) is the same as the input depth.

⁹ Other kernels we've discussed so far had weights, but pooling kernels do not: they are just stateless sliding windows.

Other than reducing computations, memory usage, and the number of parameters, a max pooling layer also introduces some level of *invariance* to small translations, as shown in [Figure 14-10](#). Here we assume that the bright pixels have a lower value than dark pixels, and we consider three images (A, B, C) going through a max pooling layer with a 2×2 kernel and stride 2. Images B and C are the same as image A, but shifted by one and two pixels to the right. As you can see, the outputs of the max pooling layer for images A and B are identical. This is what translation invariance means. For image C, the output is different: it is shifted one pixel to the right (but there is still 50% invariance). By inserting a max pooling layer every few layers in a CNN, it is possible to get some level of translation invariance at a larger scale. Moreover, max pooling offers a small amount of rotational invariance and a slight scale invariance. Such invariance (even if it is limited) can be useful in cases where the prediction should not depend on these details, such as in classification tasks.

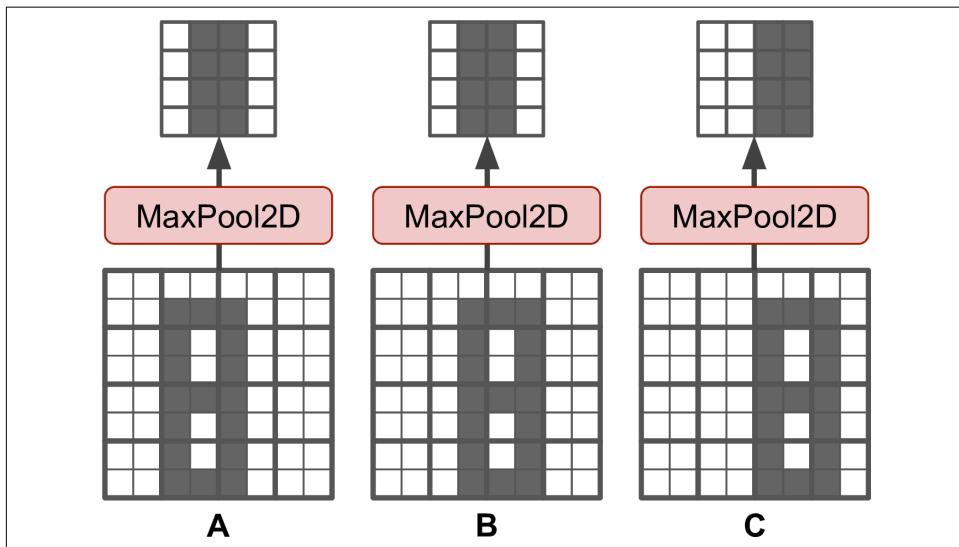


Figure 14-10. Invariance to small translations

However, max pooling has some downsides too. Firstly, it is obviously very destructive: even with a tiny 2×2 kernel and a stride of 2, the output will be two times smaller in both directions (so its area will be four times smaller), simply dropping 75% of the input values. And in some applications, invariance is not desirable. Take semantic segmentation (the task of classifying each pixel in an image according to the object that pixel belongs to, which we'll explore later in this chapter): obviously, if the input image is translated by one pixel to the right, the output should also be translated by one pixel to the right. The goal in this case is *equivariance*, not invariance: a small change to the inputs should lead to a corresponding small change in the output.

Implementing Pooling Layers With Keras

The following code creates a `MaxPooling2D` layer, alias `MaxPool2D`, using a 2×2 kernel. The strides default to the kernel size, so this layer will use a stride of 2 (both horizontally and vertically). By default, it uses “valid” padding (i.e., no padding at all):

```
max_pool = tf.keras.layers.MaxPool2D(pool_size=2)
```

To create an *average pooling layer*, just use `AveragePooling2D`, alias `AvgPool2D`, instead of `MaxPool2D`. As you might expect, it works exactly like a max pooling layer, except it computes the mean rather than the max. Average pooling layers used to be very popular, but people mostly use max pooling layers now, as they generally perform better. This may seem surprising, since computing the mean generally loses less information than computing the max. But on the other hand, max pooling preserves only the strongest features, getting rid of all the meaningless ones, so the next layers get a cleaner signal to work with. Moreover, max pooling offers stronger translation invariance than average pooling, and it requires slightly less compute.

Note that max pooling and average pooling can be performed along the depth dimension rather than the spatial dimensions, although this is not as common. This can allow the CNN to learn to be invariant to various features. For example, it could learn multiple filters, each detecting a different rotation of the same pattern (such as hand-written digits; see [Figure 14-11](#)), and the depthwise max pooling layer would ensure that the output is the same regardless of the rotation. The CNN could similarly learn to be invariant to anything else: thickness, brightness, skew, color, and so on.

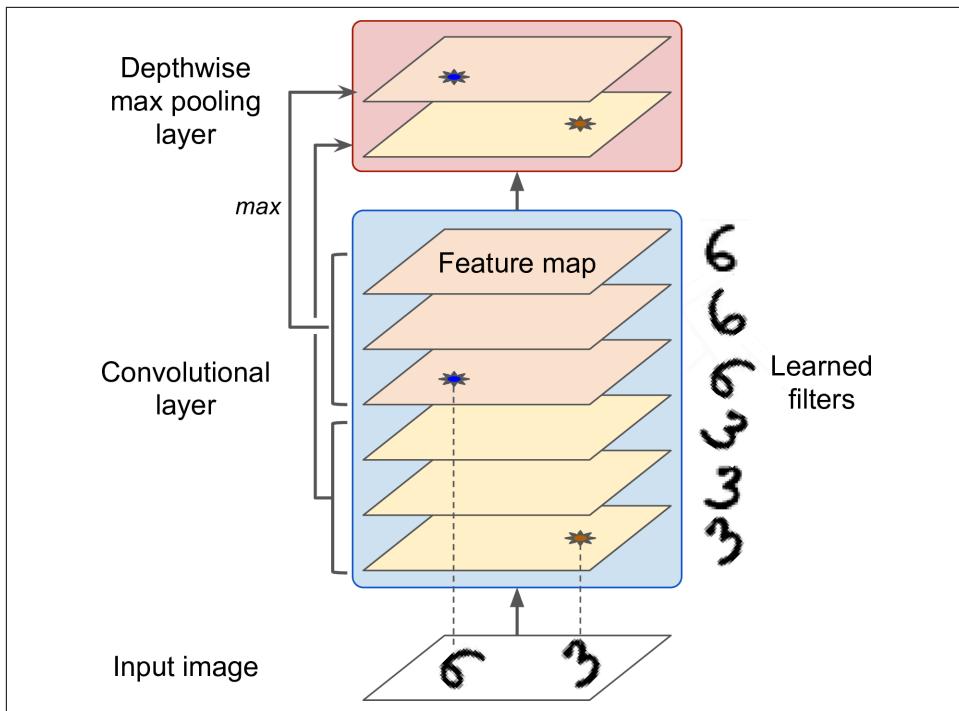


Figure 14-11. Depthwise max pooling can help the CNN learn to be invariant (to rotation in this case).

Keras does not include a depthwise max pooling layer, but it's not too difficult to implement a custom layer for that:

```
class DepthPool(tf.keras.layers.Layer):
    def __init__(self, pool_size=2, **kwargs):
        super().__init__(**kwargs)
        self.pool_size = pool_size

    def call(self, inputs):
        shape = tf.shape(inputs) # shape[-1] is the number of channels
        groups = shape[-1] // self.pool_size # number of channel groups
        new_shape = tf.concat([shape[:-1], [groups, self.pool_size]], axis=0)
        return tf.reduce_max(tf.reshape(inputs, new_shape), axis=-1)
```

This layer reshapes its inputs to split the channels into groups of the desired size (`pool_size`), then it uses `tf.reduce_max()` to compute the max of each group. This implementation assumes that the stride is equal to the pool size, which is generally what you want. Alternatively, you could use TensorFlow's `tf.nn.max_pool()` operation, and wrap in a Lambda layer to use it inside a Keras model, but sadly this op does not implement depthwise pooling for the GPU, only for the CPU.

One last type of pooling layer that you will often see in modern architectures is the *global average pooling layer*. It works very differently: all it does is compute the mean of each entire feature map (it's like an average pooling layer using a pooling kernel with the same spatial dimensions as the inputs). This means that it just outputs a single number per feature map and per instance. Although this is of course extremely destructive (most of the information in the feature map is lost), it can be useful just before the output layer, as we will see later in this chapter. To create such a layer, simply use the `GlobalAveragePooling2D` class, alias `GlobalAvgPool2D`:

```
global_avg_pool = tf.keras.layers.GlobalAvgPool2D()
```

It's equivalent to the following `Lambda` layer, which computes the mean over the spatial dimensions (height and width):

```
global_avg_pool = tf.keras.layers.Lambda(  
    lambda X: tf.reduce_mean(X, axis=[1, 2]))
```

For example, if we apply this layer to the input images, we get the mean intensity of red, green, and blue for each image:

```
>>> global_avg_pool(images)  
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=  
array([[0.64338624, 0.5971759 , 0.5824972 ],  
       [0.76306933, 0.26011038, 0.10849128]], dtype=float32)>
```

Now you know all the building blocks to create convolutional neural networks. Let's see how to assemble them.

CNN Architectures

Typical CNN architectures stack a few convolutional layers (each one generally followed by a ReLU layer), then a pooling layer, then another few convolutional layers (+ReLU), then another pooling layer, and so on. The image gets smaller and smaller as it progresses through the network, but it also typically gets deeper and deeper (i.e., with more feature maps), thanks to the convolutional layers (see [Figure 14-12](#)). At the top of the stack, a regular feedforward neural network is added, composed of a few fully connected layers (+ReLUs), and the final layer outputs the prediction (e.g., a softmax layer that outputs estimated class probabilities).

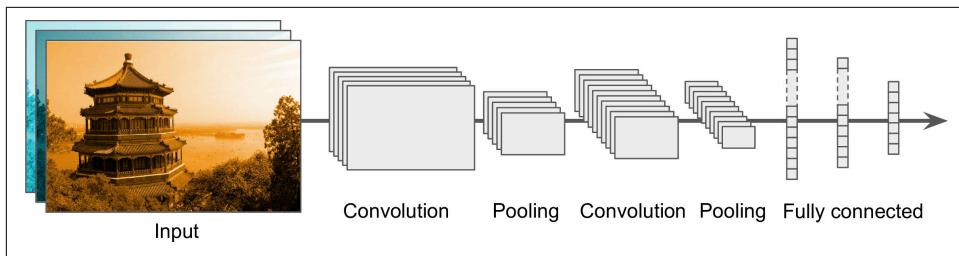


Figure 14-12. Typical CNN architecture



A common mistake is to use convolution kernels that are too large. For example, instead of using a convolutional layer with a 5×5 kernel, stack two layers with 3×3 kernels: it will use fewer parameters and require fewer computations, and it will usually perform better. One exception is for the first convolutional layer: it can typically have a large kernel (e.g., 5×5), usually with a stride of 2 or more: this will reduce the spatial dimension of the image without losing too much information, and since the input image only has three channels in general, it will not be too costly.

Here is how you can implement a basic CNN to tackle the Fashion MNIST dataset (introduced in [Chapter 10](#)):

```
from functools import partial

DefaultConv2D = partial(tf.keras.layers.Conv2D, kernel_size=3, padding="same",
                      activation="relu", kernel_initializer="he_normal")
model = tf.keras.Sequential([
    DefaultConv2D(filters=64, kernel_size=7, input_shape=[28, 28, 1]),
    tf.keras.layers.MaxPool2D(),
    DefaultConv2D(filters=128),
    DefaultConv2D(filters=128),
    tf.keras.layers.MaxPool2D(),
    DefaultConv2D(filters=256),
    DefaultConv2D(filters=256),
    tf.keras.layers.MaxPool2D(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(units=128, activation="relu",
                          kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(units=64, activation="relu",
                          kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(units=10, activation="softmax")
])
```

Let's go through this code:

- We use the `functools.partial()` function (introduced in [Chapter 11](#)) to define `DefaultConv2D`, which acts just like `Conv2D` but with different default arguments: a small kernel size of 3, “same” padding, the ReLU activation function, and its corresponding He initializer.
- Next we create the `Sequential` model. Its first layer is a `DefaultConv2D` with 64 fairly large filters (7×7). It uses the default stride of 1 because the input images are not very large. It also sets `input_shape=[28, 28, 1]`, because the images are 28×28 pixels, with a single color channel (i.e., grayscale). When you load the Fashion MNIST dataset, make sure each image has this shape: you may need to use `np.reshape()` or `np.expand_dims()` to add the channels dimension. Alternatively, you could use a `Reshape` layer as the first layer in the model.
- We then add a max pooling layer which uses the default pool size of 2, so it divides each spatial dimension by a factor of 2.
- Then we repeat the same structure twice: two convolutional layers followed by a max pooling layer. For larger images, we could repeat this structure several more times. The number of repetitions is a hyperparameter you can tune.
- Note that the number of filters doubles as we climb up the CNN toward the output layer (it is initially 64, then 128, then 256): it makes sense for it to grow, since the number of low-level features is often fairly low (e.g., small circles, horizontal lines), but there are many different ways to combine them into higher-level features. It is a common practice to double the number of filters after each pooling layer: since a pooling layer divides each spatial dimension by a factor of 2, we can afford to double the number of feature maps in the next layer without fear of exploding the number of parameters, memory usage, or computational load.
- Next is the fully connected network, composed of two hidden dense layers and a dense output layer. Since it’s a classification task with 10 classes, the output layer has 10 units and it uses the softmax activation function. Note that we must flatten the inputs just before the first dense layer, since it expects a 1D array of features for each instance. We also add two dropout layers, with a dropout rate of 50% each, to reduce overfitting.

If you compile this model using the `"sparse_categorical_crossentropy"` loss, and you fit the model to the Fashion MNIST training set, it should reach over 92% accuracy on the test set. It’s not state of the art, but it is pretty good, and clearly much better than what we achieved with dense networks in [Chapter 10](#).

Over the years, variants of this fundamental architecture have been developed, leading to amazing advances in the field. A good measure of this progress is the error rate in competitions such as the ILSVRC [ImageNet challenge](#). In this competition the top-five error rate for image classification fell from over 26% to less than 2.3%

in just six years. The top-five error rate is the number of test images for which the system's top five predictions did *not* include the correct answer. The images are fairly large (e.g., 256 pixels high) and there are 1,000 classes, some of which are really subtle (try distinguishing 120 dog breeds). Looking at the evolution of the winning entries is a good way to understand how CNNs work, and how research in Deep Learning progresses.

We will first look at the classical LeNet-5 architecture (1998), then several winners of the ILSVRC challenge: AlexNet (2012), GoogLeNet (2014), ResNet (2015), and SENet (2017). Along the way, we will also look at a few more architectures, including Xception, ResNext, DenseNet, MobileNet, CSPNet, and EfficientNet.

LeNet-5

The [LeNet-5 architecture](#)¹⁰ is perhaps the most widely known CNN architecture. As mentioned earlier, it was created by Yann LeCun in 1998 and has been widely used for handwritten digit recognition (MNIST). It is composed of the layers shown in [Table 14-1](#).

Table 14-1. LeNet-5 architecture

Layer	Type	Maps	Size	Kernel size	Stride	Activation
Out	Fully connected	—	10	—	—	RBF
F6	Fully connected	—	84	—	—	tanh
C5	Convolution	120	1×1	5×5	1	tanh
S4	Avg pooling	16	5×5	2×2	2	tanh
C3	Convolution	16	10×10	5×5	1	tanh
S2	Avg pooling	6	14×14	2×2	2	tanh
C1	Convolution	6	28×28	5×5	1	tanh
In	Input	1	32×32	—	—	—

As you can see, this looks pretty similar to our Fashion MNIST model: a stack of convolutional layers and pooling layers, followed by a dense network. Perhaps the main difference with more modern classification CNNs is the activation functions: today, we would use ReLU instead of tanh, and softmax instead of RBF. There were several other minor differences which don't really matter much, but in case you are interested, they are listed in the notebook. Yann LeCun's [website](#) also features great demos of LeNet-5 classifying digits.

¹⁰ Yann LeCun et al., “Gradient-Based Learning Applied to Document Recognition,” *Proceedings of the IEEE* 86, no. 11 (1998): 2278–2324.

AlexNet

The [AlexNet CNN architecture](#)¹¹ won the 2012 ImageNet ILSVRC challenge by a large margin: it achieved a top-five error rate of 17%, while the second best achieved only 26%! It was developed by Alex Krizhevsky (hence the name), Ilya Sutskever, and Geoffrey Hinton. It is similar to LeNet-5, only much larger and deeper, and it was the first to stack convolutional layers directly on top of one another, instead of stacking a pooling layer on top of each convolutional layer. [Table 14-2](#) presents this architecture.

Table 14-2. AlexNet architecture

Layer	Type	Maps	Size	Kernel size	Stride	Padding	Activation
Out	Fully connected	–	1,000	–	–	–	Softmax
F10	Fully connected	–	4,096	–	–	–	ReLU
F9	Fully connected	–	4,096	–	–	–	ReLU
S8	Max pooling	256	6×6	3×3	2	valid	–
C7	Convolution	256	13×13	3×3	1	same	ReLU
C6	Convolution	384	13×13	3×3	1	same	ReLU
C5	Convolution	384	13×13	3×3	1	same	ReLU
S4	Max pooling	256	13×13	3×3	2	valid	–
C3	Convolution	256	27×27	5×5	1	same	ReLU
S2	Max pooling	96	27×27	3×3	2	valid	–
C1	Convolution	96	55×55	11×11	4	valid	ReLU
In	Input	3 (RGB)	227×227	–	–	–	–

To reduce overfitting, the authors used two regularization techniques. First, they applied dropout (introduced in [Chapter 11](#)) with a 50% dropout rate during training to the outputs of layers F9 and F10. Second, they performed *data augmentation* by randomly shifting the training images by various offsets, flipping them horizontally, and changing the lighting conditions.

Data Augmentation

Data augmentation artificially increases the size of the training set by generating many realistic variants of each training instance. This reduces overfitting, making this a regularization technique. The generated instances should be as realistic as possible: ideally, given an image from the augmented training set, a human should not be able

¹¹ Alex Krizhevsky et al., “ImageNet Classification with Deep Convolutional Neural Networks,” *Proceedings of the 25th International Conference on Neural Information Processing Systems* 1 (2012): 1097–1105.

to tell whether it was augmented or not. Simply adding white noise will not help; the modifications should be learnable (white noise is not).

For example, you can slightly shift, rotate, and resize every picture in the training set by various amounts and add the resulting pictures to the training set (see [Figure 14-13](#)). To do this, you can use Keras's data augmentation layers, introduced in [Chapter 13](#) (e.g., `RandomCrop`, `RandomRotation`, etc.). This forces the model to be more tolerant to variations in the position, orientation, and size of the objects in the pictures. To produce a model that's more tolerant of different lighting conditions, you can similarly generate many images with various contrasts. In general, you can also flip the pictures horizontally (except for text, and other asymmetrical objects). By combining these transformations, you can greatly increase the size of your training set.

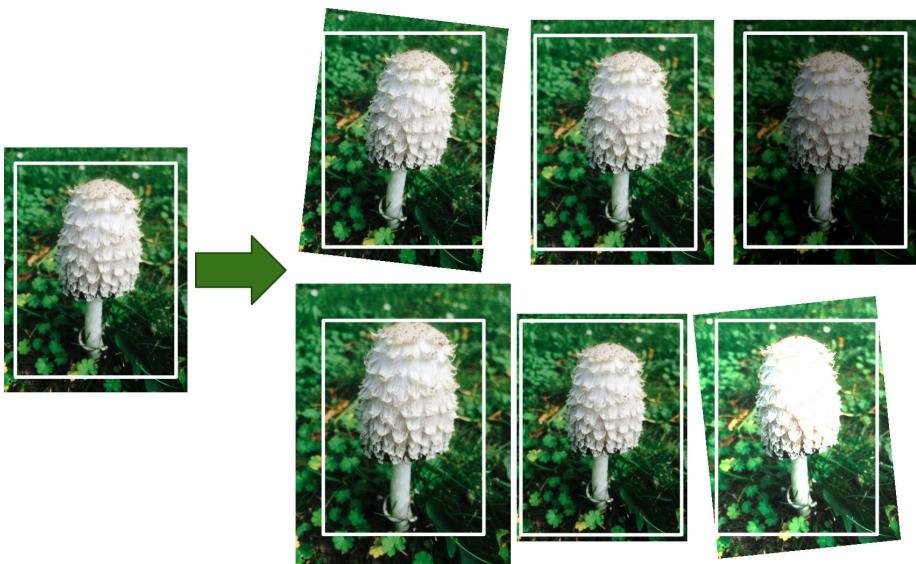


Figure 14-13. Generating new training instances from existing ones

Data augmentation is also useful when you have an unbalanced dataset: you can use it to generate more samples of the less frequent classes. This is called the *Synthetic Minority Oversampling Technique*, or SMOTE for short.

AlexNet also uses a competitive normalization step immediately after the ReLU step of layers C1 and C3, called *local response normalization* (LRN): the most strongly activated neurons inhibit other neurons located at the same position in neighboring feature maps. Such competitive activation has been observed in biological neurons. This encourages different feature maps to specialize, pushing them apart and forcing

them to explore a wider range of features, ultimately improving generalization. [Equation 14-2](#) shows how to apply LRN.

Equation 14-2. Local response normalization (LRN)

$$b_i = a_i \left(k + \alpha \sum_{j=j_{\text{low}}}^{j_{\text{high}}} a_j^2 \right)^{-\beta} \quad \text{with} \quad \begin{cases} j_{\text{high}} = \min \left(i + \frac{r}{2}, f_n - 1 \right) \\ j_{\text{low}} = \max \left(0, i - \frac{r}{2} \right) \end{cases}$$

In this equation:

- b_i is the normalized output of the neuron located in feature map i , at some row u and column v (note that in this equation we consider only neurons located at this row and column, so u and v are not shown).
- a_i is the activation of that neuron after the ReLU step, but before normalization.
- k , α , β , and r are hyperparameters. k is called the *bias*, and r is called the *depth radius*.
- f_n is the number of feature maps.

For example, if $r = 2$ and a neuron has a strong activation, it will inhibit the activation of the neurons located in the feature maps immediately above and below its own.

In AlexNet, the hyperparameters are set as follows: $r = 5$, $\alpha = 0.0001$, $\beta = 0.75$, and $k = 2$. This step can be implemented using the `tf.nn.local_response_normalization()` function (which you can wrap in a `Lambda` layer if you want to use it in a Keras model).

A variant of AlexNet called [ZF Net](#)¹² was developed by Matthew Zeiler and Rob Fergus and won the 2013 ILSVRC challenge. It is essentially AlexNet with a few tweaked hyperparameters (number of feature maps, kernel size, stride, etc.).

GoogLeNet

The [GoogLeNet architecture](#) was developed by Christian Szegedy et al. from Google Research,¹³ and it won the ILSVRC 2014 challenge by pushing the top-five error rate below 7%. This great performance came in large part from the fact that the network was much deeper than previous CNNs (as you'll see in [Figure 14-15](#)). This was made

¹² Matthew D. Zeiler and Rob Fergus, “Visualizing and Understanding Convolutional Networks,” *Proceedings of the European Conference on Computer Vision* (2014): 818-833.

¹³ Christian Szegedy et al., “Going Deeper with Convolutions,” *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2015): 1-9.

possible by subnetworks called *inception modules*,¹⁴ which allow GoogLeNet to use parameters much more efficiently than previous architectures: GoogLeNet actually has 10 times fewer parameters than AlexNet (roughly 6 million instead of 60 million).

Figure 14-14 shows the architecture of an inception module. The notation “ $3 \times 3 + 1(S)$ ” means that the layer uses a 3×3 kernel, stride 1, and “same” padding. The input signal is first fed to four different layers in parallel. All convolutional layers use the ReLU activation function. Note that top convolutional layers use different kernel sizes (1×1 , 3×3 , and 5×5), allowing them to capture patterns at different scales. Also note that every single layer uses a stride of 1 and “same” padding (even the max pooling layer), so their outputs all have the same height and width as their inputs. This makes it possible to concatenate all the outputs along the depth dimension in the final *depth concatenation layer* (i.e., stack the feature maps from all four top convolutional layers). It can be implemented using Keras’s `Concatenate` layer, using the default `axis=-1`.

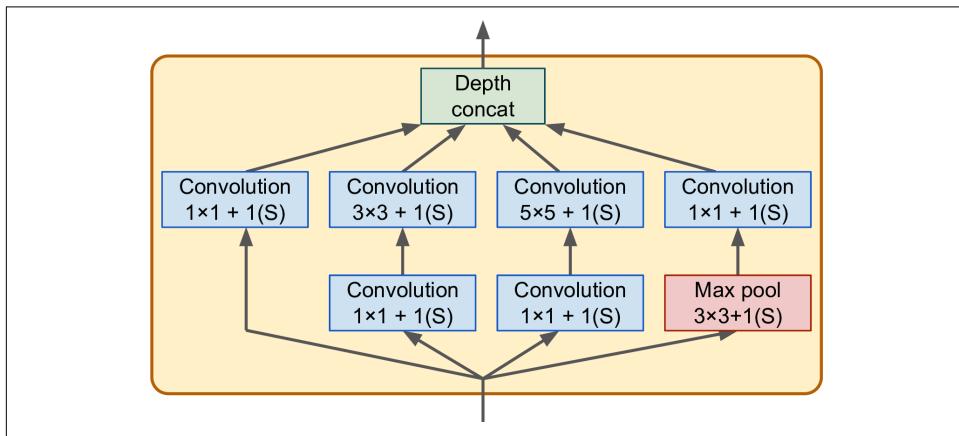


Figure 14-14. Inception module

You may wonder why inception modules have convolutional layers with 1×1 kernels. Surely these layers cannot capture any features because they look at only one pixel at a time, right? In fact, these layers serve three purposes:

- Although they cannot capture spatial patterns, they can capture patterns along the depth dimension (i.e., across channels).
- They are configured to output fewer feature maps than their inputs, so they serve as *bottleneck layers*, meaning they reduce dimensionality. This cuts the

¹⁴ In the 2010 movie *Inception*, the characters keep going deeper and deeper into multiple layers of dreams; hence the name of these modules.

computational cost and the number of parameters, speeding up training and improving generalization.

- Each pair of convolutional layers ($[1 \times 1, 3 \times 3]$ and $[1 \times 1, 5 \times 5]$) acts like a single powerful convolutional layer, capable of capturing more complex patterns. Indeed, a convolutional layer is equivalent to sweeping a dense layer across the image (at each location, it only looks at a small receptive field), and these pairs of convolutional layers are equivalent to sweeping two-layer neural networks across the image.

In short, you can think of the whole inception module as a convolutional layer on steroids, able to output feature maps that capture complex patterns at various scales.

Now let's look at the architecture of the GoogLeNet CNN (see [Figure 14-15](#)). The number of feature maps output by each convolutional layer and each pooling layer is shown before the kernel size. The architecture is so deep that it has to be represented in three columns, but GoogLeNet is actually one tall stack, including nine inception modules (the boxes with the spinning tops). The six numbers in the inception modules represent the number of feature maps output by each convolutional layer in the module (in the same order as in [Figure 14-14](#)). Note that all the convolutional layers use the ReLU activation function.

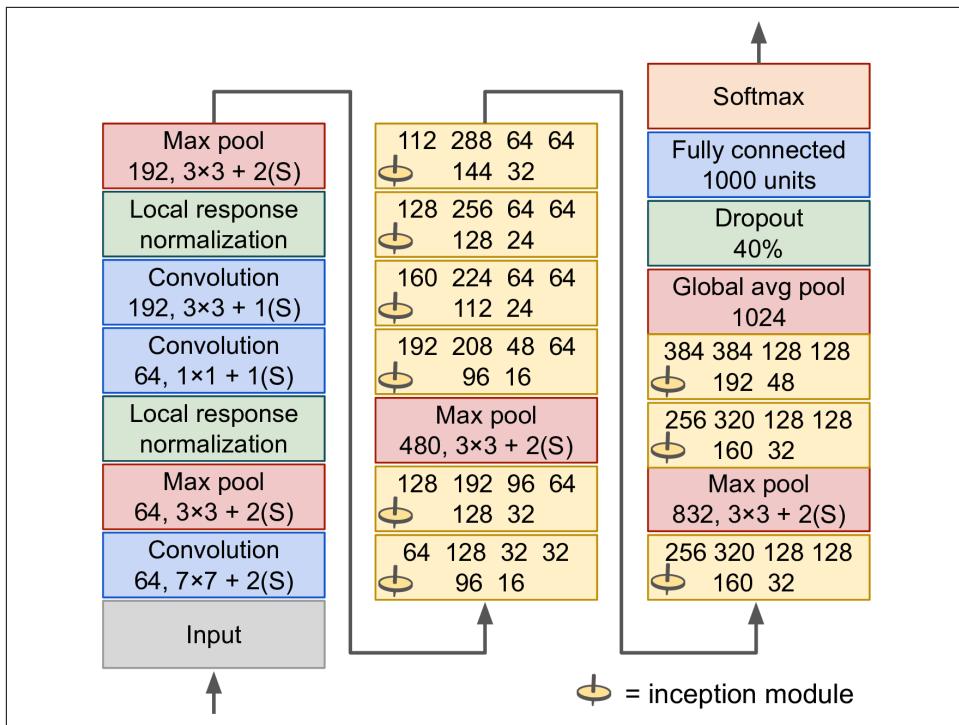


Figure 14-15. GoogLeNet architecture

Let's go through this network:

- The first two layers divide the image's height and width by 4 (so its area is divided by 16), to reduce the computational load. The first layer uses a large kernel size, 7×7 , so that much of the information is preserved.
- Then the local response normalization layer ensures that the previous layers learn a wide variety of features (as discussed earlier).
- Two convolutional layers follow, where the first acts like a bottleneck layer. As explained earlier, you can think of this pair as a single smarter convolutional layer.
- Again, a local response normalization layer ensures that the previous layers capture a wide variety of patterns.
- Next, a max pooling layer reduces the image height and width by 2, again to speed up computations.
- Then comes the CNN's *backbone*: a tall stack of nine inception modules, interleaved with a couple max pooling layers to reduce dimensionality and speed up the net.

- Next, the global average pooling layer outputs the mean of each feature map: this drops any remaining spatial information, which is fine because there was not much spatial information left at that point. Indeed, GoogLeNet input images are typically expected to be 224×224 pixels, so after 5 max pooling layers, each dividing the height and width by 2, the feature maps are down to 7×7 . Moreover, it is a classification task, not localization, so it does not matter where the object is. Thanks to the dimensionality reduction brought by this layer, there is no need to have several fully connected layers at the top of the CNN (like in AlexNet), and this considerably reduces the number of parameters in the network and limits the risk of overfitting.
- The last layers are self-explanatory: dropout for regularization, then a fully connected layer with 1,000 units (since there are 1,000 classes) and a softmax activation function to output estimated class probabilities.

The original GoogLeNet architecture also included two auxiliary classifiers plugged on top of the third and sixth inception modules. They were both composed of one average pooling layer, one convolutional layer, two fully connected layers, and a softmax activation layer. During training, their loss (scaled down by 70%) was added to the overall loss. The goal was to fight the vanishing gradients problem and regularize the network. However, it was later shown that their effect was relatively minor.

Several variants of the GoogLeNet architecture were later proposed by Google researchers, including Inception-v3 and Inception-v4, using slightly different inception modules to reach even better performance.

VGGNet

The runner-up in the ILSVRC 2014 challenge was **VGGNet**,¹⁵ developed by Karen Simonyan and Andrew Zisserman from the Visual Geometry Group (VGG) research lab at Oxford University. It had a very simple and classical architecture, with 2 or 3 convolutional layers and a pooling layer, then again 2 or 3 convolutional layers and a pooling layer, and so on (reaching a total of just 16 or 19 convolutional layers, depending on the VGG variant), plus a final dense network with 2 hidden layers and the output layer. It used only small 3×3 filters, but it had many of them.

ResNet

Kaiming He et al. won the ILSVRC 2015 challenge using a *Residual Network* (or *ResNet*),¹⁶ that delivered an astounding top-five error rate under 3.6%. The winning

¹⁵ Karen Simonyan and Andrew Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” arXiv preprint arXiv:1409.1556 (2014).

variant used an extremely deep CNN composed of 152 layers (other variants had 34, 50, and 101 layers). It confirmed the general trend: computer vision models were getting deeper and deeper, with fewer and fewer parameters. The key to being able to train such a deep network is to use *skip connections* (also called *shortcut connections*): the signal feeding into a layer is also added to the output of a layer located a bit higher up the stack. Let's see why this is useful.

When training a neural network, the goal is to make it model a target function $h(\mathbf{x})$. If you add the input \mathbf{x} to the output of the network (i.e., you add a skip connection), then the network will be forced to model $f(\mathbf{x}) = h(\mathbf{x}) - \mathbf{x}$ rather than $h(\mathbf{x})$. This is called *residual learning* (see [Figure 14-16](#)).

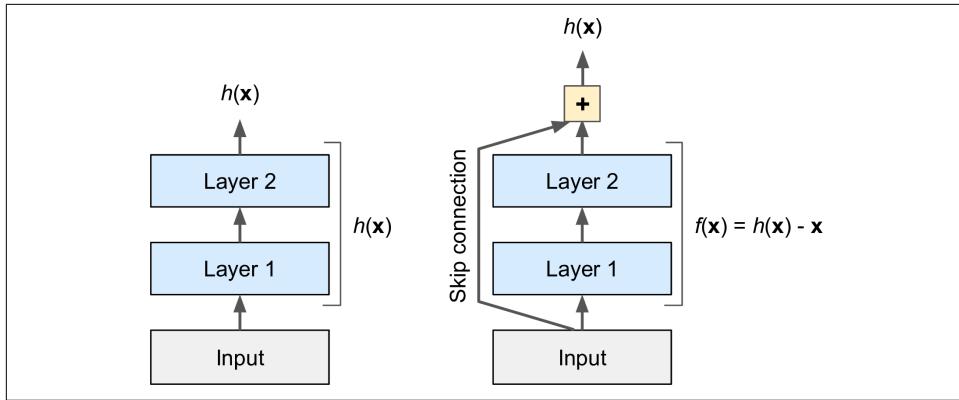


Figure 14-16. Residual learning

When you initialize a regular neural network, its weights are close to zero, so the network just outputs values close to zero. If you add a skip connection, the resulting network just outputs a copy of its inputs; in other words, it initially models the identity function. If the target function is fairly close to the identity function (which is often the case), this will speed up training considerably.

Moreover, if you add many skip connections, the network can start making progress even if several layers have not started learning yet (see [Figure 14-17](#)). Thanks to skip connections, the signal can easily make its way across the whole network. The deep residual network can be seen as a stack of *residual units* (RUs), where each residual unit is a small neural network with a skip connection.

¹⁶ Kaiming He et al., “Deep Residual Learning for Image Recognition,” arXiv preprint arXiv:1512:03385 (2015).

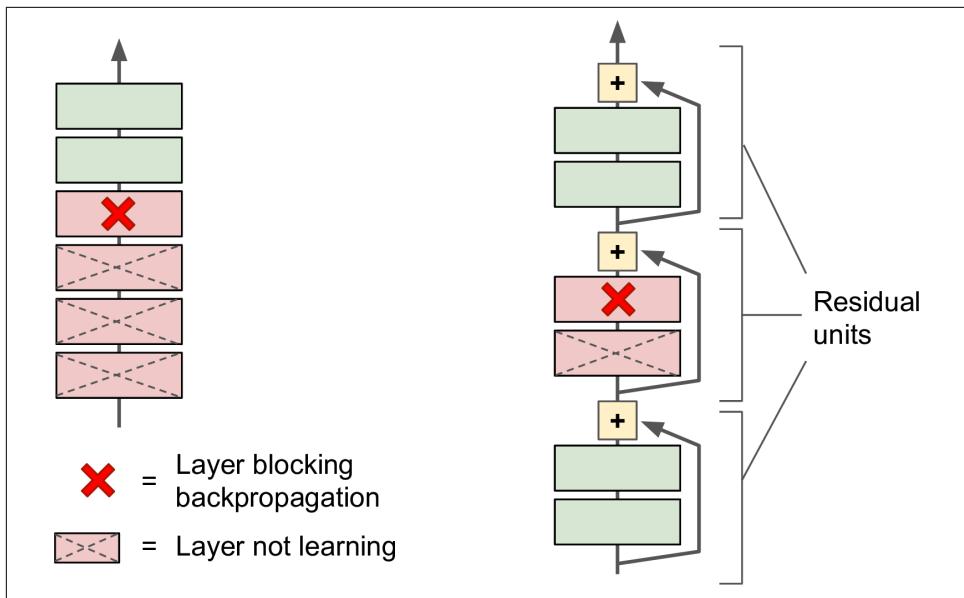


Figure 14-17. Regular deep neural network (left) and deep residual network (right)

Now let's look at ResNet's architecture (see [Figure 14-18](#)). It is surprisingly simple. It starts and ends exactly like GoogLeNet (except without a dropout layer), and in between is just a very deep stack of residual units. Each residual unit is composed of two convolutional layers (and no pooling layer!), with Batch Normalization (BN) and ReLU activation, using 3×3 kernels and preserving spatial dimensions (stride 1, "same" padding).

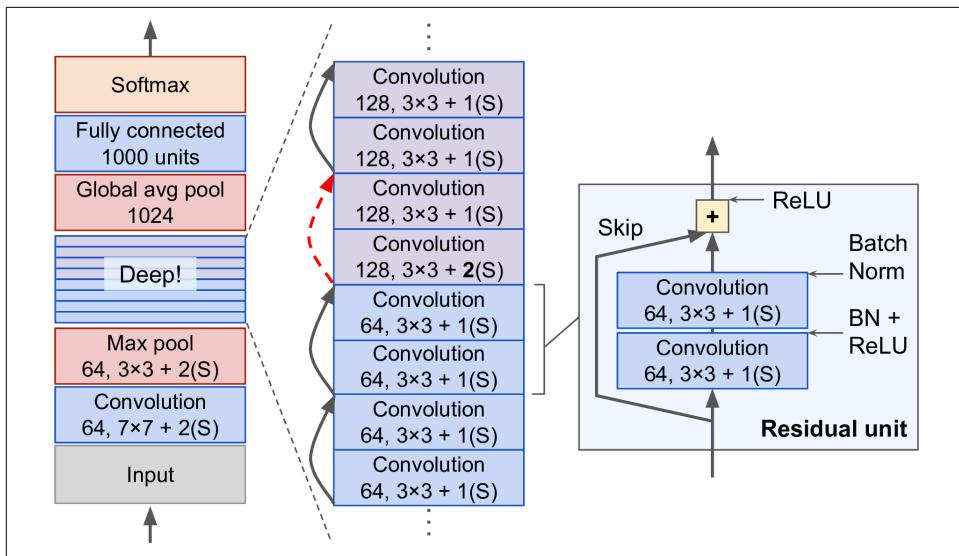


Figure 14-18. ResNet architecture

Note that the number of feature maps is doubled every few residual units, at the same time as their height and width are halved (using a convolutional layer with stride 2). When this happens, the inputs cannot be added directly to the outputs of the residual unit because they don't have the same shape (for example, this problem affects the skip connection represented by the dashed arrow in Figure 14-18). To solve this problem, the inputs are passed through a 1×1 convolutional layer with stride 2 and the right number of output feature maps (see Figure 14-19).

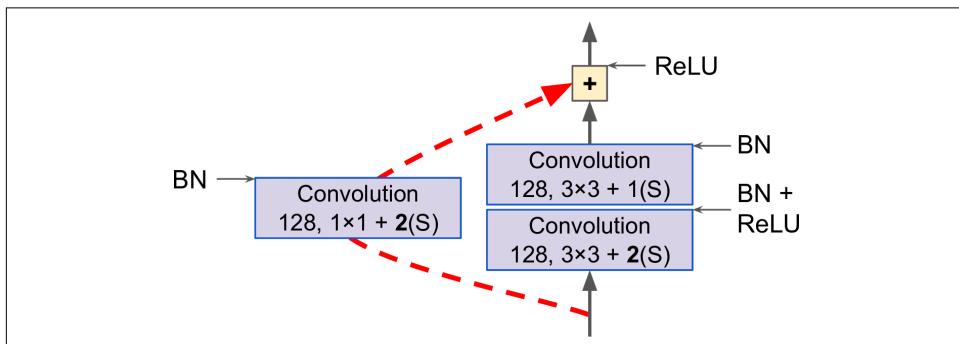


Figure 14-19. Skip connection when changing feature map size and depth

ResNet-34 is the ResNet with 34 layers (only counting the convolutional layers and the fully connected layer)¹⁷ containing 3 residual units that output 64 feature maps, 4 RUs with 128 maps, 6 RUs with 256 maps, and 3 RUs with 512 maps. We will implement this architecture later in this chapter.

ResNets deeper than that, such as ResNet-152, use slightly different residual units. Instead of two 3×3 convolutional layers with, say, 256 feature maps, they use three convolutional layers: first a 1×1 convolutional layer with just 64 feature maps (4 times less), which acts as a bottleneck layer (as discussed already), then a 3×3 layer with 64 feature maps, and finally another 1×1 convolutional layer with 256 feature maps (4 times 64) that restores the original depth. ResNet-152 contains 3 such RUs that output 256 maps, then 8 RUs with 512 maps, a whopping 36 RUs with 1,024 maps, and finally 3 RUs with 2,048 maps.



Google's [Inception-v4](#)¹⁸ architecture merged the ideas of GoogLeNet and ResNet and achieved a top-five error rate of close to 3% on ImageNet classification.

Xception

Another variant of the GoogLeNet architecture is worth noting: [Xception](#)¹⁹ (which stands for *Extreme Inception*) was proposed in 2016 by François Chollet (the author of Keras), and it significantly outperformed Inception-v3 on a huge vision task (350 million images and 17,000 classes). Just like Inception-v4, it merges the ideas of GoogLeNet and ResNet, but it replaces the inception modules with a special type of layer called a *depthwise separable convolution layer* (or *separable convolution layer* for short²⁰). These layers had been used before in some CNN architectures, but they were not as central as in the Xception architecture. While a regular convolutional layer uses filters that try to simultaneously capture spatial patterns (e.g., an oval) and cross-channel patterns (e.g., mouth + nose + eyes = face), a separable convolutional layer makes the strong assumption that spatial patterns and cross-channel patterns can be modeled separately (see [Figure 14-20](#)). Thus, it is composed of two parts: the first part applies a single spatial filter to each input feature map, then the second part

¹⁷ It is a common practice when describing a neural network to count only layers with parameters.

¹⁸ Christian Szegedy et al., "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning," arXiv preprint arXiv:1602.07261 (2016).

¹⁹ François Chollet, "Xception: Deep Learning with Depthwise Separable Convolutions," arXiv preprint arXiv:1610.02357 (2016).

²⁰ This name can sometimes be ambiguous, since spatially separable convolutions are often called "separable convolutions" as well.

looks exclusively for cross-channel patterns—it is just a regular convolutional layer with 1×1 filters.

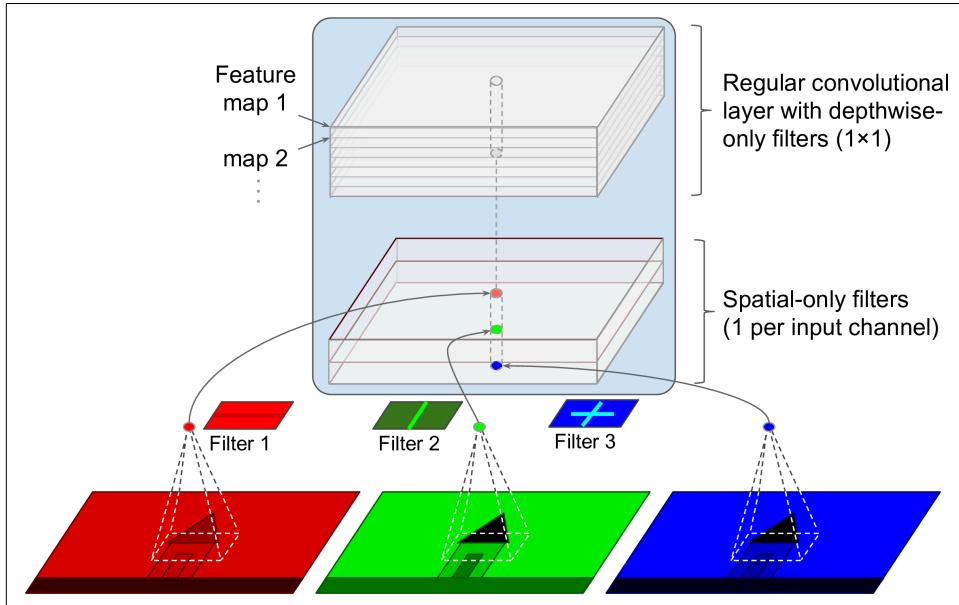


Figure 14-20. Depthwise separable convolutional layer

Since separable convolutional layers only have one spatial filter per input channel, you should avoid using them after layers that have too few channels, such as the input layer (granted, that's what Figure 14-20 represents, but it is just for illustration purposes). For this reason, the Xception architecture starts with 2 regular convolutional layers, but then the rest of the architecture uses only separable convolutions (34 in all), plus a few max pooling layers and the usual final layers (a global average pooling layer and a dense output layer).

You might wonder why Xception is considered a variant of GoogLeNet, since it contains no inception module at all. Well, as we discussed earlier, an inception module contains convolutional layers with 1×1 filters: these look exclusively for cross-channel patterns. However, the convolutional layers that sit on top of them are regular convolutional layers that look both for spatial and cross-channel patterns. So you can think of an inception module as an intermediate between a regular convolutional layer (which considers spatial patterns and cross-channel patterns jointly) and a separable convolutional layer (which considers them separately). In practice, it seems that separable convolutional layers often perform better.



Separable convolutional layers use fewer parameters, less memory, and fewer computations than regular convolutional layers, and they often perform better. You should consider using them by default, except after layers with few channels (such as the input channel). In Keras, just use `SepableConv2D` instead of `Conv2D`: it's a drop-in replacement. Keras also offers a `DepthwiseConv2D` layer which implements the first part of a depthwise separable convolutional layer (i.e., applying one spatial filter per input feature map).

SENet

The winning architecture in the ILSVRC 2017 challenge was the [Squeeze-and-Excitation Network \(SENet\)](#).²¹ This architecture extends existing architectures such as inception networks and ResNets, and boosts their performance. This allowed SENet to win the competition with an astonishing 2.25% top-five error rate! The extended versions of inception networks and ResNets are called *SE-Inception* and *SE-ResNet*, respectively. The boost comes from the fact that a SENet adds a small neural network, called an *SE block*, to every inception module or residual unit in the original architecture, as shown in Figure 14-21.

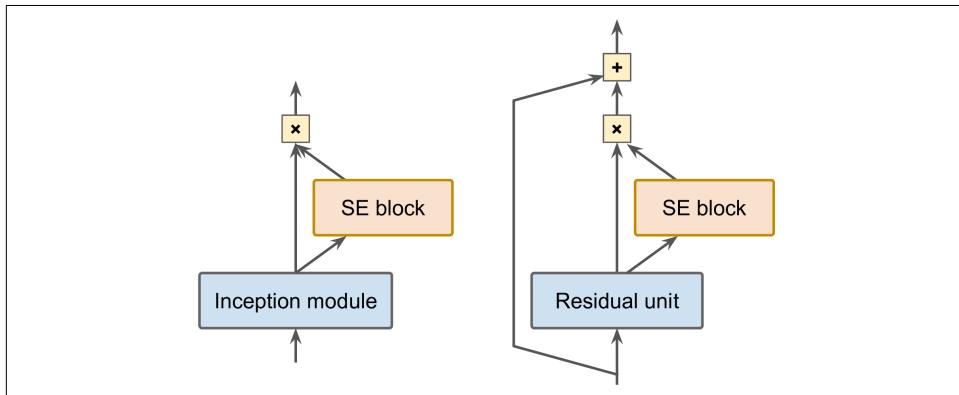


Figure 14-21. SE-Inception module (left) and SE-ResNet unit (right)

An SE block analyzes the output of the unit it is attached to, focusing exclusively on the depth dimension (it does not look for any spatial pattern), and it learns which features are usually most active together. It then uses this information to recalibrate the feature maps, as shown in Figure 14-22. For example, an SE block may learn that mouths, noses, and eyes usually appear together in pictures: if you see a mouth and

²¹ Jie Hu et al., “Squeeze-and-Excitation Networks,” *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2018): 7132–7141.

a nose, you should expect to see eyes as well. So if the block sees a strong activation in the mouth and nose feature maps, but only mild activation in the eye feature map, it will boost the eye feature map (more accurately, it will reduce irrelevant feature maps). If the eyes were somewhat confused with something else, this feature map recalibration will help resolve the ambiguity.

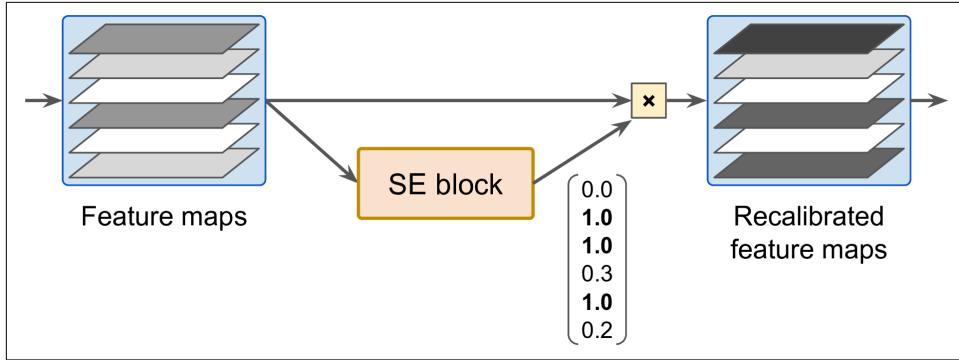


Figure 14-22. An SE block performs feature map recalibration

An SE block is composed of just three layers: a global average pooling layer, a hidden dense layer using the ReLU activation function, and a dense output layer using the sigmoid activation function (see Figure 14-23).

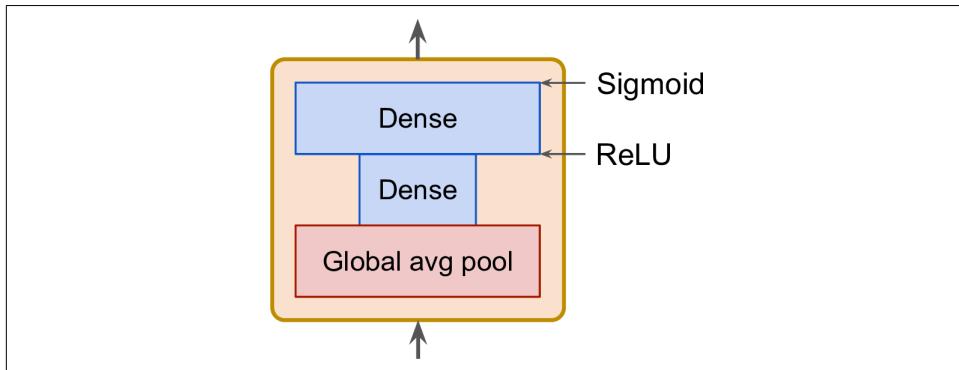


Figure 14-23. SE block architecture

As earlier, the global average pooling layer computes the mean activation for each feature map: for example, if its input contains 256 feature maps, it will output 256 numbers representing the overall level of response for each filter. The next layer is where the “squeeze” happens: this layer has significantly fewer than 256 neurons—typically 16 times fewer than the number of feature maps (e.g., 16 neurons)—so the 256 numbers get compressed into a small vector (e.g., 16 dimensions). This is a low-dimensional vector representation (i.e., an embedding) of the distribution

of feature responses. This bottleneck step forces the SE block to learn a general representation of the feature combinations (we will see this principle in action again when we discuss autoencoders in [Chapter 17](#)). Finally, the output layer takes the embedding and outputs a recalibration vector containing one number per feature map (e.g., 256), each between 0 and 1. The feature maps are then multiplied by this recalibration vector, so irrelevant features (with a low recalibration score) get scaled down while relevant features (with a recalibration score close to 1) are left alone.

Other Noteworthy Architectures

There are many other CNN architectures to explore. Here's a brief overview of some of the most noteworthy:

*ResNeXt*²²

ResNeXt improves the residual units in ResNet. Whereas the residual units in the best ResNet models just contain 3 convolution layers each, the ResNeXt residual units are composed of many parallel stacks (e.g., 32 stacks), with 3 convolutional layers each. However, the first two layers in each stack only use a few filters (e.g., just 4), so the overall number of parameters remains the same as in ResNet. Then the outputs of all stacks are added together, and the result is passed to the next residual unit (along with the skip connection).

*DenseNet*²³

A DenseNet is composed of several dense blocks, each made up of a few densely connected convolutional layers. What does “densely connected” mean? Well, the output of each layer is fed as input to every layer after it within the same block. For example, layer 4 in a block takes as input the depthwise concatenation of the outputs of layers 1, 2 and 3 in that block. Dense blocks are separated by a few transition layers. This architecture achieved excellent accuracy while using comparatively few parameters.

*MobileNet*²⁴

MobileNets are streamlined models designed to be lightweight and fast, making them popular in mobile and web applications. They are based on depthwise separable convolutional layers, like Xception. The authors proposed several variants, trading a bit of accuracy for faster and smaller models.

²² Saining Xie et al., “Aggregated Residual Transformations for Deep Neural Networks,” arXiv preprint arXiv:1611.05431 (2016).

²³ Gao Huang et al., “Densely Connected Convolutional Networks,” arXiv preprint arXiv:1608.06993 (2016).

²⁴ Andrew G. Howard et al., “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications,” arXiv preprint arxiv:1704.04861 (2017).

CSPNet²⁵

Cross Stage Partial Network (CSPNet) is a similar to DenseNet, but part of each dense block's input is concatenated directly to that block's output, without going through the block.

EfficientNet²⁶

EfficientNet is arguably the most important model in this list. The authors proposed a method to scale any CNN efficiently, by jointly increasing the depth (number of layers), width (number of filters per layer), and resolution (the size of the input image), in a principled way (explained below). This is called *compound scaling*. They used neural architecture search to find a good architecture for a scaled down version of ImageNet (with smaller and fewer images), then they used compound scaling to create larger and larger versions of this architecture. When EfficientNet models came out, they vastly outperformed all existing models, across all compute budgets, and they remain among the best models out there today.

Understanding EfficientNet's compound scaling method is helpful to gain a deeper understanding of CNNs, especially if you ever need to scale a CNN architecture. It is based on a logarithmic measure of the compute budget noted ϕ : if your compute budget doubles, then ϕ increases by 1. In other words, the number of floating-point operations available for training is proportional to 2^ϕ . Your CNN architecture's depth, width, and resolution should scale as α^ϕ , β^ϕ , and γ^ϕ , respectively. The factors α , β , and γ must be greater than 1, and $\alpha + \beta^2 + \gamma^2$ should be close to 2. The optimal values for these factors depend on the CNN's architecture. To find the optimal values for the EfficientNet architecture, the authors started with a small baseline model (EfficientNet-B0), they fixed $\phi = 1$, and they simply ran a grid search: they found $\alpha = 1.2$, $\beta = 1.1$, and $\gamma = 1.1$. They then used these factors to create several larger architectures, named EfficientNet-B1 to EfficientNet-B7, for increasing values of ϕ .

Choosing The Right CNN Architecture

OK, with so many CNN architectures, how do you choose which one is best for your project? Well, it depends on what matters most to you: accuracy? Model size (e.g., for deployment to a mobile device)? Inference speed on CPU? On GPU? **Table 14-3** lists the best pretrained models currently available in Keras (we will see how to use them later in this chapter), sorted by model size. You can find the full list at <https://keras.io/api/applications/>. For each model, the table shows the Keras class name to

²⁵ Chien-Yao Wang et al., "CSPNet: A New Backbone that can Enhance Learning Capability of CNN," arXiv preprint arXiv:1911.11929 (2019).

²⁶ Mingxing Tan and Quoc V. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," arXiv preprint arXiv:1905.11946 (2019).

use (in the `tf.keras.applications` package), the model's size in MB, the top-1 and top-5 validation accuracy on the ImageNet dataset, the number of parameters (millions), and the inference time on CPU and GPU in ms, using batches of 32 images on reasonably powerful hardware.²⁷ For each column, the best value is highlighted. As you can see, larger models are generally more accurate, but not always. For example, EfficientNetB2 outperforms InceptionV3 both in size and accuracy. I only kept InceptionV3 in the list because it is almost twice as fast as EfficientNetB2 on a CPU. Similarly, InceptionResNetV2 is fast on a CPU, and ResNet50V2 and ResNet101V2 are blazingly fast on a GPU.

Table 14-3. Pretrained models available in Keras

Class Name	Size (MB)	Top-1 Acc	Top-5 Acc	Params	CPU (ms)	GPU (ms)
MobileNetV2	14	71.3%	90.1%	3.5M	25.9	3.8
MobileNet	16	70.4%	89.5%	4.3M	22.6	3.4
NASNetMobile	23	74.4%	91.9%	5.3M	27.0	6.7
EfficientNetB0	29	77.1%	93.3%	5.3M	46.0	4.9
EfficientNetB1	31	79.1%	94.4%	7.9M	60.2	5.6
EfficientNetB2	36	80.1%	94.9%	9.2M	80.8	6.5
EfficientNetB3	48	81.6%	95.7%	12.3M	140.0	8.8
EfficientNetB4	75	82.9%	96.4%	19.5M	308.3	15.1
InceptionV3	92	77.9%	93.7%	23.9M	42.2	6.9
ResNet50V2	98	76.0%	93.0%	25.6M	45.6	4.4
EfficientNetB5	118	83.6%	96.7%	30.6M	579.2	25.3
EfficientNetB6	166	84.0%	96.8%	43.3M	958.1	40.4
ResNet101V2	171	77.2%	93.8%	44.7M	72.7	5.4
InceptionResNetV2	215	80.3%	95.3%	55.9M	130.2	10.0
EfficientNetB7	256	84.3%	97.0%	66.7M	1578.9	61.6

I hope you enjoyed this deep dive into the main CNN architectures! Now let's see how to implement one of them using Keras.

Implementing a ResNet-34 CNN Using Keras

Most CNN architectures described so far can be implemented pretty naturally using Keras (although generally you would load a pretrained network instead, as we will see). To illustrate the process, let's implement a ResNet-34 from scratch using Keras. First, let's create a `ResidualUnit` layer:

²⁷ A 92-core AMD EPYC CPU with 1BPP, 1.7T of RAM, and an Nvidia Tesla A100 GPU.

```

DefaultConv2D = partial(tf.keras.layers.Conv2D, kernel_size=3, strides=1,
                      padding="same", kernel_initializer="he_normal",
                      use_bias=False)

class ResidualUnit(tf.keras.layers.Layer):
    def __init__(self, filters, strides=1, activation="relu", **kwargs):
        super().__init__(**kwargs)
        self.activation = tf.keras.activations.get(activation)
        self.main_layers = [
            DefaultConv2D(filters, strides=strides),
            tf.keras.layers.BatchNormalization(),
            self.activation,
            DefaultConv2D(filters),
            tf.keras.layers.BatchNormalization()
        ]
        self.skip_layers = []
        if strides > 1:
            self.skip_layers = [
                DefaultConv2D(filters, kernel_size=1, strides=strides),
                tf.keras.layers.BatchNormalization()
            ]

    def call(self, inputs):
        Z = inputs
        for layer in self.main_layers:
            Z = layer(Z)
        skip_Z = inputs
        for layer in self.skip_layers:
            skip_Z = layer(skip_Z)
        return self.activation(Z + skip_Z)

```

As you can see, this code matches [Figure 14-19](#) pretty closely. In the constructor, we create all the layers we will need: the main layers are the ones on the right side of the diagram, and the skip layers are the ones on the left (only needed if the stride is greater than 1). Then in the `call()` method, we make the inputs go through the main layers and the skip layers (if any), then we add both outputs and apply the activation function.

Next, we can build ResNet-34 using a `Sequential` model, since it's really just a long sequence of layers—we can treat each residual unit as a single layer now that we have the `ResidualUnit` class. The code closely matches [Figure 14-18](#):

```

model = tf.keras.Sequential([
    DefaultConv2D(64, kernel_size=7, strides=2, input_shape=[224, 224, 3]),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Activation("relu"),
    tf.keras.layers.MaxPool2D(pool_size=3, strides=2, padding="same"),
])
prev_filters = 64
for filters in [64] * 3 + [128] * 4 + [256] * 6 + [512] * 3:
    strides = 1 if filters == prev_filters else 2

```

```
model.add(ResidualUnit(filters, strides=strides))
prev_filters = filters

model.add(tf.keras.layers.GlobalAvgPool2D())
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(10, activation="softmax"))
```

The only tricky part in this code is the loop that adds the `ResidualUnit` layers to the model: as explained earlier, the first 3 RUs have 64 filters, then the next 4 RUs have 128 filters, and so on. At each iteration, we must set the stride to 1 when the number of filters is the same as in the previous RU, or else we set it to 2, then we add the `ResidualUnit`, and finally we update `prev_filters`.

It is amazing that in about 40 lines of code, we can build the model that won the ILSVRC 2015 challenge! This demonstrates both the elegance of the ResNet model and the expressiveness of the Keras API. Implementing the other CNN architectures is a bit longer, but not much harder. However, Keras comes with several of these architectures built in, so why not use them instead?

Using Pretrained Models from Keras

In general, you won't have to implement standard models like GoogLeNet or ResNet manually, since pretrained networks are readily available with a single line of code in the `tf.keras.applications` package. For example, you can load the ResNet-50 model, pretrained on ImageNet, with the following line of code:

```
model = tf.keras.applications.ResNet50(weights="imagenet")
```

That's all! This will create a ResNet-50 model and download weights pretrained on the ImageNet dataset. To use it, you first need to ensure that the images have the right size. A ResNet-50 model expects 224×224 -pixel images (other models may expect other sizes, such as 299×299), so let's use Keras's `Resizing` layer (introduced in [Chapter 13](#)) to resize two sample images (after cropping them to the target aspect ratio):

```
images = load_sample_images()["images"]
images_resized = tf.keras.layers.Resizing(height=224, width=224,
                                         crop_to_aspect_ratio=True)(images)
```

The pretrained models assume that the images are preprocessed in a specific way. In some cases they may expect the inputs to be scaled from 0 to 1, or from -1 to 1, and so on. Each model provides a `preprocess_input()` function that you can use to preprocess your images. These functions assume that the original pixel values range from 0 to 255, which is the case here.

```
inputs = tf.keras.applications.resnet50.preprocess_input(images_resized)
```

Now we can use the pretrained model to make predictions:

```
>>> Y_proba = model.predict(inputs)
>>> Y_proba.shape
(2, 1000)
```

As usual, the output `Y_proba` is a matrix with one row per image and one column per class (in this case, there are 1,000 classes). If you want to display the top K predictions, including the class name and the estimated probability of each predicted class, use the `decode_predictions()` function. For each image, it returns an array containing the top K predictions, where each prediction is represented as an array containing the class identifier,²⁸ its name, and the corresponding confidence score:

```
top_K = tf.keras.applications.resnet50.decode_predictions(Y_proba, top=3)
for image_index in range(len(images)):
    print(f"Image #{image_index}")
    for class_id, name, y_proba in top_K[image_index]:
        print(f"  {class_id} - {name:12s} {y_proba:.2%}")
```

The output looks like this:

```
Image #0
n03877845 - palace      54.69%
n03781244 - monastery   24.72%
n02825657 - bell_cote   18.55%
Image #1
n04522168 - vase        32.66%
n11939491 - daisy       17.81%
n03530642 - honeycomb   12.06%
```

The correct classes are *palace* and *dahlia*, so the model is correct for the first image but wrong for the second. However, that's because *dahlia* is not part of the 1,000 ImageNet classes. With that in mind, *vase* is a reasonable guess (perhaps the flower is in a vase?), and *daisy* is not a bad choice either, since daliyas and daisies are both from the same Compositae family.

As you can see, it is very easy to create a pretty good image classifier using a pretrained model. As we saw in [Table 14-3](#), many other vision models are available in `tf.keras.applications`, from lightweight and fast models to large and accurate ones.

But what if you want to use an image classifier for classes of images that are not part of ImageNet? In that case, you may still benefit from the pretrained models to perform transfer learning.

²⁸ In the ImageNet dataset, each image is mapped to a word in the [WordNet dataset](#): the class ID is just a WordNet ID.

Pretrained Models for Transfer Learning

If you want to build an image classifier but you do not have enough data to train it from scratch, then it is often a good idea to reuse the lower layers of a pretrained model, as we discussed in [Chapter 11](#). For example, let's train a model to classify pictures of flowers, reusing a pretrained Xception model. First, let's load the dataset using TensorFlow Datasets (introduced in [Chapter 13](#)):

```
import tensorflow_datasets as tfds

dataset, info = tfds.load("tf_flowers", as_supervised=True, with_info=True)
dataset_size = info.splits["train"].num_examples # 3670
class_names = info.features["label"].names # ["dandelion", "daisy", ...]
n_classes = info.features["label"].num_classes # 5
```

Note that you can get information about the dataset by setting `with_info=True`. Here, we get the dataset size and the names of the classes. Unfortunately, there is only a "`train`" dataset, no test set or validation set, so we need to split the training set. Let's call `tfds.load()` again, but this time taking the first 10% of the dataset for testing, the next 15% for validation, and the remaining 75% for training:

```
test_set_raw, valid_set_raw, train_set_raw = tfds.load(
    "tf_flowers",
    split=["train[:10%]", "train[10%:25%]", "train[25%:]"],
    as_supervised=True)
```

All three datasets contain individual images. We need to batch them, but for this we must first ensure they all have the same size, or else batching will fail. We can use a `Resizing` layer for this. We must also call the `tf.keras.applications.xception.preprocess_input()` function to preprocess the images appropriately for the Xception model. Lastly, let's also shuffle the training set and use prefetching:

```
batch_size = 32
preprocess = tf.keras.Sequential([
    tf.keras.layers.Resizing(height=224, width=224, crop_to_aspect_ratio=True),
    tf.keras.layers.Lambda(tf.keras.applications.xception.preprocess_input)
])
train_set = train_set_raw.map(lambda X, y: (preprocess(X), y))
train_set = train_set.shuffle(1000, seed=42).batch(batch_size).prefetch(1)
valid_set = valid_set_raw.map(lambda X, y: (preprocess(X), y)).batch(batch_size)
test_set = test_set_raw.map(lambda X, y: (preprocess(X), y)).batch(batch_size)
```

Now each batch contains 32 images, all of them 224×224 pixels, with pixel values ranging from -1 to 1. Perfect!

Since the dataset is not very large, a bit of data augmentation will certainly help. Let's create a data augmentation model that we will embed in our final model: during training, it will randomly flip the images horizontally, rotate them a little bit, and tweak the contrast.

```
data_augmentation = tf.keras.Sequential([
    tf.keras.layers.RandomFlip(mode="horizontal", seed=42),
    tf.keras.layers.RandomRotation(factor=0.05, seed=42),
    tf.keras.layers.RandomContrast(factor=0.2, seed=42)
])
```



The `tf.keras.preprocessing.image.ImageDataGenerator` class makes it easy to load images from disk and augment them in various ways: you can shift each image, rotate it, rescale it, flip it horizontally or vertically, shear it, or apply any transformation function you want to it. This is very convenient for simple projects. However, a `tf.data` pipeline is not much more complicated, and it's generally faster. Moreover, if you have a GPU and you include the preprocessing or data augmentation layers inside your model, they will benefit from GPU acceleration during training.

Next let's load an Xception model, pretrained on ImageNet. We exclude the top of the network by setting `include_top=False`. This excludes the global average pooling layer and the dense output layer. We then add our own global average pooling layer (feeding it the output of the base model), followed by a dense output layer with one unit per class, using the softmax activation function. Finally, we wrap all this in a Keras Model:

```
base_model = tf.keras.applications.Xception(weights="imagenet",
                                              include_top=False)
avg = tf.keras.layers.GlobalAveragePooling2D()(base_model.output)
output = tf.keras.layers.Dense(n_classes, activation="softmax")(avg)
model = tf.keras.Model(inputs=base_model.input, outputs=output)
```

As explained in [Chapter 11](#), it's usually a good idea to freeze the weights of the pretrained layers, at least at the beginning of training:

```
for layer in base_model.layers:
    layer.trainable = False
```



Since our model uses the base model's layers directly, rather than the `base_model` object itself, setting `base_model.trainable=False` would have no effect.

Finally, we can compile the model and start training:

```
optimizer = tf.keras.optimizers.SGD(learning_rate=0.1, momentum=0.9)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])
history = model.fit(train_set, validation_data=valid_set, epochs=3)
```



If you are running in Colab, make sure the runtime is using a GPU: open Runtime → “Change runtime type,” select “GPU” in the “Hardware accelerator” drop-down menu, then click Save. It’s possible to train the model without a GPU, but it will be terribly slow (minutes per epoch, as opposed to seconds).

After training the model for a few epochs, its validation accuracy should reach a bit over 80%, and then stop improving. This means that the top layers are now pretty well trained, and we are ready to unfreeze some of the base model’s top layers, then continue training. For example, let’s unfreeze layers 56 and above (that’s the start of residual unit 7, out of 14, as you can see if you list the layer names):

```
for layer in base_model.layers[56:]:
    layer.trainable = True
```

Don’t forget to compile the model whenever you freeze or unfreeze layers. Also make sure to use a much lower learning rate to avoid damaging the pretrained weights:

```
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01, momentum=0.9)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])
history = model.fit(train_set, validation_data=valid_set, epochs=10)
```

This model should reach around 92% accuracy on the test set, in just a few minutes of training (with a GPU). If you tune the hyperparameters, lower the learning rate and train for quite a bit longer, you should be able to reach 95% to 97%. With that, you can start training amazing image classifiers on your own images and classes! But there’s more to computer vision than just classification. For example, what if you also want to know *where* the flower is in the picture? Let’s look at this now.

Classification and Localization

Localizing an object in a picture can be expressed as a regression task, as discussed in [Chapter 10](#): to predict a bounding box around the object, a common approach is to predict the horizontal and vertical coordinates of the object’s center, as well as its height and width. This means we have four numbers to predict. It does not require much change to the model; we just need to add a second dense output layer with four units (typically on top of the global average pooling layer), and it can be trained using the MSE loss:

```
base_model = tf.keras.applications.Xception(weights="imagenet",
                                             include_top=False)
avg = tf.keras.layers.GlobalAveragePooling2D()(base_model.output)
class_output = tf.keras.layers.Dense(n_classes, activation="softmax")(avg)
loc_output = tf.keras.layers.Dense(4)(avg)
model = tf.keras.Model(inputs=base_model.input,
                       outputs=[class_output, loc_output])
model.compile(loss=["sparse_categorical_crossentropy", "mse"],
```

```
loss_weights=[0.8, 0.2], # depends on what you care most about  
optimizer=optimizer, metrics=["accuracy"])
```

But now we have a problem: the flowers dataset does not have bounding boxes around the flowers. So, we need to add them ourselves. This is often one of the hardest and most costly parts of a Machine Learning project: getting the labels. It's a good idea to spend time looking for the right tools. To annotate images with bounding boxes, you may want to use an open source image labeling tool like VGG Image Annotator, LabelImg, OpenLabeler, or ImgLab, or perhaps a commercial tool like LabelBox or Supervisely. You may also want to consider crowdsourcing platforms such as Amazon Mechanical Turk if you have a very large number of images to annotate. However, it is quite a lot of work to set up a crowdsourcing platform, prepare the form to be sent to the workers, supervise them, and ensure that the quality of the bounding boxes they produce is good, so make sure it is worth the effort. Adriana Kovashka et al. wrote a very practical [paper](#)²⁹ about crowdsourcing in computer vision. I recommend you check it out, even if you do not plan to use crowdsourcing. If there are just a few hundred or even a couple thousand images to label, and you don't plan to do this frequently, it may be preferable to do it yourself: with the right tools, it will only take a few days, and you'll also gain a better understanding of your dataset and task.

Now let's suppose you've obtained the bounding boxes for every image in the flowers dataset (for now we will assume there is a single bounding box per image). You then need to create a dataset whose items will be batches of preprocessed images along with their class labels and their bounding boxes. Each item should be a tuple of the form `(images, (class_labels, bounding_boxes))`. Then you are ready to train your model!



The bounding boxes should be normalized so that the horizontal and vertical coordinates, as well as the height and width, all range from 0 to 1. Also, it is common to predict the square root of the height and width rather than the height and width directly: this way, a 10-pixel error for a large bounding box will not be penalized as much as a 10-pixel error for a small bounding box.

The MSE often works fairly well as a cost function to train the model, but it is not a great metric to evaluate how well the model can predict bounding boxes. The most common metric for this is the *Intersection over Union* (IoU): the area of overlap between the predicted bounding box and the target bounding box, divided

²⁹ Adriana Kovashka et al., "Crowdsourcing in Computer Vision," *Foundations and Trends in Computer Graphics and Vision* 10, no. 3 (2014): 177–243.

by the area of their union (see [Figure 14-24](#)). In Keras, it is implemented by the `tf.keras.metrics.MeanIoU` class.

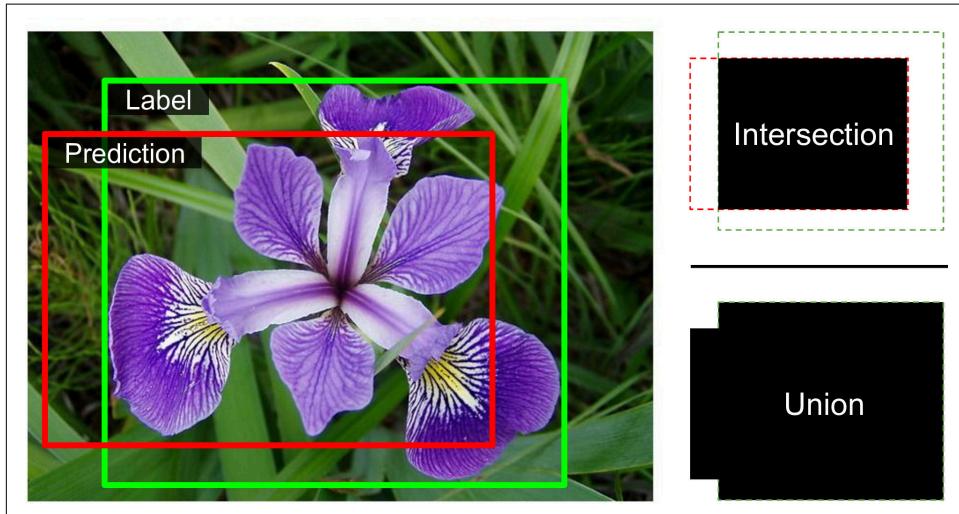


Figure 14-24. Intersection over Union (IoU) metric for bounding boxes

Classifying and localizing a single object is nice, but what if the images contain multiple objects (as is often the case in the flowers dataset)?

Object Detection

The task of classifying and localizing multiple objects in an image is called *object detection*. Until a few years ago, a common approach was to take a CNN that was trained to classify and locate a single object roughly centered in the image, then slide this CNN across the image, and make predictions at each step. The CNN was generally trained to predict not only class probabilities and a bounding box, but also an *objectness score*: this is the estimated probability that the image does indeed contain an object centered near the middle of the image. This is a binary classification output: it can be produced by a dense output layer with a single unit, using the sigmoid activation function and trained using the binary cross-entropy loss.



Instead of an objectness score, a “no-object” class was sometimes added, but in general this did not work as well: the questions “is an object present?” and “what type of object is it?” are best answered separately.

This sliding-CNN approach is illustrated in [Figure 14-25](#). In this example, the image was chopped into a 5×7 grid, and we see a CNN—the thick black rectangle—sliding across all 3×3 regions, and making predictions at each step.

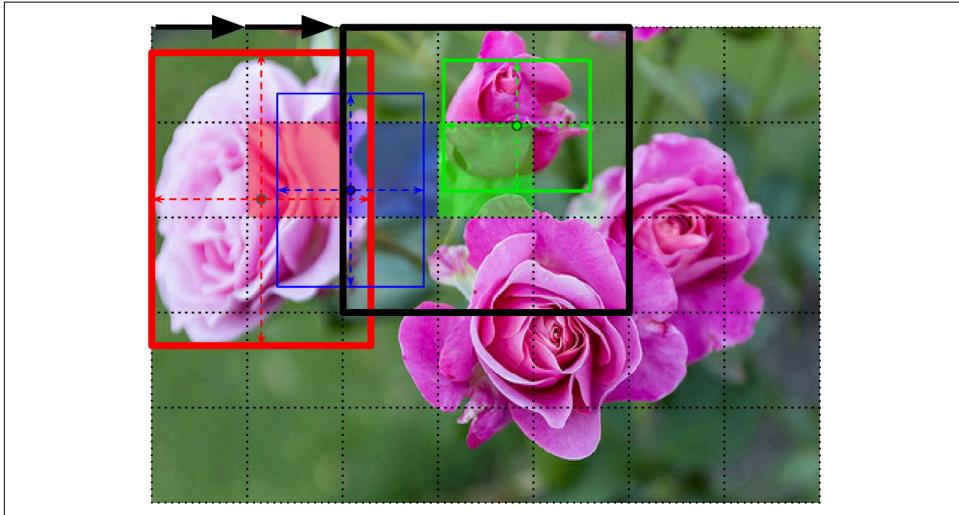


Figure 14-25. Detecting multiple objects by sliding a CNN across the image

In this figure, the CNN has already made predictions for three of these 3×3 regions:

- When looking at the top-left 3×3 region (centered on the red-shaded grid cell located in the second row and second column), it detected the leftmost rose. Notice that the predicted bounding box exceeds the boundary of this 3×3 region. That's absolutely fine: even though the CNN could not see the bottom part of the rose, it was able to make a reasonable guess as to where it might be. It also predicted class probabilities, giving a high probability to the “rose” class. Lastly, it predicted a fairly high objectness score, since the center of the bounding box lies within the central grid cell (in this figure, the objectness score is represented by the thickness of the bounding box).
- Then when looking at the next 3×3 region, one grid cell to the right (centered on the shaded blue square), it did not detect any flower centered in that region, so it predicted a very low objectness score, and therefore the predicted bounding box and class probabilities can safely be ignored. You can see that the predicted bounding box was no good anyway.
- Then, once again, it looked at the next 3×3 region, one grid cell to the right (centered on the shaded green cell), and this time it detected the rose at the top, although not perfectly since this rose is not well centered within this region, so the predicted objectness score was not very high.

You can imagine how sliding the CNN across the whole image would give you a total of 15 predicted bounding boxes, organized in a 3×5 grid, with each bounding box accompanied by its estimated class probabilities and objectness score. Since objects can have varying sizes, you may then want to slide the CNN again across larger 4×4 regions as well, to get even more bounding boxes.

This technique is fairly straightforward, but as you can see it will often detect the same object multiple times, at slightly different positions. Some post-processing is needed to get rid of all the unnecessary bounding boxes. A common approach for this is called *non-max suppression*. Here's how it works:

1. First, get rid of all the bounding boxes for which the objectness score is below some threshold: since the CNN believes there's no object at that location, the bounding box is useless.
2. Find the remaining bounding box with the highest objectness score, and get rid of all the other remaining bounding boxes that overlap a lot with it (e.g., with an IoU greater than 60%). For example, in [Figure 14-25](#), the bounding box with the max objectness score is the thick bounding box over the leftmost rose. The other bounding box that touches this same rose overlaps a lot with the max bounding box, so we will get rid of it (although in this example it would already have been removed in the previous step).
3. Repeat step two until there are no more bounding boxes to get rid of.

This simple approach to object detection works pretty well, but it requires running the CNN many times (15 times in this example), so it is quite slow. Fortunately, there is a much faster way to slide a CNN across an image: using a *fully convolutional network* (FCN).

Fully Convolutional Networks

The idea of FCNs was first introduced in a [2015 paper](#)³⁰ by Jonathan Long et al., for semantic segmentation (the task of classifying every pixel in an image according to the class of the object it belongs to). The authors pointed out that you could replace the dense layers at the top of a CNN, with convolutional layers. To understand this, let's look at an example: suppose a dense layer with 200 neurons sits on top of a convolutional layer that outputs 100 feature maps, each of size 7×7 (this is the feature map size, not the kernel size). Each neuron will compute a weighted sum of all $100 \times 7 \times 7$ activations from the convolutional layer (plus a bias term). Now let's see what happens if we replace the dense layer with a convolutional layer using

³⁰ Jonathan Long et al., "Fully Convolutional Networks for Semantic Segmentation," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2015): 3431–3440.

200 filters, each of size 7×7 , and with “valid” padding. This layer will output 200 feature maps, each 1×1 (since the kernel is exactly the size of the input feature maps and we are using “valid” padding). In other words, it will output 200 numbers, just like the dense layer did; and if you look closely at the computations performed by a convolutional layer, you will notice that these numbers will be precisely the same as those the dense layer produced. The only difference is that the dense layer’s output was a tensor of shape [*batch size*, 200], while the convolutional layer will output a tensor of shape [*batch size*, 1, 1, 200].



To convert a dense layer to a convolutional layer, the number of filters in the convolutional layer must be equal to the number of units in the dense layer, the filter size must be equal to the size of the input feature maps, and you must use “valid” padding. The stride may be set to 1 or more, as we will see shortly.

Why is this important? Well, while a dense layer expects a specific input size (since it has one weight per input feature), a convolutional layer will happily process images of any size³¹ (however, it does expect its inputs to have a specific number of channels, since each kernel contains a different set of weights for each input channel). Since an FCN contains only convolutional layers (and pooling layers, which have the same property), it can be trained and executed on images of any size!

For example, suppose we’d already trained a CNN for flower classification and localization. It was trained on 224×224 images, and it outputs 10 numbers: outputs 0 to 4 are sent through the softmax activation function, and this gives the class probabilities (one per class); output 5 is sent through the sigmoid activation function, and this gives the objectness score; outputs 6 and 7 represent the bounding box’s center coordinates, and they also go through a sigmoid activation function to ensure they range from 0 to 1; lastly, outputs 8 and 9 represent the bounding box’s height and width, and they do not go through any activation function to allow the bounding boxes to extend beyond the borders of the image. We can now convert the CNN’s dense layers to convolutional layers. In fact, we don’t even need to retrain it; we can just copy the weights from the dense layers to the convolutional layers! Alternatively, we could have converted the CNN into an FCN before training.

Now suppose the last convolutional layer before the output layer (also called the bottleneck layer) outputs 7×7 feature maps when the network is fed a 224×224 image (see the left side of [Figure 14-26](#)). If we feed the FCN a 448×448 image (see the right side of [Figure 14-26](#)), the bottleneck layer will now output 14×14 feature maps.³² Since the dense output layer was replaced by a convolutional layer

³¹ There is one small exception: a convolutional layer using “valid” padding will complain if the input size is smaller than the kernel size.

using 10 filters of size 7×7 , with “valid” padding and stride 1, the output will be composed of 10 feature maps, each of size 8×8 (since $14 - 7 + 1 = 8$). In other words, the FCN will process the whole image only once, and it will output an 8×8 grid where each cell contains 10 numbers (5 class probabilities, 1 objectness score, and 4 bounding box coordinates). It’s exactly like taking the original CNN and sliding it across the image using 8 steps per row and 8 steps per column. To visualize this, imagine chopping the original image into a 14×14 grid, then sliding a 7×7 window across this grid; there will be $8 \times 8 = 64$ possible locations for the window, hence 8 × 8 predictions. However, the FCN approach is *much* more efficient, since the network only looks at the image once. In fact, *You Only Look Once* (YOLO) is the name of a very popular object detection architecture, which we’ll look at next.

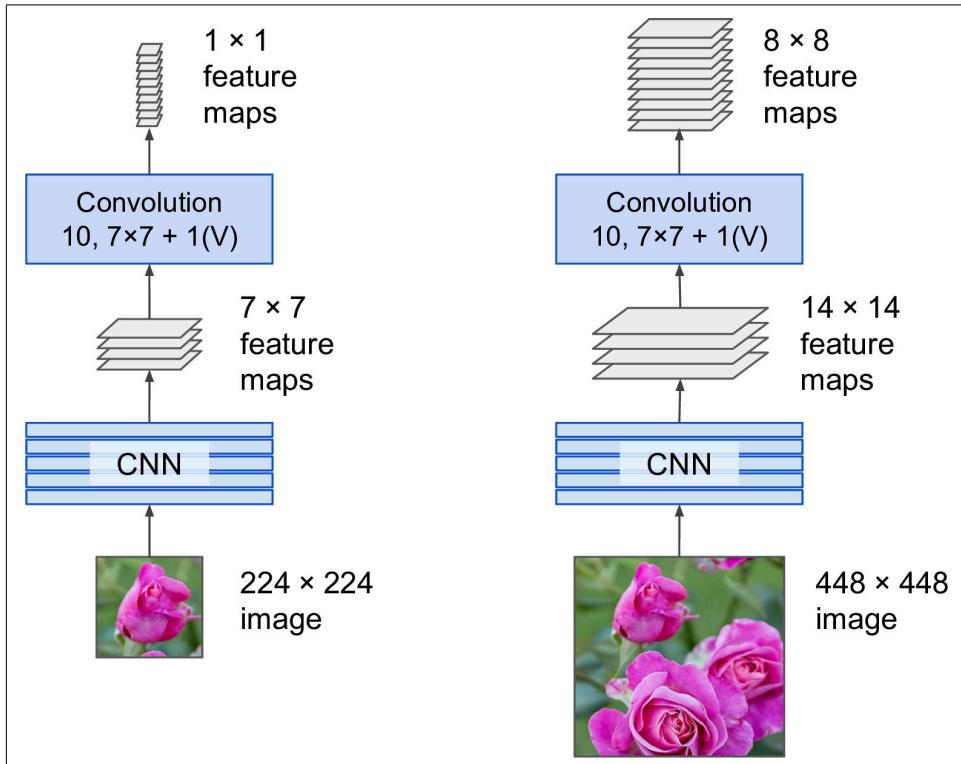


Figure 14-26. The same fully convolutional network processing a small image (left) and a large one (right)

³² This assumes we used only “same” padding in the network: indeed, “valid” padding would reduce the size of the feature maps. Moreover, 448 can be neatly divided by 2 several times until we reach 7, without any rounding error. If any layer uses a different stride than 1 or 2, then there may be some rounding error, so again the feature maps may end up being smaller.

You Only Look Once (YOLO)

YOLO is a fast and accurate object detection architecture proposed by Joseph Redmon et al. in a [2015 paper](#),³³. It is so fast that it can run in real time on a video, as seen in Redmon's [demo](#). YOLO's architecture is quite similar to the one we just discussed, but with a few important differences:

- For each grid cell, YOLO only considers objects whose bounding box center lies within that cell. The bounding box coordinates are relative to that cell, where $(0, 0)$ means the top left of that cell and $(1, 1)$ means the bottom right. However, the bounding box's height and width may extend well beyond the cell.
- It outputs two bounding boxes for each grid cell (instead of just one), which allows the model to handle cases where two objects are so close to each other that their bounding box centers lie within the same cell. Each bounding box also comes with its own objectness score.
- YOLO also outputs a class probability distribution for each grid cell, predicting 20 class probabilities per grid cell since YOLO was trained on the PASCAL VOC dataset, which contains 20 classes. This produces a coarse *class probability map*. Note that the model predicts one class probability distribution per grid cell, not per bounding box. However, it's possible to estimate class probabilities for each bounding box during post-processing, by measuring how well each bounding box matches each class in the class probability map. For example, imagine a picture of a person standing in front of a car. There will be two bounding boxes: one large horizontal one for the car, and a smaller vertical one for the person. These bounding boxes may have their centers within the same grid cell. So how can we tell which class should be assigned to each bounding box? Well, the class probability map will contain a large region where the “car” class is dominant, and inside it there will be a smaller region where the “person” class is dominant. Hopefully, the car's bounding box will roughly match the “car” region, while the person's bounding box will roughly match the “person” region: this will allow the correct class to be assigned to each bounding box.

YOLO was originally developed using Darknet, an open source Deep Learning framework initially developed in C by Joseph Redmon, but it was soon ported to TensorFlow, Keras, PyTorch, and more. It was continuously improved over the years, with YOLOv2, YOLOv3 and YOLO9000 (again by Joseph Redmon et al.), YOLOv4 (by Alexey Bochkovskiy et al.), YOLOv5 (by Glenn Jocher), and PP-YOLO (by Xiang Long et al.). Each version brought some impressive improvements in speed and accuracy, using a variety of techniques. For example, YOLOv3 boosted accuracy

³³ Joseph Redmon et al., “You Only Look Once: Unified, Real-Time Object Detection,” *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016): 779–788.

in part thanks to *anchor priors*, exploiting the fact that some bounding box shapes are more likely than others, depending on the class (e.g., people tend to have vertical bounding boxes, while cars usually don't). They also increased the number of bounding boxes per grid cell, they trained on different datasets with many more classes (up to 9,000 classes organized in a hierarchy in the case of YOLO9000), they added skip connections to recover some of the spatial resolution that is lost in the CNN (we will discuss this shortly, when we look at semantic segmentation), and much more. There are also many variants of these models, such as YOLOv4-tiny, which is optimized to be trained on less powerful machines, and which can run extremely fast (at over 1,000 frames per second!), but with a slightly lower *mean Average Precision* (mAP).

Mean Average Precision (mAP)

A very common metric used in object detection tasks is the *mean Average Precision* (mAP). “Mean Average” sounds a bit redundant, doesn’t it? To understand this metric, let’s go back to two classification metrics we discussed in [Chapter 3](#): precision and recall. Remember the trade-off: the higher the recall, the lower the precision. You can visualize this in a precision/recall curve (see [Figure 3-6](#)). To summarize this curve into a single number, we could compute its area under the curve (AUC). But note that the precision/recall curve may contain a few sections where precision actually goes up when recall increases, especially at low recall values (you can see this at the top left of [Figure 3-6](#)). This is one of the motivations for the mAP metric.

Suppose the classifier has 90% precision at 10% recall, but 96% precision at 20% recall. There’s really no trade-off here: it simply makes more sense to use the classifier at 20% recall rather than at 10% recall, as you will get both higher recall and higher precision. So instead of looking at the precision *at* 10% recall, we should really be looking at the *maximum* precision that the classifier can offer with *at least* 10% recall. It would be 96%, not 90%. Therefore, one way to get a fair idea of the model’s performance is to compute the maximum precision you can get with at least 0% recall, then 10% recall, 20%, and so on up to 100%, and then calculate the mean of these maximum precisions. This is called the *Average Precision* (AP) metric. Now when there are more than two classes, we can compute the AP for each class, and then compute the mean AP (mAP). That’s it!

In an object detection system, there is an additional level of complexity: what if the system detected the correct class, but at the wrong location (i.e., the bounding box is completely off)? Surely we should not count this as a positive prediction. One approach is to define an IOU threshold: for example, we may consider that a prediction is correct only if the IOU is greater than, say, 0.5, and the predicted class is correct. The corresponding mAP is generally noted mAP@0.5 (or mAP@50%, or sometimes just AP₅₀). In some competitions (such as the PASCAL VOC challenge), this is what is done. In others (such as the COCO competition), the mAP is computed for different IOU thresholds (0.50, 0.55, 0.60, ..., 0.95), and the final metric is the

mean of all these mAPs (noted mAP@[.50:.95] or mAP@[.50:0.05:.95]). Yes, that's a mean mean average.

Many object detection models are available on TensorFlow Hub, often with pre-trained weights, such as YOLOv5³⁴, SSD³⁵, Faster R-CNN³⁶, or EfficientDet³⁷.

SSD and EfficientDet are “look once” detection models, similar to YOLO. EfficientDet is based on the EfficientNet convolutional architecture. Faster R-CNN is more complex: the image first goes through a CNN, then the output is passed to a *Region Proposal Network* (RPN) that proposes bounding boxes that are most likely to contain an object, and a classifier is then run for each bounding box, based on the cropped output of the CNN. The best place to start using these models is TensorFlow Hub’s excellent [object detection tutorial](#).

So far, we’ve only considered detecting objects in single images. But what about videos? Objects must not only be detected in each frame, they must also be tracked over time. Let’s take a quick look at object tracking now.

Object Tracking

Object tracking is a challenging task since objects move, they may grow or shrink as they get closer or further away from the camera, their appearance may change as they turn around or move to different lighting conditions or backgrounds, they may be temporarily occluded by other objects, and so on.

One of the most popular *object tracking* systems is DeepSORT³⁸. It is based on a combination of classical algorithms and Deep Learning:

- It uses *Kalman Filters* to estimate the most likely current position of an object given prior detections, and assuming that objects tend to move at a constant speed.

³⁴ You can find YOLOv3, YOLOv4, and their tiny variants in the TensorFlow Models project at <https://hml.info/yolof>.

³⁵ Wei Liu et al., “SSD: Single Shot Multibox Detector,” *Proceedings of the 14th European Conference on Computer Vision* 1 (2016): 21–37.

³⁶ Shaoqing Ren et al., “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks,” *Proceedings of the 28th International Conference on Neural Information Processing Systems* 1 (2015): 91–99.

³⁷ Mingxing Tan et al., “EfficientDet: Scalable and Efficient Object Detection,” arXiv preprint arXiv:1911.09070 (2019).

³⁸ Nicolai Wojke et al., “Simple Online and Realtime Tracking with a Deep Association Metric,” arXiv preprint arXiv:1703.07402 (2017).

- It also uses a Deep Learning model to measure the resemblance between new detections and existing tracked objects.
- Lastly, it uses the *Hungarian algorithm* to map new detections to existing tracked objects (or to new tracked objects): this algorithm efficiently finds the combination of mappings that minimizes the distance between the detections and the predicted positions of tracked objects, while also minimizing the appearance discrepancy.

For example, imagine a red ball that just bounced on a blue ball. Based on the previous positions of the balls, the Kalman Filter will predict that the balls will go through each other: indeed, it assumes that objects move at a constant speed, so it will not expect the bounce. If the Hungarian algorithm only considered positions, then it would happily map the new detections to the wrong balls, as if they just went through each other and swapped colors. But thanks to the resemblance measure, the Hungarian algorithm will notice the problem. Assuming the balls are not too similar, the algorithm will map the new detections to the correct balls.



There are a few DeepSORT implementations available on GitHub. For example, this TensorFlow implementation of YOLOv4 + Deep-SORT: <https://github.com/theAIGuysCode/yolov4-deepsort>.

So far we have located objects using bounding boxes. This is often sufficient, but sometimes you need to locate objects with much more precision, for example to remove the background behind a person during a videoconference call. Let's see how to go down to the pixel level.

Semantic Segmentation

In *semantic segmentation*, each pixel is classified according to the class of the object it belongs to (e.g., road, car, pedestrian, building, etc.), as shown in [Figure 14-27](#). Note that different objects of the same class are *not* distinguished. For example, all the bicycles on the right side of the segmented image end up as one big lump of pixels. The main difficulty in this task is that when images go through a regular CNN, they gradually lose their spatial resolution (due to the layers with strides greater than 1); so, a regular CNN may end up knowing that there's a person somewhere in the bottom left of the image, but it will not be much more precise than that.



Figure 14-27. Semantic segmentation

Just like for object detection, there are many different approaches to tackle this problem, some quite complex. However, a fairly simple solution was proposed in the 2015 paper by Jonathan Long et al. we discussed earlier, on Fully Convolutional Networks. The authors start by taking a pretrained CNN and turning it into an FCN. The CNN applies an overall stride of 32 to the input image (i.e., if you add up all the strides greater than 1), meaning the last layer outputs feature maps that are 32 times smaller than the input image. This is clearly too coarse, so they add a single *upsampling layer* that multiplies the resolution by 32.

There are several solutions available for upsampling (increasing the size of an image), such as bilinear interpolation, but that only works reasonably well up to $\times 4$ or $\times 8$. Instead, they use a *transposed convolutional layer*:³⁹ it is equivalent to first stretching the image by inserting empty rows and columns (full of zeros), then performing a regular convolution (see Figure 14-28). Alternatively, some people prefer to think of it as a regular convolutional layer that uses fractional strides (e.g., 1/2 in Figure 14-28). The transposed convolutional layer can be initialized to perform something close to linear interpolation, but since it is a trainable layer, it will learn to do better during training. In Keras, you can use the Conv2DTranspose layer.

³⁹ This type of layer is sometimes referred to as a *deconvolution layer*, but it does *not* perform what mathematicians call a deconvolution, so this name should be avoided.

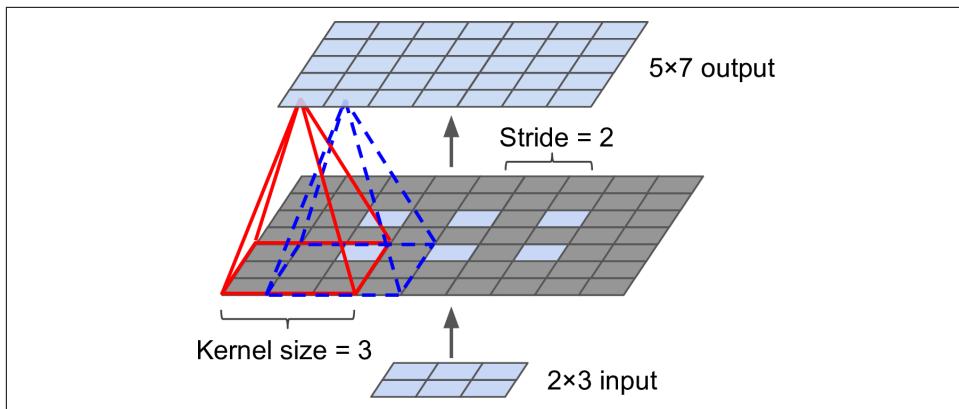


Figure 14-28. Upsampling using a transposed convolutional layer



In a transposed convolutional layer, the stride defines how much the input will be stretched, not the size of the filter steps, so the larger the stride, the larger the output (unlike for convolutional layers or pooling layers).

Other Keras Convolution Layers

Keras also offers a few other kinds of convolutional layers:

`tf.keras.layers.Conv1D`

A convolutional layer for 1D inputs, such as time series or text (sequences of letters or words), as we will see in [Chapter 15](#).

`tf.keras.layers.Conv3D`

A convolutional layer for 3D inputs, such as 3D PET scans.

`dilation_rate`

Setting the `dilation_rate` hyperparameter of any convolutional layer to a value of 2 or more creates an *à-trous convolutional layer* (“à trous” is French for “with holes”). This is equivalent to using a regular convolutional layer with a filter dilated by inserting rows and columns of zeros (i.e., holes). For example, a 1×3 filter equal to $[[1, 2, 3]]$ may be dilated with a *dilation rate* of 4, resulting in a *dilated filter* of $[[1, 0, 0, 0, 2, 0, 0, 0, 3]]$. This lets the convolutional layer have a larger receptive field at no computational price and using no extra parameters.

Using transposed convolutional layers for upsampling is OK, but still too imprecise. To do better, the authors added skip connections from lower layers: for example, they

upsampled the output image by a factor of 2 (instead of 32), and they added the output of a lower layer that had this double resolution. Then they upsampled the result by a factor of 16, leading to a total upsampling factor of 32 (see [Figure 14-29](#)). This recovered some of the spatial resolution that was lost in earlier pooling layers. In their best architecture, they used a second similar skip connection to recover even finer details from an even lower layer. In short, the output of the original CNN goes through the following extra steps: upsample $\times 2$, add the output of a lower layer (of the appropriate scale), upsample $\times 2$, add the output of an even lower layer, and finally upsample $\times 8$. It is even possible to scale up beyond the size of the original image: this can be used to increase the resolution of an image, which is a technique called *super-resolution*.

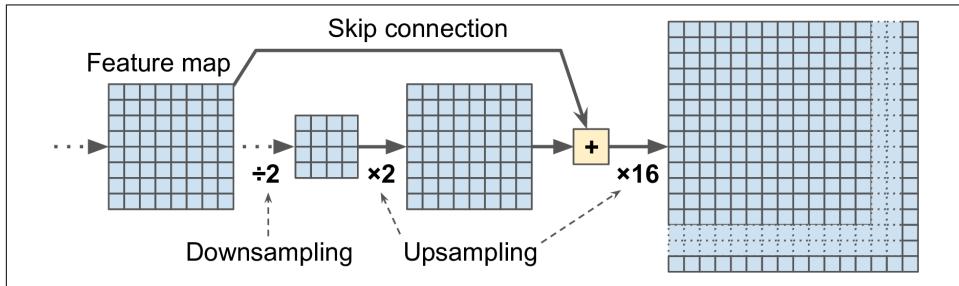


Figure 14-29. Skip layers recover some spatial resolution from lower layers

Instance segmentation is similar to semantic segmentation, but instead of merging all objects of the same class into one big lump, each object is distinguished from the others (e.g., it identifies each individual bicycle). For example the *Mask R-CNN* architecture, proposed in a [2017 paper⁴⁰](#) by Kaiming He et al., extends the Faster R-CNN model by additionally producing a pixel mask for each bounding box. So not only do you get a bounding box around each object, with a set of estimated class probabilities, but you also get a pixel mask that locates pixels in the bounding box that belong to the object. This model is available on TensorFlow Hub, pretrained on the COCO 2017 dataset. But the field is moving fast, so if you want to try the latest and greatest models, please check out the state-of-the-art section on <https://paperswithcode.com/>.

As you can see, the field of Deep Computer Vision is vast and moving fast, with all sorts of architectures popping out every year. Almost all of them are based on convolutional neural networks, but since 2020 another neural net architecture has entered the computer vision space: Transformers (which we will discuss in [Chapter 16](#)). The progress made over the last decade has been astounding, and researchers are now focusing on harder and harder problems, such as *adversarial learning* (which attempts

⁴⁰ Kaiming He et al., “Mask R-CNN,” arXiv preprint arXiv:1703.06870 (2017).

to make the network more resistant to images designed to fool it), explainability (understanding why the network makes a specific classification), realistic *image generation* (which we will come back to in [Chapter 17](#)), *single-shot learning* (a system that can recognize an object after it has seen it just once), predicting the next frames in a video, combining text and image tasks, and more.

Now on to the next chapter, where we will look at how to process sequential data such as time series using recurrent neural networks and convolutional neural networks.

Exercises

1. What are the advantages of a CNN over a fully connected DNN for image classification?
2. Consider a CNN composed of three convolutional layers, each with 3×3 kernels, a stride of 2, and “same” padding. The lowest layer outputs 100 feature maps, the middle one outputs 200, and the top one outputs 400. The input images are RGB images of 200×300 pixels.

What is the total number of parameters in the CNN? If we are using 32-bit floats, at least how much RAM will this network require when making a prediction for a single instance? What about when training on a mini-batch of 50 images?

3. If your GPU runs out of memory while training a CNN, what are five things you could try to solve the problem?
4. Why would you want to add a max pooling layer rather than a convolutional layer with the same stride?
5. When would you want to add a local response normalization layer?
6. Can you name the main innovations in AlexNet, compared to LeNet-5? What about the main innovations in GoogLeNet, ResNet, SENet, Xception, and EfficientNet?
7. What is a fully convolutional network? How can you convert a dense layer into a convolutional layer?
8. What is the main technical difficulty of semantic segmentation?
9. Build your own CNN from scratch and try to achieve the highest possible accuracy on MNIST.
10. Use transfer learning for large image classification, going through these steps:
 - a. Create a training set containing at least 100 images per class. For example, you could classify your own pictures based on the location (beach, mountain, city, etc.), or alternatively you can use an existing dataset (e.g., from TensorFlow Datasets).
 - b. Split it into a training set, a validation set, and a test set.

- c. Build the input pipeline, apply the appropriate preprocessing operations, and optionally add data augmentation.
 - d. Fine-tune a pretrained model on this dataset.
11. Go through TensorFlow's [Style Transfer tutorial](#). It is a fun way to generate art using Deep Learning.

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab3>.

Processing Sequences Using RNNs and CNNs

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 15th chapter of the final book. The GitHub repo is <https://github.com/ageron/handsonml3>.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

Predicting the future is something you do all the time, whether you are finishing a friend’s sentence or anticipating the smell of coffee at breakfast. In this chapter we will discuss recurrent neural networks (RNNs)—a class of nets that can predict the future (well, up to a point). RNNs can analyze time series data, such as the number of daily active users on your website, the hourly temperature in your city, your home’s daily power consumption, or the trajectories of nearby cars, and more. Once an RNN learns past patterns in the data, it is able to use its knowledge to forecast the future, assuming of course that past patterns still hold in the future.

More generally, RNNs can work on sequences of arbitrary lengths, rather than on fixed-sized inputs. For example, they can take sentences, documents, or audio samples as input, making them extremely useful for natural language processing applications such as automatic translation or speech-to-text.

In this chapter, we will first go through the fundamental concepts underlying RNNs and how to train them using backpropagation through time. Then, we will use them to forecast a time series. Along the way, we will look at the popular ARMA family of models, often used to forecast time series, and use them as baselines to compare with our RNNs. After that, we'll explore the two main difficulties that RNNs face:

- Unstable gradients (discussed in [Chapter 11](#)), which can be alleviated using various techniques, including *recurrent dropout* and *recurrent layer normalization*.
- A (very) limited short-term memory, which can be extended using LSTM and GRU cells.

RNNs are not the only types of neural networks capable of handling sequential data. For small sequences, a regular dense network can do the trick, and for very long sequences, such as audio samples or text, convolutional neural networks can actually work quite well too. We will discuss both of these possibilities, and we will finish this chapter by implementing a *WaveNet*—a CNN architecture capable of handling sequences of tens of thousands of time steps. Let's get started!

Recurrent Neurons and Layers

Up to now we have focused on feedforward neural networks, where the activations flow only in one direction, from the input layer to the output layer. A recurrent neural network looks very much like a feedforward neural network, except it also has connections pointing backward.

Let's look at the simplest possible RNN, composed of one neuron receiving inputs, producing an output, and sending that output back to itself, as shown in [Figure 15-1](#) (left). At each *time step* t (also called a *frame*), this *recurrent neuron* receives the inputs $\mathbf{x}_{(t)}$ as well as its own output from the previous time step, $\hat{\mathbf{y}}_{(t-1)}$. Since there is no previous output at the first time step, it is generally set to 0. We can represent this tiny network against the time axis, as shown in [Figure 15-1](#) (right). This is called *unrolling the network through time* (it's the same recurrent neuron represented once per time step).

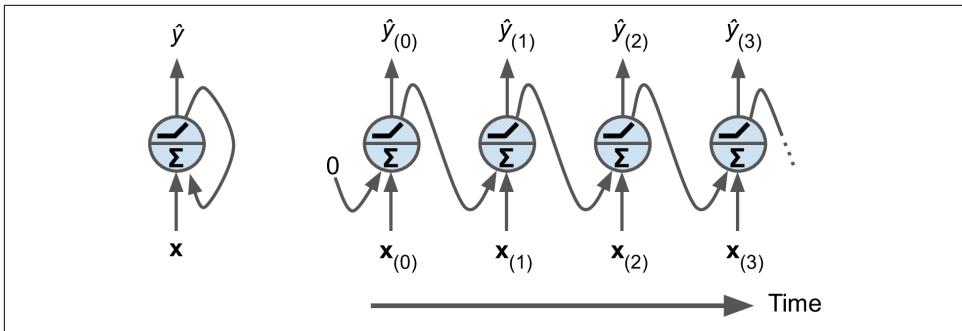


Figure 15-1. A recurrent neuron (left) unrolled through time (right)

You can easily create a layer of recurrent neurons. At each time step t , every neuron receives both the input vector $x_{(t)}$ and the output vector from the previous time step $\hat{y}_{(t-1)}$, as shown in Figure 15-2. Note that both the inputs and outputs are now vectors (when there was just a single neuron, the output was a scalar).

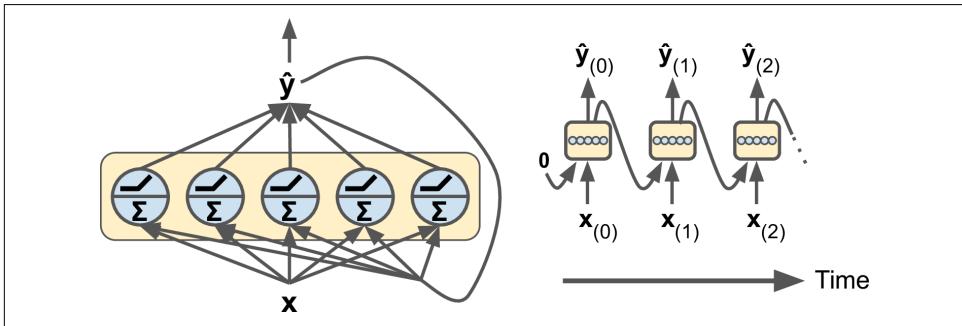


Figure 15-2. A layer of recurrent neurons (left) unrolled through time (right)

Each recurrent neuron has two sets of weights: one for the inputs $x_{(t)}$ and the other for the outputs of the previous time step, $\hat{y}_{(t-1)}$. Let's call these weight vectors w_x and $w_{\hat{y}}$. If we consider the whole recurrent layer instead of just one recurrent neuron, we can place all the weight vectors in two weight matrices: W_x and $W_{\hat{y}}$.

The output vector of the whole recurrent layer can then be computed pretty much as you might expect, as shown in [Equation 15-1](#), where \mathbf{b} is the bias vector and $\phi(\cdot)$ is the activation function (e.g., ReLU¹).

Equation 15-1. Output of a recurrent layer for a single instance

$$\hat{\mathbf{y}}_{(t)} = \varphi(\mathbf{W}_x^\top \mathbf{x}_{(t)} + \mathbf{W}_{\hat{y}}^\top \hat{\mathbf{y}}_{(t-1)} + \mathbf{b})$$

Just as with feedforward neural networks, we can compute a recurrent layer's output in one shot for an entire mini-batch by placing all the inputs at time step t into an input matrix $\mathbf{X}_{(t)}$ (see [Equation 15-2](#)).

Equation 15-2. Outputs of a layer of recurrent neurons for all instances in a mini-batch

$$\begin{aligned}\hat{\mathbf{Y}}_{(t)} &= \varphi(\mathbf{X}_{(t)} \mathbf{W}_x + \hat{\mathbf{Y}}_{(t-1)} \mathbf{W}_{\hat{y}} + \mathbf{b}) \\ &= \varphi([\mathbf{X}_{(t)} \quad \hat{\mathbf{Y}}_{(t-1)}] \mathbf{W} + \mathbf{b}) \text{ with } \mathbf{W} = \begin{bmatrix} \mathbf{W}_x \\ \mathbf{W}_{\hat{y}} \end{bmatrix}\end{aligned}$$

In this equation:

- $\hat{\mathbf{Y}}_{(t)}$ is an $m \times n_{\text{neurons}}$ matrix containing the layer's outputs at time step t for each instance in the mini-batch (m is the number of instances in the mini-batch and n_{neurons} is the number of neurons).
- $\mathbf{X}_{(t)}$ is an $m \times n_{\text{inputs}}$ matrix containing the inputs for all instances (n_{inputs} is the number of input features).
- \mathbf{W}_x is an $n_{\text{inputs}} \times n_{\text{neurons}}$ matrix containing the connection weights for the inputs of the current time step.
- $\mathbf{W}_{\hat{y}}$ is an $n_{\text{neurons}} \times n_{\text{neurons}}$ matrix containing the connection weights for the outputs of the previous time step.
- \mathbf{b} is a vector of size n_{neurons} containing each neuron's bias term.
- The weight matrices \mathbf{W}_x and $\mathbf{W}_{\hat{y}}$ are often concatenated vertically into a single weight matrix \mathbf{W} of shape $(n_{\text{inputs}} + n_{\text{neurons}}) \times n_{\text{neurons}}$ (see the second line of [Equation 15-2](#)).

¹ Note that many researchers prefer to use the hyperbolic tangent (tanh) activation function in RNNs rather than the ReLU activation function. For example, Vu Pham et al.'s 2013 paper "[Dropout Improves Recurrent Neural Networks for Handwriting Recognition](#)". ReLU-based RNNs are also possible, as shown in Quoc V. Le et al.'s 2015 paper "[A Simple Way to Initialize Recurrent Networks of Rectified Linear Units](#)".

- The notation $[X_{(t)} \hat{Y}_{(t-1)}]$ represents the horizontal concatenation of the matrices $X_{(t)}$ and $\hat{Y}_{(t-1)}$.

Notice that $\hat{Y}_{(t)}$ is a function of $X_{(t)}$ and $\hat{Y}_{(t-1)}$, which is a function of $X_{(t-1)}$ and $\hat{Y}_{(t-2)}$, which is a function of $X_{(t-2)}$ and $\hat{Y}_{(t-3)}$, and so on. This makes $\hat{Y}_{(t)}$ a function of all the inputs since time $t = 0$ (that is, $X_{(0)}, X_{(1)}, \dots, X_{(t)}$). At the first time step, $t = 0$, there are no previous outputs, so they are typically assumed to be all zeros.

Memory Cells

Since the output of a recurrent neuron at time step t is a function of all the inputs from previous time steps, you could say it has a form of *memory*. A part of a neural network that preserves some state across time steps is called a *memory cell* (or simply a *cell*). A single recurrent neuron, or a layer of recurrent neurons, is a very basic cell, capable of learning only short patterns (typically about 10 steps long, but this varies depending on the task). Later in this chapter, we will look at some more complex and powerful types of cells capable of learning longer patterns (roughly 10 times longer, but again, this depends on the task).

A cell's state at time step t , denoted $h_{(t)}$ (the “ h ” stands for “hidden”), is a function of some inputs at that time step and its state at the previous time step: $h_{(t)} = f(x_{(t)}, h_{(t-1)})$. Its output at time step t , denoted $\hat{y}_{(t)}$, is also a function of the previous state and the current inputs. In the case of the basic cells we have discussed so far, the output is just equal to the state, but in more complex cells this is not always the case, as shown in Figure 15-3.

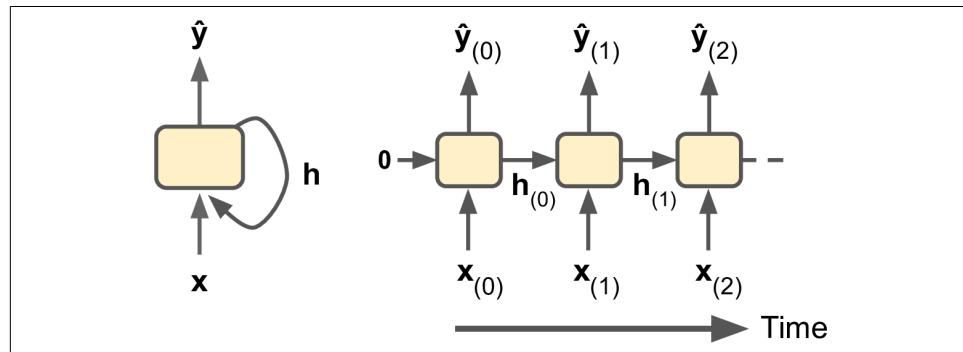


Figure 15-3. A cell's hidden state and its output may be different

Input and Output Sequences

An RNN can simultaneously take a sequence of inputs and produce a sequence of outputs (see the top-left network in Figure 15-4). This type of *sequence-to-sequence network* is useful to forecast time series, such as your home's daily power consumption: you feed it the data over the last N days, and you train it to output the

power consumption shifted by one day into the future (i.e., from $N - 1$ days ago to tomorrow).

Alternatively, you could feed the network a sequence of inputs and ignore all outputs except for the last one (see the top-right network in [Figure 15-4](#)). In other words, this is a *sequence-to-vector network*. For example, you could feed the network a sequence of words corresponding to a movie review, and the network would output a sentiment score (e.g., from 0 [hate] to 1 [love]).

Conversely, you could feed the network the same input vector over and over again at each time step and let it output a sequence (see the bottom-left network of [Figure 15-4](#)). This is a *vector-to-sequence network*. For example, the input could be an image (or the output of a CNN), and the output could be a caption for that image.

Lastly, you could have a sequence-to-vector network, called an *encoder*, followed by a vector-to-sequence network, called a *decoder* (see the bottom-right network of [Figure 15-4](#)). For example, this could be used for translating a sentence from one language to another. You would feed the network a sentence in one language, the encoder would convert this sentence into a single vector representation, and then the decoder would decode this vector into a sentence in another language. This two-step model, called an *encoder-decoder*², works much better than trying to translate on the fly with a single sequence-to-sequence RNN (like the one represented at the top left): the last words of a sentence can affect the first words of the translation, so you need to wait until you have seen the whole sentence before translating it. We will go through the implementation of an encoder–decoder in [Chapter 16](#) (as we will see, it is a bit more complex than what [Figure 15-4](#) suggests).

² Nal Kalchbrenner and Phil Blunsom, “Recurrent Continuous Translation Models” (2013).

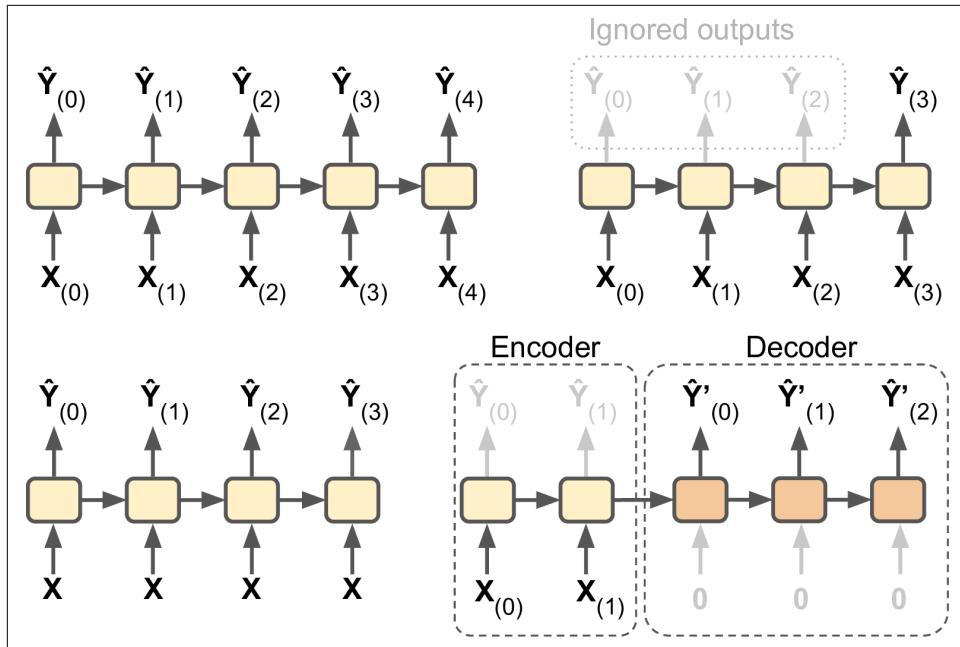


Figure 15-4. Seq-to-seq (top left), seq-to-vector (top right), vector-to-seq (bottom left), and encoder-decoder (bottom right) networks

This versatility sounds promising, but how do you train a recurrent neural network?

Training RNNs

To train an RNN, the trick is to unroll it through time (like we just did) and then use regular backpropagation (see [Figure 15-5](#)). This strategy is called *backpropagation through time* (BPTT).

Just like in regular backpropagation, there is a first forward pass through the unrolled network (represented by the dashed arrows). Then the output sequence is evaluated using a loss function $\mathcal{L}(Y_{(0)}, Y_{(1)}, \dots, Y_{(T)}; \hat{Y}_{(0)}, \hat{Y}_{(1)}, \dots, \hat{Y}_{(T)})$ (where $Y_{(i)}$ is the i^{th} target, $\hat{Y}_{(i)}$ is the i^{th} prediction, and T is the max time step). Note that this loss function may ignore some outputs. For example, in a sequence-to-vector RNN, all outputs are ignored except for the very last one. In [Figure 15-5](#), the loss function is computed based on the last three outputs only. The gradients of that loss function are then propagated backward through the unrolled network (represented by the solid arrows). In this example, since the outputs $\hat{Y}_{(0)}$ and $\hat{Y}_{(1)}$ are not used to compute the loss, the gradients do not flow backward through them, only through $\hat{Y}_{(2)}$, $\hat{Y}_{(3)}$, and $\hat{Y}_{(4)}$. Moreover, since the same parameters W and b are used at each time step, their gradients will be tweaked multiple times during backprop. Once the backward phase

is complete and all the gradients have been computed, BPTT can perform a Gradient Descent step to update the parameters (this is no different from regular backprop).

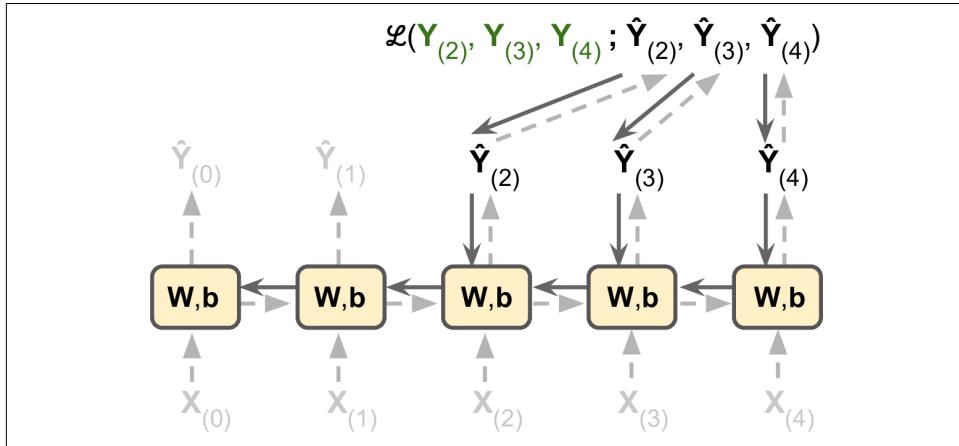


Figure 15-5. Backpropagation through time

Fortunately, Keras takes care of all of this complexity for you, as we will see. But before we get there, let's load a time series and start analyzing it using classical tools to better understand what we're dealing with, and to get some baseline metrics.

Forecasting a Time Series

All right! Let's pretend you've just been hired as a data scientist by Chicago's Transit Authority. Your first task is to build a model capable of forecasting the number of passengers that will ride on bus and rail the next day. You have access to daily ridership data since 2001. Let's start by loading and cleaning up the data:³

```
import pandas as pd
from pathlib import Path

path = Path("datasets/ridership/CTA_-_Ridership_-_Daily_Boarding_Totals.csv")
df = pd.read_csv(path, parse_dates=["service_date"])
df.columns = ["date", "day_type", "bus", "rail", "total"] # shorter names
df = df.sort_values("date").set_index("date")
df = df.drop("total", axis=1) # no need for total, it's just bus + rail
df = df.drop_duplicates() # remove duplicated months (2011-10 and 2014-07)
```

We load the CSV file, set short column names, sort the rows by date, remove the redundant “total” column, and drop duplicate rows. Let's check what the first few rows look like:

³ The latest data from the Chicago Transit Authority is available at [Chicago Data Portal](#).

```
>>> df.head()
      day_type    bus    rail
date
2001-01-01      U 297192 126455
2001-01-02      W 780827 501952
2001-01-03      W 824923 536432
2001-01-04      W 870021 550011
2001-01-05      W 890426 557917
```

On January 1st, 2001, 297,192 people boarded a bus in Chicago, and 126,455 boarded a train. The “day_type” column contains “W” for Weekdays, “A” for Saturdays, and “U” for Sundays or Holidays. Let’s plot 3 months of data (see [Figure 15-6](#)). Note that Pandas includes both the start and end month in the range, all the way up to the 31st of May.

This is a *time series*: data with values at different time steps, usually at regular intervals. More specifically, since there are multiple values per time step, this is called a *multivariate time series*. If we only looked at the “bus” column, it would be a *univariate time series*, with a single value per time step. Predicting future values (i.e., forecasting), is the most typical task when dealing with time series, and this is what we will focus on in this chapter. Other tasks include imputation (i.e., filling in missing past values), classification, anomaly detection, and more.

Now let’s plot the bus and rail riderships over a few months in 2019, to see what it looks like:

```
import matplotlib.pyplot as plt

df["2019-03":"2019-05"].plot(grid=True, marker=".", figsize=(8, 3.5))
plt.show()
```

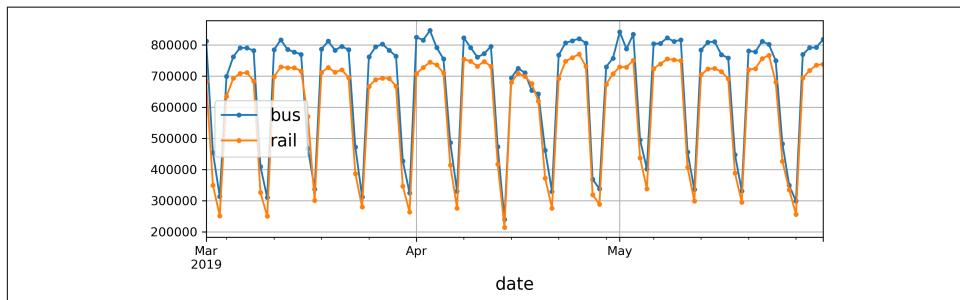


Figure 15-6. Daily ridership in Chicago

A similar pattern is clearly repeated every week. This is called a weekly *seasonality*. In fact, it’s so strong in this case that forecasting tomorrow’s ridership by just copying the values from a week earlier will yield reasonably good results. This is called *naive forecasting*: simply copying a past value to make our forecast. Naive forecasting is often a great baseline, and it can even be tricky to beat in some cases.



In general, naive forecasting means copying the latest known value (e.g., forecasting that tomorrow will be the same as today). However, in our case, copying the value from the previous week works better, due to the strong weekly seasonality.

To visualize these naive forecasts, let's overlay the two time series (for bus and rail) as well as the same time series lagged by one week (i.e., shifted towards the right) using dotted lines, and let's also plot the difference between the two (i.e., the value at time t minus the value at time $t - 7$). This is called *differencing* (see Figure 15-7):

```
diff_7 = df[["bus", "rail"]].diff(7)[ "2019-03": "2019-05"]

fig, axs = plt.subplots(2, 1, sharex=True, figsize=(8, 5))
df.plot(ax=axs[0], legend=False, marker=".") # original time series
df.shift(7).plot(ax=axs[0], grid=True, legend=False, linestyle=":") # lagged
diff_7.plot(ax=axs[1], grid=True, marker=".") # 7-day difference time series
plt.show()
```

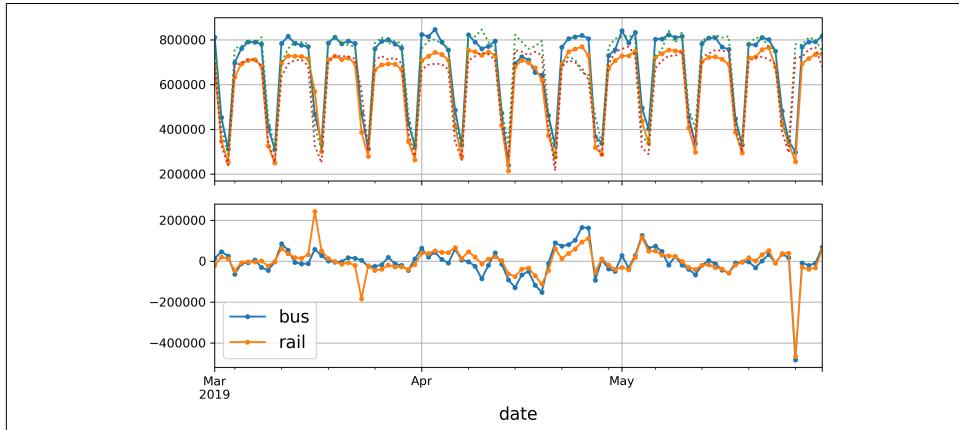


Figure 15-7. Time series overlaid with 7-day lagged time series (top), and difference between t and $t - 7$ (bottom)

Not too bad! Notice how closely the lagged time series track the actual time series. When a time series is correlated with a lagged version of itself, we say that the time series is *autocorrelated*. Notice how most differences are fairly small, except at the end of May. Maybe there was a holiday at that time? Let's check the “day_type” column:

```
>>> list(df.loc["2019-05-25": "2019-05-27"]["day_type"])
['A', 'U', 'U']
```

Indeed, there was a long weekend back then: the Monday was the Memorial Day holiday. We could use this column to improve our forecasts, but for now let's just measure the mean absolute error over the 3-month period we're arbitrarily focusing on—March, April, and May 2019—just to get a rough idea:

```
>>> diff_7.abs().mean()
bus      43915.608696
rail     42143.271739
dtype: float64
```

Our naive forecasts get an MAE of about 43,916 bus riders, and about 42,143 rail riders. It's hard to tell at a glance how good or bad this is, so let's put the forecast errors into perspective by dividing them by the target values:

```
>>> targets = df[["bus", "rail"]][["2019-03":"2019-05"]]
>>> (diff_7 / targets).abs().mean()
bus      0.082938
rail     0.089948
dtype: float64
```

What we just computed is called the *mean absolute percentage error* (MAPE): it looks like our naive forecasts give us a MAPE of roughly 8.3% for bus and 9.0% for rail. It's interesting to note that the MAE for the rail forecasts look slightly better than the MAE for the bus forecasts, while the opposite is true for the MAPE. That's because the bus ridership is larger than the rail ridership, so naturally the forecast errors are also larger, but when we put the errors into perspective, it turns out that the bus forecasts are actually slightly better than the rail forecasts.



The MAE, MAPE, or MSE are among the most common metrics you can use to evaluate your forecasts. As always, choosing the right metric depends on the task. For example, if your project suffers quadratically more from large errors than from small ones, then the MSE may be preferable, as it strongly penalizes large errors.

Looking at the time series, there doesn't appear to be any significant monthly seasonality, but let's check whether there's any yearly seasonality. We'll look at the data from 2001 to 2019. To reduce the risk of data snooping, we'll ignore more recent data for now. Let's also plot a 12-month rolling average for each series to visualize long-term trends (see [Figure 15-8](#)).

```
period = slice("2001", "2019")
df_monthly = df.resample('M').mean() # compute the mean for each month
rolling_average_12_months = df_monthly[period].rolling(window=12).mean()

fig, ax = plt.subplots(figsize=(8, 4))
df_monthly[period].plot(ax=ax, marker=".")
rolling_average_12_months.plot(ax=ax, grid=True, legend=False)
plt.show()
```

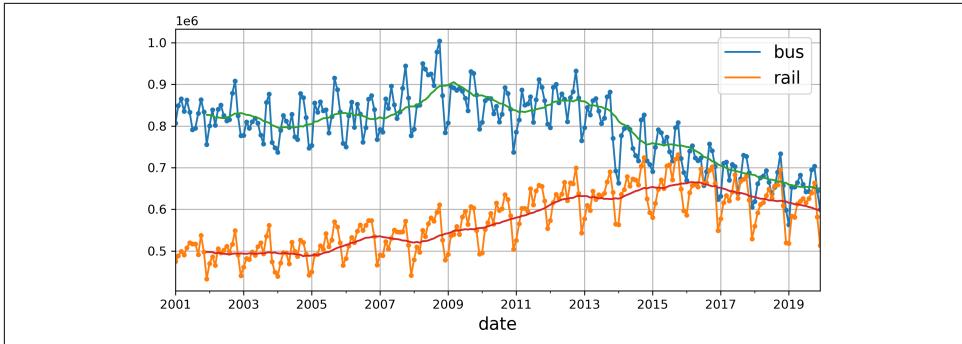


Figure 15-8. Yearly seasonality and long-term trends

Yep! There's definitely some yearly seasonality as well, although it is noisier than the weekly seasonality, and more visible on the rail series than on the bus series: we see peaks and troughs at roughly the same dates each year. Let's check what we get if we plot the 12-month difference:

```
df_monthly.diff(12)[period].plot(grid=True, marker=".", figsize=(8, 3))
plt.show()
```

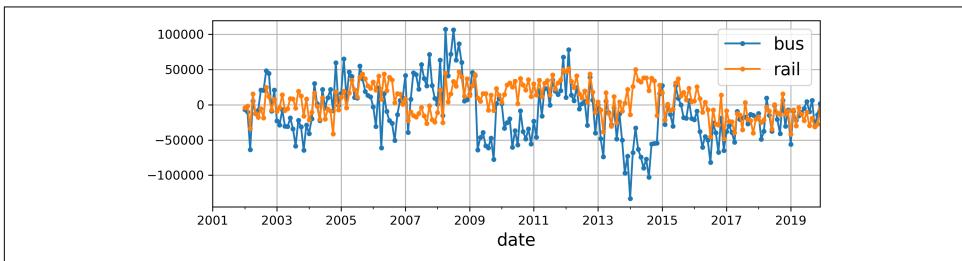


Figure 15-9. 12-month difference

Notice how differencing not only removed the yearly seasonality, but it also removed the long-term trends. For example, the linear downward trend present in the time series from 2016 to 2019 became a roughly constant negative value in the differenced time series. In fact, differencing is a common technique used to remove trend and seasonality from a time series: it's easier to study a *stationary* time series, meaning one whose statistical properties remain constant over time, without any seasonality or trends. Once you're able to make accurate forecasts on the differenced time series, it's easy to turn them into forecasts for the actual time series by just adding back the past values that were previously subtracted.

Now you may be thinking that we're only trying to predict tomorrow's ridership, so the long-term patterns matter much less than the short term ones. You're right, but still, we may be able to improve performance slightly by taking long-term patterns into account. For example, daily bus ridership dropped by about 2,500 in October

2017, which represents about 570 less passengers each week, so if we were at the end of October 2017, it would make sense to forecast tomorrow's ridership by copying the value from last week, minus 570. Accounting for the trend will make your forecasts a bit more accurate on average.

Now that you're familiar with the ridership time series, as well as some of the most important concepts in time series analysis, including seasonality, trend, differencing, and moving averages, let's take a quick look at a very popular family of statistical models that are commonly used to analyze time series.

The ARMA Model Family

Let's start with the *Autoregressive Moving Average* (ARMA) model, developed by Herman Wold in the 1930s: it computes its forecasts using a simple weighted sum of lagged values, and corrects these forecasts by adding a moving average, very much like we just discussed. Specifically, the moving average component is computed using a weighted sum of the last few forecast errors. [Equation 15-3](#) shows how the model makes its forecasts.

Equation 15-3. Forecasting using an ARMA model

$$\hat{y}(t) = \sum_{i=1}^p \alpha_i y(t-i) + \sum_{i=1}^q \theta_i \epsilon(t-i)$$

with $\epsilon(t) = y(t) - \hat{y}(t)$

- $\hat{y}_{(t)}$ is the model's forecast for time step t
- $y_{(t)}$ is the time series' value at time step t
- The first sum in the equation is the weighted sum of the past p values of the time series, using the learned weights α_i . The number p is a hyperparameter, and it determines how far back into the past the model should look. This sum is the *autoregressive* component of the model: it performs regression based on past values.
- The second sum is the weighted sum over the past q forecast errors $\epsilon_{(t)}$, using the learned weights θ_i . The number q is a hyperparameter. This sum is the moving average component of the model.

Importantly, this model assumes that the time series is stationary. If it is not, then differencing may help. Using differencing over a single time step will produce an approximation of the derivative of the time series: indeed, it will give the slope of the series at each time step. This means that it will eliminate any linear trend, transforming it into a constant value. For example, if you apply one-step differencing to the series [3, 5, 7, 9, 11], you get the differenced series [2, 2, 2, 2]. If the original time series has a quadratic trend instead of a linear trend, then a single round of

differencing will not be enough. For example, the series [1, 4, 9, 16, 25, 36], becomes [3, 5, 7, 9, 11] after one round of differencing. But if you run differencing for a second round, then you get [2, 2, 2, 2]. So running two rounds of differencing will eliminate quadratic trends. More generally, running d consecutive rounds of differencing computes an approximation of the d^{th} order derivative of the time series, so it will eliminate polynomial trends, up to degree d . This hyperparameter d is called the *order of integration*.

Differencing is the central contribution of the *Autoregressive Integrated Moving Average* (ARIMA) model, introduced in 1970 by George Box and Gwilym Jenkins in their book *Time Series Analysis*: this model runs d rounds of differencing to make the time series more stationary, then it applies a regular ARMA model. When making forecasts, it uses this ARMA model, then it adds back the terms that were subtracted by differencing.

One last member of the ARMA family is the Seasonal ARIMA (SARIMA) model: it models the time series in the same way as ARIMA, but it additionally models a seasonal component for a given frequency (e.g., weekly), using the exact same ARIMA approach. It has a total of 7 hyperparameters: the same p , d , and q , hyperparameters as ARIMA, plus additional P , D , and Q hyperparameters to model the seasonal pattern, and lastly the period of the seasonal pattern, noted s . The hyperparameters P , D , and Q are just like p , d , and q , but they are used to model the time series at $t - s$, $t - 2s$, $t - 3s$, etc.

Let's see how to fit a SARIMA model to the rail time series, and use it to make a forecast for tomorrow's ridership. We'll pretend today is the last day of May 2019, and we want to forecast the rail ridership for "tomorrow", the 1st of June, 2019. For this, we can use the *statsmodels* library, which contains many different statistical models, including the ARMA model and its variants, implemented by the ARIMA class:

```
from statsmodels.tsa.arima.model import ARIMA

origin, today = "2019-01-01", "2019-05-31"
rail_series = df.loc[origin:today]["rail"].asfreq("D")
model = ARIMA(rail_series,
              order=(1, 0, 0),
              seasonal_order=(0, 1, 1, 7))
model = model.fit()
y_pred = model.forecast() # returns 427,758.6
```

- In this code example, we start by importing the ARIMA class, then we take the rail ridership data from the start of 2019 and up to "today", and we use `asfreq("D")` to set the time series' frequency to daily: this doesn't change the data at all in this case, since it's already daily, but without this the ARIMA class would have to guess the frequency, and it would display a warning.

- Next, we create an ARIMA instance, passing it all the data until “today”, and we set the model hyperparameters: `order=(1, 0, 0)` means that $p = 1$, $d = 0$, $q = 0$, and `seasonal_order=(0, 1, 1, 7)` means that $P = 0$, $D = 1$, $Q = 1$, and $s = 7$. Notice that the statsmodels API differs a bit from Scikit-Learn’s API, since we pass the data to the model at construction time, instead of passing it to the `fit()` method.
- Next, we fit the model, and we use it to make a forecast for “tomorrow”, the 1st of June, 2019.

The forecast is 427,759 passengers, when in fact there were 379,044. Yikes, we’re 12.9% off, that’s pretty bad. It’s actually slightly worse than naive forecasting, which forecasts 426,932, off by 12.6%. But perhaps we were just unlucky that day? To check this, we can run the same code in a loop to make forecasts for every day in March, April, and May, and compute the MAE over that period:

```
origin, start_date, end_date = "2019-01-01", "2019-03-01", "2019-05-31"
time_period = pd.date_range(start_date, end_date)
rail_series = df.loc[origin:end_date]["rail"].asfreq("D")
y_preds = []
for today in time_period.shift(-1):
    model = ARIMA(rail_series[origin:today], # train on data up to "today"
                  order=(1, 0, 0),
                  seasonal_order=(0, 1, 1, 7))
    model = model.fit() # note that we retrain the model every day!
    y_pred = model.forecast()[0]
    y_preds.append(y_pred)

y_preds = pd.Series(y_preds, index=time_period)
mae = (y_preds - rail_series[time_period]).abs().mean() # returns 32,040.7
```

Ah, that’s much better! The MAE is about 32,041, which is significantly lower than the MAE we got with naive forecasting (42,143). So although the model is not perfect, it still beats naive forecasting by a large margin, on average.

At this point, you may be wondering how to pick good hyperparameters for the SARIMA model. Well, there are several methods, but the simplest to understand and to get started is the bruteforce approach: just run a grid search. For each model you want to evaluate (i.e., each hyperparameter combination), you can run the preceding code example, changing only the hyperparameter values. Good p , q , P , and Q values are usually fairly small (typically 0 to 2, sometimes up to 5 or 6), and d , and D are typically 0 or 1, sometimes 2. As for s , it’s just the main seasonal pattern’s period: in our case it’s 7 since there’s a strong weekly seasonality. The model with the lowest

MAE wins. Of course, you can replace the MAE with another metric if it better matches your business objective. And that's it!⁴

Preparing The Data For Machine Learning Models

Now that we have two baselines, naive forecasting and SARIMA, let's try to use the Machine Learning models we covered so far to forecast this time series, starting with a basic linear model. Our goal will be to forecast tomorrow's ridership based on the ridership of the past 8 weeks of data (56 days). The inputs to our model will therefore be sequences (usually a single sequence per day once the model is in production), each containing 56 values from time steps $t - 55$ to t . For each input sequence, the model will output a single value: the forecast for time step $t + 1$.

But what will we use as training data? Well, that's the trick: we will use every 56-day window from the past as training data, and the target for each window will be the value immediately following it.

Keras actually has a nice utility function called `tf.keras.utils.timeseries_dataset_from_array()` to help us prepare the training set. It takes a time series as input, and it builds a `tf.data` dataset (introduced in [Chapter 13](#)) containing all the windows of the desired length, as well as their corresponding targets. Here's an example which takes a time series containing the numbers 0 to 5, and creates a dataset containing all the windows of length 3, with their corresponding targets, grouped into batches of size 2:

```
import tensorflow as tf

my_series = [0, 1, 2, 3, 4, 5]
my_dataset = tf.keras.utils.timeseries_dataset_from_array(
    my_series,
    targets=my_series[3:], # the targets are 3 steps into the future
    sequence_length=3,
    batch_size=2
)
```

Let's inspect the contents of this dataset:

```
>>> list(my_dataset)
[(<tf.Tensor: shape=(2, 3), dtype=int32, numpy=
array([[0, 1, 2],
       [1, 2, 3]]), dtype=int32>),
 <tf.Tensor: shape=(2,), dtype=int32, numpy=array([3, 4], dtype=int32)>),
```

⁴ There are other more principled approaches to selecting good hyperparameters, based on analyzing the *autocorrelation function* (ACF) and *partial autocorrelation function* (PACF), or minimizing the AIC or BIC metrics (introduced in [Chapter 9](#)) to penalize models that use too many parameters and reduce the risk of overfitting the data, but grid search is a good place to start. For more details on the ACF-PACF approach, check out this very nice [post by Jason Brownlee](#).

```
<tf.Tensor: shape=(1, 3), dtype=int32, numpy=array([[2, 3, 4]], dtype=int32)>,
<tf.Tensor: shape=(1,), dtype=int32, numpy=array([5], dtype=int32)>]
```

Each sample in the dataset is a window of length 3, along with its corresponding target (i.e., the value immediately after the window). The windows are [0, 1, 2], [1, 2, 3], and [2, 3, 4], and their respective targets are 3, 4, and 5. Since there are 3 windows in total, which is not a multiple of the batch size, the last batch only contains 1 window instead of 2.

Another way to get the same result is to use the `window()` method of `tf.data`'s `Dataset` class. It's more complex, but it gives you full control, which will come in handy later in this chapter, so let's see how it works. The `window()` method returns a dataset of window datasets, for example:

```
>>> for window_dataset in tf.data.Dataset.range(6).window(4, shift=1):
...     for element in window_dataset:
...         print(f"{element}", end=" ")
...     print()
...
0 1 2 3
1 2 3 4
2 3 4 5
3 4 5
4 5
5
```

In this example, the dataset contains 6 windows, each shifted by 1 step compared to the previous one, and the last 3 windows are smaller because they've reached the end of the series. In general you'll want to get rid of these smaller windows by passing `drop_remainder=True` to the `window()` method.

The `window()` method returns a *nested dataset*, analogous to a list of lists. This is useful when you want to transform each window by calling its dataset methods (e.g., to shuffle them or batch them). However, we cannot use a nested dataset directly for training, as our model will expect tensors as input, not datasets.

Therefore, we must call the `flat_map()` method: it converts a nested dataset into a *flat dataset* (one that contains tensors, not datasets). For example, suppose {1, 2, 3} represents a dataset containing the sequence of tensors 1, 2, and 3. If you flatten the nested dataset {{1, 2}, {3, 4, 5, 6}}, you get back the flat dataset {1, 2, 3, 4, 5, 6}.

Moreover, the `flat_map()` method takes a function as an argument, which allows you to transform each dataset in the nested dataset before flattening. For example, if you pass the function `lambda ds: ds.batch(2)` to `flat_map()`, then it will transform the nested dataset {{1, 2}, {3, 4, 5, 6}} into the flat dataset {[1, 2], [3, 4], [5, 6]}: it's a dataset containing 3 tensors, each of size 2.

With that in mind, we are ready to flatten our dataset:

```

>>> dataset = tf.data.Dataset.range(6).window(4, shift=1, drop_remainder=True)
>>> dataset = dataset.flat_map(lambda window_dataset: window_dataset.batch(4))
>>> for window_tensor in dataset:
...     print(f"window_tensor={window_tensor}")
...
[0 1 2 3]
[1 2 3 4]
[2 3 4 5]

```

Since each window dataset contains exactly 4 items, calling `batch(4)` on a window produces a single tensor of size 4. Great! We now have a dataset containing consecutive windows represented as tensors. Let's create a little helper function to make it easier to extract windows from a dataset:

```

def to_windows(dataset, length):
    dataset = dataset.window(length, shift=1, drop_remainder=True)
    return dataset.flat_map(lambda window_ds: window_ds.batch(length))

```

The last step is to split each window into inputs and targets, using the `map()` method. We can also group the resulting windows into batches of size 2:

```

>>> dataset = to_windows(tf.data.Dataset.range(6), 4) # 3 inputs + 1 target = 4
>>> dataset = dataset.map(lambda window: (window[:-1], window[-1]))
>>> list(dataset.batch(2))
[(<tf.Tensor: shape=(2, 3), dtype=int64, numpy=
  array([[0, 1, 2],
         [1, 2, 3]]),<tf.Tensor: shape=(2,), dtype=int64, numpy=array([3, 4])>),
 (<tf.Tensor: shape=(1, 3), dtype=int64, numpy=array([[2, 3, 4]])>,
 <tf.Tensor: shape=(1,), dtype=int64, numpy=array([5])>)]

```

As you can see, we now have the same output as we got earlier with the `timeseries_dataset_from_array()` function (with a bit more effort, but it will be worthwhile soon).

Now before we start training, we need to split our data into a training period, a validation period, and a test period. We will focus on the rail ridership for now. We will also scale it down by a factor of one million, to ensure the values are near the 0-1 range: this plays nicely with the default weight initialization and learning rate.

```

rail_train = df["rail"]["2016-01":"2018-12"] / 1e6
rail_valid = df["rail"]["2019-01":"2019-05"] / 1e6
rail_test = df["rail"]["2019-06":] / 1e6

```



When dealing with time series, you generally want to split across time. However, in some cases you may be able to split along other dimensions, which will give you a longer time period to train on. For example, if you have data about the financial health of 10,000 companies from 2001 to 2019, you might be able to split this data across the different companies. It's very likely that many of these companies will be strongly correlated, though (e.g., whole economic sectors may go up or down jointly), and if you have correlated companies across the training set and the test set, your test set will not be as useful, as its measure of the generalization error will be optimistically biased.

Next, let's use `timeseries_dataset_from_array()` to create datasets for training and validation. Since Gradient Descent expects the instances in the training set to be independent and identically distributed (IID), as we saw in [Chapter 4](#), we must set the argument `shuffle=True` to just shuffle the training windows (but not their contents).

```
seq_length = 56
train_ds = tf.keras.utils.timeseries_dataset_from_array(
    rail_train.to_numpy(),
    targets=rail_train[seq_length:],
    sequence_length=seq_length,
    batch_size=32,
    shuffle=True,
    seed=42
)
valid_ds = tf.keras.utils.timeseries_dataset_from_array(
    rail_valid.to_numpy(),
    targets=rail_valid[seq_length:],
    sequence_length=seq_length,
    batch_size=32
)
```

And now we're ready to build and train any regression model we want!

Forecasting Using a Linear Model

Let's try a basic linear model first. We will use the Huber loss, which usually works better than minimizing the MAE directly, as discussed in [Chapter 10](#). We'll also use early stopping:

```
tf.random.set_seed(42)
model = tf.keras.Sequential([
    tf.keras.layers.Dense(1, input_shape=[seq_length])
])
early_stopping_cb = tf.keras.callbacks.EarlyStopping(
    monitor="val_mae", patience=50, restore_best_weights=True)
opt = tf.keras.optimizers.SGD(learning_rate=0.02, momentum=0.9)
model.compile(loss=tf.keras.losses.Huber(), optimizer=opt, metrics=["mae"])
```

```
history = model.fit(train_ds, validation_data=valid_ds, epochs=500,
                     callbacks=[early_stopping_cb])
```

This model reaches a validation MAE of about 37,866 (your mileage may vary). That's better than naive forecasting, but worse than the SARIMA model.⁵

Can we do better with an RNN? Let's see!

Forecasting Using a Simple RNN

Let's try the most basic RNN, containing a single recurrent layer with just one recurrent neuron, as we saw in [Figure 15-1](#):

```
model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(1, input_shape=[None, 1])
])
```

All recurrent layers in Keras expect 3-dimensional inputs of shape [*batch size, time steps, dimensionality*], where *dimensionality* is 1 for univariate time series and more for multivariate time series. Recall that the `input_shape` argument ignores the first dimension (i.e., the batch size), and since recurrent layers can accept input sequences of any length, we can set the second dimension to `None`, which means "any size". Lastly, since we're dealing with a univariate time series, we need the last dimension's size to be 1. This is why we specified the input shape `[None, 1]`: it means "univariate sequences of any length". Note that the datasets actually contain inputs of shape [*batch size, time steps*], so we're missing the last dimension, of size 1, but Keras is kind enough to add it for us in this case.

This model works exactly as we saw earlier: the initial state $h_{(\text{init})}$ is set to 0, and it is passed to a single recurrent neuron, along with the value of the first time step, $x_{(0)}$. The neuron computes a weighted sum of these values plus the bias term, and it applies the activation function to the result, using the hyperbolic tangent function by default. The result is the first output, y_0 . In a simple RNN, this output is also the new state h_0 . This new state is passed to the same recurrent neuron along with the next input value, $x_{(1)}$, and the process is repeated until the last time step. At the end, the layer just outputs the last value: in our case the sequences are 56 steps long, so the last value is y_{55} . All of this is performed simultaneously for every sequence in the batch, 32 in this case.

⁵ Note that the validation period starts on the 1st of January 2019, so the first prediction is for the 26th of February 2019, 8 weeks later. When we evaluated the baseline models, we used predictions starting on the 1st of March instead. But this should be close enough.



By default, recurrent layers in Keras only return the final output. To make them return one output per time step, you must set `return_sequences=True`, as we will see.

So that's our first recurrent model! It's a sequence-to-vector model. Since there's a single output neuron, the output vector has a size of 1.

Now if you compile, train, and evaluate this model just like the previous model, you will find that it's no good at all: its validation MAE is greater than 100,000! Ouch. That was to be expected for two reasons:

- First, the model only has a single recurrent neuron, so the only data it can use to make a prediction at each time step is the input value at the current time step and the output value from the previous time step. That's not much to go on! In other words, the RNN's memory is extremely limited: it's just a single number, its previous output. And let's count how many parameters this model has: since there's just one recurrent neuron with only two input values, the whole model only has three parameters (two weights plus a bias term). That's far from enough for this time series. In contrast, our previous model could look at all 56 previous values at once, and it had a total of 57 parameters.
- Second, since the default activation function is `tanh`, the recurrent layer can only output values between `-1` and `+1`. But the time series contains values from `0` to about `1.4`. So there's no way it can predict values between `1.0` and `1.4`.

Let's fix both of these issues: we will create a model with a larger recurrent layer, containing 32 recurrent neurons, and we will add a dense output layer on top of it, with a single output neuron and no activation function. The recurrent layer will be able to carry much more information from one time step to the next, and the dense output layer will project the final output from 32 dimensions down to 1, without any value range constraints:

```
univar_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, input_shape=[None, 1]),
    tf.keras.layers.Dense(1) # no activation function by default
])
```

Now if you compile, fit, and evaluate this model just like earlier, you will find that its validation MAE reaches 27,703. That's the best model we trained so far, and it even beats the SARIMA model: we're doing pretty good!



We've only normalized the time series, without removing its trend and seasonality, and yet the model still performs well. This is convenient, as it makes it possible to quickly search for promising models without worrying too much about preprocessing. However, to get the best performance, you may want to try making the time series more stationary, for example using differencing.

Forecasting Using a Deep RNN

It is quite common to stack multiple layers of cells, as shown in [Figure 15-10](#). This gives you a *deep RNN*.

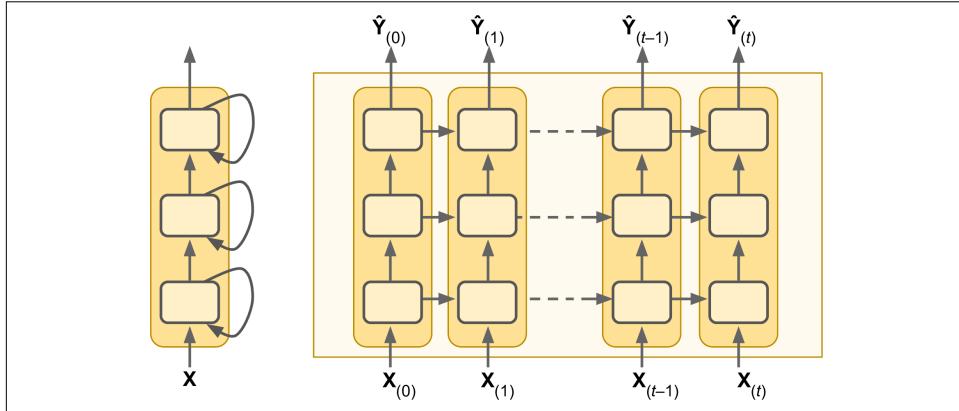


Figure 15-10. Deep RNN (left) unrolled through time (right)

Implementing a deep RNN with Keras is straightforward: just stack recurrent layers. In the following example, we use three `SimpleRNN` layers (but we could use any other type of recurrent layer instead, such as an `LSTM` layer or a `GRU` layer, which we will discuss shortly). The first two are sequence-to-sequence layers, and the last one is a sequence-to-vector layer. Finally, the `Dense` layer produces the model's forecast (you can think of it as a vector-to-vector layer). So this model is just like the model represented in [Figure 15-10](#), except the outputs $\hat{Y}_{(0)}$ to $\hat{Y}_{(t-1)}$ are ignored, and there's a dense layer on top of $\hat{Y}_{(t)}$, which outputs the actual forecast.

```
deep_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, return_sequences=True, input_shape=[None, 1]),
    tf.keras.layers.SimpleRNN(32, return_sequences=True),
    tf.keras.layers.SimpleRNN(32),
    tf.keras.layers.Dense(1)
])
```



Make sure to set `return_sequences=True` for all recurrent layers (except the last one, if you only care about the last output). If you forget to set this parameter for one recurrent layer, it will output a 2D array containing only the output of the last time step, instead of a 3D array containing outputs for all time steps. The next recurrent layer will complain that you are not feeding it sequences in the expected 3D format.

If you train and evaluate this model, you will find that it reaches an MAE of about 31,211. That's better than both baselines, but it doesn't beat our "shallow" RNN. It looks like this RNN is a bit too large for our task.

Forecasting Multivariate Time Series

A great quality of neural networks is their flexibility: in particular, they can deal with multivariate time series with almost no change to their architecture. For example, let's try to forecast the rail time series using both the bus and rail data as input. In fact, let's also throw in the day type! Since we can always know in advance whether tomorrow is going to be a weekday, a weekend, or a holiday, we can shift the day type series one day into the future, so that the model is given tomorrow's day type as input. For simplicity, let's do this processing using Pandas:

```
df_mulvar = df[["bus", "rail"]] / 1e6 # use both bus & rail series as input
df_mulvar["next_day_type"] = df["day_type"].shift(-1) # we know tomorrow's type
df_mulvar = pd.get_dummies(df_mulvar) # one-hot encode the day type
```

Now `df_mulvar` is a DataFrame with 5 columns: the bus and rail data, plus three columns containing the one-hot encoding of the next day's type, as there are three possible day types (W, A, and U). Next we can proceed much like we did earlier. First, split the data into three periods, for training, validation, and testing:

```
mulvar_train = df_mulvar["2016-01":"2018-12"]
mulvar_valid = df_mulvar["2019-01":"2019-05"]
mulvar_test = df_mulvar["2019-06":]
```

Next, create the datasets:

```
train_mulvar_ds = tf.keras.utils.timeseries_dataset_from_array(
    mulvar_train.to_numpy(), # use all 5 columns as input
    targets=mulvar_train["rail"][seq_length:], # forecast only the rail series
    [...] # the other 4 arguments are the same as earlier
)
valid_mulvar_ds = tf.keras.utils.timeseries_dataset_from_array(
    mulvar_valid.to_numpy(),
    targets=mulvar_valid["rail"][seq_length:],
    [...] # the other 2 arguments are the same as earlier
)
```

And finally create the RNN:

```

mulvar_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, input_shape=[None, 5]),
    tf.keras.layers.Dense(1)
])

```

Notice that the only difference with the `univar_model` RNN we built earlier is the input shape: at each time step, the model now receives 5 inputs instead of 1. This model actually reaches a validation MAE of 22,062. Now we're making big progress!

In fact, it's not too hard to make the RNN forecast both the bus and the rail ridership: all you need to do is change the targets when creating the datasets, setting them to `mulvar_train[["bus", "rail"]][seq_length:]` for the training set, and `mulvar_valid[["bus", "rail"]][seq_length:]` for the validation set. You must also add an extra neuron in the output Dense layer, since it must now make 2 forecasts: one for tomorrow's bus ridership, and the other for rail. And that's all there is to it!

As we discussed in [Chapter 10](#), using a single model for multiple related tasks often performs better than using a separate model for each task, since features learned for one task may be useful for the other tasks, and also because having to perform well across multiple tasks prevents the model from overfitting (it's a form of regularization). However, it depends on the task, and in this particular case, the multitask RNN that forecasts both the bus and the rail ridership doesn't perform quite as well as dedicated models that forecast one or the other (using all 5 columns as input). Still, it reaches a validation MAE of 25,330 for rail and 26,369 for bus, which is pretty good.

Forecasting Several Time Steps Ahead

So far we have only predicted the value at the next time step, but we could just as easily have predicted the value several steps ahead by changing the targets appropriately (e.g., to predict the ridership two weeks from now, just change the targets to be the value 14 days ahead instead of 1 day ahead). But what if we want to predict the next 14 values?

The first option is to take the `univar_model` RNN we trained earlier for the rail time series, make it predict the next value, then add that value to the inputs, acting as if the predicted value had actually occurred, and use the model again to predict the following value, and so on, as in the following code:

```

import numpy as np

X = rail_valid.to_numpy()[:, np.newaxis, :seq_length, np.newaxis]
for step_ahead in range(14):
    y_pred_one = univar_model.predict(X)
    X = np.concatenate([X, y_pred_one.reshape(1, 1, 1)], axis=1)

```

In this code, we take the rail ridership of the first 56 days of the validation period, and we convert the data to a NumPy array of shape [1, 56, 1] (recall that recurrent layers expect 3D inputs). Then we repeatedly use the model to forecast the next value, and we append each forecast to the input series, along the time axis (`axis=1`). The resulting forecasts are plotted in [Figure 15-11](#).

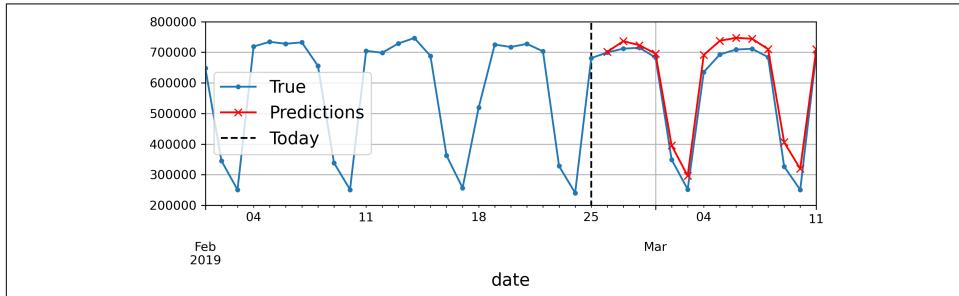


Figure 15-11. Forecasting 14 steps ahead, 1 step at a time



If the model makes an error at one time step, then the forecasts for the following time steps are impacted as well: the errors tend to accumulate. So it's preferable to use this technique only for a small number of steps.

The second option is to train an RNN to predict the next 14 values in one shot. We can still use a sequence-to-vector model, but it will output 14 values instead of 1. However, we first need to change the targets to be vectors containing the next 14 values. To do this, we can use `timeseries_dataset_from_array()` again, but this time asking it to create datasets without targets (`targets=None`), and with longer sequences, of length `seq_length + 14`. Then we can use the datasets' `map()` method to apply a custom function to each batch of sequences, splitting them into inputs and targets. In this example, we use the multivariate time series as input (using all 5 columns), and we forecast the rail ridership for the next 14 days:⁶

```
def split_inputs_and_targets(mulvar_series, ahead=14, target_col=1):
    return mulvar_series[:, :-ahead], mulvar_series[:, -ahead:, target_col]

ahead_train_ds = tf.keras.utils.timeseries_dataset_from_array(
    mulvar_train.to_numpy(),
    targets=None,
    sequence_length=seq_length + 14,
    [...] # the other 3 arguments are the same as earlier
```

⁶ Feel free to play around with this model. For example, you can try forecasting both the bus and rail riderships for the next 14 days. You'll need to tweak the targets to include both the bus and rail riderships, and make your model output 28 forecasts instead of 14.

```

).map(split_inputs_and_targets)
ahead_valid_ds = tf.keras.utils.timeseries_dataset_from_array(
    mulvar_valid.to_numpy(),
    targets=None,
    sequence_length=seq_length + 14,
    batch_size=32
).map(split_inputs_and_targets)

```

Now we just need the output layer to have 14 units instead of 1:

```

ahead_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, input_shape=[None, 5]),
    tf.keras.layers.Dense(14)
])

```

After training this model, you can predict the next 14 values at once like this:

```

X = mulvar_valid.to_numpy()[:, np.newaxis, :seq_length] # shape [1, 56, 5]
Y_pred = ahead_model.predict(X) # shape [1, 14]

```

This approach works quite well. Its forecast for the next day are obviously better than its forecast for 14 days into the future, but it doesn't accumulate errors like the previous approach did. However, we can still do better, using a sequence-to-sequence (or *seq2seq*) model.

Forecasting Using a Sequence-To-Sequence Model

Instead of training the model to forecast the next 14 values only at the very last time step, we can train it to forecast the next 14 values at each and every time step. In other words, we can turn this sequence-to-vector RNN into a sequence-to-sequence RNN. The advantage of this technique is that the loss will contain a term for the output of the RNN at each and every time step, not just for the output at the last time step. This means there will be many more error gradients flowing through the model, and they won't have to flow through time as much since they will come from the output of each time step, not just the last one. This will both stabilize and speed up training.

To be clear, at time step 0 the model will output a vector containing the forecasts for time steps 1 to 14, then at time step 1 the model will forecast time steps 2 to 15, and so on. In other words, the targets are sequences of consecutive windows, shifted by 1 time step at each time step. The target is not a vector anymore, but a sequence of the same length as the inputs, containing a 14-dimensional vector at each step. Preparing the datasets is not trivial, since each instance has a window as input, and a sequence of windows as output. One way to do this is to use the `to_windows()` utility function we created earlier, twice in a row, to get windows of consecutive windows. For example, let's turn the series of numbers 0 to 6 into a dataset containing sequences of 4 consecutive windows, each of length 3:

```

>>> my_series = tf.data.Dataset.range(7)
>>> dataset = to_windows(to_windows(my_series, 3), 4)

```

```
>>> list(dataset)
[<tf.Tensor: shape=(4, 3), dtype=int64, numpy=
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4],
       [3, 4, 5]])>,
 <tf.Tensor: shape=(4, 3), dtype=int64, numpy=
array([[1, 2, 3],
       [2, 3, 4],
       [3, 4, 5],
       [4, 5, 6]])>]
```

Now we can use the `map()` method to split these windows of windows into inputs and targets:

```
>>> dataset = dataset.map(lambda S: (S[:, 0], S[:, 1:]))
>>> list(dataset)
[(<tf.Tensor: shape=(4,), dtype=int64, numpy=array([0, 1, 2, 3])>,
 <tf.Tensor: shape=(4, 2), dtype=int64, numpy=
array([[1, 2],
       [2, 3],
       [3, 4],
       [4, 5]])>),
 (<tf.Tensor: shape=(4,), dtype=int64, numpy=array([1, 2, 3, 4])>,
 <tf.Tensor: shape=(4, 2), dtype=int64, numpy=
array([[2, 3],
       [3, 4],
       [4, 5],
       [5, 6]])>)]
```

Now the dataset contains sequences of length 4 as inputs, and the targets are sequences containing the next 2 steps, for each time step. For example, the first input sequence is [0, 1, 2, 3], and its corresponding targets are [[1, 2], [2, 3], [3, 4], [4, 5]], which are the next two values for each time step. If you're like me, you will probably need a bit of time to wrap your head around this. Take your time!



It may be surprising that the targets contain values that appear in the inputs. Isn't that cheating? Fortunately, not at all: at each time step, an RNN only knows about past time steps, it cannot look ahead. It is said to be a *causal* model.

Let's create another little utility function to prepare the datasets for our sequence-to-sequence model. It will also take care of shuffling (optional) and batching:

```
def to_seq2seq_dataset(series, seq_length=56, ahead=14, target_col=1,
                      batch_size=32, shuffle=False, seed=None):
    ds = to_windows(tf.data.Dataset.from_tensor_slices(series), ahead + 1)
    ds = to_windows(ds, seq_length).map(lambda S: (S[:, 0], S[:, 1:, 1]))
    if shuffle:
```

```
    ds = ds.shuffle(8 * batch_size, seed=seed)
    return ds.batch(batch_size)
```

Now we can use this function to create the datasets:

```
seq2seq_train = to_seq2seq_dataset(mulvar_train, shuffle=True, seed=42)
seq2seq_valid = to_seq2seq_dataset(mulvar_valid)
```

And lastly, we can build the sequence-to-sequence model:

```
seq2seq_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, return_sequences=True, input_shape=[None, 5]),
    tf.keras.layers.Dense(14)
])
```

It is almost identical to our previous model: the only difference is that we set `return_sequences=True` in the `SimpleRNN` layer. This way, it will output a sequence of vectors (each of size 32), instead of outputting a single vector at the last time step. The `Dense` layer is smart enough to handle sequences as input: it will be applied at each time step, taking a 32-dimensional vector as input, and outputting a 14-dimensional vector. In fact, another way to get the exact same result is to use a `Conv1D` layer with a kernel size of 1: `Conv1D(14, kernel_size=1)`.



Keras offers a `TimeDistributed` layer which lets you apply any vector-to-vector layer to every vector in the input sequences, at every time step. It does this efficiently, by reshaping the inputs so that each time step is treated as a separate instance, then it reshapes the layer's outputs to recover the time dimension. In our case, we don't need it since the `Dense` layer already supports sequences as inputs.

The training code is the same as usual. During training, all the model's outputs are used, but after training only the output of the very last time step matters, and the rest can be ignored. For example, we can forecast the rail ridership for the next 14 days like this:

```
X = mulvar_valid.to_numpy()[:, :seq_length]
y_pred_14 = seq2seq_model.predict(X)[0, -1] # only the last time step's output
```

If you evaluate this model's forecasts for $t + 1$, you will find a validation MAE of 25,519. For $t + 2$, it's 26,274. And the performance continues to drop gradually as the model tries to forecast further into the future. At $t + 14$, the MAE is 34,322.



You can combine both approaches to forecasting multiple steps ahead: for example, you can train a model that forecasts 14 days ahead, then take its output and append it to the inputs, then run the model again to get forecasts for the following 14 days, and possibly repeat the process.

Simple RNNs can be quite good at forecasting time series or handling other kinds of sequences, but they do not perform as well on long time series or sequences. Let's discuss why and see what we can do about it.

Handling Long Sequences

To train an RNN on long sequences, we must run it over many time steps, making the unrolled RNN a very deep network. Just like any deep neural network it may suffer from the unstable gradients problem, discussed in [Chapter 11](#): it may take forever to train, or training may be unstable. Moreover, when an RNN processes a long sequence, it will gradually forget the first inputs in the sequence. Let's look at both these problems, starting with the unstable gradients problem.

Fighting the Unstable Gradients Problem

Many of the tricks we used in deep nets to alleviate the unstable gradients problem can also be used for RNNs: good parameter initialization, faster optimizers, dropout, and so on. However, nonsaturating activation functions (e.g., ReLU) may not help as much here. In fact, they may actually lead the RNN to be even more unstable during training.

Why? Well, suppose Gradient Descent updates the weights in a way that increases the outputs slightly at the first time step. Because the same weights are used at every time step, the outputs at the second time step may also be slightly increased, and those at the third, and so on until the outputs explode—and a nonsaturating activation function does not prevent that.

You can reduce this risk by using a smaller learning rate. Or you can simply use a saturating activation function like the hyperbolic tangent (this explains why it is the default).

In much the same way, the gradients themselves can explode. If you notice that training is unstable, you may want to monitor the size of the gradients (e.g., using TensorBoard) and perhaps use Gradient Clipping.

Moreover, Batch Normalization cannot be used as efficiently with RNNs as with deep feedforward nets. In fact, you cannot use it between time steps, only between recurrent layers. To be more precise, it is technically possible to add a BN layer to a memory cell (as we will see shortly) so that it will be applied at each time step (both on the inputs for that time step and on the hidden state from the previous step). However, the same BN layer will be used at each time step, with the same parameters, regardless of the actual scale and offset of the inputs and hidden state. In practice, this does not yield good results, as was demonstrated by César Laurent et al. in a [2015](#)

[paper](#):⁷ the authors found that BN was slightly beneficial only when it was applied to the layer's inputs, not to the hidden states. In other words, it was slightly better than nothing when applied between recurrent layers (i.e., vertically in [Figure 15-10](#)), but not within recurrent layers (i.e., horizontally). In Keras, you can apply BN between layers simply by adding a `BatchNormalization` layer before each recurrent layer, but it will slow down training, and it may not help much.

Another form of normalization often works better with RNNs: *Layer Normalization*. This idea was introduced by Jimmy Lei Ba et al. in a [2016 paper](#):⁸ it is very similar to Batch Normalization, but instead of normalizing across the batch dimension, Layer Normalization normalizes across the features dimension. One advantage is that it can compute the required statistics on the fly, at each time step, independently for each instance. This also means that it behaves the same way during training and testing (as opposed to BN), and it does not need to use exponential moving averages to estimate the feature statistics across all instances in the training set, like BN does. Like BN, Layer Normalization learns a scale and an offset parameter for each input. In an RNN, it is typically used right after the linear combination of the inputs and the hidden states.

Let's use Keras to implement Layer Normalization within a simple memory cell. To do this, we need to define a custom memory cell, which is just like a regular layer, except its `call()` method takes two arguments: the `inputs` at the current time step and the hidden `states` from the previous time step. Note that the `states` argument is a list containing one or more tensors. In the case of a simple RNN cell it contains a single tensor equal to the outputs of the previous time step, but other cells may have multiple state tensors (e.g., an `LSTMCell` has a long-term state and a short-term state, as we will see shortly). A cell must also have a `state_size` attribute and an `output_size` attribute. In a simple RNN, both are simply equal to the number of units. The following code implements a custom memory cell that will behave like a `SimpleRNNCell`, except it will also apply Layer Normalization at each time step:

```
class LNSimpleRNNCell(tf.keras.layers.Layer):
    def __init__(self, units, activation="tanh", **kwargs):
        super().__init__(**kwargs)
        self.state_size = units
        self.output_size = units
        self.simple_rnn_cell = tf.keras.layers.SimpleRNNCell(units,
                                                               activation=None)
        self.layer_norm = tf.keras.layers.LayerNormalization()
        self.activation = tf.keras.activations.get(activation)
```

⁷ César Laurent et al., "Batch Normalized Recurrent Neural Networks," *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing* (2016): 2657–2661.

⁸ Jimmy Lei Ba et al., "Layer Normalization," arXiv preprint arXiv:1607.06450 (2016).

```
def call(self, inputs, states):
    outputs, new_states = self.simple_rnn_cell(inputs, states)
    norm_outputs = self.activation(self.layer_norm(outputs))
    return norm_outputs, [norm_outputs]
```

The walk through this code:

- Our `LNSimpleRNNCell` class inherits from the `tf.keras.layers.Layer` class, just like any custom layer.
- The constructor takes the number of units and the desired activation function, and it sets the `state_size` and `output_size` attributes, then creates a `SimpleRNNCell` with no activation function (because we want to perform Layer Normalization after the linear operation but before the activation function).⁹ Then the constructor creates the `LayerNormalization` layer, and finally it fetches the desired activation function.
- The `call()` method starts by applying the simple RNN cell, which computes a linear combination of the current inputs and the previous hidden states, and it returns the result twice (indeed, in a `SimpleRNNCell`, the outputs are just equal to the hidden states: in other words, `new_states[0]` is equal to `outputs`, so we can safely ignore `new_states` in the rest of the `call()` method). Next, the `call()` method applies Layer Normalization, followed by the activation function. Finally, it returns the outputs twice: once as the outputs, and once as the new hidden states. To use this custom cell, all we need to do is create a `tf.keras.layers.RNN` layer, passing it a cell instance:

```
custom_ln_model = tf.keras.Sequential([
    tf.keras.layers.RNN(LNSimpleRNNCell(32), return_sequences=True,
                        input_shape=[None, 5]),
    tf.keras.layers.Dense(14)
])
```

Similarly, you could create a custom cell to apply dropout between each time step. But there's a simpler way: most recurrent layers and cells provided by Keras have `dropout` and `recurrent_dropout` hyperparameters: the former defines the dropout rate to apply to the inputs, and the latter defines the dropout rate for the hidden states, between time steps. No need to create a custom cell to apply dropout at each time step in an RNN.

⁹ It would have been simpler to inherit from `SimpleRNNCell` instead so that we wouldn't have to create an internal `SimpleRNNCell` or handle the `state_size` and `output_size` attributes, but the goal here was to show how to create a custom cell from scratch.



When forecasting time series, it is often useful to have some error bars along with your predictions. For this, one approach is to use MC Dropout, introduced in [Chapter 11](#): use `recurrent_dropout` during training, then keep dropout active at inference time by calling the model using `model(x, training=True)`. Repeat this several times to get multiple slightly different forecasts, then compute the mean and standard deviation of these predictions, for each time step.

With these techniques, you can alleviate the unstable gradients problem and train an RNN much more efficiently. Now let's look at how to deal with the short-term memory problem.

Tackling the Short-Term Memory Problem

Due to the transformations that the data goes through when traversing an RNN, some information is lost at each time step. After a while, the RNN's state contains virtually no trace of the first inputs. This can be a showstopper. Imagine Dory the fish¹⁰ trying to translate a long sentence; by the time she's finished reading it, she has no clue how it started. To tackle this problem, various types of cells with long-term memory have been introduced. They have proven so successful that the basic cells are not used much anymore. Let's first look at the most popular of these long-term memory cells: the LSTM cell.

LSTM cells

The *Long Short-Term Memory* (LSTM) cell was proposed in 1997¹¹ by Sepp Hochreiter and Jürgen Schmidhuber and gradually improved over the years by several researchers, such as [Alex Graves](#), [Haşim Sak](#),¹² and [Wojciech Zaremba](#).¹³ If you consider the LSTM cell as a black box, it can be used very much like a basic cell, except it will perform much better; training will converge faster, and it will detect longer-term patterns in the data. In Keras, you can simply use the `LSTM` layer instead of the `SimpleRNN` layer:

```
model = tf.keras.Sequential([
    tf.keras.layers.LSTM(32, return_sequences=True, input_shape=[None, 5]),
```

¹⁰ A character from the animated movies *Finding Nemo* and *Finding Dory* who has short-term memory loss.

¹¹ Sepp Hochreiter and Jürgen Schmidhuber, “Long Short-Term Memory,” *Neural Computation* 9, no. 8 (1997): 1735–1780.

¹² Haşim Sak et al., “Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition,” arXiv preprint arXiv:1402.1128 (2014).

¹³ Wojciech Zaremba et al., “Recurrent Neural Network Regularization,” arXiv preprint arXiv:1409.2329 (2014).

```

    tf.keras.layers.Dense(14)
])

```

Alternatively, you could use the general-purpose `tf.keras.layers.RNN` layer, giving it an `LSTMCell` as an argument. However, the LSTM layer uses an optimized implementation when running on a GPU (see [Chapter 19](#)), so in general it is preferable to use it (the RNN layer is mostly useful when you define custom cells, as we did earlier).

So how does an LSTM cell work? Its architecture is shown in [Figure 15-12](#). If you don't look at what's inside the box, the LSTM cell looks exactly like a regular cell, except that its state is split into two vectors: $\mathbf{h}_{(t)}$ and $\mathbf{c}_{(t)}$ ("c" stands for "cell"). You can think of $\mathbf{h}_{(t)}$ as the short-term state and $\mathbf{c}_{(t)}$ as the long-term state.

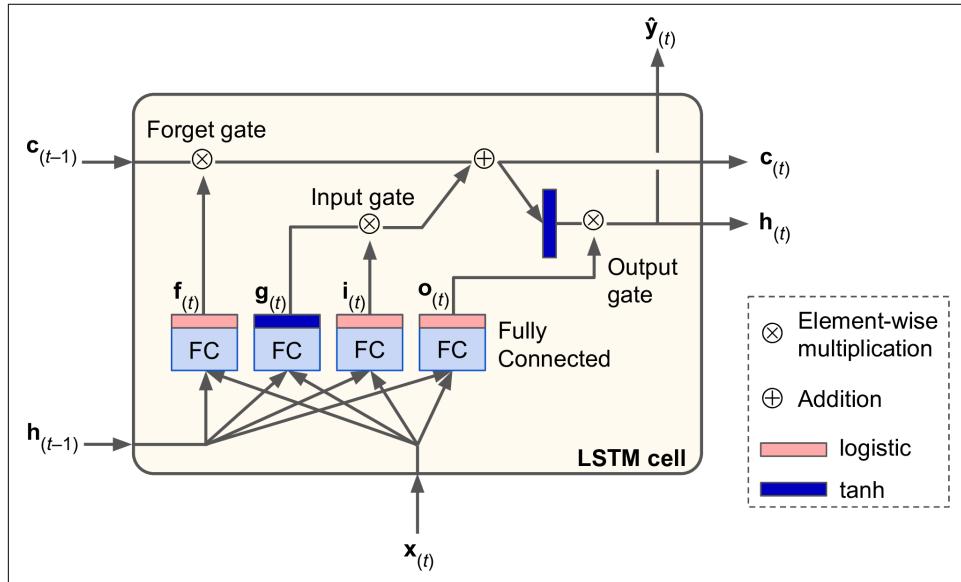


Figure 15-12. LSTM cell

Now let's open the box! The key idea is that the network can learn what to store in the long-term state, what to throw away, and what to read from it. As the long-term state $\mathbf{c}_{(t-1)}$ traverses the network from left to right, you can see that it first goes through a *forget gate*, dropping some memories, and then it adds some new memories via the addition operation (which adds the memories that were selected by an *input gate*). The result $\mathbf{c}_{(t)}$ is sent straight out, without any further transformation. So, at each time step, some memories are dropped and some memories are added. Moreover, after the addition operation, the long-term state is copied and passed through the tanh function, and then the result is filtered by the *output gate*. This produces the short-term state $\mathbf{h}_{(t)}$ (which is equal to the cell's output for this time step, $\hat{\mathbf{y}}_{(t)}$). Now let's look at where new memories come from and how the gates work.

First, the current input vector $\mathbf{x}_{(t)}$ and the previous short-term state $\mathbf{h}_{(t-1)}$ are fed to four different fully connected layers. They all serve a different purpose:

- The main layer is the one that outputs $\mathbf{g}_{(t)}$. It has the usual role of analyzing the current inputs $\mathbf{x}_{(t)}$ and the previous (short-term) state $\mathbf{h}_{(t-1)}$. In a basic cell, there is nothing other than this layer, and its output goes straight out to $\mathbf{y}_{(t)}$ and $\mathbf{h}_{(t)}$. In contrast, in an LSTM cell this layer's output does not go straight out, but instead its most important parts are stored in the long-term state (and the rest is dropped).
- The three other layers are *gate controllers*. Since they use the logistic activation function, the outputs range from 0 to 1. As you can see, the gate controllers' outputs are fed to element-wise multiplication operations: if they output 0s they close the gate, and if they output 1s they open it. Specifically:
 - The *forget gate* (controlled by $\mathbf{f}_{(t)}$) controls which parts of the long-term state should be erased.
 - The *input gate* (controlled by $\mathbf{i}_{(t)}$) controls which parts of $\mathbf{g}_{(t)}$ should be added to the long-term state.
 - Finally, the *output gate* (controlled by $\mathbf{o}_{(t)}$) controls which parts of the long-term state should be read and output at this time step, both to $\mathbf{h}_{(t)}$ and to $\mathbf{y}_{(t)}$.

In short, an LSTM cell can learn to recognize an important input (that's the role of the input gate), store it in the long-term state, preserve it for as long as it is needed (that's the role of the forget gate), and extract it whenever it is needed. This explains why these cells have been amazingly successful at capturing long-term patterns in time series, long texts, audio recordings, and more.

Equation 15-4 summarizes how to compute the cell's long-term state, its short-term state, and its output at each time step for a single instance (the equations for a whole mini-batch are very similar).

Equation 15-4. LSTM computations

$$\begin{aligned}\mathbf{i}_{(t)} &= \sigma(\mathbf{W}_{xi}^T \mathbf{x}_{(t)} + \mathbf{W}_{hi}^T \mathbf{h}_{(t-1)} + \mathbf{b}_i) \\ \mathbf{f}_{(t)} &= \sigma(\mathbf{W}_{xf}^T \mathbf{x}_{(t)} + \mathbf{W}_{hf}^T \mathbf{h}_{(t-1)} + \mathbf{b}_f) \\ \mathbf{o}_{(t)} &= \sigma(\mathbf{W}_{xo}^T \mathbf{x}_{(t)} + \mathbf{W}_{ho}^T \mathbf{h}_{(t-1)} + \mathbf{b}_o) \\ \mathbf{g}_{(t)} &= \tanh(\mathbf{W}_{xg}^T \mathbf{x}_{(t)} + \mathbf{W}_{hg}^T \mathbf{h}_{(t-1)} + \mathbf{b}_g) \\ \mathbf{c}_{(t)} &= \mathbf{f}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{i}_{(t)} \otimes \mathbf{g}_{(t)} \\ \mathbf{y}_{(t)} &= \mathbf{h}_{(t)} = \mathbf{o}_{(t)} \otimes \tanh(\mathbf{c}_{(t)})\end{aligned}$$

In this equation:

- \mathbf{W}_{xi} , \mathbf{W}_{xf} , \mathbf{W}_{xo} , \mathbf{W}_{xg} are the weight matrices of each of the four layers for their connection to the input vector $\mathbf{x}_{(t)}$.
- \mathbf{W}_{hi} , \mathbf{W}_{hf} , \mathbf{W}_{ho} , and \mathbf{W}_{hg} are the weight matrices of each of the four layers for their connection to the previous short-term state $\mathbf{h}_{(t-1)}$.
- \mathbf{b}_i , \mathbf{b}_f , \mathbf{b}_o , and \mathbf{b}_g are the bias terms for each of the four layers. Note that TensorFlow initializes \mathbf{b}_f to a vector full of 1s instead of 0s. This prevents forgetting everything at the beginning of training.

There are several variants of the LSTM cell. One particularly popular variant is the GRU cell, which we will look at now.

GRU cells

The *Gated Recurrent Unit* (GRU) cell (see Figure 15-13) was proposed by Kyunghyun Cho et al. in a 2014 paper¹⁴ that also introduced the encoder–decoder network we discussed earlier.

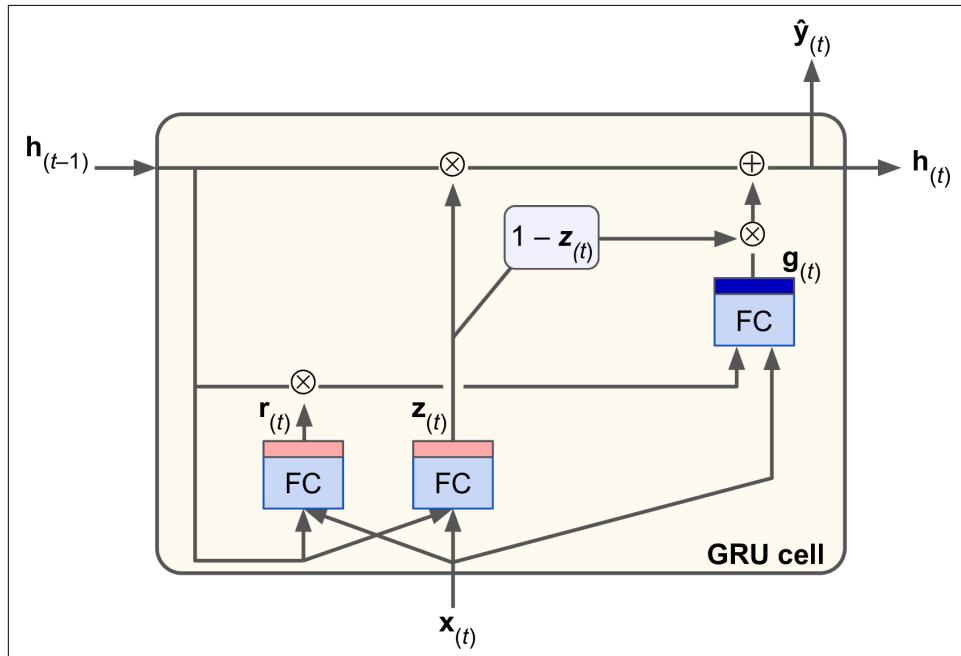


Figure 15-13. GRU cell

¹⁴ Kyunghyun Cho et al., “Learning Phrase Representations Using RNN Encoder–Decoder for Statistical Machine Translation,” *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing* (2014): 1724–1734.

The GRU cell is a simplified version of the LSTM cell, and it seems to perform just as well¹⁵ (which explains its growing popularity). These are the main simplifications:

- Both state vectors are merged into a single vector $\mathbf{h}_{(t)}$.
- A single gate controller $\mathbf{z}_{(t)}$ controls both the forget gate and the input gate. If the gate controller outputs a 1, the forget gate is open ($= 1$) and the input gate is closed ($1 - 1 = 0$). If it outputs a 0, the opposite happens. In other words, whenever a memory must be stored, the location where it will be stored is erased first. This is actually a frequent variant to the LSTM cell in and of itself.
- There is no output gate; the full state vector is output at every time step. However, there is a new gate controller $\mathbf{r}_{(t)}$ that controls which part of the previous state will be shown to the main layer ($\mathbf{g}_{(t)}$).

[Equation 15-5](#) summarizes how to compute the cell's state at each time step for a single instance.

Equation 15-5. GRU computations

$$\begin{aligned}\mathbf{z}_{(t)} &= \sigma(\mathbf{W}_{xz}^T \mathbf{x}_{(t)} + \mathbf{W}_{hz}^T \mathbf{h}_{(t-1)} + \mathbf{b}_z) \\ \mathbf{r}_{(t)} &= \sigma(\mathbf{W}_{xr}^T \mathbf{x}_{(t)} + \mathbf{W}_{hr}^T \mathbf{h}_{(t-1)} + \mathbf{b}_r) \\ \mathbf{g}_{(t)} &= \tanh(\mathbf{W}_{xg}^T \mathbf{x}_{(t)} + \mathbf{W}_{hg}^T (\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)}) + \mathbf{b}_g) \\ \mathbf{h}_{(t)} &= \mathbf{z}_{(t)} \otimes \mathbf{h}_{(t-1)} + (1 - \mathbf{z}_{(t)}) \otimes \mathbf{g}_{(t)}\end{aligned}$$

Keras provides a `tf.keras.layers.GRU` layer: using it is just a matter of replacing `SimpleRNN` or `LSTM` with `GRU`. It also provides a `tf.keras.layers.GRUCell`, in case you want to create a custom cell based on a GRU cell.

LSTM and GRU cells are one of the main reasons behind the success of RNNs. Yet while they can tackle much longer sequences than simple RNNs, they still have a fairly limited short-term memory, and they have a hard time learning long-term patterns in sequences of 100 time steps or more, such as audio samples, long time series, or long sentences. One way to solve this is to shorten the input sequences, for example using 1D convolutional layers.

Using 1D convolutional layers to process sequences

In [Chapter 14](#), we saw that a 2D convolutional layer works by sliding several fairly small kernels (or filters) across an image, producing multiple 2D feature maps

¹⁵ A 2015 paper by Klaus Greff et al., “[LSTM: A Search Space Odyssey](#)”, seems to show that all LSTM variants perform roughly the same.

(one per kernel). Similarly, a 1D convolutional layer slides several kernels across a sequence, producing a 1D feature map per kernel. Each kernel will learn to detect a single very short sequential pattern (no longer than the kernel size). If you use 10 kernels, then the layer's output will be composed of 10 1-dimensional sequences (all of the same length), or equivalently you can view this output as a single 10-dimensional sequence. This means that you can build a neural network composed of a mix of recurrent layers and 1D convolutional layers (or even 1D pooling layers). If you use a 1D convolutional layer with a stride of 1 and "same" padding, then the output sequence will have the same length as the input sequence. But if you use "valid" padding or a stride greater than 1, then the output sequence will be shorter than the input sequence, so make sure you adjust the targets accordingly.

For example, the following model is the same as earlier, except it starts with a 1D convolutional layer that downsamples the input sequence by a factor of 2, using a stride of 2. The kernel size is larger than the stride, so all inputs will be used to compute the layer's output, and therefore the model can learn to preserve the useful information, dropping only the unimportant details. By shortening the sequences, the convolutional layer may help the GRU layers detect longer patterns. So we can afford to double the input sequence length to 112 days. Note that we must also crop off the first three time steps in the targets: indeed, the kernel's size is 4, so the first output of the convolutional layer will be based on the input time steps 0 to 3, so the first forecasts will be for time steps 4 to 17 (instead of time steps 1 to 14). Moreover, we must downsample the targets by a factor of 2, because of the stride.

```
conv_rnn_model = tf.keras.Sequential([
    tf.keras.layers.Conv1D(filters=32, kernel_size=4, strides=2,
                          activation="relu", input_shape=[None, 5]),
    tf.keras.layers.GRU(32, return_sequences=True),
    tf.keras.layers.Dense(14)
])

longer_train = to_seq2seq_dataset(mulvar_train, seq_length=112,
                                  shuffle=True, seed=42)
longer_valid = to_seq2seq_dataset(mulvar_valid, seq_length=112)
downsampled_train = longer_train.map(lambda X, Y: (X, Y[:, 3::2]))
downsampled_valid = longer_valid.map(lambda X, Y: (X, Y[:, 3::2]))
[...] # compile and fit the model using the downsampled datasets
```

If you train and evaluate this model, you will find that it outperforms the previous model (by a small margin). In fact, it is actually possible to use only 1D convolutional layers and drop the recurrent layers entirely!

WaveNet

In a [2016 paper](#),¹⁶ Aaron van den Oord and other DeepMind researchers introduced a novel architecture called *WaveNet*. They stacked 1D convolutional layers, doubling the dilation rate (how spread apart each neuron's inputs are) at every layer: the first convolutional layer gets a glimpse of just two time steps at a time, while the next one sees four time steps (its receptive field is four time steps long), the next one sees eight time steps, and so on (see [Figure 15-14](#)). This way, the lower layers learn short-term patterns, while the higher layers learn long-term patterns. Thanks to the doubling dilation rate, the network can process extremely large sequences very efficiently.

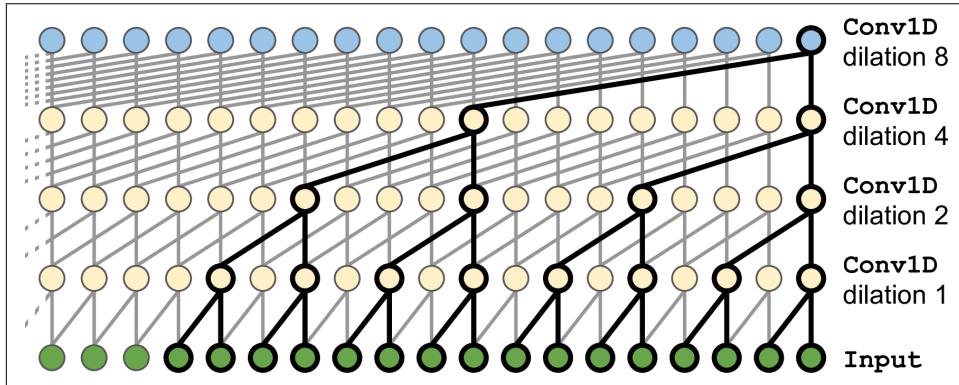


Figure 15-14. WaveNet architecture

The authors of the paper actually stacked 10 convolutional layers with dilation rates of 1, 2, 4, 8, ..., 256, 512, then they stacked another group of 10 identical layers (also with dilation rates 1, 2, 4, 8, ..., 256, 512), then again another identical group of 10 layers. They justified this architecture by pointing out that a single stack of 10 convolutional layers with these dilation rates will act like a super-efficient convolutional layer with a kernel of size 1,024 (except way faster, more powerful, and using significantly fewer parameters), which is why they stacked 3 such blocks. They also left-padded the input sequences with a number of zeros equal to the dilation rate before every layer, to preserve the same sequence length throughout the network.

Here is how to implement a simplified WaveNet to tackle the same sequences as earlier:¹⁷

```
wavenet_model = tf.keras.Sequential()
wavenet_model.add(tf.keras.layers.Input(shape=[None, 5]))
```

¹⁶ Aaron van den Oord et al., “WaveNet: A Generative Model for Raw Audio,” arXiv preprint arXiv:1609.03499 (2016).

¹⁷ The complete WaveNet uses a few more tricks, such as skip connections like in a ResNet, and *Gated Activation Units* similar to those found in a GRU cell. Please see the notebook for more details.

```
for rate in (1, 2, 4, 8) * 2:  
    wavenet_model.add(tf.keras.layers.Conv1D(  
        filters=32, kernel_size=2, padding="causal", activation="relu",  
        dilation_rate=rate))  
wavenet_model.add(tf.keras.layers.Conv1D(filters=14, kernel_size=1))
```

This Sequential model starts with an explicit input layer—this is simpler than trying to set `input_shape` only on the first layer. Then it continues with a 1D convolutional layer using “causal” padding—it is like “same” padding, except the zeros are appended only at the start of the input sequence, instead of both sides. This ensures that the convolutional layer does not peek into the future when making predictions. Then we add similar pairs of layers using growing dilation rates: 1, 2, 4, 8, and again 1, 2, 4, 8. Finally, we add the output layer: a convolutional layer with 14 filters of size 1 and without any activation function. As we saw earlier, such a convolutional layer is equivalent to a `Dense` layer with 14 units. Thanks to the causal padding, every convolutional layer outputs a sequence of the same length as its input sequence, so the targets we use during training can be the full 112-day sequences: no need to crop them or downsample them.

The models we’ve discussed in this section offer similar performance for the ridership forecasting task, but they may vary significantly depending on the task and the amount of available data. In the WaveNet paper, the authors achieved state-of-the-art performance on various audio tasks (hence the name of the architecture), including text-to-speech tasks, producing incredibly realistic voices across several languages. They also used the model to generate music, one audio sample at a time. This feat is all the more impressive when you realize that a single second of audio can contain tens of thousands of time steps—even LSTMs and GRUs cannot handle such long sequences.



If you evaluate our best Chicago ridership models on the test period, starting in 2020, you will find that they perform much worse than expected! Why is that? Well, that’s when the Covid-19 pandemic started, which greatly affected public transportation. As mentioned earlier, these models will only work well if the patterns they learned from the past continue in the future. In any case, before deploying a model to production, verify that it works well on recent data. And once it’s in production, make sure to monitor its performance regularly.

With that, you can now tackle all sorts of time series! In [Chapter 16](#), we will continue to explore RNNs, and we will see how they can tackle various NLP tasks as well.

Exercises

1. Can you think of a few applications for a sequence-to-sequence RNN? What about a sequence-to-vector RNN, and a vector-to-sequence RNN?
2. How many dimensions must the inputs of an RNN layer have? What does each dimension represent? What about its outputs?
3. If you want to build a deep sequence-to-sequence RNN, which RNN layers should have `return_sequences=True`? What about a sequence-to-vector RNN?
4. Suppose you have a daily univariate time series, and you want to forecast the next seven days. Which RNN architecture should you use?
5. What are the main difficulties when training RNNs? How can you handle them?
6. Can you sketch the LSTM cell's architecture?
7. Why would you want to use 1D convolutional layers in an RNN?
8. Which neural network architecture could you use to classify videos?
9. Train a classification model for the SketchRNN dataset, available in TensorFlow Datasets.
10. Download the **Bach chorales** dataset and unzip it. It is composed of 382 chorales composed by Johann Sebastian Bach. Each chorale is 100 to 640 time steps long, and each time step contains 4 integers, where each integer corresponds to a note's index on a piano (except for the value 0, which means that no note is played). Train a model—recurrent, convolutional, or both—that can predict the next time step (four notes), given a sequence of time steps from a chorale. Then use this model to generate Bach-like music, one note at a time: you can do this by giving the model the start of a chorale and asking it to predict the next time step, then appending these time steps to the input sequence and asking the model for the next note, and so on. Also make sure to check out **Google's Coconet model**, which was used for a nice Google doodle about Bach.

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab3>.

Natural Language Processing with RNNs and Attention

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 16th chapter of the final book. The GitHub repo is <https://github.com/ageron/handsonml3>.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

When Alan Turing imagined his famous [Turing test](#)¹ in 1950, he proposed a way to evaluate a machine’s ability to match human intelligence. He could have tested for many things, such as the ability to recognize cats in pictures, play chess, compose music, or escape a maze, but, interestingly, he chose a linguistic task. More specifically, he devised a *chatbot* capable of fooling its interlocutor into thinking it was human.² This test does have its weaknesses: a set of hardcoded rules can fool unsuspecting or naive humans (e.g., the machine could give vague predefined answers in response to some keywords; it could pretend that it is joking or drunk, to get a pass

¹ Alan Turing, “Computing Machinery and Intelligence,” *Mind* 49 (1950): 433–460.

² Of course, the word *chatbot* came much later. Turing called his test the *imitation game*: machine A and human B chat with human interrogator C via text messages; the interrogator asks questions to figure out which one is the machine (A or B). The machine passes the test if it can fool the interrogator, while the human B must try to help the interrogator.

on its weirdest answers; or it could escape difficult questions by answering them with its own questions), and many aspects of human intelligence are utterly ignored (e.g., the ability to interpret nonverbal communication such as facial expressions, or to learn a manual task). But the test does highlight the fact that mastering language is arguably *Homo sapiens*'s greatest cognitive ability.

Can we build a machine that can master written and spoken language? This is the ultimate goal of *Natural Language Processing* (NLP) research, but it's a bit too broad, so in practice researchers focus on more specific tasks, such as text classification, translation, summarization, question answering, and many more.

A common approach for natural language tasks is to use recurrent neural networks. We will therefore continue to explore RNNs (introduced in [Chapter 15](#)), starting with a *character RNN*, trained to predict the next character in a sentence. This will allow us to generate some original text. We will first use a *stateless RNN* (which learns on random portions of text at each iteration, without any information on the rest of the text), then we will build a *stateful RNN* (which preserves the hidden state between training iterations and continues reading where it left off, allowing it to learn longer patterns). Next, we will build an RNN to perform sentiment analysis (e.g., reading movie reviews and extracting the rater's feeling about the movie), this time treating sentences as sequences of words, rather than characters. Then we will show how RNNs can be used to build an encoder–decoder architecture capable of performing neural machine translation (NMT), translating English to Spanish.

In the second part of this chapter, we will explore *attention mechanisms*. As their name suggests, these are neural network components that learn to select the part of the inputs that the rest of the model should focus on at each time step. First, we will boost the performance of an RNN-based encoder–decoder architecture using attention. Then, we will drop RNNs altogether and use a very successful attention-only architecture, called the *Transformer*, to build a translation model. We will then discuss some of the most important advances in NLP in the last few years, including incredibly powerful language models such as GPT and BERT, both based on Transformers. Lastly, we will see how to get started with the excellent Transformers library by Hugging Face.

Let's start with a simple and fun model that can write like Shakespeare (sort of).

Generating Shakespearean Text Using a Character RNN

In a famous [2015 blog post](#) titled “The Unreasonable Effectiveness of Recurrent Neural Networks,” Andrej Karpathy showed how to train an RNN to predict the next character in a sentence. This *Char-RNN* can then be used to generate novel text, one character at a time. Here is a small sample of the text generated by a Char-RNN model after it was trained on all of Shakespeare’s work:

PANDARUS:

Alas, I think he shall be come approached and the day
When little strain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Not exactly a masterpiece, but it is still impressive that the model was able to learn words, grammar, proper punctuation, and more, just by learning to predict the next character in a sentence. This is our first example of a *language model*. Similar (but much more powerful) language models, discussed later in this chapter, are at the core of modern NLP. So let's build a Char-RNN, step by step, starting with the creation of the dataset.

Creating the Training Dataset

First, let's download all of Shakespeare's work, using Keras's handy `tf.keras.utils.get_file()` function. The data is loaded from Andrej Karpathy's [Char-RNN project](#):

```
import tensorflow as tf

shakespeare_url = "https://homl.info/shakespeare" # shortcut URL
filepath = tf.keras.utils.get_file("shakespeare.txt", shakespeare_url)
with open(filepath) as f:
    shakespeare_text = f.read()
```

Let's print the first few lines:

```
>>> print(shakespeare_text[:80])
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.
```

Looks like Shakespeare all right! Next, let's use a `tf.keras.layers.TextVectorization` layer (introduced in [Chapter 13](#)) to encode this text. We set `split="character"` to get character-level encoding rather than the default word-level encoding, and `standardize="lower"` to convert the text to lowercase (which will simplify the task):

```
text_vec_layer = tf.keras.layers.TextVectorization(split="character",
                                                    standardize="lower")
text_vec_layer.adapt([shakespeare_text])
encoded = text_vec_layer([shakespeare_text])[0]
```

Each character is now mapped to an integer, starting at 2. The `TextVectorization` layer reserved value 0 for padding tokens, and it reserved 1 for unknown characters. We won't need either of these tokens for now, so let's subtract 2 from the character

IDs, and compute the number of distinct characters and the total number of characters:

```
encoded -= 2 # drop tokens 0 (pad) and 1 (unknown), which we will not use
n_tokens = text_vec_layer.vocabulary_size() - 2 # number of distinct chars = 39
dataset_size = len(encoded) # total number of chars = 1,115,394
```

Next, just like we did in [Chapter 15](#), we can turn this very long sequence into a dataset of windows which we can then use to train a sequence-to-sequence RNN. The targets will be similar to the inputs, but shifted by one time-step into the “future”. For example, one sample in the dataset may be a sequence of character IDs representing the text “to be or not to b” (without the final “e”),³ and the corresponding target—a sequence of character IDs representing the text “o be or not to be” (with the final “e”, but without the leading “t”). Let’s write a small utility function to convert a long sequence of character IDs into a dataset of input/target window pairs:

```
def to_dataset(sequence, length, shuffle=False, seed=None, batch_size=32):
    ds = tf.data.Dataset.from_tensor_slices(sequence)
    ds = ds.window(length + 1, shift=1, drop_remainder=True)
    ds = ds.flat_map(lambda window_ds: window_ds.batch(length + 1))
    if shuffle:
        ds = ds.shuffle(buffer_size=100_000, seed=seed)
    ds = ds.batch(batch_size)
    return ds.map(lambda window: (window[:, :-1], window[:, 1:])).prefetch(1)
```

This function starts much like the `to_windows()` custom utility function we created in [Chapter 15](#):

- It takes a sequence as input (i.e., the encoded text), and creates a dataset containing all the windows of the desired length.
- It increases the length by one, since we need the next character for the target.
- Then, it shuffles the windows (optionally), batches them, splits them into input/output pairs, and lastly it activates prefetching.

[Figure 16-1](#) summarizes the dataset preparation steps: it shows windows of length 11, and a batch size of 3. The start index of each window is indicated next to it.

³ In this chapter, when discussing input/output strings, I chose to violate the American punctuation rule that requires dots and commas to be inside the quotes, to avoid any confusion. My apologies to the purists.

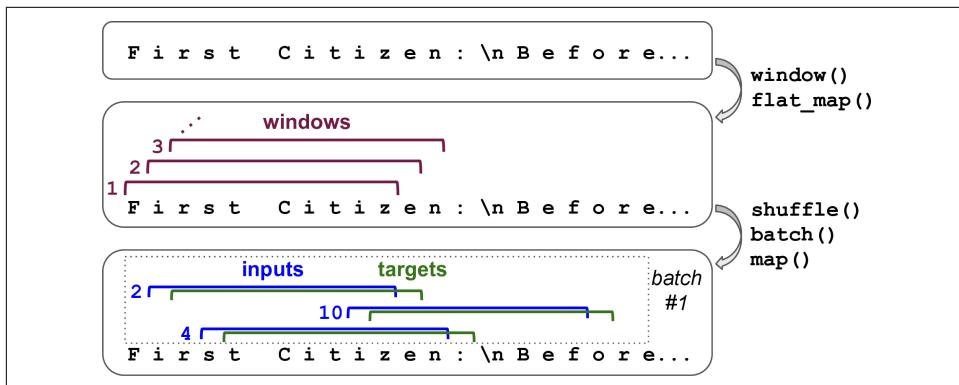


Figure 16-1. Preparing a dataset of shuffled windows

Now we're ready to create the training set, the validation set, and the test set. We will use roughly 90% of the text for training, 5% for validation, and 5% for testing.

```
length = 100
tf.random.set_seed(42)
train_set = to_dataset(encoded[:1_000_000], length=length, shuffle=True,
                      seed=42)
valid_set = to_dataset(encoded[1_000_000:1_060_000], length=length)
test_set = to_dataset(encoded[1_060_000:], length=length)
```



We set the window length to 100, but you can try tuning it: it's easier and faster to train RNNs on shorter input sequences, but the RNN will not be able to learn any pattern longer than `length`, so don't make it too small.

That's it! Preparing the dataset was the hardest part. Now let's create the model.

Building and Training the Char-RNN Model

Since our dataset is reasonably large, and modeling language is quite a difficult task, we need more than a simple RNN with a few recurrent neurons. Let's build and train a model with one GRU layer composed of 128 units (you can try tweaking the number of layers and units later, if needed):

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(input_dim=n_tokens, output_dim=16),
    tf.keras.layers.GRU(128, return_sequences=True),
    tf.keras.layers.Dense(n_tokens, activation="softmax")
])
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam",
               metrics=["accuracy"])
model_ckpt = tf.keras.callbacks.ModelCheckpoint(
```

```
"my_shakespeare_model", monitor="val_accuracy", save_best_only=True)
history = model.fit(train_set, validation_data=valid_set, epochs=10,
 callbacks=[model_ckpt])
```

Let's go over this code:

- We use an `Embedding` layer as the first layer, to encode the character IDs (embeddings were introduced in [Chapter 13](#)). The `Embedding` layer's number of input dimensions is the number of distinct character IDs, and the number of output dimensions is a hyperparameter you can tune—we'll set it to 16 for now. Whereas the inputs of the `Embedding` layer will be 2D tensors of shape $[batch\ size, window\ length]$, the output of the `Embedding` layer will be a 3D tensor of shape $[batch\ size, window\ length, embedding\ size]$.
- We use a `Dense` layer for the output layer: it must have 39 units (`n_tokens`) because there are 39 distinct characters in the text, and we want to output a probability for each possible character (at each time step). The 39 output probabilities should sum up to 1 at each time step, so we apply the softmax activation function to the outputs of the `Dense` layer.
- Lastly, we compile this model, using the `"sparse_categorical_crossentropy"` loss and a Nadam optimizer, and we train the model for several epochs,⁴ using a `ModelCheckpoint` callback to save the best model (in terms of validation accuracy) as training progresses.



If you are running this code on Colab with a GPU activated, then training should take roughly one to two hours. You can reduce the number of epochs if you don't want to wait that long, but of course the model's accuracy will probably be lower. If the Colab session times out, make sure to reconnect quickly, or else the Colab Runtime will be destroyed.

This model does not handle text preprocessing, so let's wrap it in a final model containing the `tf.keras.layers.TextVectorization` layer as the first layer, plus a `tf.keras.layers.Lambda` layer to subtract 2 from the character IDs since we're not using the padding and unknown tokens for now:

```
shakespeare_model = tf.keras.Sequential([
    text_vec_layer,
    tf.keras.layers.Lambda(lambda X: X - 2), # no <PAD> or <UNK> tokens
    model
])
```

⁴ Since the input windows overlap, the concept of epoch is not so clear in this case: during each epoch (as implemented by Keras), the model will actually see the same character multiple times.

And now let's use it to predict the next character in a sentence:

```
>>> y_proba = shakespeare_model.predict(["To be or not to b"])[0, -1]
>>> y_pred = tf.argmax(y_proba) # choose the most probable character ID
>>> text_vec_layer.get_vocabulary()[y_pred + 2]
'e'
```

Great, the model correctly predicted the next character. Now let's use this model to pretend we're Shakespeare!

Generating Fake Shakespearean Text

To generate new text using the Char-RNN model, we could feed it some text, make the model predict the most likely next letter, add it at the end of the text, then give the extended text to the model to guess the next letter, and so on. This is called *greedy decoding*. But in practice this often leads to the same words being repeated over and over again. Instead, we can sample the next character randomly, with a probability equal to the estimated probability, using TensorFlow's `tf.random.categorical()` function. This will generate more diverse and interesting text. The `categorical()` function samples random class indices, given the class log probabilities (logits). For example:

```
>>> log_probas = tf.math.log([[0.5, 0.4, 0.1]]) # probas = 50%, 40%, and 10%
>>> tf.random.set_seed(42)
>>> tf.random.categorical(log_probas, num_samples=8) # draw 8 samples
<tf.Tensor: shape=(1, 8), dtype=int64, numpy=array([[0, 1, 0, 2, 1, 0, 0, 1]])>
```

To have more control over the diversity of the generated text, we can divide the logits by a number called the *temperature*, which we can tweak as we wish. A temperature close to 0 favors high-probability characters, while a high temperature gives all characters an equal probability. Lower temperatures are typically preferred when generating fairly rigid and precise text, such as mathematical equations, while higher temperatures are preferred when generating more diverse and creative text. The following `next_char()` custom helper function uses this approach to pick the next character to add to the input text:

```
def next_char(text, temperature=1):
    y_proba = shakespeare_model.predict([text])[0, -1:]
    rescaled_logits = tf.math.log(y_proba) / temperature
    char_id = tf.random.categorical(rescaled_logits, num_samples=1)[0, 0]
    return text_vec_layer.get_vocabulary()[char_id + 2]
```

Next, we can write another small helper function that will repeatedly call `next_char()` to get the next character and append it to the given text:

```
def extend_text(text, n_chars=50, temperature=1):
    for _ in range(n_chars):
        text += next_char(text, temperature)
    return text
```

We are now ready to generate some text! Let's try with different temperature values:

```
>>> tf.random.set_seed(42)
>>> print(extend_text("To be or not to be", temperature=0.01))
To be or not to be the duke
as it is a proper strange death,
and the
>>> print(extend_text("To be or not to be", temperature=1))
To be or not to behold?

second push:
gremio, lord all, a sistermen,
>>> print(extend_text("To be or not to be", temperature=100))
To be or not to bef ,mt'&o3fpadm!$wh!nse?bws3est--vgerdjw?c-yewzng
```

Shakespeare seems to be suffering from a heatwave. To generate more convincing text, a common technique consists in only sampling from the top k characters, or only from the smallest set of top characters whose total probability exceeds some threshold (this is called *nucleus sampling*). Or you could use *beam search*, which we will discuss later in this chapter. And of course you could also try using more GRU layers and more neurons per layer, train for longer, and add some regularization if needed. Moreover, the model is currently incapable of learning patterns longer than length, which is just 100 characters. You could try making this window larger, but it will also make training harder, and even LSTM and GRU cells cannot handle very long sequences. Alternatively, you could use a stateful RNN.

Stateful RNN

Until now, we have only used *stateless RNNs*: at each training iteration the model starts with a hidden state full of zeros, then it updates this state at each time step, and after the last time step, it throws it away as it is not needed anymore. What if we instructed the RNN to preserve this final state after processing a training batch and use it as the initial state for the next training batch? This way the model could learn long-term patterns despite only backpropagating through short sequences. This is called a *stateful RNN*. Let's go over how to build one.

First, note that a stateful RNN only makes sense if each input sequence in a batch starts exactly where the corresponding sequence in the previous batch left off. So the first thing we need to do to build a stateful RNN is to use sequential and nonoverlapping input sequences (rather than the shuffled and overlapping sequences we used to train stateless RNNs). When creating the `tf.data.Dataset`, we must therefore use `shift=length` (instead of `shift=1`) when calling the `window()` method. Moreover, we must *not* call the `shuffle()` method.

Unfortunately, batching is much harder when preparing a dataset for a stateful RNN than it is for a stateless RNN. Indeed, if we were to call `batch(32)`, then 32

consecutive windows would be put in the same batch, and the following batch would not continue each of these windows where it left off. The first batch would contain windows 1 to 32 and the second batch would contain windows 33 to 64, so if you consider, say, the first window of each batch (i.e., windows 1 and 33), you can see that they are not consecutive. The simplest solution to this problem is to just use a batch size of 1. The following `to_dataset_for_stateful_rnn()` custom utility function uses this strategy to prepare a dataset for a stateful RNN:

```
def to_dataset_for_stateful_rnn(sequence, length):
    ds = tf.data.Dataset.from_tensor_slices(sequence)
    ds = ds.window(length + 1, shift=length, drop_remainder=True)
    ds = ds.flat_map(lambda window: window.batch(length + 1)).batch(1)
    return ds.map(lambda window: (window[:, :-1], window[:, 1:])).prefetch(1)

stateful_train_set = to_dataset_for_stateful_rnn(encoded[:1_000_000], length)
stateful_valid_set = to_dataset_for_stateful_rnn(encoded[1_000_000:1_060_000],
                                                length)
stateful_test_set = to_dataset_for_stateful_rnn(encoded[1_060_000:], length)
```

Figure 16-2 summarizes the main steps of this function.

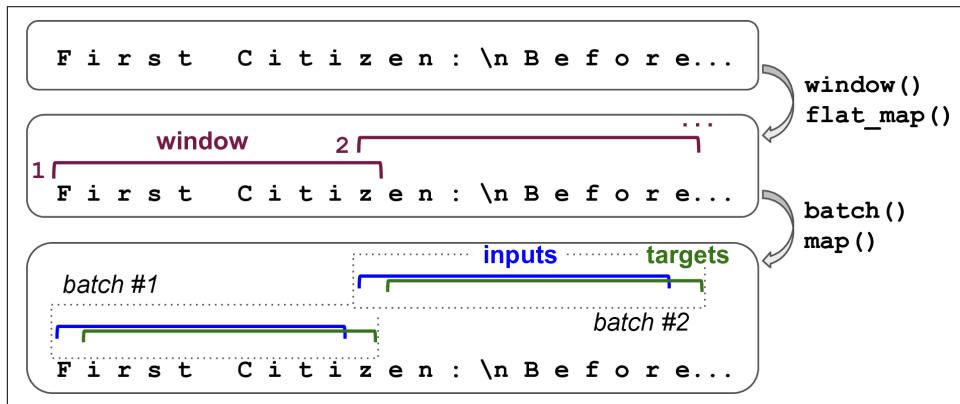


Figure 16-2. Preparing a dataset of consecutive sequence fragments for a stateful RNN

Batching is harder, but it is not impossible. For example, we could chop Shakespeare's text into 32 texts of equal length, create one dataset of consecutive input sequences for each of them, and finally use `tf.data.Dataset.zip(datasets).map(lambda *windows: tf.stack(windows))` to create proper consecutive batches, where the n^{th} input sequence in a batch starts off exactly where the n^{th} input sequence ended in the previous batch (see the notebook for the full code).

Now, let's create the stateful RNN. First, we need to set the `stateful` argument to `True` when creating each recurrent layer. Second, the stateful RNN needs to know the batch size, since it will preserve a state for each input sequence in the batch. Therefore we must set the `batch_input_shape` argument in the first layer. Note that

we can leave the second dimension unspecified, since the input sequences could have any length:

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(input_dim=n_tokens, output_dim=16,
                              batch_input_shape=[1, None]),
    tf.keras.layers.GRU(128, return_sequences=True, stateful=True),
    tf.keras.layers.Dense(n_tokens, activation="softmax")
])
```

At the end of each epoch, we need to reset the states before we go back to the beginning of the text. For this, we can use a small custom Keras callback:

```
class ResetStatesCallback(tf.keras.callbacks.Callback):
    def on_epoch_begin(self, epoch, logs):
        self.model.reset_states()
```

And now we can compile the model and train it using our callback:

```
model.compile(loss="sparse_categorical_crossentropy", optimizer="nadam",
              metrics=["accuracy"])
history = model.fit(stateful_train_set, validation_data=stateful_valid_set,
                     epochs=10, callbacks=[ResetStatesCallback(), model_ckpt])
```



After this model is trained, it will only be possible to use it to make predictions for batches of the same size as were used during training. To avoid this restriction, create an identical *stateless* model, and copy the stateful model's weights to this model.

Interestingly, although a CharRNN model is just trained to predict the next character, this seemingly simple task actually requires it to learn some higher-level tasks as well. For example, to find the next character after “Great movie, I really “, it’s helpful to understand that the sentence is very positive, so what follows is more likely to be the letter “l” (for “loved”) rather than “h” (for “hated”). In fact, in a [2017 paper by OpenAI](#)⁵, the authors trained a big CharRNN-like model on a large dataset, and they found that one of the neurons acted as an excellent sentiment-analysis classifier: although the model was trained without any labels, the *sentiment neuron*—as they called it—reached state-of-the-art performance on sentiment-analysis benchmarks. This foreshadowed and motivated unsupervised pretraining in NLP.

But before we explore unsupervised pretraining, let’s turn our attention to word-level models, and use them in a supervised fashion for sentiment analysis. In the process we will learn how to handle sequences of variable lengths using masking.

⁵ Alec Radford et al., “Learning to Generate Reviews and Discovering Sentiment,” arXiv preprint arXiv:1704.01444 (2017).

Sentiment Analysis

Generating text can be fun and instructive, but in real life projects, one of the most common applications of NLP is text classification—especially sentiment analysis. If image classification on the MNIST dataset is the “Hello world!” of computer vision, then sentiment analysis on the IMDb reviews dataset is the “Hello world!” of natural language processing. The IMDb dataset consists of 50,000 movie reviews in English (25,000 for training, 25,000 for testing) extracted from the famous [Internet Movie Database](#), along with a simple binary target for each review indicating whether it is negative (0) or positive (1). Just like MNIST, the IMDb reviews dataset is popular for good reasons: it is simple enough to be tackled on a laptop in a reasonable amount of time, but challenging enough to be fun and rewarding.

Let’s load the IMDb dataset using the TensorFlow Datasets library (introduced in [Chapter 13](#)). We’ll use the first 90% of the training set for training, and the remaining 10% for validation:

```
import tensorflow_datasets as tfds

raw_train_set, raw_valid_set, raw_test_set = tfds.load(
    name="imdb_reviews",
    split=["train[:90%]", "train[90%:]", "test"],
    as_supervised=True
)
tf.random.set_seed(42)
train_set = raw_train_set.shuffle(5000, seed=42).batch(32).prefetch(1)
valid_set = raw_valid_set.batch(32).prefetch(1)
test_set = raw_test_set.batch(32).prefetch(1)
```



Keras also includes a function for loading the IMDb dataset, if you prefer: `tf.keras.datasets.imdb.load_data()`. The reviews are already preprocessed as sequences of word IDs.

Let’s inspect a few reviews:

```
>>> for review, label in raw_train_set.take(4):
...     print(review.numpy().decode("utf-8"))
...     print("Label:", label.numpy())
...
This was an absolutely terrible movie. Don't be lured in by Christopher [...]
Label: 0
I have been known to fall asleep during films, but this is usually due to [...]
Label: 0
Mann photographs the Alberta Rocky Mountains in a superb fashion, and [...]
Label: 0
This is the kind of film for a snowy Sunday afternoon when the rest of the [...]
Label: 1
```

Some reviews are easy to classify. For example, the first review includes the words “terrible movie” in the very first sentence. But in many cases things are not that simple. For example, the third review starts off positively, even though it’s ultimately a negative review (label 0).

To build a model for this task, we need to preprocess the text, but this time we will chop it into words instead of characters. For this, we can use the `tf.keras.layers.TextVectorization` layer again. Note that it uses spaces to identify word boundaries, which will not work well in some languages. For example, Chinese writing does not use spaces between words, Vietnamese uses spaces even within words, and German often attaches multiple words together, without spaces. Even in English, spaces are not always the best way to tokenize text: think of “San Francisco” or “#ILoveDeepLearning.”

Fortunately, there are solutions to address these issues. In a [2016 paper](#)⁶ by Rico Sennrich et al. from the University of Edinburgh, the authors explored several methods to tokenize and detokenize text at the subword level. This way, even if your model encounters a rare word it has never seen before, it can still reasonably guess what it means. For example, suppose it has never seen the word “smartest” during training, but perhaps it learned the word “smart” and it also learned that the suffix “est” means “the most”, so it can infer the meaning of “smartest”. One of the techniques the authors evaluated is *byte pair encoding* (BPE). BPE works by splitting the whole training set into individual characters (including spaces), then repeatedly merging the most frequent adjacent pairs until the vocabulary reaches the desired size.

A [2018 paper](#)⁷ by Taku Kudo at Google, further improved subword tokenization, often removing the need for language-specific preprocessing prior to tokenization. Moreover, the paper proposed a novel regularization technique called *subword regularization*, which improves accuracy and robustness by introducing some randomness in tokenization during training: for example, “New England” may be tokenized as “New” + “England”, or “New” + “Eng” + “land”, or simply “New England” (just one token). Google’s [SentencePiece](#) project provides an open source implementation, which is described in a [paper](#)⁸ by Taku Kudo and John Richardson.

The TensorFlow team also released the [TensorFlow Text](#) library, which implements various tokenization strategies, including [WordPiece](#)⁹ (a variant of byte pair encod-

⁶ Rico Sennrich et al., “Neural Machine Translation of Rare Words with Subword Units,” *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics 1* (2016): 1715–1725.

⁷ Taku Kudo, “Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates,” arXiv preprint arXiv:1804.10959 (2018).

⁸ Taku Kudo and John Richardson, “SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing,” arXiv preprint arXiv:1808.06226 (2018).

ing). And last but not least, the *tokenizers* library by Hugging Face implements a wide range of extremely fast tokenizers.

However, for the IMDb task in English, using spaces for token boundaries should be good enough. So let's go ahead creating a `TextVectorization` layer and adapting it to the training set. We will limit the vocabulary 1,000 tokens, including the most frequent 998 words, plus a padding token and a token for unknown words, since it's unlikely that very rare words will be important for this task, and limiting the vocabulary size will reduce the number of parameters the model needs to learn:

```
vocab_size = 1000
text_vec_layer = tf.keras.layers.TextVectorization(max_tokens=vocab_size)
text_vec_layer.adapt(train_set.map(lambda reviews, labels: reviews))
```

Finally we can create the model and train it:

```
embed_size = 128
tf.random.set_seed(42)
model = tf.keras.Sequential([
    text_vec_layer,
    tf.keras.layers.Embedding(vocab_size, embed_size),
    tf.keras.layers.GRU(128),
    tf.keras.layers.Dense(1, activation="sigmoid")
])
model.compile(loss="binary_crossentropy", optimizer="nadam",
               metrics=["accuracy"])
history = model.fit(train_set, validation_data=valid_set, epochs=2)
```

The first layer is the `TextVectorization` layer we just prepared, followed by an `Embedding` layer which will convert word IDs into embeddings. The embedding matrix needs to have one row per token in the vocabulary (`vocab_size`) and one column per embedding dimension (this example uses 128 dimensions, but this is a hyperparameter you could tune). Next we use a `GRU` layer, and a `Dense` layer with a single neuron and the sigmoid activation function, since this is a binary classification task: the model's output will be the estimated probability that the review expresses a positive sentiment regarding the movie. We then compile the model, and we fit it on the dataset we prepared earlier, for a couple epochs (or you can train for longer to get better results).

Sadly, if you run this code, you will generally find that the model fails to learn anything at all: the accuracy remains close to 50%, no better than random chance. Why is that? Well the reviews have different lengths, so when the `TextVectorization` layer converts them to sequences of token IDs, it pads the shorter sequences using the padding token (with ID 0) to make them as long as the longest sequence in the batch.

⁹ Yonghui Wu et al., “Google’s Neural Machine Translation System: Bridging the Gap Between Human and Machine Translation,” arXiv preprint arXiv:1609.08144 (2016).

As a result, most sequences end with many padding tokens, often dozens or even hundreds of them. Even though we're using a GRU layer, which is much better than a SimpleRNN layer, its short term memory is still not great. So when it goes through many padding tokens, it ends up forgetting what the review was about! One solution is to feed the model with batches of equal-length sentences (which also speeds up training). Another solution is to make the RNN ignore the padding tokens. This can be done using masking.

Masking

Making our model ignore padding tokens is trivial using Keras: simply add `mask_zero=True` when creating the `Embedding` layer. This means that padding tokens (whose ID is 0) will be ignored by all downstream layers. That's all! If you retrain the model above for a few epochs, you will find that the validation accuracy quickly reaches over 80%.

The way this works is that the `Embedding` layer creates a *mask tensor* equal to `tf.math.not_equal(inputs, 0)`: it is a Boolean tensor with the same shape as the inputs, and it is equal to `False` anywhere the token IDs are 0, or `True` otherwise. This mask tensor is then automatically propagated by the model to the next layer. If that layer's `call()` method has a `mask` argument, then it automatically receives the mask. This allows the layer to ignore the appropriate time steps. Each layer may handle the mask differently, but in general they simply ignore masked time steps (i.e., time steps for which the mask is `False`). For example, when a recurrent layer encounters a masked time step, it simply copies the output from the previous time step.

Next, if the layer's `supports_masking` attribute is `True`, then the mask is automatically propagated to the next layer. It keeps propagating this way for as long as the layers have `support_masking=True`. For example, a recurrent layer's `supports_masking` attribute is `True` when `return_sequences=True`, but it's `False` when `return_sequences=False` since there's no need for a mask anymore in this case. So if you have a model with several recurrent layers with `return_sequences=True`, followed by a recurrent layer with `return_sequences=False`, then the mask will automatically propagate up to the last recurrent layer: that layer will use the mask to ignore masked steps, but it will not propagate the mask any further. Similarly, if you set `mask_zero=True` when creating the `Embedding` layer in the sentiment analysis model we just built, then the GRU layer will receive and use the mask automatically, but it will not propagate it any further, since `return_sequences` is not set to `True`.



Some layers need to update the mask before propagating it to the next layer: they do so by implementing the `compute_mask()` method, which takes two arguments: the inputs and the previous mask. It then computes the updated mask and returns it. The default implementation of `compute_mask()` just returns the previous mask unchanged.

Many Keras layers support masking: `SimpleRNN`, `GRU`, `LSTM`, `Bidirectional`, `TimeDistributed`, `Dense`, `Add`, and a few others (all in the `tf.keras.layers` package). However, convolutional layers (including `Conv1D`) do not support masking—it's not obvious how they would do so anyway.

If the mask propagates all the way to the output, then it gets applied to the losses as well, so the masked time steps will not contribute to the loss (their loss will be 0). This assumes that the model outputs sequences, which is not the case in our sentiment analysis model.



The `LSTM` and `GRU` layers have an optimized implementation for GPUs, based on Nvidia's cuDNN library. However, this implementation only supports masking if all the padding tokens are at the end of the sequences. It also requires you to use the default values for several hyperparameters: `activation`, `recurrent_activation`, `recurrent_dropout`, `unroll`, `use_bias`, and `reset_after`. If that's not the case, then these layers will fall back to the (much slower) default GPU implementation.

If you want to implement your own custom layer with masking support, you should add a `mask` argument to the `call()` method, and obviously make the method use the mask. Additionally, if the mask must be propagated to the next layers, then you should set `self.supports_masking = True` in the constructor. If the mask must be updated before it is propagated, then you must implement the `compute_mask()` method.

If your model does not start with an `Embedding` layer, you may use the `tf.keras.layers.Masking` layer instead: by default, it sets the mask to `tf.math.reduce_any(tf.math.not_equal(x, 0), axis=-1)`, meaning that time steps where the last dimension is full of zeros will be masked out in subsequent layers.

Using masking layers and automatic mask propagation works best for simple models. It will not always work for more complex models, such as when you need to mix `Conv1D` layers with recurrent layers. In such cases, you will need to explicitly compute the mask and pass it to the appropriate layers, using either the Functional API or the Subclassing API. For example, the following model is equivalent to the previous

model, except it is built using the Functional API and handles masking manually. It also adds a bit of dropout since the previous model was overfitting slightly:

```
inputs = tf.keras.layers.Input(shape=[], dtype=tf.string)
token_ids = text_vec_layer(inputs)
mask = tf.math.not_equal(token_ids, 0)
Z = tf.keras.layers.Embedding(vocab_size, embed_size)(token_ids)
Z = tf.keras.layers.GRU(128, dropout=0.2)(Z, mask=mask)
outputs = tf.keras.layers.Dense(1, activation="sigmoid")(Z)
model = tf.keras.Model(inputs=[inputs], outputs=[outputs])
```

One last approach to masking is to feed the model with ragged tensors.¹⁰ In practice, all you need to do is to set `ragged=True` when creating the `TextVectorization` layer, so that the input sequences are represented as ragged tensors:

```
>>> text_vec_layer_ragged = tf.keras.layers.TextVectorization(
...     max_tokens=vocab_size, ragged=True)
...
>>> text_vec_layer_ragged.adapt(train_set.map(lambda reviews, labels: reviews))
>>> text_vec_layer_ragged(["Great movie!", "This is DiCaprio's best role."])
<tf.RaggedTensor [[86, 18], [11, 7, 1, 116, 217]]>
```

Compare this ragged tensor representation with the regular tensor representation, which uses padding tokens:

```
>>> text_vec_layer(["Great movie!", "This is DiCaprio's best role."])
<tf.Tensor: shape=(2, 5), dtype=int64, numpy=
array([[ 86,  18,   0,   0,   0],
       [ 11,    7,   1, 116, 217]])>
```

Keras's recurrent layers have built-in support for ragged tensors, so there's nothing else you need to do: just use this `TextVectorization` layer in your model—that's all! There's no need to pass `mask_zero=True` or handle masks explicitly—it's all implemented for you. That's convenient! However, as of early 2022, the support for ragged tensors in Keras is still fairly recent, so there are a few rough edges. For example, it is currently not possible to use ragged tensors as targets when running on the GPU (but this may be resolved by the time you read these lines).

Whichever masking approach you prefer, after training this model for a few epochs, it will become quite good at judging whether a review is positive or not. If you use the `tf.keras.callbacks.TensorBoard()` callback, you can visualize the embeddings in TensorBoard as they are being learned: it is fascinating to see words like “awesome” and “amazing” gradually cluster on one side of the embedding space, while words like “awful” and “terrible” cluster on the other side. Some words are not as positive as you might expect (at least with this model), such as the word “good”, presumably because many negative reviews contain the phrase “not good”.

¹⁰ Ragged tensors were introduced in [Chapter 12](#), and they are detailed in [Appendix C](#).

Reusing Pretrained Embeddings and Language Models

It's impressive that the model is able to learn useful word embeddings based on just 25,000 movie reviews. Imagine how good the embeddings would be if we had billions of reviews to train on! Unfortunately, we don't, but perhaps we can reuse word embeddings trained on some other (very) large text corpus (e.g., Amazon reviews, available on TensorFlow Datasets), even if it is not composed of movie reviews? After all, the word "amazing" generally has the same meaning whether you use it to talk about movies or anything else. Moreover, perhaps embeddings would be useful for sentiment analysis even if they were trained on another task: since words like "awesome" and "amazing" have a similar meaning, they will likely cluster in the embedding space even for other tasks (e.g., predicting the next word in a sentence). If all positive words and all negative words form clusters, then this will be helpful for sentiment analysis. So, instead of training word embeddings, we could just download and use pretrained embeddings, such as Google's [Word2vec embeddings](#), Stanford's [GloVe embeddings](#), or Facebook's [FastText embeddings](#).

Using pretrained word embeddings was popular for several years, but this approach has its limits. In particular, a word has a single representation, no matter the context. For example, the word "right" is encoded the same way in "left and right" and "right and wrong", even though it means two very different things. To address this limitation, a [2018 paper¹¹](#) by Matthew Peters introduced *Embeddings from Language Models* (ELMo): these are contextualized word embeddings learned from the internal states of a deep bidirectional language model. Instead of just using pretrained embeddings in your model, you reuse part of a pretrained language model.

At roughly the same time, the [Universal Language Model Fine-tuning \(ULMFiT\) paper¹²](#) by Jeremy Howard and Sebastian Ruder demonstrated the effectiveness of unsupervised pretraining for NLP tasks: the authors trained an LSTM language model on a huge text corpus using self-supervised learning (i.e., generating the labels automatically from the data), then they fine-tuned it on various tasks. Their model outperformed the state of the art on six text classification tasks by a large margin (reducing the error rate by 18–24% in most cases). Moreover, the authors showed that by fine-tuning the pretrained model on just 100 labeled examples, the model could achieve the same performance as one trained from scratch on 10,000 examples. Before the ULMFiT paper, using pretrained models was only the norm in computer vision, but in the context of NLP, pretraining was limited to word embeddings. This

¹¹ Matthew Peters et al., "Deep Contextualized Word Representations," *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* 1 (2018): 2227–2237.

¹² Jeremy Howard and Sebastian Ruder, "Universal Language Model Fine-Tuning for Text Classification," *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics* 1 (2018): 328–339.

paper marked the beginning of a new era in NLP: today, reusing pretrained language models is the norm.

For example, let's build a classifier based on the Universal Sentence Encoder, a model architecture introduced in the [2018 *Universal Sentence Encoder* paper](#)¹³ by a team of Google researchers. This model is based on the Transformer architecture, which we will look at later in this chapter. Conveniently, this model is available in TensorFlow Hub:

```
import os
import tensorflow_hub as hub

os.environ["TFHUB_CACHE_DIR"] = "my_tfhub_cache"
model = tf.keras.Sequential([
    hub.KerasLayer("https://tfhub.dev/google/universal-sentence-encoder/4",
                  trainable=True, dtype=tf.string, input_shape=[]),
    tf.keras.layers.Dense(64, activation="relu"),
    tf.keras.layers.Dense(1, activation="sigmoid")
])
model.compile(loss="binary_crossentropy", optimizer="nadam",
               metrics=["accuracy"])
model.fit(train_set, validation_data=valid_set, epochs=10)
```



This model is quite large—close to one gigabyte in size—so it may take a while to download. By default, TensorFlow Hub modules are saved to a temporary directory, and they get downloaded again and again every time you run your program. To avoid that, you must set the `TFHUB_CACHE_DIR` environment variable to a directory of your choice: the modules will then be saved there, and only downloaded once.

Note that the last part of the TensorFlow Hub module URL specifies that we want version 4 of the model. This versioning ensures that if a new module version is released on TF Hub, it will not break our model. Conveniently, if you just enter this URL in a web browser, you will get the documentation for this module.

Also note that we set `trainable=True` when creating the `hub.KerasLayer`. This way, the pretrained Universal Sentence Encoder is fine-tuned during training: some of its weights are tweaked via backprop. Not all TensorFlow Hub modules are fine-tunable, so make sure to check the documentation for each pretrained module you're interested in.

After training, this model should reach a validation accuracy of over 90%. That's actually really good: if you try to perform the task yourself, you will probably do only

¹³ Daniel Cer et al., “Universal Sentence Encoder,” arXiv preprint arXiv:1803.11175 (2018)

marginally better since many reviews contain both positive and negative comments. Classifying these ambiguous reviews is like flipping a coin.

OK, so far we have looked at text generation using Char-RNN, and sentiment analysis with word-level RNN models (based on trainable embeddings) and using a powerful pretrained language model from TensorFlow Hub. In the next section, we will go through another important NLP task: *neural machine translation* (NMT).

An Encoder–Decoder Network for Neural Machine Translation

Let's begin with a simple [neural machine translation \(NMT\) model](#)¹⁴ that will translate English sentences to Spanish (see [Figure 16-3](#)).

In short, the architecture is as follows: English sentences are fed as inputs to the encoder, and the decoder outputs the Spanish translations. Note that the Spanish translations are also used as inputs to the decoder during training, but shifted back by one step. In other words, during training the decoder is given as input the word that it *should* have output at the previous step, regardless of what it actually output. This is called *teacher forcing*—a technique that significantly speeds up training and improves the model's performance. For the very first word, the decoder is given the start-of-sequence (SOS) token, and the decoder is expected to end the sentence with an end-of-sequence (EOS) token.

Each word is initially represented by its ID (e.g., 854 for the word “soccer”). Next, an Embedding layer returns the word embedding. These word embeddings are then fed to the encoder and the decoder.

¹⁴ Ilya Sutskever et al., “Sequence to Sequence Learning with Neural Networks,” arXiv:1409.3215 (2014).

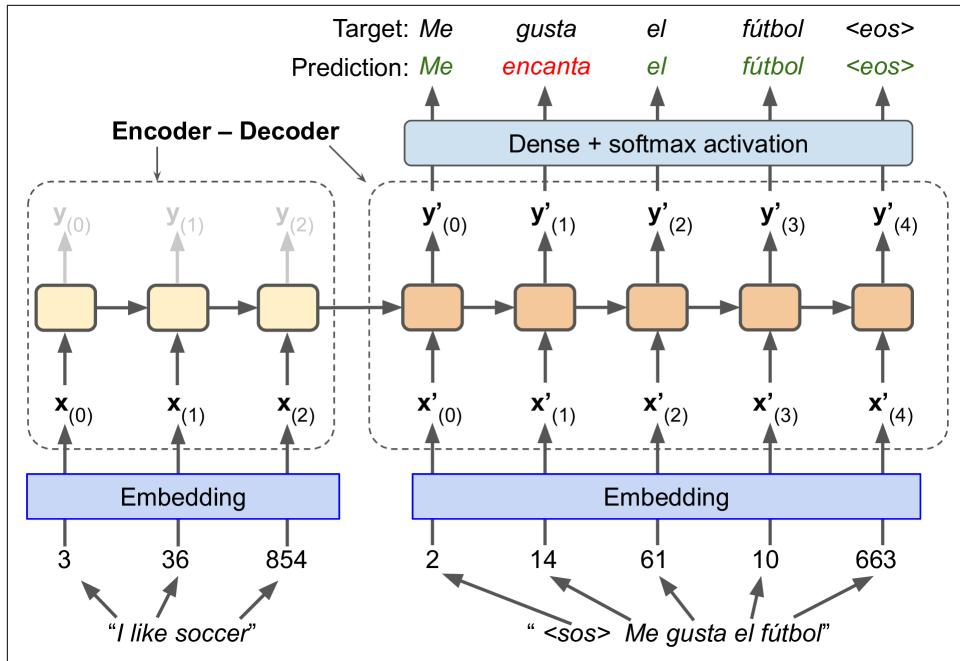


Figure 16-3. A simple machine translation model

At each step, the decoder outputs a score for each word in the output vocabulary (i.e., Spanish), then the softmax activation function turns these scores into probabilities. For example, at the first step the word “Me” may have a probability of 7%, “Yo” may have a probability of 1%, and so on. The word with the highest probability is output. This is very much like a regular classification task, and indeed you can train the model using the “`sparse_categorical_crossentropy`” loss, much like we did in the Char-RNN model.

Note that at inference time (after training), you will not have the target sentence to feed to the decoder. Instead, you need to feed it the word that it has just output at the previous step, as shown in [Figure 16-4](#) (this will require an embedding lookup that is not shown in the diagram).

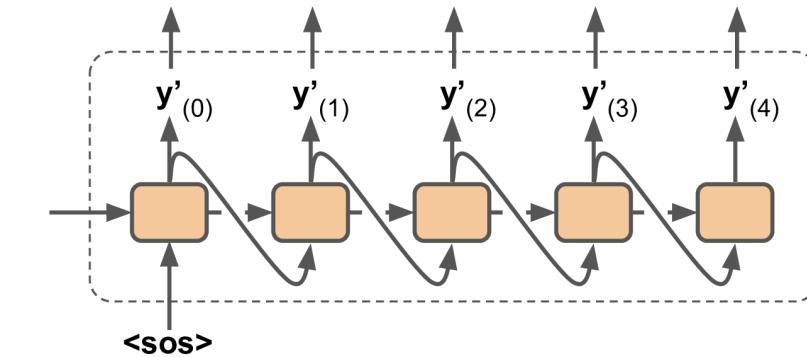


Figure 16-4. At inference time, the decoder is fed as input the word it just output at the previous time step



In a [2015 paper¹⁵](#) by Samy Bengio et al, the authors proposed to gradually switch from feeding the decoder the previous *target* token to feeding it the previous *output* token, during training.

So let's build and train this model! First, we need to download a dataset of English/Spanish sentence pairs:¹⁶

```
url = "https://storage.googleapis.com/download.tensorflow.org/data/spa-eng.zip"
path = tf.keras.utils.get_file("spa-eng.zip", origin=url, cache_dir="datasets",
                               extract=True)
text = (Path(path).with_name("spa-eng") / "spa.txt").read_text()
```

Each line contains an English sentence and the corresponding Spanish translation, separated by a tab. We'll start by removing the Spanish characters “í” and “¿”, which the `TextVectorization` layer doesn't handle, then we will parse the sentence pairs, and shuffle them. Finally, we will split them into two separate lists: one per language.

```
import numpy as np

text = text.replace("í", "").replace("¿", "")
pairs = [line.split("\t") for line in text.splitlines()]
np.random.shuffle(pairs)
sentences_en, sentences_es = zip(*pairs) # separates the pairs into 2 lists
```

¹⁵ Samy Bengio et al., “Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks,” arXiv preprint arXiv:1506.03099 (2015).

¹⁶ This dataset is composed of sentence pairs created by contributors of the [Tatoeba Project](#). About 120,000 sentence pairs were selected by the authors of the website <https://manythings.org/anki>. This dataset is released under the Creative Commons Attribution 2.0 France license. Other language pairs are available.

Let's take a look at the first three sentence pairs:

```
>>> for i in range(3):
...     print(sentences_en[i], "=>", sentences_es[i])
...
How boring! => Qué aburrimiento!
I love sports. => Adoro el deporte.
Would you like to swap jobs? => Te gustaría que intercambiemos los trabajos?
```

Next, let's create two `TextVectorization` layers—one per language—and let's adapt them to the text:

```
vocab_size = 1000
max_length = 50
text_vec_layer_en = tf.keras.layers.TextVectorization(
    vocab_size, output_sequence_length=max_length)
text_vec_layer_es = tf.keras.layers.TextVectorization(
    vocab_size, output_sequence_length=max_length)
text_vec_layer_en.adapt(sentences_en)
text_vec_layer_es.adapt([f"startofseq {s} endofseq" for s in sentences_es])
```

There are a few things to note here:

- We limit the vocabulary size to 1,000, which is quite small. That's because the training set is not very large, and because using a small value will speed up training. State-of-the-art translation models typically use a much larger vocabulary (e.g., 30,000), a much larger training set (Gigabytes), and a much larger model (hundreds or even thousands of megabytes). For example, check out the Opus-MT models by the University of Helsinki, or the M2M-100 model by Facebook.
- Since all sentences in the dataset have a maximum of 50 words, we set `output_sequence_length` to 50: this way the input sequences will automatically be padded with 0s until they are all 50 tokens long. If there was any sentence longer than 50 tokens in the training set, it would be cropped to 50 tokens.
- For the Spanish text, we add “startofseq” and “endofseq” to each sentence when adapting the `TextVectorization` layer: we will use these words as SOS and EOS tokens. You could use any other words, as long as they are not actual Spanish words.

Let's inspect the first 10 tokens in both vocabularies. They start with the padding token, the unknown token, the SOS and EOS tokens (only in the Spanish vocabulary), then the actual words, sorted by decreasing frequency:

```
>>> text_vec_layer_en.get_vocabulary()[:10]
['', '[UNK]', 'the', 'i', 'to', 'you', 'tom', 'a', 'is', 'he']
>>> text_vec_layer_es.get_vocabulary()[:10]
['', '[UNK]', 'startofseq', 'endofseq', 'de', 'que', 'a', 'no', 'tom', 'la']
```

Next, let's create the training set and the validation set (you could also create a test set if you needed it). We will use the first 100,000 sentence pairs for training, and the rest for validation. The decoder's inputs are the Spanish sentences plus an SOS token prefix. The targets are the Spanish sentences plus an EOS suffix.

```
X_train = tf.constant(sentences_en[:100_000])
X_valid = tf.constant(sentences_en[100_000:])
X_train_dec = tf.constant([f"startofseq {s}" for s in sentences_es[:100_000]])
X_valid_dec = tf.constant([f"startofseq {s}" for s in sentences_es[100_000:]])
Y_train = text_vec_layer_es([f"{s} endofseq" for s in sentences_es[:100_000]])
Y_valid = text_vec_layer_es([f"{s} endofseq" for s in sentences_es[100_000:]])
```

OK, we're now ready to build our translation model. We will use the functional API for that since the model is not sequential. Indeed, it requires two text inputs—one for the encoder, and one for the decoder—so let's start with that:

```
encoder_inputs = tf.keras.layers.Input(shape=[], dtype=tf.string)
decoder_inputs = tf.keras.layers.Input(shape=[], dtype=tf.string)
```

Next, we need to encode these sentences using the `TextVectorization` layers we prepared earlier, followed by an `Embedding` layer for each language, with `mask_zero=True` to ensure masking is handled automatically. The embedding size is a hyperparameter you can tune, as always.

```
embed_size = 128
encoder_input_ids = text_vec_layer_en(encoder_inputs)
decoder_input_ids = text_vec_layer_es(decoder_inputs)
encoder_embedding_layer = tf.keras.layers.Embedding(vocab_size, embed_size,
                                                    mask_zero=True)
decoder_embedding_layer = tf.keras.layers.Embedding(vocab_size, embed_size,
                                                    mask_zero=True)
encoder_embeddings = encoder_embedding_layer(encoder_input_ids)
decoder_embeddings = decoder_embedding_layer(decoder_input_ids)
```



When the languages share many words, you may get better performance using the same embedding layer for both the encoder and the decoder.

Now let's create the encoder and pass it the embedded inputs:

```
encoder = tf.keras.layers.LSTM(512, return_state=True)
encoder_outputs, *encoder_state = encoder(encoder_embeddings)
```

To keep things simple, we just used a single LSTM layer, but you could stack several of them. We also set `return_state=True` to get a reference to the layer's final state. Since we're using an LSTM layer, there are actually two states: the short-term state and the long-term state. The layer returns these states separately, which is why we had to

write `*encoder_state` to group both states in a list.¹⁷ Now we can use this (double) state as the initial state of the decoder:

```
decoder = tf.keras.layers.LSTM(512, return_sequences=True)
decoder_outputs = decoder(decoder_embeddings, initial_state=encoder_state)
```

Next we can pass the decoder's outputs through a `Dense` layer with the softmax activation function to get the word probabilities for each step:

```
output_layer = tf.keras.layers.Dense(vocab_size, activation="softmax")
Y_proba = output_layer(decoder_outputs)
```

Optimizing the output layer

When the output vocabulary is large, outputting a probability for each and every possible word can be quite slow. If the target vocabulary contains, say, 50,000 Spanish words instead of 1,000, then the decoder would output 50,000-dimensional vectors, and then computing the softmax function over such a large vector would be very computationally intensive. To avoid this, one solution is to look only at the logits output by the model for the correct word and for a random sample of incorrect words, then compute an approximation of the loss based only on these logits. This *sampled softmax* technique was introduced in 2015 by Sébastien Jean et al.¹⁸ In TensorFlow you can use the `tf.nn.sampled_softmax_loss()` function for this during training and use the normal softmax function at inference time (sampled softmax cannot be used at inference time because it requires knowing the target).

Another thing you can do to speed up training—which is compatible with sampled softmax—is to tie the weights of the output layer to the transpose of the decoder's embedding matrix (we will see how to tie weights in [Chapter 17](#)). This significantly reduces the number of model parameters, which speeds up training and may sometimes improve the model's accuracy as well, especially if you don't have a lot of training data. The embedding matrix is equivalent to one-hot encoding followed by a linear layer with no bias term and no activation function that maps the one-hot vectors to the embedding space. The output layer does the reverse. So if the model can find an embedding matrix whose transpose is close to its inverse (i.e., such a matrix is called an *orthogonal matrix*), then there's no need to learn a separate set of weights for the output layer.

And that's it! We just need to create the Keras Model, compile it, and train it:

¹⁷ In Python, if you run `a, *b = [1, 2, 3, 4]`, then `a` equals 1 and `b` equals `[2, 3, 4]`.

¹⁸ Sébastien Jean et al., “On Using Very Large Target Vocabulary for Neural Machine Translation,” *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing* 1 (2015): 1–10.

```

model = tf.keras.Model(inputs=[encoder_inputs, decoder_inputs],
                       outputs=[Y_proba])
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam",
              metrics=["accuracy"])
model.fit((X_train, X_train_dec), Y_train, epochs=10,
          validation_data=((X_valid, X_valid_dec), Y_valid))

```

After training, we can use the model to translate new English sentences to Spanish. But it's not as simple as calling `model.predict()`, because the decoder expects as input the word that was predicted at the previous time step. One way to do this is to write a custom memory cell that keeps track of the previous output and feeds it to the encoder at the next time step. However, to keep things simple, we can just call the model multiple times, predicting one extra word at each round. Let's write a little utility function for that:

```

def translate(sentence_en):
    translation = ""
    for word_idx in range(max_length):
        X = np.array([sentence_en]) # encoder input
        X_dec = np.array(["startofseq " + translation]) # decoder input
        y_proba = model.predict((X, X_dec))[0, word_idx] # last token's probas
        predicted_word_id = np.argmax(y_proba)
        predicted_word = text_vec_layer_es.get_vocabulary()[predicted_word_id]
        if predicted_word == "endofseq":
            break
        translation += " " + predicted_word
    return translation.strip()

```

The function simply keeps predicting one word at a time, gradually completing the translation, and it stops once it reaches the EOS token. Let's give it a try!

```

>>> translate("I like soccer")
'me gusta el fútbol'

```

Hurray, it works! Well, at least it does with very short sentences. If you try playing with this model for a while, you will find that it's not bilingual yet, and in particular it really struggles with longer sentences. For example:

```

>>> translate("I like soccer and also going to the beach")
'me gusta el fútbol y a veces mismo al bus'

```

The translation says “I like soccer and sometimes even the bus”. So how can you improve it? One way is to increase the training set size and add more LSTM layers in both the encoder and the decoder. But this will only get you so far, so let's look at more sophisticated techniques, starting with *bidirectional recurrent layers*.

Bidirectional RNNs

At each time step, a regular recurrent layer only looks at past and present inputs before generating its output. In other words, it is *causal*, meaning it cannot look

into the future. This type of RNN makes sense when forecasting time series, or in the decoder of a sequence-to-sequence (seq2seq) model. But for tasks like text classification, or in the encoder of a seq2seq model, it is often preferable to look ahead at the next words before encoding a given word.

For example, consider the phrases “the right arm”, “the right person”, and “the right to criticize”: to properly encode the word “right”, you need to look ahead. One solution is to run two recurrent layers on the same inputs, one reading the words from left to right and the other reading them from right to left. Then combine their outputs at each time step, typically by concatenating them. This is what a *bidirectional recurrent layer* does (see [Figure 16-5](#)).

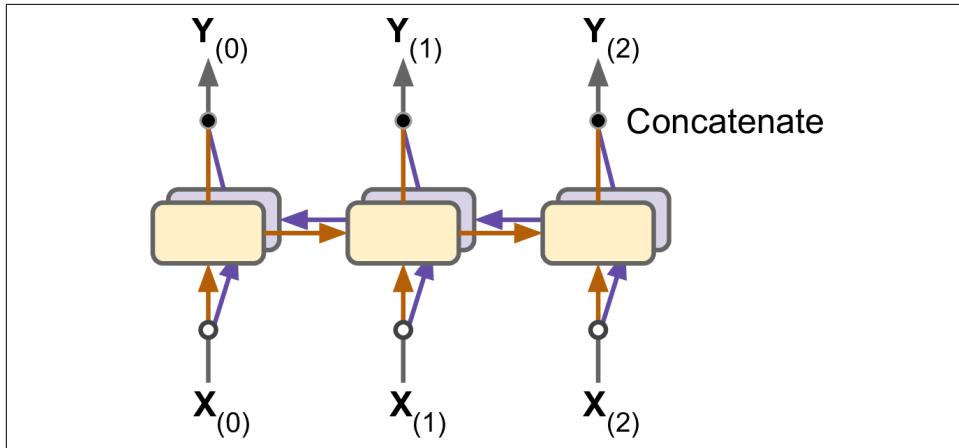


Figure 16-5. A bidirectional recurrent layer

To implement a bidirectional recurrent layer in Keras, just wrap a recurrent layer in a `tf.keras.layers.Bidirectional` layer. For example, the following bidirectional layer could be used as the encoder in our translation model:

```
encoder = tf.keras.layers.Bidirectional(  
    tf.keras.layers.LSTM(256, return_state=True))
```



The `Bidirectional` layer will create a clone of the `GRU` layer (but in the reverse direction), and it will run both and concatenate their outputs. So although the `GRU` layer has 10 units, the `Bidirectional` layer will output 20 values per time step.

There's just one problem. This layer will now return four states instead of two: the final short-term and long-term states of the forward `LSTM` layer, and the final short-term and long-term states of the backward `LSTM` layer. We cannot use this quadruple state directly as the initial state of the decoder's `LSTM` layer, since it expects just two

states (short-term and long-term). We cannot make the decoder bidirectional, since it must remain causal: otherwise it would cheat during training and it would not work. Instead, we can concatenate the two short-term states, and also concatenate the two long-term states:

```
encoder_outputs, *encoder_state = encoder(encoder_embeddings)
encoder_state = [tf.concat(encoder_state[::2], axis=-1), # short-term (0 & 2)
                 tf.concat(encoder_state[1::2], axis=-1)] # long-term (1 & 3)
```

Now let's look at another popular technique that can greatly improve the performance of a translation model at inference time: beam search.

Beam Search

Suppose you have trained an encoder–decoder model, and you use it to translate the sentence “I like soccer” to Spanish. You are hoping that it will output the proper translation “me gusta el fútbol”, but unfortunately it outputs “me gustan los jugadores”, which means “I like the players”. Looking at the training set, you notice many sentences such as “I like cars”, which translates to “me gustan los autos”, so it wasn't absurd for the model to output “me gustan los” after seeing “I like”. Unfortunately, in this case it was a mistake since “soccer” is singular. The model could not go back and fix it, so it tried to complete the sentence as best it could, in this case using the word “jugadores”. How can we give the model a chance to go back and fix mistakes it made earlier? One of the most common solutions is *beam search*: it keeps track of a short list of the k most promising sentences (say, the top three), and at each decoder step it tries to extend them by one word, keeping only the k most likely sentences. The parameter k is called the *beam width*.

For example, suppose you use the model to translate the sentence “I like soccer” using beam search with a beam width of 3 (see [Figure 16-6](#)). At the first decoder step, the model will output an estimated probability for each possible first word in the translated sentence. Suppose the top three words are “me” (75% estimated probability), “a” (3%), and “como” (1%). That's our short list so far. Next, we use the model to find the next word for each sentence. For the first sentence (“me”), perhaps the model outputs a probability of 36% for the word “gustan”, 32% for the word “gusta”, 16% for the word “encanta”, and so on. Note that these are actually *conditional* probabilities, given that the sentence starts with “me”. For the second sentence (“a”), the model might output a conditional probability of 50% for the word “mi”, and so on. Assuming the vocabulary has 1,000 words, we will end up with 1,000 probabilities per sentence.

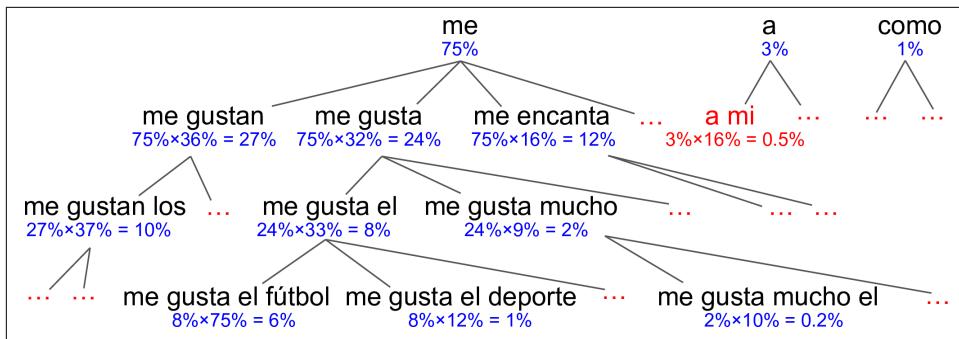


Figure 16-6. Beam search, with a beam width of 3

Next, we compute the probabilities of each of the 3,000 two-word sentences we considered ($3 \times 1,000$). We do this by multiplying the estimated conditional probability of each word by the estimated probability of the sentence it completes. For example, the estimated probability of the sentence “me” was 75%, while the estimated conditional probability of the word “gustan” (given that the first word is “me”) was 36%, so the estimated probability of the sentence “me gustan” is $75\% \times 36\% = 27\%$. After computing the probabilities of all 3,000 two-word sentences, we keep only the top 3. In this example they all start with the word “me”: “me gustan” (27%), “me gusta” (24%), and “me encanta” (12%). Right now, the sentence “me gustan” is winning, but “me gusta” has not been eliminated.

Then we repeat the same process: we use the model to predict the next word in each of these three sentences, and we compute the probabilities of all 3,000 three-word sentences we considered. Perhaps the top three are now “me gustan los” (10%), “me gusta el” (8%), and “me gusta mucho” (2%). At the next step we may get “me gusta el fútbol” (6%), “me gusta mucho el” (1%), and “me gusta el deporte” (0.2%). Notice that “me gustan” was eliminated, and the correct translation is now ahead. We boosted our encoder–decoder model’s performance without any extra training, simply by using it more wisely.



The TensorFlow Addons library includes a full seq2seq API which lets you build encoder–decoder models with attention, including beam search, and more. However, its documentation is currently very limited. Implementing beam search is a good exercise, so give it a try! Check out the notebook for a possible solution.

With all this, you can get reasonably good translations for fairly short sentences. Unfortunately, this model will be really bad at translating long sentences. Once again, the problem comes from the limited short-term memory of RNNs. *Attention mechanisms* are the game-changing innovation that addressed this problem.

Attention Mechanisms

Consider the path from the word “soccer” to its translation “fútbol” back in [Figure 16-3](#): it is quite long! This means that a representation of this word (along with all the other words) needs to be carried over many steps before it is actually used. Can’t we make this path shorter?

This was the core idea in a landmark [2014 paper¹⁹](#) by Dzmitry Bahdanau et al., where the authors introduced a technique that allowed the decoder to focus on the appropriate words (as encoded by the encoder) at each time step. For example, at the time step where the decoder needs to output the word “fútbol”, it will focus its attention on the word “soccer”. This means that the path from an input word to its translation is now much shorter, so the short-term memory limitations of RNNs have much less impact. Attention mechanisms revolutionized neural machine translation (and Deep Learning in general), allowing a significant improvement in the state of the art, especially for long sentences (e.g., over 30 words).



The most common metric used in NMT is the BiLingual Evaluation Understudy (BLEU) score, which compares each translation produced by the model with several good translations produced by humans: it counts the number of n -grams (sequences of n words) that appear in any of the target translations and adjusts the score to take into account the frequency of the produced n -grams in the target translations.

[Figure 16-7](#) shows our encoder-decoder model with an added attention mechanism. On the left, you have the encoder and the decoder. Instead of just sending the encoder’s final hidden state to the decoder, as well as the previous target word at each step (which is still done, although it is not shown in the figure), we now send all of the encoder’s outputs to the decoder as well. Since the decoder cannot deal with all these encoder outputs at once, they need to be aggregated: at each time step, the decoder’s memory cell computes a weighted sum of all the encoder outputs. This determines which words it will focus on at this step. The weight $\alpha_{(t,i)}$ is the weight of the i^{th} encoder output at the t^{th} decoder time step. For example, if the weight $\alpha_{(3,2)}$ is much larger than the weights $\alpha_{(3,0)}$ and $\alpha_{(3,1)}$, then the decoder will pay much more attention to the encoder’s output for word #2 (“soccer”) than to the other two outputs, at least at this time step. The rest of the decoder works just like earlier: at each time step the memory cell receives the inputs we just discussed, plus the hidden state from the previous time step, and finally (although it is not represented in the

¹⁹ Dzmitry Bahdanau et al., “Neural Machine Translation by Jointly Learning to Align and Translate,” arXiv preprint arXiv:1409.0473 (2014).

diagram) it receives the target word from the previous time step (or at inference time, the output from the previous time step).

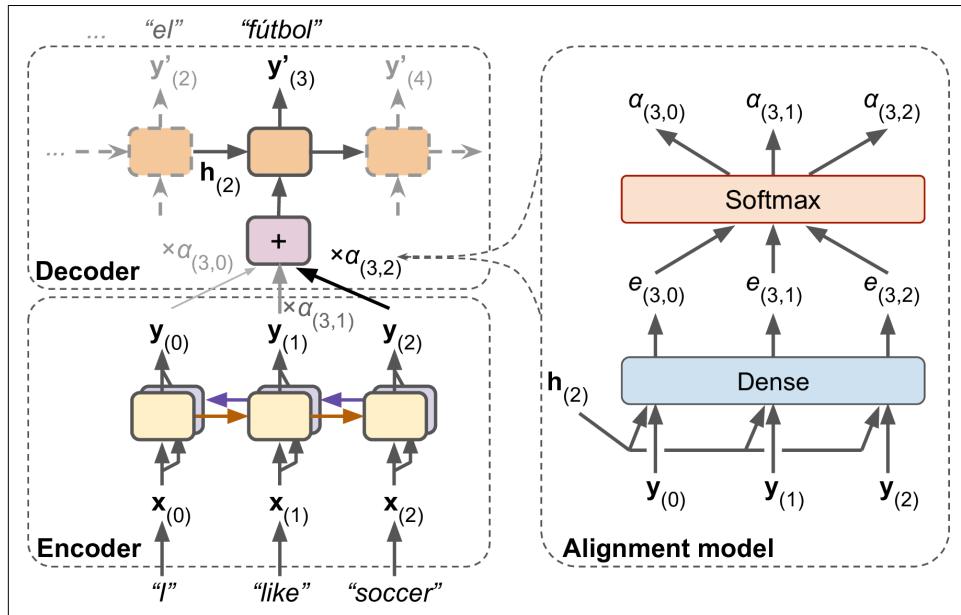


Figure 16-7. Neural machine translation using an encoder–decoder network with an attention model

But where do these $\alpha_{(t,i)}$ weights come from? Well, they are generated by a small neural network called an *alignment model* (or an *attention layer*), which is trained jointly with the rest of the encoder–decoder model. This alignment model is illustrated on the righthand side of Figure 16-7. It starts with a **Dense** layer composed of a single neuron which processes each of the encoder’s outputs, along with the decoder’s previous hidden state (e.g., $h_{(2)}$). This layer outputs a score (or energy) for each encoder output (e.g., $e_{(3,2)}$): this score measures how well each output is aligned with the decoder’s previous hidden state. For example, in Figure 16-7, the model has already output “me gusta el” (meaning “I like”), so it’s now expecting a noun: the word “soccer” is the one that best aligns with the current state, so it gets a high score. Finally, all the scores go through a softmax layer to get a final weight for each encoder output (e.g., $\alpha_{(3,2)}$). All the weights for a given decoder time step add up to 1. This particular attention mechanism is called *Bahdanau attention* (named after the 2014 paper’s first author). Since it concatenates the encoder output with the decoder’s previous hidden state, it is sometimes called *concatenative attention* (or *additive attention*).



If the input sentence is n words long, and assuming the output sentence is about as long, then this model will need to compute about n^2 weights. Fortunately, this quadratic computational complexity is still tractable because even long sentences don't have thousands of words.

Another common attention mechanism, known as *Luong attention* or *multiplicative attention*, was proposed shortly after, in 2015²⁰, by Minh-Thang Luong et al. Because the goal of the alignment model is to measure the similarity between one of the encoder's outputs and the decoder's previous hidden state, the authors proposed to simply compute the *dot product* (see Chapter 4) of these two vectors, as this is often a fairly good similarity measure, and modern hardware can compute it very efficiently. For this to be possible, both vectors must have the same dimensionality. The dot product gives a score, and all the scores (at a given decoder time step) go through a softmax layer to give the final weights, just like in Bahdanau attention. Another simplification Minh-Thang Luong et al. proposed was to use the decoder's hidden state at the current time step rather than at the previous time step (i.e., $\mathbf{h}_{(t)}$ rather than $\mathbf{h}_{(t-1)}$), then to use the output of the attention mechanism (noted $\tilde{\mathbf{h}}_{(t)}$) directly to compute the decoder's predictions (rather than using it to compute the decoder's current hidden state). The researchers also proposed a variant of the dot product mechanism where the encoder outputs first go through a fully-connected layer (without a bias term) before the dot products are computed. This is called the "general" dot product approach. The researchers compared both dot product approaches with the concatenative attention mechanism (adding a rescaling parameter vector \mathbf{v}), and they observed that the dot product variants performed better than concatenative attention. For this reason, concatenative attention is much less used now. The equations for these three attention mechanisms are summarized in Equation 16-1.

Equation 16-1. Attention mechanisms

$$\tilde{\mathbf{h}}_{(t)} = \sum_i \alpha_{(t, i)} \mathbf{y}_{(i)}$$

with $\alpha_{(t, i)} = \frac{\exp(e_{(t, i)})}{\sum_{i'} \exp(e_{(t, i')})}$

and $e_{(t, i)} = \begin{cases} \mathbf{h}_{(t)}^\top \mathbf{y}_{(i)} & \text{dot} \\ \mathbf{h}_{(t)}^\top \mathbf{W} \mathbf{y}_{(i)} & \text{general} \\ \mathbf{v}^\top \tanh(\mathbf{W}[\mathbf{h}_{(t)}; \mathbf{y}_{(i)}]) & \text{concat} \end{cases}$

²⁰ Minh-Thang Luong et al., "Effective Approaches to Attention-Based Neural Machine Translation," *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing* (2015): 1412–1421.

Keras provides a `tf.keras.layers.Attention` layer for Luong attention, and an `AdditiveAttention` layer for Bahdanau attention. Let's add Luong attention to our encoder-decoder model. Since we will need to pass all the encoder's outputs to the `Attention` layer, we first need to set `return_sequences=True` when creating the encoder:

```
encoder = tf.keras.layers.Bidirectional(  
    tf.keras.layers.LSTM(256, return_sequences=True, return_state=True))
```

Next, we need to create the attention layer and pass it the decoder's states and the encoder's outputs. However, to access the decoder's states at each step we would need to write a custom memory cell. For simplicity, let's use the decoder's outputs instead of its states: in practice this works well too and it's much easier to code. Then we just pass the attention layer's outputs directly to the output layer, as suggested in the Luong attention paper:

```
attention_layer = tf.keras.layers.Attention()  
attention_outputs = attention_layer([decoder_outputs, encoder_outputs])  
output_layer = tf.keras.layers.Dense(vocab_size, activation="softmax")  
Y_proba = output_layer(attention_outputs)
```

And that's it! If you train this model, you will find that it now handles much longer sentences, for example:

```
>>> translate("I like soccer and also going to the beach")  
'me gusta el fútbol y también ir a la playa'
```

In short, the attention layer provides a way to focus the attention of the model on part of the inputs. But there's actually another way to think of this layer: it acts as a differentiable memory retrieval mechanism.

For example, let's suppose the encoder analyzed the input sentence "I like soccer", and it managed to understand that the word "I" is the subject and the word "like" is the verb, so it encoded this information in its outputs for these words. Now suppose the decoder has already translated the subject, and it thinks that it should translate the verb next. For this, it needs to fetch the verb from the input sentence. This is analog to a dictionary lookup: it's as if the encoder had created a dictionary {"subject": "They", "verb": "played", ...} and the decoder wanted to look up the value that corresponds to the key "verb".

However, the model does not have discrete tokens to represent the keys (like "subject" or "verb"); instead, it has vectorized representations of these concepts—which it learned during training—so the query it will use for the lookup will not perfectly match any key in the dictionary. The solution is to compute a similarity measure between the query and each key in the dictionary, and then use the softmax function to convert these similarity scores to weights that add up to 1. As we saw earlier, that's exactly what the attention layer does. If the key that represents the verb is by far the most similar to the query, then that key's weight will be close to 1.

Next, the attention layer computes a weighted sum of the corresponding values: if the weight of the “verb” key is close to 1, then the weighted sum will be very close to the representation of the word “played”.

This is why the Keras `Attention` and `AdditiveAttention` layers both expect a list as input, containing two or three items: the *queries*, the *keys*, and optionally the *values*. If you do not pass any values, then they are automatically equal to the keys. So looking at the previous code example again, the decoder outputs are the queries, and the encoder outputs are both the keys and the values. For each decoder output (i.e., each query), the attention layer returns a weighted sum of the encoder outputs (i.e., the keys/values) that are most similar to the decoder output.

The bottom line is: an attention mechanism is a trainable memory retrieval system. It is so powerful that you can actually build state-of-the-art models using only attention mechanisms. Enter the Transformer architecture.

Attention Is All You Need: The Original Transformer Architecture

In a groundbreaking 2017 paper,²¹ a team of Google researchers suggested that “Attention Is All You Need.” They created an architecture called the *Transformer*, which significantly improved the state-of-the-art in NMT without using any recurrent or convolutional layers,²² just attention mechanisms (plus embedding layers, dense layers, normalization layers, and a few other bits and pieces). Because the model is not recurrent, it doesn’t suffer as much from the vanishing or exploding gradients issue as RNNs, it can be trained in less steps, it’s easier to parallelize across multiple GPUs, and it can better capture long-range patterns than RNNs. The original 2017 Transformer architecture is represented in Figure 16-8.

²¹ Ashish Vaswani et al., “Attention Is All You Need,” *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017): 6000–6010.

²² Since the Transformer uses time-distributed dense layers, you could argue that it uses 1D convolutional layers with a kernel size of 1.

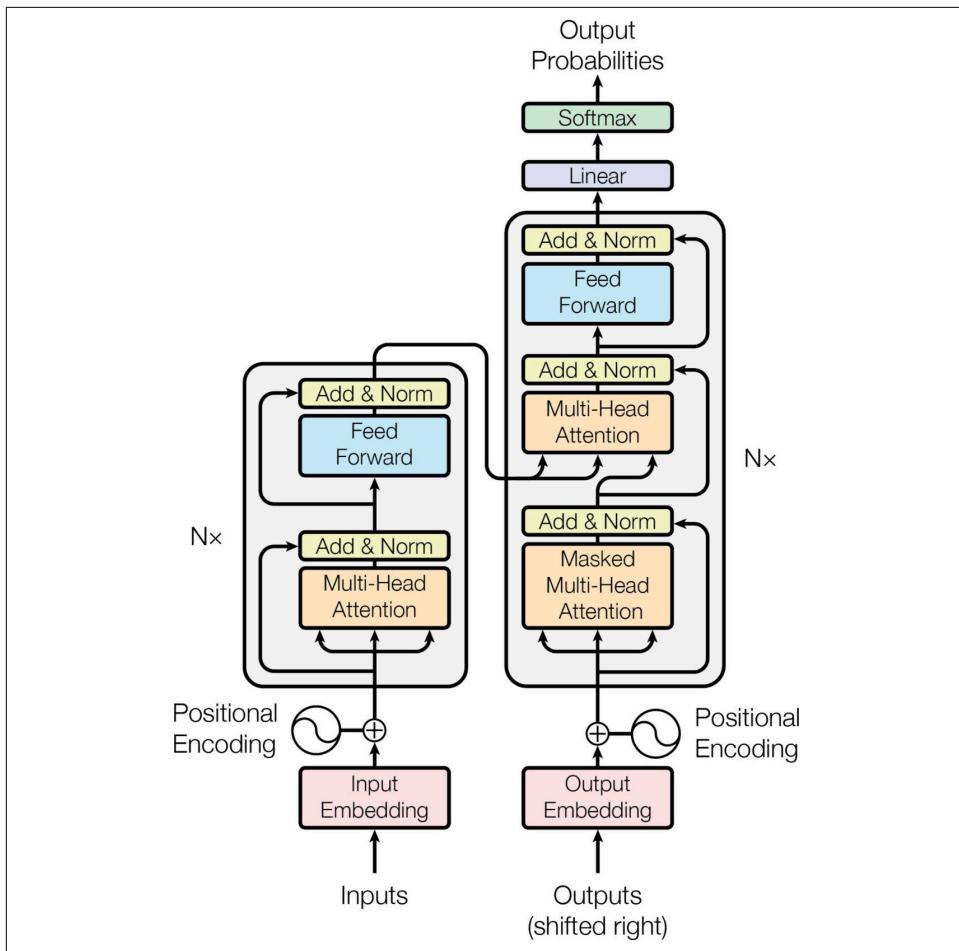


Figure 16-8. The original 2017 Transformer architecture²³

In short, the left part of Figure 16-8 is the encoder, and the right part is the decoder. Each embedding layer outputs a 3D tensor of shape $[\text{batch size}, \text{sequence length}, \text{embedding size}]$. After that, the tensors are gradually transformed as they flow through the Transformer, but their shape remains the same.

If you use the Transformer for NMT, then during training you must feed the English sentences to the encoder, and the corresponding Spanish translations to the decoder, with an extra SOS token inserted at the start of each sentence. At inference time, you must call the Transformer multiple times, producing the translations one word at a

²³ This is figure 1 from the “Attention Is All You Need” paper, reproduced with the kind permission of the authors.

time, and feeding the partial translations to the decoder at each round, just like we did earlier in the `translate()` function.

The encoder's role is to gradually transform the inputs—word representations of the English sentence—until each word's representation perfectly captures the meaning of the word, in the context of the sentence. For example, if you feed the encoder with the sentence “I like soccer”, then the word “like” will start off with a rather vague representation, since this word could mean different things in different contexts: think of “I like soccer” versus “It's like that”. But after going through the encoder, the word's representation should capture the correct meaning of “like” in the given sentence (i.e., to be fond of), as well as any other information that may be required for translation (e.g., it's a verb).

The decoder's role is to gradually transform each word representation in the translated sentence into a word representation of the next word in the translation. For example, if the sentence to translate is “I like soccer”, and the decoder's input sentence is “<SOS> me gusta el fútbol”, then after going through the decoder, the word representation of the word “el” will end up transformed into a representation of the word “fútbol”. Similarly, the representation of the word “fútbol” will be transformed into a representation of the EOS token.

After going through the decoder, each word representation goes through a final Dense layer with a softmax activation function, which will hopefully output a high probability to the correct next word, and a low probability to all other words. The predicted sentence should be “me gusta el fútbol <EOS>”.

That was the big picture, now let's walk through [Figure 16-8](#) in more detail:

- First, notice that both the encoder and the decoder contain modules that are stacked N times. In the paper, $N = 6$. The final outputs of the whole encoder stack are fed to the decoder at each of these N levels.
- Zooming in, you can see that you are already familiar with most components: there are two embedding layers, several skip connections, each of them followed by a layer normalization layer, several feedforward modules that are composed of two dense layers each (the first one using the ReLU activation function, the second with no activation function), and finally the output layer is a dense layer using the softmax activation function. You can also sprinkle a bit of dropout after the attention layers and the feedforward modules, if needed. Since all of these layers are time-distributed, each word is treated independently from all the others. But how can we translate a sentence by looking at the words completely separately? Well, we can't, so that's where the new components come in:
 - The encoder's *Multi-Head Attention* layer updates each word representation by attending to (i.e., paying attention to) all other words in the same sentence. That's where the vague representation of the word “like” becomes a richer

and more accurate representation, capturing its precise meaning in the given sentence. We will discuss exactly how this works shortly.

- The decoder's *Masked Multi-Head Attention* layer does the same thing, but when it processes a word, it doesn't attend to words located after it: it's a causal layer. For example, when it processes the word "gusta", it only attends to the words "<SOS> me gusta", and it ignores the words "el fútbol" (or else that would be cheating).
- The decoder's upper *Multi-Head Attention* layer is where the decoder pays attention to the words in the English sentence. This is called *cross-attention*, not *self-attention* in this case. For example, the decoder will probably pay close attention to the word "soccer" when it processes the word "el" and transforms its representation into a representation of the word "fútbol".
- The *positional encodings* are dense vectors (much like word embeddings) that represent the position of each word in the sentence. The n^{th} positional encoding is added to the word embedding of the n^{th} word in each sentence. This is needed because all layers in the Transformer architecture ignore word positions: without positional encodings, you could shuffle the input sequences, and it would just shuffle the output sequences in the same way. Obviously, the order of words matters, which is why we need to give positional information to the Transformer somehow: adding positional encodings to the word representations is a good way to achieve this.



The first two arrows going into each Multi-Head Attention layer in [Figure 16-8](#) represent the keys and values, and the third arrow represents the queries. In the self-attention layers, all three are equal to the word representations output by the previous layer, while in the decoder's upper attention layer, the keys and values are equal to the encoder's final word representations, while the queries are equal to the word representations output by the previous layer.

Let's go through the novel components of the Transformer architecture in more detail, starting with the positional encodings.

Positional encodings

A positional encoding is a dense vector that encodes the position of a word within a sentence: the i^{th} positional encoding is added to the word embedding of the i^{th} word in the sentence. The easiest way to implement this is to use an `Embedding` layer, and make it encode all the positions from 0 to the maximum sequence length in the batch, then add the result to the word embeddings. The rules of broadcasting will ensure

that the positional encodings get applied to every input sequence. For example, here is how to add positional encodings to the encoder and decoder inputs:

```
max_length = 50 # max length in the whole training set
embed_size = 128
pos_embed_layer = tf.keras.layers.Embedding(max_length, embed_size)
batch_max_len_enc = tf.shape(encoder_embeddings)[1]
encoder_in = encoder_embeddings + pos_embed_layer(tf.range(batch_max_len_enc))
batch_max_len_dec = tf.shape(decoder_embeddings)[1]
decoder_in = decoder_embeddings + pos_embed_layer(tf.range(batch_max_len_dec))
```

Note that this implementation assumes that the embeddings are represented as regular tensors, not ragged tensors.²⁴ The encoder and the decoder share the same Embedding layer for the positional encodings, since they have the same embedding size (this is often the case).

Instead of using trainable positional encodings, the authors of the Transformer paper chose to use fixed positional encodings, based on the sine and cosine functions at different frequencies. The positional encoding matrix \mathbf{P} is defined in [Equation 16-2](#) and represented at the top of [Figure 16-9](#) (transposed), where $P_{p,i}$ is the i^{th} component of the encoding for the word located at the p^{th} position in the sentence.

Equation 16-2. Sine/cosine positional encodings

$$P_{p,i} = \begin{cases} \sin(p/10000^{i/d}) & \text{if } i \text{ is even} \\ \cos(p/10000^{(i-1)/d}) & \text{if } i \text{ is odd} \end{cases}$$

²⁴ It's possible to use ragged tensors instead, if you are using the latest version of TensorFlow.

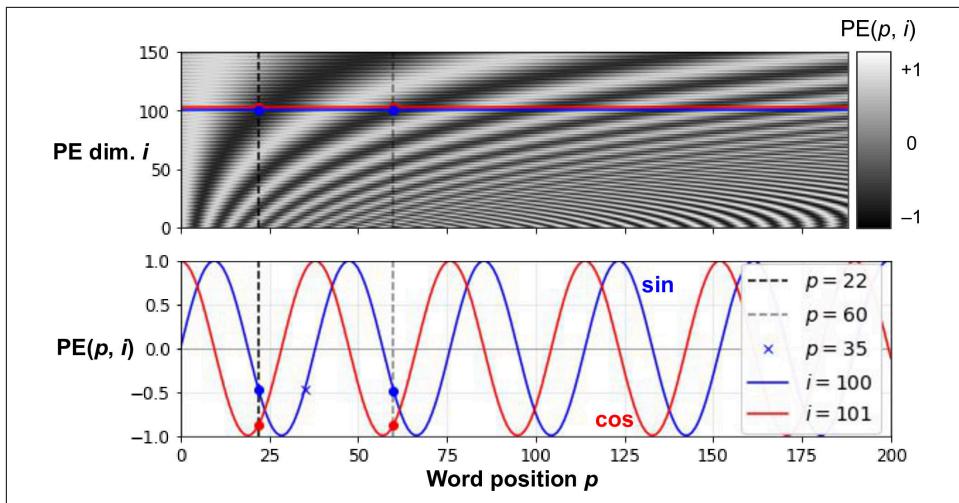


Figure 16-9. Sine/cosine positional encoding matrix (transposed, top) with a focus on two values of i (bottom)

This solution can give the same performance as trainable positional encodings, and it can extend to arbitrarily long sentences without adding any parameters to the model (however, when there is a large amount of pretraining data, trainable positional encodings are usually favored). After these positional encodings are added to the word embeddings, the rest of the model has access to the absolute position of each word in the sentence because there is a unique positional encoding for each position (e.g., the positional encoding for the word located at the 22nd position in a sentence is represented by the vertical dashed line at the top left of Figure 16-9, and you can see that it is unique to that position). Moreover, the choice of oscillating functions (sine and cosine) makes it possible for the model to learn relative positions as well. For example, words located 38 words apart (e.g., at positions $p = 22$ and $p = 60$) always have the same positional encoding values in the encoding dimensions $i = 100$ and $i = 101$, as you can see in Figure 16-9. This explains why we need both the sine and the cosine for each frequency: if we only used the sine (the blue wave at $i = 100$), the model would not be able to distinguish positions $p = 22$ and $p = 35$ (marked by a cross).

There is no `PositionalEncoding` layer in TensorFlow, but it is not too hard to create one. For efficiency reasons, we precompute the positional encoding matrix in the constructor. The `call()` method just truncates this encoding matrix to the max length of the input sequences, and it adds them to the inputs. We also set `supports_masking=True` to propagate the input's automatic mask to the next layer.

```
class PositionalEncoding(tf.keras.layers.Layer):
    def __init__(self, max_length, embed_size, dtype=tf.float32, **kwargs):
        super().__init__(dtype=dtype, **kwargs)
```

```

assert embed_size % 2 == 0, "embed_size must be even"
p, i = np.meshgrid(np.arange(max_length),
                   2 * np.arange(embed_size // 2))
pos_emb = np.empty((1, max_length, embed_size))
pos_emb[0, :, ::2] = np.sin(p / 10_000 ** (i / embed_size)).T
pos_emb[0, :, 1::2] = np.cos(p / 10_000 ** (i / embed_size)).T
self.pos_encodings = tf.constant(pos_emb.astype(self.dtype))
self.supports_masking = True

def call(self, inputs):
    batch_max_length = tf.shape(inputs)[1]
    return inputs + self.pos_encodings[:, :batch_max_length]

```

Let's use this layer to add the positional encoding to the encoder's inputs:

```

pos_embed_layer = PositionalEncoding(max_length, embed_size)
encoder_in = pos_embed_layer(encoder_embeddings)
decoder_in = pos_embed_layer(decoder_embeddings)

```

Now let's look deeper into the heart of the Transformer model: the Multi-Head Attention layer.

Multi-Head Attention

To understand how a Multi-Head Attention layer works, we must first understand the *Scaled Dot-Product Attention* layer, which it is based on. Its equation is shown in [Equation 16-3](#), in a vectorized form. It's the same as Luong attention, except for a scaling factor.

Equation 16-3. Scaled Dot-Product Attention

$$\text{Attention } (\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax} \left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_{keys}}} \right) \mathbf{V}$$

In this equation:

- \mathbf{Q} is a matrix containing one row per *query*. Its shape is $[n_{\text{queries}}, d_{\text{keys}}]$, where n_{queries} is the number of queries and d_{keys} is the number of dimensions of each query and each key.
- \mathbf{K} is a matrix containing one row per *key*. Its shape is $[n_{\text{keys}}, d_{\text{keys}}]$, where n_{keys} is the number of keys and values.
- \mathbf{V} is a matrix containing one row per *value*. Its shape is $[n_{\text{keys}}, d_{\text{values}}]$, where d_{values} is the number of dimensions of each value.
- The shape of $\mathbf{Q} \mathbf{K}^T$ is $[n_{\text{queries}}, n_{\text{keys}}]$: it contains one similarity score for each query/key pair. To prevent this matrix from being huge, the input sequences must not be too long (we will discuss how to overcome this limitation later in this chapter). The output of the softmax function has the same shape, but all rows

sum up to 1. The final output has a shape of $[n_{\text{queries}}, d_{\text{values}}]$: there is one row per query, where each row represents the query result (a weighted sum of the values).

- The scaling factor $1 / (\sqrt{d_{\text{keys}}})$ scales down the similarity scores to avoid saturating the softmax function, which would lead to tiny gradients.
- It is possible to mask out some key/value pairs by adding a very large negative value to the corresponding similarity scores, just before computing the softmax. This is useful in the Masked Multi-Head Attention layer.

If you set `use_scale=True` when create a `tf.keras.layers.Attention` layer, then it will create an additional parameter that lets the layer learn how to properly downscale the similarity scores. The Scaled Dot-Product Attention used in the Transformer model is almost the same, except it always scales the similarity scores by the same factor $1 / (\sqrt{d_{\text{keys}}})$.

Note that the `Attention` layer's inputs are just like **Q**, **K**, and **V**, except with an extra batch dimension (the first dimension). Internally, the layer computes all the attention scores for all sentences in the batch with just one call to `tf.matmul(queries, keys)`: this makes it extremely efficient. Indeed, in TensorFlow, if **A** and **B** are tensors with more than two dimensions—say, of shape [2, 3, 4, 5] and [2, 3, 5, 6] respectively—then `tf.matmul(A, B)` will treat these tensors as 2×3 arrays where each cell contains a matrix, and it will multiply the corresponding matrices: the matrix at the i^{th} row and j^{th} column in **A** will be multiplied by the matrix at the i^{th} row and j^{th} column in **B**. Since the product of a 4×5 matrix with a 5×6 matrix is a 4×6 matrix, `tf.matmul(A, B)` will return an array of shape [2, 3, 4, 6].

Now we're ready to look at the Multi-Head Attention layer. Its architecture is shown in [Figure 16-10](#).

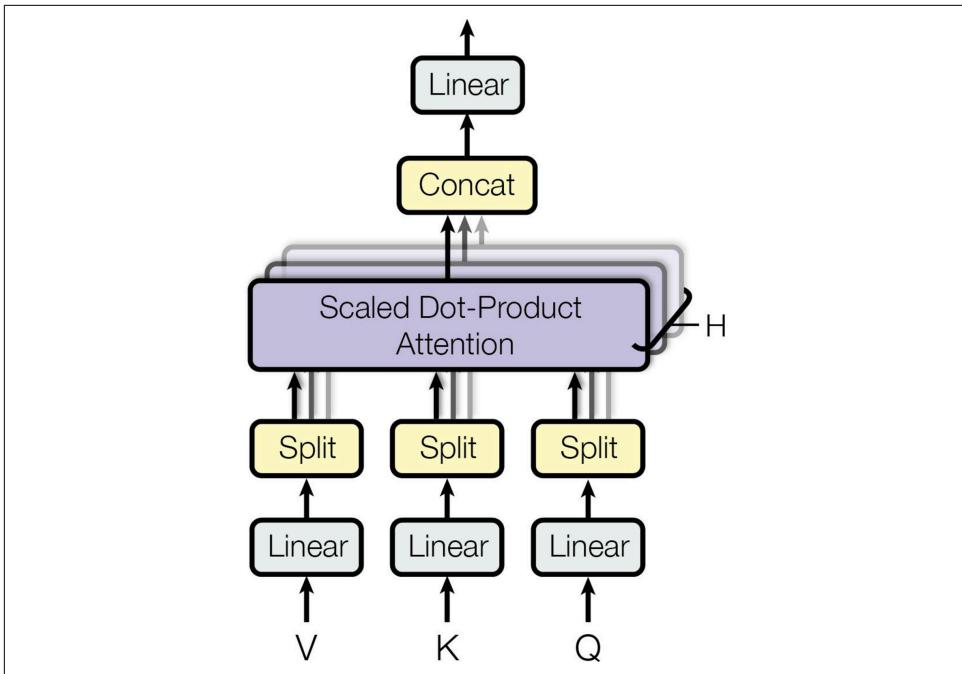


Figure 16-10. Multi-Head Attention layer architecture²⁵

As you can see, it is just a bunch of Scaled Dot-Product Attention layers, each preceded by a linear transformation of the values, keys, and queries (i.e., a time-distributed dense layer with no activation function). All the outputs are simply concatenated, and they go through a final linear transformation (again, time-distributed).

But why? What is the intuition behind this architecture? Well, consider once again the word “like” in the sentence “I like soccer”. The encoder was smart enough to encode the fact that it is a verb. But the word representation also includes its position in the text, thanks to the positional encodings, and it probably includes many other features that are useful for its translation, such as the fact that it is in the present tense. In short, the word representation encodes many different characteristics of the word. If we just used a single Scaled Dot-Product Attention layer, we would only be able to query all of these characteristics in one shot.

This is why the Multi-Head Attention layer applies *multiple* different linear transformations of the values, keys, and queries: this allows the model to apply many different projections of the word representation into different subspaces, each focusing on a subset of the word’s characteristics. Perhaps one of the linear layers will project the

²⁵ This is the right part of figure 2 from the “Attention is all you need” paper, reproduced with the kind authorization of the authors.

word representation into a subspace where all that remains is the information that the word is a verb, another linear layer will extract just the fact that it is present tense, and so on. Then the Scaled Dot-Product Attention layers implement the lookup phase, and finally we concatenate all the results and project them back to the original space.

Keras includes a `tf.keras.layers.MultiHeadAttention` layer, so we now have everything we need to build the rest of the Transformer. Let's start with the full encoder, exactly like in [Figure 16-8](#), except using a stack of two blocks ($N = 2$) instead of six, since we don't have a huge training set, and adding a bit of dropout as well:

```
N = 2 # instead of 6
num_heads = 8
dropout_rate = 0.1
n_units = 128 # for the first dense layer in each feedforward block
encoder_pad_mask = tf.math.not_equal(encoder_input_ids, 0)[:, tf.newaxis]
Z = encoder_in
for _ in range(N):
    skip = Z
    attn_layer = tf.keras.layers.MultiHeadAttention(
        num_heads=num_heads, key_dim=embed_size, dropout=dropout_rate)
    Z = attn_layer(Z, value=Z, attention_mask=encoder_pad_mask)
    Z = tf.keras.layers.LayerNormalization()(tf.keras.layers.Add()([Z, skip]))
    skip = Z
    Z = tf.keras.layers.Dense(n_units, activation="relu")(Z)
    Z = tf.keras.layers.Dense(embed_size)(Z)
    Z = tf.keras.layers.Dropout(dropout_rate)(Z)
    Z = tf.keras.layers.LayerNormalization()(tf.keras.layers.Add()([Z, skip]))
```

This code should be mostly straightforward except for one thing: masking. As of May 2022, the `MultiHeadAttention` layer does not support automatic masking²⁶ so we must handle it manually. How can we do that?

Well, the `MultiHeadAttention` layer accepts an `attention_mask` argument which is a boolean tensor of shape $[batch\ size, max\ query\ length, max\ value\ length]$: for every token in every query sequence, this mask indicates which tokens in the corresponding value sequence should be attended to. We want to tell the `MultiHeadAttention` layer to ignore all the padding tokens in the values. So we first compute the padding mask using `tf.math.not_equal(encoder_input_ids, 0)`. This returns a boolean tensor of shape $[batch\ size, max\ sequence\ length]$. We then insert a second axis using $[:, tf.newaxis]$, to get a mask of shape $[batch\ size, 1, max\ sequence\ length]$. This allows us to use this mask as the `attention_mask` when calling the `MultiHeadAttention` layer: thanks to broadcasting, the same mask will be used for all tokens in each query. This way, the padding tokens in the values will be ignored correctly.

²⁶ This will most likely change by the time you read this. Check out Keras issue #16248 for more details. When this happens, there will be no need to set the `attention_mask` argument, and therefore no need to create `encoder_pad_mask`.

However, the layer will compute outputs for every single query token, including the padding tokens. We need to mask the outputs that correspond to these padding tokens. Recall that we used `mask_zero` in the `Embedding` layers, and we set `supports_masking` to `True` in the `PositionalEncoding` layer, so the automatic mask was propagated all the way to the `MultiHeadAttention` layer's inputs (`encoder_in`). We can use this to our advantage in the skip connection: indeed, the `Add` layer supports automatic masking, so when we add `Z` and `skip` (which is initially equal to `encoder_in`), the output get automatically masked correctly.²⁷ Yikes! Masking required much more explanation than code.

Now on to the decoder! Once again, masking is going to be the only tricky part, so let's start with that. The first Multi-Head Attention layer is a self-attention layer, like in the encoder, but it is a *masked* Multi-Head Attention layer, meaning it is causal: it should ignore all tokens in the future. So we need two masks: a padding mask, and a causal mask. Let's create them:

```
decoder_pad_mask = tf.math.not_equal(decoder_input_ids, 0)[ :, tf.newaxis]
causal_mask = tf.linalg.band_part( # creates a lower triangular matrix
    tf.ones((batch_max_len_dec, batch_max_len_dec), tf.bool), -1, 0)
```

The padding mask is exactly like the one we created for the encoder, except it's based on the decoder's inputs rather than the encoder's. The causal mask is created using the `tf.linalg.band_part()` function, which takes a tensor and returns a copy with all the values outside a diagonal band set to zero. With these arguments, we get a square matrix of size `batch_max_len_dec` (the max length of the input sequences in the batch), with 1s in the lower left triangle, and 0s in the upper right. If we use this mask as the attention mask, we will get exactly what we want: the first query token will only attend to the first value token, the second will only attend to the first two, the third will only attend to the first three, and so on. In other words, query tokens cannot attend to any value token in the future.

Let's now build the decoder:

```
encoder_outputs = Z # let's save the encoder's final outputs
Z = decoder_in # the decoder starts with its own inputs
for _ in range(N):
    skip = Z
    attn_layer = tf.keras.layers.MultiHeadAttention(
        num_heads=num_heads, key_dim=embed_size, dropout=dropout_rate)
    Z = attn_layer(Z, value=Z, attention_mask=causal_mask & decoder_pad_mask)
    Z = tf.keras.layers.LayerNormalization()(tf.keras.layers.Add()([Z, skip]))
    skip = Z
    attn_layer = tf.keras.layers.MultiHeadAttention(
        num_heads=num_heads, key_dim=embed_size, dropout=dropout_rate)
```

²⁷ Currently `Z + skip` does not support automatic masking, which is why we had to write `tf.keras.layers.Add()([Z, skip])` instead. Again, this may change by the time you read this.

```

Z = attn_layer(Z, value=encoder_outputs, attention_mask=encoder_pad_mask)
Z = tf.keras.layers.LayerNormalization()(tf.keras.layers.Add()([Z, skip]))
skip = Z
Z = tf.keras.layers.Dense(n_units, activation="relu")(Z)
Z = tf.keras.layers.Dense(embed_size)(Z)
Z = tf.keras.layers.LayerNormalization()(tf.keras.layers.Add()([Z, skip]))

```

For the first attention layer, we use `causal_mask` & `decoder_pad_mask` to mask both the padding tokens and future tokens. The causal mask only has two dimensions, it's missing the batch dimension, but that's okay since broadcasting ensures that it gets copied across all the instances in the batch.

For the second attention layer, there's nothing special. The only thing to note is that we are using `encoder_pad_mask`, not `decoder_pad_mask`, because this attention layer uses the encoder's final outputs as its values.

We're almost done. We just need to add the final output layer, create the model, compile it, and train it:

```

Y_proba = tf.keras.layers.Dense(vocab_size, activation="softmax")(Z)
model = tf.keras.Model(inputs=[encoder_inputs, decoder_inputs],
                       outputs=[Y_proba])
model.compile(loss="sparse_categorical_crossentropy", optimizer="nadam",
               metrics=["accuracy"])
model.fit((X_train, X_train_dec), Y_train, epochs=10,
          validation_data=((X_valid, X_valid_dec), Y_valid))

```

Congratulations! You've built a full Transformer from scratch, and trained it for automatic translation. This is getting quite advanced!



The Keras team has created a new [Keras-NLP project](#), including an API to build a Transformer more easily. You may also be interested in the new [Keras-CV project for computer vision](#).

But the field didn't stop there. Let's now explore some of the recent advances.

An Avalanche of Transformer Models

The year 2018 has been called the “ImageNet moment for NLP”. Since then, progress has been astounding, with larger and larger Transformer-based architectures trained on immense datasets.

First, the [GPT paper²⁸](#) by Alec Radford and other OpenAI researchers once again demonstrated the effectiveness of unsupervised pretraining, like the ELMo and ULM-

²⁸ Alec Radford et al., “Improving Language Understanding by Generative Pre-Training” (2018).

FiT papers did, but this time using a Transformer-like architecture. The authors pretrained a large but fairly simple architecture composed of a stack of 12 Transformer modules using only Masked Multi-Head Attention layers, like in the original transformer’s decoder. They trained it on a very large dataset, using the same autoregressive technique we used for our Shakespeare CharRNN: just predict the next token. This is a form of self-supervised learning. Then they fine-tuned it on various language tasks, using only minor adaptations for each task. The tasks were quite diverse: they included text classification, *entailment* (whether sentence A entails sentence B),²⁹ similarity (e.g., “Nice weather today” is very similar to “It is sunny”), and question answering (given a few paragraphs of text giving some context, the model must answer some multiple-choice questions).

Then Google’s [BERT paper](#)³⁰ came out: it also demonstrates the effectiveness of self-supervised pretraining on a large corpus, using a similar architecture to GPT but with non-masked Multi-Head Attention layers only, like in the original Transformer’s encoder. This means that the model is naturally bidirectional; hence the B in BERT (*Bidirectional Encoder Representations from Transformers*). Most importantly, the authors proposed two pretraining tasks that explain most of the model’s strength:

Masked language model (MLM)

Each word in a sentence has a 15% probability of being masked, and the model is trained to predict the masked words. For example, if the original sentence is “She had fun at the birthday party,” then the model may be given the sentence “She <mask> fun at the <mask> party” and it must predict the words “had” and “birthday” (the other outputs will be ignored). To be more precise, each selected word has an 80% chance of being masked, a 10% chance of being replaced by a random word (to reduce the discrepancy between pretraining and fine-tuning, since the model will not see <mask> tokens during fine-tuning), and a 10% chance of being left alone (to bias the model toward the correct answer).

Next sentence prediction (NSP)

The model is trained to predict whether two sentences are consecutive or not. For example, it should predict that “The dog sleeps” and “It snores loudly” are consecutive sentences, while “The dog sleeps” and “The Earth orbits the Sun” are not consecutive. Later research showed that NSP was not as important as was initially thought, so it was dropped in most later architectures.

²⁹ For example, the sentence “Jane had a lot of fun at her friend’s birthday party” entails “Jane enjoyed the party,” but it is contradicted by “Everyone hated the party” and it is unrelated to “The Earth is flat.”

³⁰ Jacob Devlin et al., “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* 1 (2019).

The model is trained on these two tasks simultaneously (see [Figure 16-11](#)). For the NSP task, the authors inserted a class token (<CLS>) at the start of every input, and the corresponding output token represents the model's prediction: sentence B follows sentence A, or it does not. The two input sentences are concatenated, separated only by a special separation token (<SEP>), and they are fed as input to the model. To help the model know which sentence each input token belongs to, a *segment embedding* is added on top of each token's positional embeddings: there are just two possible segment embeddings, one for sentence A, and one for sentence B. For the MLM task, some input words are masked (as we just saw) and the model tries to predict what those words were. The loss is only computed on the NSP prediction and the masked tokens, not on the unmasked ones.

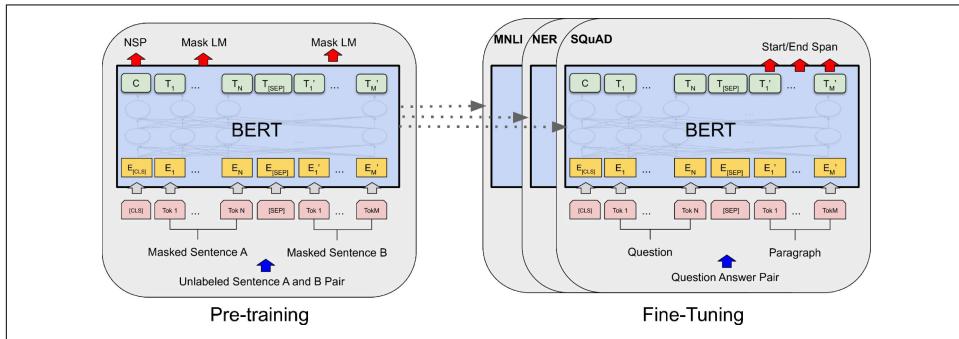


Figure 16-11. BERT training and fine-tuning process³¹

After this unsupervised pretraining phase on a very large corpus of text, the model is then fine-tuned on many different tasks, changing very little for each task. For example, for text classification such as sentiment analysis, all output tokens are ignored except for the first one, corresponding to the class token, and a new output layer replaces the previous one, which was just a binary classification layer for NSP.

In February 2019, just a few months after BERT was published, Alec Radford, Jeffrey Wu, and other OpenAI researchers published the [GPT-2 paper](#),³² which proposed a very similar architecture to GPT, but larger still (with over 1.5 billion parameters!). The researchers showed that the new and improved GPT model could perform *zero-shot learning* (ZSL), meaning it could achieve good performance on many tasks without any fine-tuning. This was just the start of a race towards larger and larger models: Google's [Switch Transformers](#)³³ (introduced in January 2021) used 1 trillion

³¹ This is figure 1 from the paper, reproduced with the kind authorization of the authors.

³² Alec Radford et al., "Language Models Are Unsupervised Multitask Learners" (2019).

³³ William Fedus et al., "Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity" (2021).

parameters, and soon much larger models came out, such as the Wu Dao 2.0 model by the Beijing Academy of Artificial Intelligence (BAII), in June 2021.

Sadly, this trend toward gigantic models implies that only well-funded organizations can afford to train such models: it can easily cost hundreds of thousands of dollars or more. And the energy required to train a single model corresponds to an American household's electricity consumption for several years: not eco-friendly at all. Many of these models are just too big to even be used on regular hardware: they wouldn't fit in RAM, and they would be horribly slow. Lastly, some are so costly that they are not released publicly.

Luckily, ingenious researchers are finding new ways to downsize Transformers and make them more data-efficient. For example, the **DistilBERT model**,³⁴ introduced in October 2019 by Victor Sanh et al. from Hugging Face, is a small and fast Transformer model based on BERT. It is available on Hugging Face's excellent model hub, along with thousands of others—we will see an example later in this chapter. DistilBERT was trained using *distillation* (hence the name): this means transferring knowledge from a teacher model to a student one, which is usually much smaller than the teacher model. This is typically done by using the teacher's predicted probabilities for each training instance as targets for the student. Surprisingly, distillation often works better than training the student from scratch on the same dataset as the teacher! Indeed, the student benefits from the teacher's more nuanced labels.

Many more Transformer architectures came out after BERT almost every month, often improving on the state-of-the-art across all NLP tasks: XLNet (June 2019), RoBERTa (July 2019), StructBERT (August 2019), ALBERT (September 2019), T5 (October 2019), ELECTRA (March 2020), GPT3 (May 2020), DeBERTa (June 2020), Switch Transformers (January 2021), Wu Dao 2.0 (June 2021), Gopher (December 2021), GPT-NeoX-20B (February 2022), Chinchilla (March 2022), OPT (May 2022), and the list goes on and on. Each of these models brought new ideas and techniques,³⁵ but I particularly like Google's **T5 paper**³⁶ by Google researchers: it framed all NLP tasks as text-to-text, using an encoder-decoder Transformer. For example, to translate "I like soccer" to Spanish, you can just call the model with the following input sentence: "translate English to Spanish: I like soccer" and it outputs "me gusta el fútbol". To summarize a paragraph, you just enter "summarize:" followed by the paragraph, and it outputs the summary. For classification, you only need to change the prefix to "classify:" and the model outputs the class name, as text. And so on. This

³⁴ Victor Sanh et al., "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter" (2019).

³⁵ Mariya Yao summarized many of these models in this post: <https://hom.info/yaopost>.

³⁶ Colin Raffel et al., "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer" (2019).

simplifies using the model, and it also makes it possible to pretrain it on even more tasks.

Last but not least, in April 2022, Google researchers used a new large-scale training platform named *Pathways* (which we will briefly discuss in [Chapter 19](#)) to train a humongous language model named *Pathways Language Model* (PaLM)³⁷, with a whopping 540 billion parameters, using over 6,000 TPUs. Other than its incredible size, this model is a standard Transformer, using decoders only (i.e., with masked multi-head attention layers), with just a few tweaks (please see the paper for the details). This model reached incredible performance in all sorts of NLP tasks, particularly in Natural Language Understanding (NLU). It's capable of impressive feats, such as explaining jokes, giving detailed step-by-step answers to questions, or even coding. This is in part due to the model's size, but also thanks to a technique called *Chain-of-Thought Prompting*³⁸, which was introduced a couple months earlier by another team of Google researchers.

In question answering tasks, regular prompting typically includes a few examples of questions and answers, such as: “Q: *Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now? A: 11.*” The prompt then continues with the actual question, such as “Q: *John takes care of 10 dogs. Each dog takes .5 hours a day to walk and take care of their business. How many hours a week does he spend taking care of dogs? A:*”, and the model’s job is to append the answer: in this case, “35.” But with Chain-of-Thought Prompting, the example answers include all the reasoning steps that lead to the conclusion. For example, instead of “A: 11,” the prompt contains “A: *Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11.*” This encourages the model to give a detailed answer to the actual question, such as “*John takes care of 10 dogs. Each dog takes .5 hours a day to walk and take care of their business. So that is 10 x .5 = 5 hours a day. 5 hours a day x 7 days a week = 35 hours a week. The answer is 35 hours a week.*” This is an actual example from the paper! Not only does the model give the right answer much more frequently than using regular prompting—we’re encouraging the model to think things through—it also provides all the reasoning steps, which can be useful to better understand the rationale behind a model’s answer.

Transformers have taken over NLP, but they didn’t stop there: they soon expanded to computer vision as well.

³⁷ Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, et al., “PaLM: Scaling Language Modeling with Pathways” (2022).

³⁸ Jason Wei et al., “Chain of Thought Prompting Elicits Reasoning in Large Language Models” (2022)

Vision Transformers

One of the first applications of attention mechanisms beyond NMT was in generating image captions using **visual attention**:³⁹ a convolutional neural network first processes the image and outputs some feature maps, then a decoder RNN equipped with an attention mechanism generates the caption, one word at a time. At each decoder time step (i.e., each word), the decoder uses the attention model to focus on just the right part of the image. For example, in [Figure 16-12](#), the model generated the caption “A woman is throwing a frisbee in a park”, and you can see what part of the input image the decoder focused its attention on when it was about to output the word “frisbee”: clearly, most of its attention was focused on the frisbee.



Figure 16-12. Visual attention: an input image (left) and the model's focus before producing the word “frisbee” (right)⁴⁰

Explainability

One extra benefit of attention mechanisms is that they make it easier to understand what led the model to produce its output. This is called *explainability*. It can be especially useful when the model makes a mistake: for example, if an image of a dog walking in the snow is labeled as “a wolf walking in the snow”, then you can go back and check what the model focused on when it output the word “wolf”. You may find that it was paying attention not only to the dog, but also to the snow, hinting at a possible explanation: perhaps the way the model learned to distinguish dogs from

³⁹ Kelvin Xu et al., “Show, Attend and Tell: Neural Image Caption Generation with Visual Attention,” *Proceedings of the 32nd International Conference on Machine Learning* (2015): 2048–2057.

⁴⁰ This is a part of figure 3 from the paper. It is reproduced with the kind authorization of the authors.

wolves is by checking whether or not there's a lot of snow around. You can then fix this by training the model with more images of wolves without snow, and dogs with snow. This example comes from a great [2016 paper](#)⁴¹ by Marco Tulio Ribeiro et al. that uses a different approach to explainability: learning an interpretable model locally around a classifier's prediction.

In some applications, explainability is not just a tool to debug a model; it can be a legal requirement—think of a system deciding whether or not it should grant you a loan.

When Transformers came out in 2017 and people started to experiment with them beyond NLP, they were first used alongside CNNs, without replacing them. Instead, Transformers were generally used to replace RNNs, for example, in image captioning models. Transformers became slightly more visual in a [2020 paper](#)⁴² by Facebook researchers, which proposed a hybrid CNN-Transformer architecture for object detection. Once again, the CNN first processes the input images and outputs a set of feature maps, then these feature maps are converted to sequences and fed to a transformer, which outputs bounding box predictions. But again, most of the visual work is still done by the CNN.

Then, in October 2020, a team of Google researchers released a [paper](#)⁴³ that introduced a fully Transformer-based vision model, called a *Vision Transformer* (ViT). The idea is surprisingly simple: just chop the image into little 16×16 squares, and treat the sequence of squares as if it they were a sequence of word representations. To be more precise, the squares are first flattened into $16 \times 16 \times 3 = 768$ dimensional vectors—the 3 is for the 3 RGB color channels—then these vectors go through a linear layer that transforms them but retains their dimensionality. The resulting sequence of vectors can then be treated just like a sequence of word embeddings: this means adding positional embeddings, and passing the result to the Transformer. That's it! This paper beat the state-of-the-art on ImageNet image classification, but to be fair they had to use over 300 million additional images for training. This makes sense since Transformers don't have as many *inductive biases* as Convolution Neural Nets, so they need extra data just to learn things that CNNs implicitly assume.

⁴¹ Marco Tulio Ribeiro et al., “Why Should I Trust You?: Explaining the Predictions of Any Classifier,” *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2016): 1135–1144.

⁴² Nicolas Carion et al., “End-to-End Object Detection with Transformers” (2020).

⁴³ Alexey Dosovitskiy et al., “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale” (2020).



An inductive bias is an implicit assumption made by the model, due to its architecture. For example, linear models implicitly assume that the data is, well, linear. CNNs implicitly assume that patterns learned in one location will likely be useful in other locations as well. RNNs implicitly assume that the inputs are ordered, and that recent tokens are more important than older ones. The more inductive biases a model has, assuming they are correct, the less training data the model will require. But if the implicit assumptions are wrong, then the model may perform poorly even if it is trained on a large dataset.

Just two months later, a team of Facebook researchers released a [paper⁴⁴](#) that introduced *Data-efficient Image Transformers* (DeiT). Their model reached competitive results on ImageNet without requiring any additional data for training. The model's architecture is virtually the same as the original ViT, but they used a distillation technique to transfer knowledge from state-of-the-art CNN models to their model.

Then in March 2021, DeepMind released an important [paper⁴⁵](#) which introduced the *Perceiver* architecture. It is a *multimodal* Transformer, meaning you can feed it text, images, audio, or virtually any other *modality*. Until then, Transformers were restricted to fairly short sequences, because of the performance and RAM bottleneck in the attention layers. This excluded modalities such as audio or video, and it forced researchers to treat images as sequences of patches, rather than sequences of pixels. The bottleneck is due to self-attention, where every token must attend to every other token. Indeed, if the input sequence has M tokens, then the attention layer must compute an $M \times M$ matrix, which can be huge if M is very large. The Perceiver solves this problem by gradually improving a fairly short *latent representation* of the inputs, composed of N tokens—typically just a few hundred. The word *latent* means hidden, or internal. The model uses cross-attention layers only, feeding them the latent representation as the queries, and the (possibly large) inputs as the values. This only requires computing an $M \times N$ matrix, so the computational complexity is linear with regards to M , instead of quadratic. After going through several cross-attention layers, if everything goes well, the latent representations end up capturing everything that matters in the inputs. The authors also suggested sharing the weights between consecutive cross-attention layers: if you do that, then the Perceiver effectively becomes an RNN. Indeed, the shared cross-attention layers can be seen as the same memory cell at different time steps, and the latent representation corresponds to the cell's context vector. The same inputs are repeatedly fed to the memory cell at every time step. It looks like RNNs are not dead after all!

⁴⁴ Hugo Touvron et al., “Training data-efficient image transformers & distillation through attention” (2020).

⁴⁵ Andrew Jaegle et al., “Perceiver: General Perception with Iterative Attention” (2021).

Just a month later, Mathilde Caron et al. introduced **DINO** “Emerging Properties in Self-Supervised Vision Transformers” (2021), an impressive vision Transformer trained entirely without labels, using self-supervision, and capable of high-accuracy semantic segmentation. The model is duplicated during training, with one network acting as a teacher, and the other acting as a student. Gradient descent only affects the student, while the teacher’s weights are just an exponential moving average of the student’s weights. The student is trained to match the teacher’s predictions: since they’re almost the same model, this is called *self-distillation*. At each training step, the input images are augmented in different ways for the teacher and the student, so they don’t see the exact same image, but their predictions must match. This forces them to come up with high-level representations. To prevent *mode collapse*, where both the student and the teacher would always output the same thing, completely ignoring the inputs, DINO keeps track of a moving average of the teacher’s outputs, and it tweaks the teacher’s predictions to ensure that they remained centered on zero, on average. DINO also forces the teacher to have high confidence in its predictions: this is called *sharpening*. Together, these techniques preserve diversity in the teacher’s outputs.

In a [2021 paper](#),⁴⁶ Google researchers showed how to scale ViTs up or down, depending on the amount of data. They managed to create a huge 2 billion parameter model that reached over 90.4% top-1 accuracy on ImageNet. Conversely, they also trained a scaled down model that reached over 84.8% top-1 accuracy on ImageNet, using only 10,000 images: that’s just 10 images per class!

And progress in visual Transformers has continued steadily to this day. For example, in March 2022, a [paper](#)⁴⁷ by Mitchell Wortsman et al. demonstrated that it’s possible to first train multiple Transformers, then average their weights to create a new and improved model. This is similar to an ensemble, except there’s just one model in the end, which also means there’s no inference time penalty.

Now the latest trend in Transformers consists in building large multimodal models, often capable of zero-shot or few-shot learning. For example, [OpenAI’s 2021 CLIP paper](#)⁴⁸ proposed a large Transformer model pretrained to match captions with images: this task allows it to learn excellent image representations, and the model can then be used directly for tasks such as image classification using simple text prompts such as “a photo of a cat”. Soon after, OpenAI released the [DALL-E paper](#)⁴⁹, capable of generating amazing images based on text prompts. The [DALL-E 2 paper](#)⁵⁰

⁴⁶ Xiaohua Zhai et al., “Scaling Vision Transformers” (2021).

⁴⁷ Mitchell Wortsman et al., “Model soups: averaging weights of multiple fine-tuned models improves accuracy without increasing inference time” (2022).

⁴⁸ Alec Radford et al., “Learning Transferable Visual Models From Natural Language Supervision,” (2021).

⁴⁹ Aditya Ramesh et al., “Zero-Shot Text-to-Image Generation,” (2021).

⁵⁰ “Hierarchical Text-Conditional Image Generation with CLIP Latents,” (2022).

generates even higher quality images using a diffusion model (see [Chapter 17](#)). In April 2022, DeepMind released the [Flamingo paper](#)⁵¹, which introduced a family of models pretrained on a wide variety of tasks across multiple modalities, including text, images, and videos. A single model can be used across very different tasks, such as question-answering, image captioning, and more. Soon after, in May 2022, DeepMind introduced the [GATO model](#)⁵², again a multimodal model, but which can be used as a policy for a Reinforcement Learning agent (RL will be introduced in [Chapter 18](#)). The same Transformer can chat with you, caption images, play Atari games, control (simulated) robotic arms, and more. And all that with “only” 1.2 billion parameters. And the adventure continues!



These astounding advances have led some researchers to claim that Human-Level AI is near, that “scale is all you need,” and that some of these models may be “slightly conscious.” Others point out that despite the amazing progress, these models still lack the reliability and adaptability of human intelligence, our ability to reason symbolically, to generalize based on a single example, and more.

As you can see, Transformers are everywhere! And the good news is that you generally won’t have to implement Transformers yourself since many excellent pretrained models are readily available for download via TensorFlow Hub or Hugging Face’s model hub. We have already seen how to use a model from TF Hub, so let’s close this chapter by taking a quick look at Hugging Face’s ecosystem.

Hugging Face’s Transformers Library

It’s impossible to talk about Transformers today without mentioning Hugging Face, an A.I. company that has built a whole ecosystem of easy-to-use open source tools for NLP, vision, and beyond. The central component of their ecosystem is the *Transformers library*, which allows you to easily download a pretrained model, including its corresponding tokenizer, and then fine-tune it on your own dataset, if needed. Plus the library supports TensorFlow, PyTorch, and JAX (with the Flax library).

The simplest way to use the Transformers library is to use the `transformers.pipeline()` function: you just specify which task you want, such as sentiment analysis, and it downloads a default pretrained model, ready to be used—it really couldn’t be any simpler:

⁵¹ Jean-Baptiste Alayrac et al., “Flamingo: a Visual Language Model for Few-Shot Learning.” (2022).

⁵² Scott Reed et al., “A Generalist Agent” (2022).

```
from transformers import pipeline

classifier = pipeline("sentiment-analysis") # many other tasks are available
result = classifier("The actors were very convincing.")
```

The result is a Python list containing one dictionary per input text:

```
>>> result
[{'label': 'POSITIVE', 'score': 0.9998071789741516}]
```

In this example, the model correctly found that the sentence is positive, with around 99.98% confidence. Of course, you can also pass a batch of sentences to the model:

```
>>> classifier(["I am from India.", "I am from Iraq."])
[{'label': 'POSITIVE', 'score': 0.9896161556243896},
 {'label': 'NEGATIVE', 'score': 0.9811071157455444}]
```

Bias and fairness

As the output suggests, this specific classifier loves Indians, but is severely biased against Iraqis. You can try this code with your own country or city. Such an undesirable bias generally comes in large part from the training data itself: in this case, there were plenty of negative sentences related to the wars in Iraq in the training data. This bias was then amplified during the fine-tuning process since the model was forced to choose between just two classes: positive or negative. If you add a neutral class when fine-tuning, then the country bias mostly disappears. But the training data is not the only source of bias: the model's architecture, the type of loss or regularization used for training, the optimizer, all of these can affect what the model ends up learning. Even a mostly unbiased model can be *used* in a biased way, much like survey questions can be biased.

Understanding bias in A.I. and mitigating its negative effects is still an area of active research, but one thing is certain: you should pause and think before you rush to deploy a model to production. Ask yourself how the model could do harm, even indirectly. For example, if the model's predictions are used to decide whether or not to give someone a loan, the process should be fair. So make sure you evaluate the model's performance not just on average over the whole test set, but across various subsets as well: for example, you may find that although the model works very well on average, its performance is abysmal for some categories of people. You may also want to run counterfactual tests: for example, you may want to check that the model's predictions do not change when you simply switch someone's gender.

If the model works well on average, it's tempting to push it to production and move on to something else, especially if it's just one component of a much larger system. But in general, if you don't fix such issues, no one else will, and your model may end up doing more harm than good. The solution depends on the problem: it

may require rebalancing the dataset, fine-tuning on a different dataset, switching to another pretrained model, tweaking the model’s architecture or hyperparameters, etc.

The `pipeline()` function uses the default model for the given task. For example, for text classification such as sentiment analysis, at the time of writing, it defaults to “distilbert-base-uncased-finetuned-sst-2-english”—a DistilBERT model with an uncased tokenizer, trained on the English Wikipedia and on a corpus of English books, and fine-tuned on the Stanford Sentiment Treebank v2 (SST 2) task. It’s also possible to manually specify a different model. For example, you could use a DistilBERT model fine-tuned on the Multi-Genre Natural Language Inference (MNLI) task, which classifies two sentences into three classes: contradiction, neutral, or entailment. Here is how:

```
>>> model_name = "huggingface/distilbert-base-uncased-finetuned-mnli"
>>> classifier_mnli = pipeline("text-classification", model=model_name)
>>> classifier_mnli("She loves me. [SEP] She loves me not.")
[{'label': 'contradiction', 'score': 0.9790192246437073}]
```



You can find the available models at <https://huggingface.co/models>, and the list of tasks at <https://huggingface.co/tasks>.

The pipeline API is very simple and convenient, but sometimes you will need more control. For such cases, the Transformers library provides many classes, including all sorts of tokenizers, models, configurations, callbacks, and much more. For example, let’s load the same DistilBERT model, along with its corresponding tokenizer, using the `TFAutoModelForSequenceClassification` and `AutoTokenizer` classes:

```
from transformers import AutoTokenizer, TFAutoModelForSequenceClassification

tokenizer = AutoTokenizer.from_pretrained(model_name)
model = TFAutoModelForSequenceClassification.from_pretrained(model_name)
```

Next, let’s tokenize a couple pairs of sentences. In this code, we activate padding and specify that we want TensorFlow tensors instead of Python lists:

```
token_ids = tokenizer(["I like soccer. [SEP] We all love soccer!",
                      "Joe lived for a very long time. [SEP] Joe is old."],
                      padding=True, return_tensors="tf")
```



Instead of passing “Sentence 1 [SEP] Sentence 2” to the tokenizer, you can equivalently pass it a tuple: (“Sentence 1”, “Sentence 2”).

The output is a dictionary-like instance of the `BatchEncoding` class, which contains the sequences of token IDs, as well as a mask containing 0s for the padding tokens:

```
>>> token_ids
{'input_ids': <tf.Tensor: shape=(2, 15), dtype=int32, numpy=
array([[ 101, 1045, 2066, 4715, 1012, 102, 2057, 2035, 2293, 4715, 999,
       102,     0,     0,     0],
       [ 101, 3533, 2973, 2005, 1037, 2200, 2146, 2051, 1012, 102, 3533,
       2003, 2214, 1012, 102]], dtype=int32)>,
'attention_mask': <tf.Tensor: shape=(2, 15), dtype=int32, numpy=
array([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]], dtype=int32)>}
```

If you set `return_token_type_ids=True` when calling the tokenizer, you will also get an extra tensor that indicates which sentence each token belongs to. This is needed by some models, but not DistilBERT.

Next, we can directly pass this `BatchEncoding` object to the model: it returns a `TFSequenceClassifierOutput` object containing its predicted class logits:

```
>>> outputs = model(token_ids)
>>> outputs
TFSequenceClassifierOutput(loss=None, logits=[<tf.Tensor: [...] numpy=
array([[-2.1123817, 1.1786783, 1.4101017],
       [-0.01478387, 1.0962474, -0.9919954]], dtype=float32)>], [...])
```

Lastly, we can apply the softmax activation function to convert these logits to class probabilities, and use the `argmax()` function to predict the class with the highest probability, for each input sentence pair:

```
>>> Y_probas = tf.keras.activations.softmax(outputs.logits)
>>> Y_probas
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[0.01619702, 0.43523544, 0.5485676],
       [0.08672056, 0.85204804, 0.06123142]], dtype=float32)>
>>> Y_pred = tf.argmax(Y_probas, axis=1)
>>> Y_pred # 0 = contradiction, 1 = entailment, 2 = neutral
<tf.Tensor: shape=(2,), dtype=int64, numpy=array([2, 1])>
```

In this example, the model correctly classifies the first sentence pair as neutral—the fact that I like soccer does not imply that everyone else does—and the second pair as an entailment—Joe must indeed be quite old.

If you wish to fine-tune this model on your own dataset, you can train the model as usual with Keras since it's just a regular Keras model with a few extra methods. However, because the model outputs logits instead of probabilities, you must use the `tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)` loss instead of the usual "sparse_categorical_crossentropy" loss. Moreover, the model does not support `BatchEncoding` inputs during training, so we must use its `data` attribute to get a regular dictionary instead.

```

sentences = [("Sky is blue", "Sky is red"), ("I love her", "She loves me")]
X_train = tokenizer(sentences, padding=True, return_tensors="tf").data
y_train = tf.constant([0, 2]) # contradiction, neutral
loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
model.compile(loss=loss, optimizer="adam", metrics=["accuracy"])
history = model.fit(X_train, y_train, epochs=2)

```

Hugging Face has also built a *Datasets library* that you can use to easily download a standard dataset (such as IMDb) or a custom one, and use it to fine-tune your model. It's similar to TensorFlow Datasets, but it also provides tools to perform common preprocessing tasks on the fly, such as masking. The list of datasets is available at <https://huggingface.co/datasets>.

This should get you started with Hugging Face's ecosystem. To learn more, you can head over to <https://huggingface.co/docs> for the documentation, which includes many tutorial notebooks, videos, the full API, and more. I also recommend you check out the O'Reilly book *Natural Language Processing with Transformers: Building Language Applications with Hugging Face* by Lewis Tunstall, Leandro von Werra, and Thomas Wolf—all from the Hugging Face team.

In the next chapter we will discuss how to learn deep representations in an unsupervised way using autoencoders, and we will use generative adversarial networks (GANs) to produce images and more!

Exercises

1. What are the pros and cons of using a stateful RNN versus a stateless RNN?
2. Why do people use encoder-decoder RNNs rather than plain sequence-to-sequence RNNs for automatic translation?
3. How can you deal with variable-length input sequences? What about variable-length output sequences?
4. What is beam search and why would you use it? What tool can you use to implement it?
5. What is an attention mechanism? How does it help?
6. What is the most important layer in the Transformer architecture? What is its purpose?
7. When would you need to use sampled softmax?
8. *Embedded Reber grammars* were used by Hochreiter and Schmidhuber in [their paper](#) about LSTMs. They are artificial grammars that produce strings such as “BPBTSXXVPSEPE.” Check out Jenny Orr’s [nice introduction](#) to this topic. Choose a particular embedded Reber grammar (such as the one represented on Jenny Orr’s page), then train an RNN to identify whether a string respects that

grammar or not. You will first need to write a function capable of generating a training batch containing about 50% strings that respect the grammar, and 50% that don't.

9. Train an encoder–decoder model that can convert a date string from one format to another (e.g., from “April 22, 2019” to “2019-04-22”).
10. Go through the example on Keras’ site for [Natural language image search with a Dual Encoder](#). You will learn how to build a model capable of representing both images and text within the same embedding space. This makes it possible to search for images using a text prompt, like in the CLIP model by OpenAI.
11. Use the Hugging Face Transformers library to download a pretrained language model capable of generating text (e.g., GPT), and try generating more convincing Shakespearean text. You will need to use the model’s `generate()` method—see Hugging Face’s documentation for more details.

Solutions to these exercises are available at the end of this chapter’s notebook, at <https://homl.info/colab3>.

Autoencoders, GANs, and Diffusion Models

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 17th chapter of the final book. The GitHub repo is <https://github.com/ageron/handsonml3>.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

Autoencoders are artificial neural networks capable of learning dense representations of the input data, called *latent representations* or *codings*, without any supervision (i.e., the training set is unlabeled). These codings typically have a much lower dimensionality than the input data, making autoencoders useful for dimensionality reduction (see [Chapter 8](#)), especially for visualization purposes. Autoencoders also act as feature detectors, and they can be used for unsupervised pretraining of deep neural networks (as we discussed in [Chapter 11](#)). Lastly, some autoencoders are *generative models*: they are capable of randomly generating new data that looks very similar to the training data. For example, you could train an autoencoder on pictures of faces, and it would then be able to generate new faces.

Generative adversarial networks (GANs) are also neural nets capable of generating data. In fact, they can generate pictures of faces so convincing that it is hard to believe the people they represent do not exist. You can judge so for yourself by visiting <https://thispersondoesnotexist.com/>, a website that shows faces generated by a GAN architecture called *StyleGAN*. You can also check out <https://thisrentaldoesno>

texist.com/ to see some generated Airbnb listings. GANs are now widely used for super resolution (increasing the resolution of an image), **colorization**, powerful image editing (e.g., replacing photo bombers with realistic background), turning a simple sketch into a photorealistic image, predicting the next frames in a video, augmenting a dataset (to train other models), generating other types of data (such as text, audio, and time series), identifying the weaknesses in other models to strengthen them, and more.

Lastly, *diffusion models* have also joined the generative learning party. In 2021, they managed to generate more diverse and higher quality images than GANs, while also being much easier to train. However, diffusion models are much slower to run.

Autoencoders, GANs, and diffusion models are all unsupervised, they all learn latent representations, they can all be used as generative models, and they have many similar applications. However, they work very differently:

- Autoencoders simply learn to copy their inputs to their outputs. This may sound like a trivial task, but we will see that constraining the network in various ways can make it rather difficult. For example, you can limit the size of the latent representations, or you can add noise to the inputs and train the network to recover the original inputs. These constraints prevent the autoencoder from trivially copying the inputs directly to the outputs, which forces it to learn efficient ways of representing the data. In short, the codings are byproducts of the autoencoder learning the identity function under some constraints.
- GANs are composed of two neural networks: a *generator* that tries to generate data that looks similar to the training data, and a *discriminator* that tries to tell real data from fake data. This architecture is very original in Deep Learning in that the generator and the discriminator compete against each other during training: the generator is often compared to a criminal trying to make realistic counterfeit money, while the discriminator is like the police investigator trying to tell real money from fake. *Adversarial training* (training competing neural networks) is widely considered as one of the most important ideas of the 2010s. In 2016, Yann LeCun even said that it was “the most interesting idea in the last 10 years in Machine Learning.”
- A *Denoising Diffusion Probabilistic Model* (DDPM) is trained to remove a tiny bit of noise from an image. If you then take an image entirely full of Gaussian noise, and repeatedly run the diffusion model on that image, a high quality image will gradually emerge, similar to the training images (but not identical).

In this chapter we will start by exploring in more depth how autoencoders work and how to use them for dimensionality reduction, feature extraction, unsupervised pretraining, or as generative models. This will naturally lead us to GANs. We will start by building a simple GAN to generate fake images, but we will see that training

is often quite difficult. We will discuss the main difficulties you will encounter with adversarial training, as well as some of the main techniques to work around these difficulties. And lastly we will build and train a DDPM and use it to generate images. Let's start with autoencoders!

Efficient Data Representations

Which of the following number sequences do you find the easiest to memorize?

- 40, 27, 25, 36, 81, 57, 10, 73, 19, 68
- 50, 48, 46, 44, 42, 40, 38, 36, 34, 32, 30, 28, 26, 24, 22, 20, 18, 16, 14

At first glance, it would seem that the first sequence should be easier, since it is much shorter. However, if you look carefully at the second sequence, you will notice that it is just the list of even numbers from 50 down to 14. Once you notice this pattern, the second sequence becomes much easier to memorize than the first because you only need to remember the pattern (i.e., decreasing even numbers) and the starting and ending numbers (i.e., 50 and 14). Note that if you could quickly and easily memorize very long sequences, you would not care much about the existence of a pattern in the second sequence. You would just learn every number by heart, and that would be that. The fact that it is hard to memorize long sequences is what makes it useful to recognize patterns, and hopefully this clarifies why constraining an autoencoder during training pushes it to discover and exploit patterns in the data.

The relationship between memory, perception, and pattern matching was [famously studied by William Chase and Herbert Simon in the early 1970s](#).¹ They observed that expert chess players were able to memorize the positions of all the pieces in a game by looking at the board for just five seconds, a task that most people would find impossible. However, this was only the case when the pieces were placed in realistic positions (from actual games), not when the pieces were placed randomly. Chess experts don't have a much better memory than you and I; they just see chess patterns more easily, thanks to their experience with the game. Noticing patterns helps them store information efficiently.

Just like the chess players in this memory experiment, an autoencoder looks at the inputs, converts them to an efficient latent representation, and then spits out something that (hopefully) looks very close to the inputs. An autoencoder is always composed of two parts: an *encoder* (or *recognition network*) that converts the inputs to a latent representation, followed by a *decoder* (or *generative network*) that converts the internal representation to the outputs (see [Figure 17-1](#)).

¹ William G. Chase and Herbert A. Simon, "Perception in Chess," *Cognitive Psychology* 4, no. 1 (1973): 55–81.

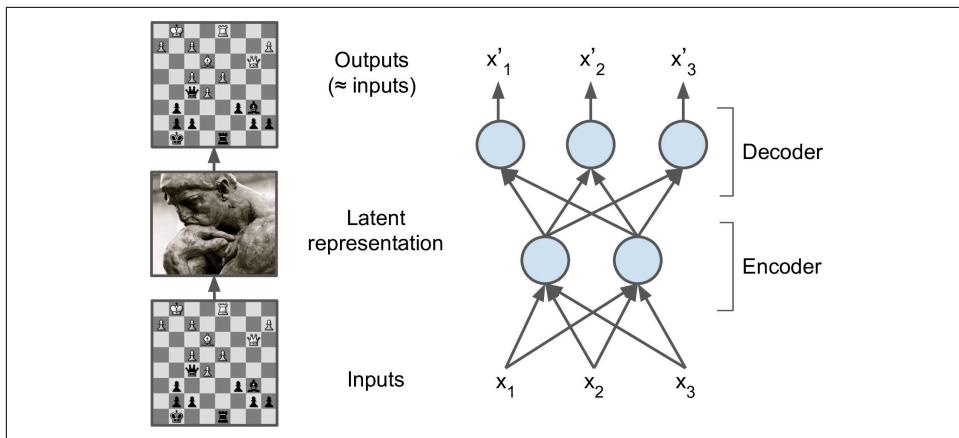


Figure 17-1. The chess memory experiment (left) and a simple autoencoder (right)

As you can see, an autoencoder typically has the same architecture as a Multi-Layer Perceptron (MLP; see [Chapter 10](#)), except that the number of neurons in the output layer must be equal to the number of inputs. In this example, there is just one hidden layer composed of two neurons (the encoder), and one output layer composed of three neurons (the decoder). The outputs are often called the *reconstructions* because the autoencoder tries to reconstruct the inputs. The cost function contains a *reconstruction loss* that penalizes the model when the reconstructions are different from the inputs.

Because the internal representation has a lower dimensionality than the input data (it is 2D instead of 3D), the autoencoder is said to be *undercomplete*. An undercomplete autoencoder cannot trivially copy its inputs to the codings, yet it must find a way to output a copy of its inputs. It is forced to learn the most important features in the input data (and drop the unimportant ones).

Let's see how to implement a very simple undercomplete autoencoder for dimensionality reduction.

Performing PCA with an Undercomplete Linear Autoencoder

If the autoencoder uses only linear activations and the cost function is the mean squared error (MSE), then it ends up performing Principal Component Analysis (PCA; see [Chapter 8](#)).

The following code builds a simple linear autoencoder to perform PCA on a 3D dataset, projecting it to 2D:

```

import tensorflow as tf

encoder = tf.keras.Sequential([tf.keras.layers.Dense(2)])
decoder = tf.keras.Sequential([tf.keras.layers.Dense(3)])
autoencoder = tf.keras.Sequential([encoder, decoder])

optimizer = tf.keras.optimizers.SGD(learning_rate=0.5)
autoencoder.compile(loss="mse", optimizer=optimizer)

```

This code is really not very different from all the MLPs we built in past chapters, but there are a few things to note:

- We organized the autoencoder into two subcomponents: the encoder and the decoder. Both are regular Sequential models with a single Dense layer each, and the autoencoder is a Sequential model containing the encoder followed by the decoder (remember that a model can be used as a layer in another model).
- The autoencoder's number of outputs is equal to the number of inputs (i.e., 3).
- To perform PCA, we do not use any activation function (i.e., all neurons are linear), and the cost function is the MSE. That's because PCA is a linear transformation. We will see more complex and non-linear autoencoders shortly.

Now let's train the model on the same simple generated 3D dataset we used in [Chapter 8](#) and use it to encode that dataset (i.e., project it to 2D):

```

X_train = [...] # generate a 3D dataset, like in Chapter 8
history = autoencoder.fit(X_train, X_train, epochs=500, verbose=False)
codings = encoder.predict(X_train)

```

Note that `X_train` is used as both the inputs and the targets. [Figure 17-2](#) shows the original 3D dataset (on the left) and the output of the autoencoder's hidden layer (i.e., the coding layer, on the right). As you can see, the autoencoder found the best 2D plane to project the data onto, preserving as much variance in the data as it could (just like PCA).

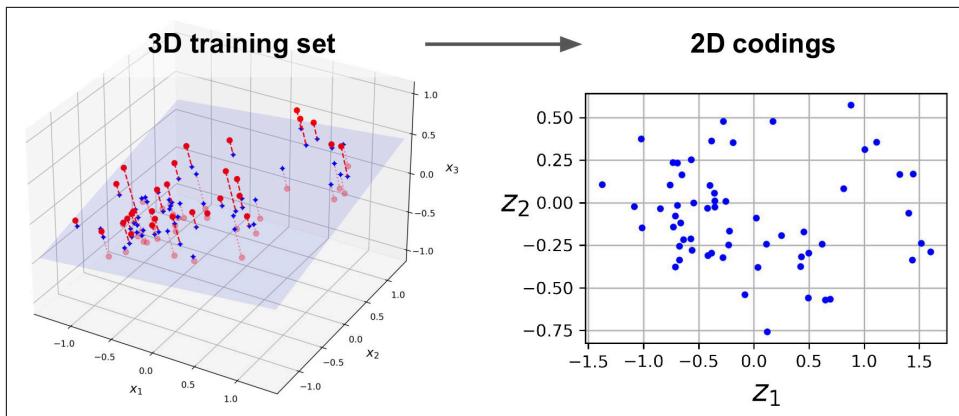


Figure 17-2. Approximate PCA performed by an undercomplete linear autoencoder



You can think of autoencoders as a form of self-supervised learning since it is based on a supervised learning technique with automatically generated labels, in this case simply equal to the inputs.

Stacked Autoencoders

Just like other neural networks we have discussed, autoencoders can have multiple hidden layers. In this case they are called *stacked autoencoders* (or *deep autoencoders*). Adding more layers helps the autoencoder learn more complex codings. That said, one must be careful not to make the autoencoder too powerful. Imagine an encoder so powerful that it just learns to map each input to a single arbitrary number (and the decoder learns the reverse mapping). Obviously such an autoencoder will reconstruct the training data perfectly, but it will not have learned any useful data representation in the process, and it is unlikely to generalize well to new instances.

The architecture of a stacked autoencoder is typically symmetrical with regard to the central hidden layer (the coding layer). To put it simply, it looks like a sandwich. For example, an autoencoder for Fashion MNIST (introduced in [Chapter 10](#)) may have 784 inputs, followed by a hidden layer with 100 neurons, then a central hidden layer of 30 neurons, then another hidden layer with 100 neurons, and an output layer with 784 neurons. This stacked autoencoder is represented in [Figure 17-3](#).

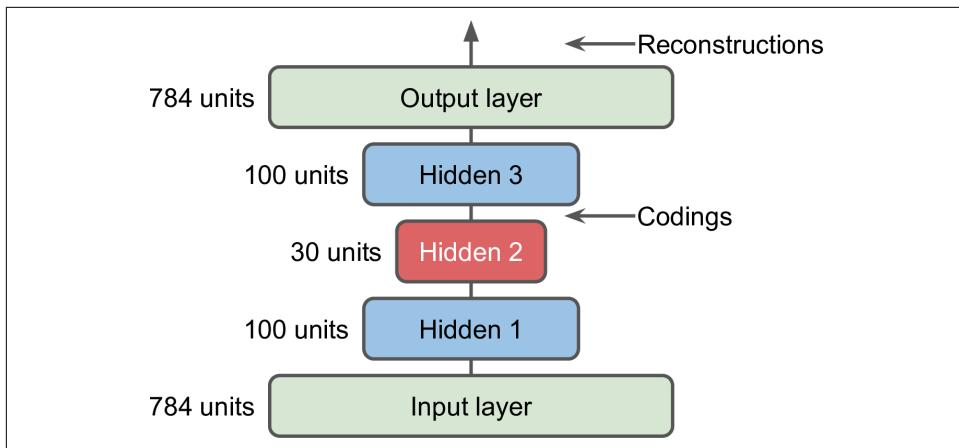


Figure 17-3. Stacked autoencoder

Implementing a Stacked Autoencoder Using Keras

You can implement a stacked autoencoder very much like a regular deep MLP:

```

stacked_encoder = tf.keras.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(30, activation="relu"),
])
stacked_decoder = tf.keras.Sequential([
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(28 * 28),
    tf.keras.layers.Reshape([28, 28])
])
stacked_ae = tf.keras.Sequential([stacked_encoder, stacked_decoder])

stacked_ae.compile(loss="mse", optimizer="adam")
history = stacked_ae.fit(X_train, X_train, epochs=20,
                         validation_data=(X_valid, X_valid))

```

Let's go through this code:

- Just like earlier, we split the autoencoder model into two submodels: the encoder and the decoder.
- The encoder takes 28×28 -pixel grayscale images, flattens them so that each image is represented as a vector of size 784, then processes these vectors through two `Dense` layers of diminishing sizes (100 units then 30 units), both using the ReLU activation function. For each input image, the encoder outputs a vector of size 30.

- The decoder takes codings of size 30 (output by the encoder) and processes them through two `Dense` layers of increasing sizes (100 units then 784 units), and it reshapes the final vectors into 28×28 arrays so the decoder's outputs have the same shape as the encoder's inputs.
- When compiling the stacked autoencoder, we use the MSE loss, and Nadam optimization.
- Finally, we train the model using `X_train` as both the inputs and the targets. Similarly, we use `X_valid` as both the validation inputs and targets.

Visualizing the Reconstructions

One way to ensure that an autoencoder is properly trained is to compare the inputs and the outputs: the differences should not be too significant. Let's plot a few images from the validation set, as well as their reconstructions:

```
import numpy as np

def plot_reconstructions(model, images=X_valid, n_images=5):
    reconstructions = np.clip(model.predict(images[:n_images]), 0, 1)
    fig = plt.figure(figsize=(n_images * 1.5, 3))
    for image_index in range(n_images):
        plt.subplot(2, n_images, 1 + image_index)
        plt.imshow(images[image_index], cmap="binary")
        plt.axis("off")
        plt.subplot(2, n_images, 1 + n_images + image_index)
        plt.imshow(reconstructions[image_index], cmap="binary")
        plt.axis("off")

plot_reconstructions(stacked_ae)
plt.show()
```

Figure 17-4 shows the resulting images.

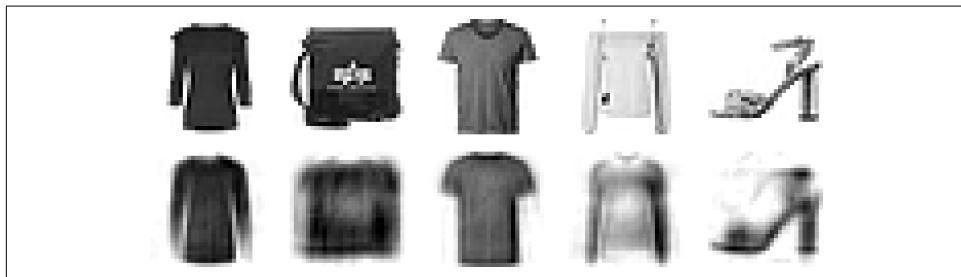


Figure 17-4. Original images (top) and their reconstructions (bottom)

The reconstructions are recognizable, but a bit too lossy. We may need to train the model for longer, or make the encoder and decoder deeper, or make the codings larger. But if we make the network too powerful, it will manage to make perfect

reconstructions without having learned any useful patterns in the data. For now, let's go with this model.

Visualizing the Fashion MNIST Dataset

Now that we have trained a stacked autoencoder, we can use it to reduce the dataset's dimensionality. For visualization, this does not give great results compared to other dimensionality reduction algorithms (such as those we discussed in [Chapter 8](#)), but one big advantage of autoencoders is that they can handle large datasets, with many instances and many features. So one strategy is to use an autoencoder to reduce the dimensionality down to a reasonable level, then use another dimensionality reduction algorithm for visualization. Let's use this strategy to visualize Fashion MNIST. First, we use the encoder from our stacked autoencoder to reduce the dimensionality down to 30, then we use Scikit-Learn's implementation of the t-SNE algorithm to reduce the dimensionality down to 2 for visualization:

```
from sklearn.manifold import TSNE

X_valid_compressed = stacked_encoder.predict(X_valid)
tsne = TSNE(init="pca", learning_rate="auto", random_state=42)
X_valid_2D = tsne.fit_transform(X_valid_compressed)
```

Now we can plot the dataset:

```
plt.scatter(X_valid_2D[:, 0], X_valid_2D[:, 1], c=y_valid, s=10, cmap="tab10")
plt.show()
```

[Figure 17-5](#) shows the resulting scatterplot, beautified a bit by displaying some of the images. The t-SNE algorithm identified several clusters which match the classes reasonably well (each class is represented by a different color).

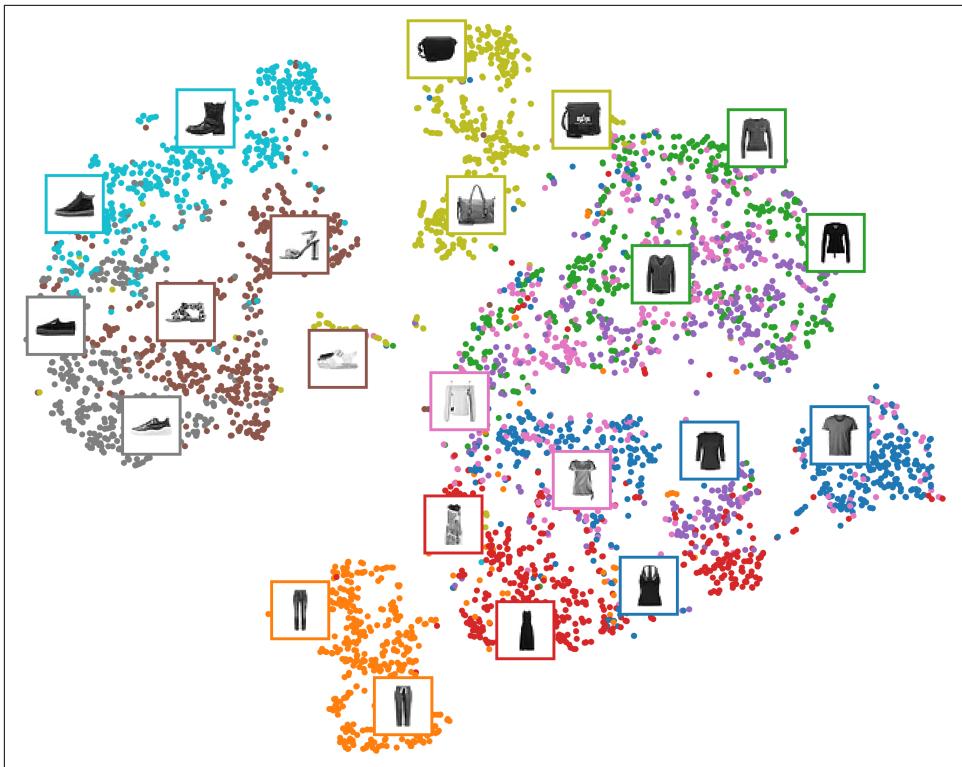


Figure 17-5. Fashion MNIST visualization using an autoencoder followed by t-SNE

So, autoencoders can be used for dimensionality reduction. Another application is for unsupervised pretraining.

Unsupervised Pretraining Using Stacked Autoencoders

As we discussed in [Chapter 11](#), if you are tackling a complex supervised task but you do not have a lot of labeled training data, one solution is to find a neural network that performs a similar task and reuse its lower layers. This makes it possible to train a high-performance model using little training data because your neural network won't have to learn all the low-level features; it will just reuse the feature detectors learned by the existing network.

Similarly, if you have a large dataset but most of it is unlabeled, you can first train a stacked autoencoder using all the data, then reuse the lower layers to create a neural network for your actual task and train it using the labeled data. For example, [Figure 17-6](#) shows how to use a stacked autoencoder to perform unsupervised pre-training for a classification neural network. When training the classifier, if you really

don't have much labeled training data, you may want to freeze the pretrained layers (at least the lower ones).

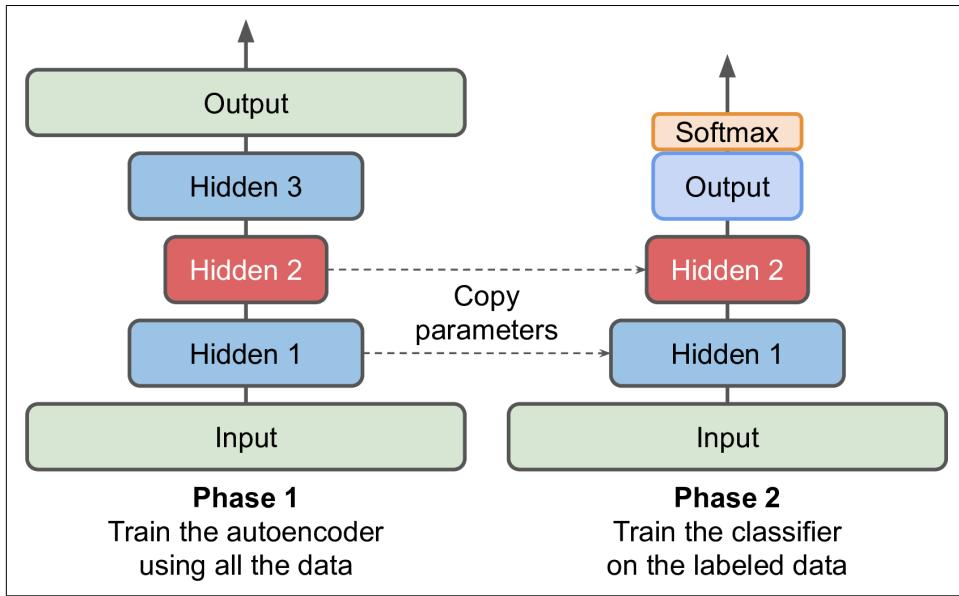


Figure 17-6. Unsupervised pretraining using autoencoders



Having plenty of unlabeled data and little labeled data is common. Building a large unlabeled dataset is often cheap (e.g., a simple script can download millions of images off the internet), but labeling those images (e.g., classifying them as cute or not) can usually be done reliably only by humans. Labeling instances is time-consuming and costly, so it's normal to have only a few thousand human-labeled instances, or even less.

There is nothing special about the implementation: just train an autoencoder using all the training data (labeled plus unlabeled), then reuse its encoder layers to create a new neural network (see the exercises at the end of this chapter for an example).

Next, let's look at a few techniques for training stacked autoencoders.

Tying Weights

When an autoencoder is neatly symmetrical, like the one we just built, a common technique is to *tie* the weights of the decoder layers to the weights of the encoder layers. This halves the number of weights in the model, speeding up training and limiting the risk of overfitting. Specifically, if the autoencoder has a total of N layers (not counting the input layer), and \mathbf{W}_L represents the connection weights of the L^{th}

layer (e.g., layer 1 is the first hidden layer, layer $N/2$ is the coding layer, and layer N is the output layer), then the decoder layer weights can be defined as: $\mathbf{W}_L = \mathbf{W}_{N-L+1}^\top$ (with $L = N/2+1, \dots, N$).

To tie weights between layers using Keras, let's define a custom layer:

```
class DenseTranspose(tf.keras.layers.Layer):
    def __init__(self, dense, activation=None, **kwargs):
        super().__init__(**kwargs)
        self.dense = dense
        self.activation = tf.keras.activations.get(activation)

    def build(self, batch_input_shape):
        self.biases = self.add_weight(name="bias",
                                      shape=self.dense.input_shape[-1],
                                      initializer="zeros")
        super().build(batch_input_shape)

    def call(self, inputs):
        Z = tf.matmul(inputs, self.dense.weights[0], transpose_b=True)
        return self.activation(Z + self.biases)
```

This custom layer acts like a regular Dense layer, but it uses another Dense layer's weights, transposed (setting `transpose_b=True` is equivalent to transposing the second argument, but it's more efficient as it performs the transposition on the fly within the `matmul()` operation). However, it uses its own bias vector. Next, we can build a new stacked autoencoder, much like the previous one, but with the decoder's Dense layers tied to the encoder's Dense layers:

```
dense_1 = tf.keras.layers.Dense(100, activation="relu")
dense_2 = tf.keras.layers.Dense(30, activation="relu")

tied_encoder = tf.keras.Sequential([
    tf.keras.layers.Flatten(),
    dense_1,
    dense_2
])

tied_decoder = tf.keras.Sequential([
    DenseTranspose(dense_2, activation="relu"),
    DenseTranspose(dense_1),
    tf.keras.layers.Reshape([28, 28])
])

tied_ae = tf.keras.Sequential([tied_encoder, tied_decoder])
```

This model achieves roughly the same reconstruction error as the previous model, using almost half the number of parameters.

Training One Autoencoder at a Time

Rather than training the whole stacked autoencoder in one go like we just did, it is possible to train one shallow autoencoder at a time, then stack all of them into a single stacked autoencoder (hence the name), as shown in Figure 17-7. This technique is not used as much these days, but you may still run into papers that talk about “greedy layerwise training,” so it’s good to know what it means.

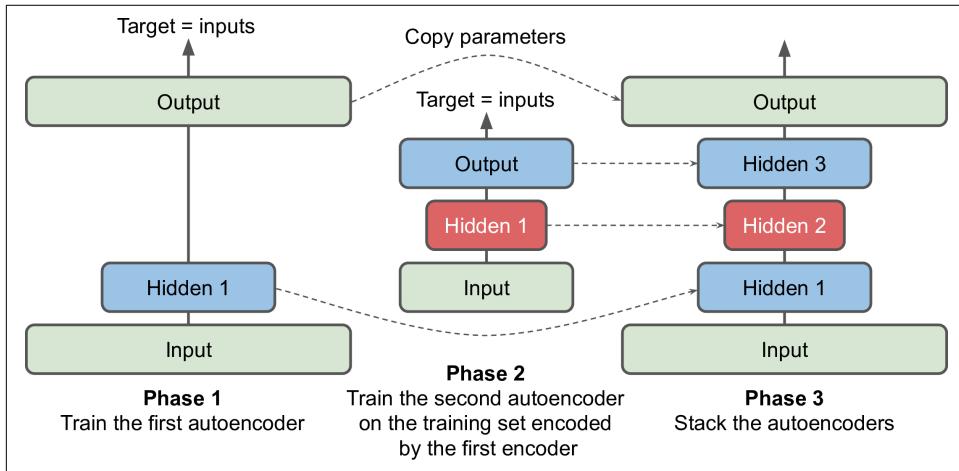


Figure 17-7. Training one autoencoder at a time

During the first phase of training, the first autoencoder learns to reconstruct the inputs. Then we encode the whole training set using this first autoencoder, and this gives us a new (compressed) training set. We then train a second autoencoder on this new dataset. This is the second phase of training. Finally, we build a big sandwich using all these autoencoders, as shown in Figure 17-7 (i.e., we first stack the hidden layers of each autoencoder, then the output layers in reverse order). This gives us the final stacked autoencoder (see the “Training One Autoencoder at a Time” section in the notebook for an implementation). We could easily train more autoencoders this way, building a very deep stacked autoencoder.

As we discussed earlier, one of the triggers of the Deep Learning tsunami was the discovery in 2006 by [Geoffrey Hinton et al.](#) that deep neural networks can be pretrained in an unsupervised fashion, using this greedy layerwise approach. They used restricted Boltzmann machines (RBMs; see <https://homl.info/extras>) for this purpose, but in [2007 Yoshua Bengio et al. showed](#)² that autoencoders worked just as well. For several years this was the only efficient way to train deep nets, until many of

² Yoshua Bengio et al., "Greedy Layer-Wise Training of Deep Networks," *Proceedings of the 19th International Conference on Neural Information Processing Systems* (2006): 153–160.

the techniques introduced in [Chapter 11](#) made it possible to just train a deep net in one shot.

Autoencoders are not limited to dense networks: you can also build convolutional autoencoders. Let's look at these now.

Convolutional Autoencoders

If you are dealing with images, then the autoencoders we have seen so far will not work well (unless the images are very small): as we saw in [Chapter 14](#), convolutional neural networks are far better suited than dense networks to work with images. So if you want to build an autoencoder for images (e.g., for unsupervised pretraining or dimensionality reduction), you will need to build a *convolutional autoencoder*.³ The encoder is a regular CNN composed of convolutional layers and pooling layers. It typically reduces the spatial dimensionality of the inputs (i.e., height and width) while increasing the depth (i.e., the number of feature maps). The decoder must do the reverse (upscale the image and reduce its depth back to the original dimensions), and for this you can use transpose convolutional layers (alternatively, you could combine upsampling layers with convolutional layers). Here is a basic convolutional autoencoder for Fashion MNIST:

```
conv_encoder = tf.keras.Sequential([
    tf.keras.layers.Reshape([28, 28, 1]),
    tf.keras.layers.Conv2D(16, 3, padding="same", activation="relu"),
    tf.keras.layers.MaxPool2D(pool_size=2), # output: 14 x 14 x 16
    tf.keras.layers.Conv2D(32, 3, padding="same", activation="relu"),
    tf.keras.layers.MaxPool2D(pool_size=2), # output: 7 x 7 x 32
    tf.keras.layers.Conv2D(64, 3, padding="same", activation="relu"),
    tf.keras.layers.MaxPool2D(pool_size=2), # output: 3 x 3 x 64
    tf.keras.layers.Conv2D(30, 3, padding="same", activation="relu"),
    tf.keras.layers.GlobalAvgPool2D() # output: 30
])
conv_decoder = tf.keras.Sequential([
    tf.keras.layers.Dense(3 * 3 * 16),
    tf.keras.layers.Reshape((3, 3, 16)),
    tf.keras.layers.Conv2DTranspose(32, 3, strides=2, activation="relu"),
    tf.keras.layers.Conv2DTranspose(16, 3, strides=2, padding="same",
                                  activation="relu"),
    tf.keras.layers.Conv2DTranspose(1, 3, strides=2, padding="same"),
    tf.keras.layers.Reshape([28, 28])
])
conv_ae = tf.keras.Sequential([conv_encoder, conv_decoder])
```

³ Jonathan Masci et al., “Stacked Convolutional Auto-Encoders for Hierarchical Feature Extraction,” *Proceedings of the 21st International Conference on Artificial Neural Networks* 1 (2011): 52–59.

It's also possible to create autoencoders with other architecture types, such as RNNs (see the notebook for an example).

OK, let's step back for a second. So far we have seen various kinds of autoencoders (basic, stacked, and convolutional), and we have looked at how to train them (either in one shot or layer by layer). We also looked at a couple applications: data visualization and unsupervised pretraining.

Up to now, in order to force the autoencoder to learn interesting features, we have limited the size of the coding layer, making it undercomplete. There are actually many other kinds of constraints that can be used, including ones that allow the coding layer to be just as large as the inputs, or even larger, resulting in an *overcomplete autoencoder*. So let's look at a few more kinds of autoencoders: denoising autoencoders, sparse autoencoders, and variational autoencoders.

Denoising Autoencoders

Another way to force the autoencoder to learn useful features is to add noise to its inputs, training it to recover the original, noise-free inputs. This idea has been around since the 1980s (e.g., it is mentioned in Yann LeCun's 1987 master's thesis). In a [2008 paper](#),⁴ Pascal Vincent et al. showed that autoencoders could also be used for feature extraction. In a [2010 paper](#),⁵ Vincent et al. introduced *stacked denoising autoencoders*.

The noise can be pure Gaussian noise added to the inputs, or it can be randomly switched-off inputs, just like in dropout (introduced in [Chapter 11](#)). [Figure 17-8](#) shows both options.

⁴ Pascal Vincent et al., "Extracting and Composing Robust Features with Denoising Autoencoders," *Proceedings of the 25th International Conference on Machine Learning* (2008): 1096–1103.

⁵ Pascal Vincent et al., "Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion," *Journal of Machine Learning Research* 11 (2010): 3371–3408.

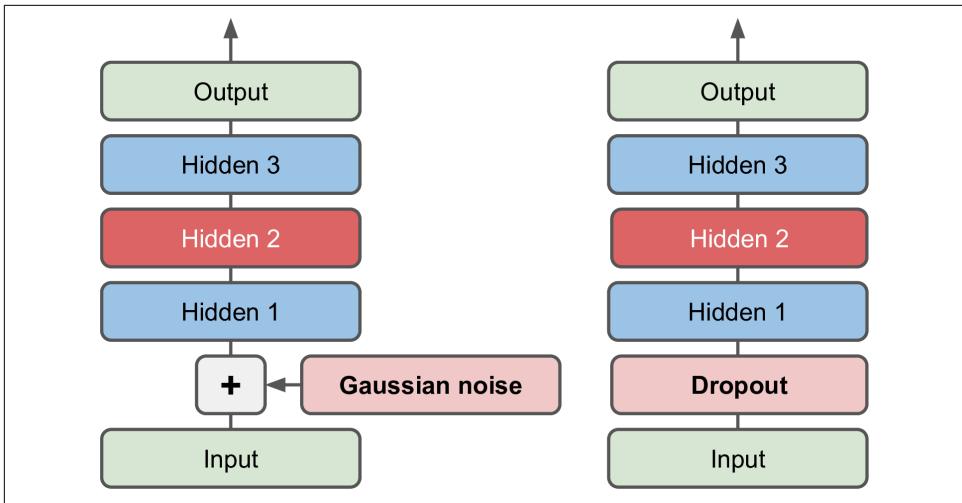


Figure 17-8. Denoising autoencoders, with Gaussian noise (left) or dropout (right)

The implementation is straightforward: it is a regular stacked autoencoder with an additional `Dropout` layer applied to the encoder's inputs (or you could use a `GaussianNoise` layer instead). Recall that the `Dropout` layer is only active during training (and so is the `GaussianNoise` layer):

```
dropout_encoder = tf.keras.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(30, activation="relu")
])
dropout_decoder = tf.keras.Sequential([
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(28 * 28),
    tf.keras.layers.Reshape([28, 28])
])
dropout_ae = tf.keras.Sequential([dropout_encoder, dropout_decoder])
```

Figure 17-9 shows a few noisy images (with half the pixels turned off), and the images reconstructed by the dropout-based denoising autoencoder. Notice how the autoencoder guesses details that are actually not in the input, such as the top of the white shirt (bottom row, fourth image). As you can see, not only can denoising autoencoders be used for data visualization or unsupervised pretraining, like the other autoencoders we've discussed so far, but they can also be used quite simply and efficiently to remove noise from images.

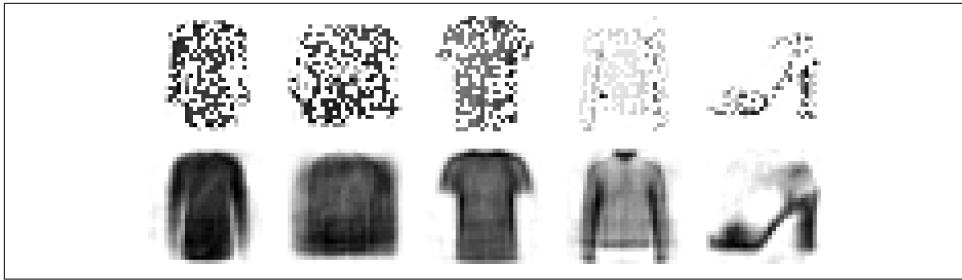


Figure 17-9. Noisy images (top) and their reconstructions (bottom)

Sparse Autoencoders

Another kind of constraint that often leads to good feature extraction is *sparsity*: by adding an appropriate term to the cost function, the autoencoder is pushed to reduce the number of active neurons in the coding layer. For example, it may be pushed to have on average only 5% significantly active neurons in the coding layer. This forces the autoencoder to represent each input as a combination of a small number of activations. As a result, each neuron in the coding layer typically ends up representing a useful feature (if you could speak only a few words per month, you would probably try to make them worth listening to).

A simple approach is to use the sigmoid activation function in the coding layer (to constrain the codings to values between 0 and 1), use a large coding layer (e.g., with 300 units), and add some ℓ_1 regularization to the coding layer's activations. The decoder is just a regular decoder.

```
sparse_l1_encoder = tf.keras.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(300, activation="sigmoid"),
    tf.keras.layers.ActivityRegularization(l1=1e-4)
])
sparse_l1_decoder = tf.keras.Sequential([
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(28 * 28),
    tf.keras.layers.Reshape([28, 28])
])
sparse_l1_ae = tf.keras.Sequential([sparse_l1_encoder, sparse_l1_decoder])
```

This `ActivityRegularization` layer just returns its inputs, but as a side effect it adds a training loss equal to the sum of absolute values of its inputs. This only affects training. Equivalently, you could remove the `ActivityRegularization` layer and set `activity_regularizer=tf.keras.regularizers.l1(1e-4)` in the previous layer. This penalty will encourage the neural network to produce codings close to 0, but since it will also be penalized if it does not reconstruct the inputs correctly, it will have to output at least a few nonzero values. Using the ℓ_1 norm rather than

the ℓ_2 norm will push the neural network to preserve the most important codings while eliminating the ones that are not needed for the input image (rather than just reducing all codings).

Another approach, which often yields better results, is to measure the actual sparsity of the coding layer at each training iteration, and penalize the model when the measured sparsity differs from a target sparsity. We do so by computing the average activation of each neuron in the coding layer, over the whole training batch. The batch size must not be too small, or else the mean will not be accurate.

Once we have the mean activation per neuron, we want to penalize the neurons that are too active, or not active enough, by adding a *sparsity loss* to the cost function. For example, if we measure that a neuron has an average activation of 0.3, but the target sparsity is 0.1, it must be penalized to activate less. One approach could be simply adding the squared error $(0.3 - 0.1)^2$ to the cost function, but in practice a better approach is to use the Kullback–Leibler (KL) divergence (briefly discussed in [Chapter 4](#)), which has much stronger gradients than the mean squared error, as you can see in [Figure 17-10](#).

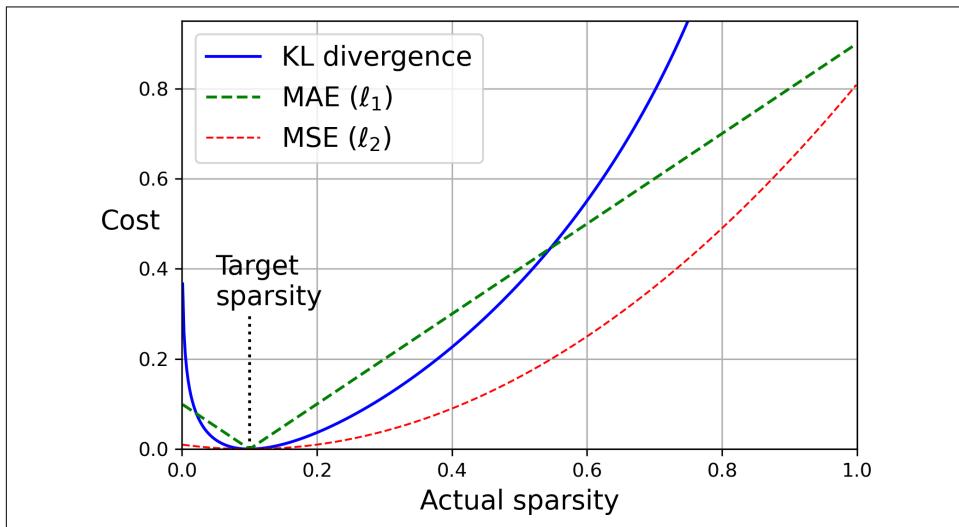


Figure 17-10. Sparsity loss

Given two discrete probability distributions P and Q , the KL divergence between these distributions, noted $D_{\text{KL}}(P \parallel Q)$, can be computed using [Equation 17-1](#).

Equation 17-1. Kullback–Leibler divergence

$$D_{\text{KL}}(P \parallel Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

In our case, we want to measure the divergence between the target probability p that a neuron in the coding layer will activate and the actual probability q , estimated by measuring the mean activation over the training batch. So the KL divergence simplifies to [Equation 17-2](#).

Equation 17-2. KL divergence between the target sparsity p and the actual sparsity q

$$D_{\text{KL}}(p \parallel q) = p \log \frac{p}{q} + (1 - p) \log \frac{1 - p}{1 - q}$$

Once we have computed the sparsity loss for each neuron in the coding layer, we sum up these losses and add the result to the cost function. In order to control the relative importance of the sparsity loss and the reconstruction loss, we can multiply the sparsity loss by a sparsity weight hyperparameter. If this weight is too high, the model will stick closely to the target sparsity, but it may not reconstruct the inputs properly, making the model useless. Conversely, if it is too low, the model will mostly ignore the sparsity objective and will not learn any interesting features.

We now have all we need to implement a sparse autoencoder based on the KL divergence. First, let's create a custom regularizer to apply KL divergence regularization:

```
kl_divergence = tf.keras.losses.kullback_leibler_divergence

class KLDivergenceRegularizer(tf.keras.regularizers.Regularizer):
    def __init__(self, weight, target):
        self.weight = weight
        self.target = target

    def __call__(self, inputs):
        mean_activities = tf.reduce_mean(inputs, axis=0)
        return self.weight * (
            kl_divergence(self.target, mean_activities) +
            kl_divergence(1. - self.target, 1. - mean_activities))
```

Now we can build the sparse autoencoder, using the `KLDivergenceRegularizer` for the coding layer's activations:

```
kld_reg = KLDivergenceRegularizer(weight=5e-3, target=0.1)
sparse_kl_encoder = tf.keras.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(300, activation="sigmoid",
                         activity_regularizer=kld_reg)
])
sparse_kl_decoder = tf.keras.Sequential([
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(28 * 28),
    tf.keras.layers.Reshape([28, 28])
```

```
])
sparse_kl_ae = tf.keras.Sequential([sparse_kl_encoder, sparse_kl_decoder])
```

After training this sparse autoencoder on Fashion MNIST, the coding layer will have roughly 10% sparsity.

Variational Autoencoders

An important category of autoencoders was [introduced in 2013](#) by Diederik Kingma and Max Welling and quickly became one of the most popular types of autoencoders: *variational autoencoders* (VAEs).⁶

They are quite different from all the autoencoders we have discussed so far, in these particular ways:

- They are *probabilistic autoencoders*, meaning that their outputs are partly determined by chance, even after training (as opposed to denoising autoencoders, which use randomness only during training).
- Most importantly, they are *generative autoencoders*, meaning that they can generate new instances that look like they were sampled from the training set.

Both these properties make them rather similar to RBMs, but they are easier to train, and the sampling process is much faster (with RBMs you need to wait for the network to stabilize into a “thermal equilibrium” before you can sample a new instance). As their name suggests, variational autoencoders perform variational Bayesian inference, which is an efficient way of carrying out approximate Bayesian inference. Recall that Bayesian inference means updating a probability distribution based on new data, using equations derived from Bayes’ theorem. The original distribution is called the *prior*, while the updated distribution is called the *posterior*. In our case, we want to find a good approximation of the data distribution. Once we have that, we can sample from it.

Let’s take a look at how VAEs work. [Figure 17-11](#) (left) shows a variational autoencoder. You can recognize the basic structure of all autoencoders, with an encoder followed by a decoder (in this example, they both have two hidden layers), but there is a twist: instead of directly producing a coding for a given input, the encoder produces a *mean coding* μ and a standard deviation σ . The actual coding is then sampled randomly from a Gaussian distribution with mean μ and standard deviation σ . After that the decoder decodes the sampled coding normally. The right part of the diagram shows a training instance going through this autoencoder. First, the encoder produces μ and σ , then a coding is sampled randomly (notice that it is not

⁶ Diederik Kingma and Max Welling, “Auto-Encoding Variational Bayes,” arXiv preprint arXiv:1312.6114 (2013).

exactly located at μ), and finally this coding is decoded; the final output resembles the training instance.

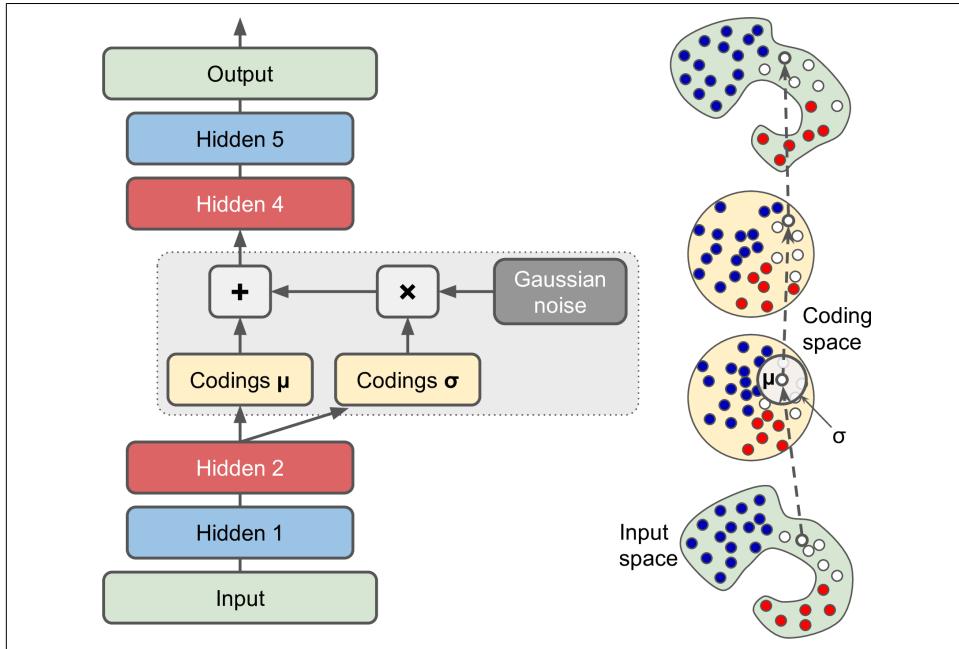


Figure 17-11. Variational autoencoder (left) and an instance going through it (right)

As you can see in the diagram, although the inputs may have a very convoluted distribution, a variational autoencoder tends to produce codings that look as though they were sampled from a simple Gaussian distribution:⁷ during training, the cost function (discussed next) pushes the codings to gradually migrate within the coding space (also called the *latent space*) to end up looking like a cloud of Gaussian points. One great consequence is that after training a variational autoencoder, you can very easily generate a new instance: just sample a random coding from the Gaussian distribution, decode it, and voilà!

Now, let's look at the cost function. It is composed of two parts. The first is the usual reconstruction loss that pushes the autoencoder to reproduce its inputs. We can use the MSE for this, as we did earlier. The second is the *latent loss* that pushes the autoencoder to have codings that look as though they were sampled from a simple Gaussian distribution: it is the KL divergence between the target distribution (i.e., the Gaussian distribution) and the actual distribution of the codings. The math is a bit more complex than with the sparse autoencoder, in particular because of the

⁷ Variational autoencoders are actually more general; the codings are not limited to Gaussian distributions.

Gaussian noise, which limits the amount of information that can be transmitted to the coding layer. This pushes the autoencoder to learn useful features. Luckily, the equations simplify, so the latent loss can be computed using [Equation 17-3](#):⁸

Equation 17-3. Variational autoencoder's latent loss

$$\mathcal{L} = -\frac{1}{2} \sum_{i=1}^n [1 + \log(\sigma_i^2) - \sigma_i^2 - \mu_i^2]$$

In this equation, \mathcal{L} is the latent loss, n is the codings' dimensionality, and μ_i and σ_i are the mean and standard deviation of the i^{th} component of the codings. The vectors μ and σ (which contain all the μ_i and σ_i) are output by the encoder, as shown in [Figure 17-11](#) (left).

A common tweak to the variational autoencoder's architecture is to make the encoder output $\gamma = \log(\sigma^2)$ rather than σ . The latent loss can then be computed as shown in [Equation 17-4](#). This approach is more numerically stable and speeds up training.

Equation 17-4. Variational autoencoder's latent loss, rewritten using $\gamma = \log(\sigma^2)$

$$\mathcal{L} = -\frac{1}{2} \sum_{i=1}^n [1 + \gamma_i - \exp(\gamma_i) - \mu_i^2]$$

Let's start building a variational autoencoder for Fashion MNIST (as shown in [Figure 17-11](#), but using the γ tweak). First, we will need a custom layer to sample the codings, given μ and γ :

```
class Sampling(tf.keras.layers.Layer):
    def call(self, inputs):
        mean, log_var = inputs
        return tf.random.normal(tf.shape(log_var)) * tf.exp(log_var / 2) + mean
```

This `Sampling` layer takes two inputs: `mean` (μ) and `log_var` (γ). It uses the function `tf.random.normal()` to sample a random vector (of the same shape as γ) from the Gaussian distribution, with mean 0 and standard deviation 1. Then it multiplies it by $\exp(\gamma / 2)$ (which is equal to σ , as you can verify mathematically), and finally it adds μ and returns the result. This samples a codings vector from the Gaussian distribution with mean μ and standard deviation σ .

Next, we can create the encoder, using the Functional API because the model is not entirely sequential:

⁸ For more mathematical details, check out the original paper on variational autoencoders, or Carl Doersch's [great tutorial](#) (2016).

```

codings_size = 10

inputs = tf.keras.layers.Input(shape=[28, 28])
Z = tf.keras.layers.Flatten()(inputs)
Z = tf.keras.layers.Dense(150, activation="relu")(Z)
Z = tf.keras.layers.Dense(100, activation="relu")(Z)
codings_mean = tf.keras.layers.Dense(codings_size)(Z) # μ
codings_log_var = tf.keras.layers.Dense(codings_size)(Z) # γ
codings = Sampling()([codings_mean, codings_log_var])
variational_encoder = tf.keras.Model(
    inputs=[inputs], outputs=[codings_mean, codings_log_var, codings])

```

Note that the Dense layers that output `codings_mean` (μ) and `codings_log_var` (γ) have the same inputs (i.e., the outputs of the second Dense layer). We then pass both `codings_mean` and `codings_log_var` to the Sampling layer. Finally, the `variational_encoder` model has three outputs. Only the `codings` are required, but we add `codings_mean` and `codings_log_var` as well, in case you want to inspect their values. Now let's build the decoder:

```

decoder_inputs = tf.keras.layers.Input(shape=[codings_size])
x = tf.keras.layers.Dense(100, activation="relu")(decoder_inputs)
x = tf.keras.layers.Dense(150, activation="relu")(x)
x = tf.keras.layers.Dense(28 * 28)(x)
outputs = tf.keras.layers.Reshape([28, 28])(x)
variational_decoder = tf.keras.Model(inputs=[decoder_inputs], outputs=[outputs])

```

For this decoder, we could have used the Sequential API instead of the Functional API, since it is really just a simple stack of layers, virtually identical to many of the decoders we have built so far. Finally, let's build the variational autoencoder model:

```

_, _, codings = variational_encoder(inputs)
reconstructions = variational_decoder(codings)
variational_ae = tf.keras.Model(inputs=[inputs], outputs=[reconstructions])

```

Note that we ignore the first two outputs of the encoder (we only want to feed the codings to the decoder). Lastly, we must add the latent loss and the reconstruction loss:

```

latent_loss = -0.5 * tf.reduce_sum(
    1 + codings_log_var - tf.exp(codings_log_var) - tf.square(codings_mean),
    axis=-1)
variational_ae.add_loss(tf.reduce_mean(latent_loss) / 784.)

```

We first apply [Equation 17-4](#) to compute the latent loss for each instance in the batch, summing over the last axis. Then we compute the mean loss over all the instances in the batch, and we divide the result by 784 to ensure it has the appropriate scale compared to the reconstruction loss. Indeed, the variational autoencoder's reconstruction loss is supposed to be the sum of the pixel reconstruction errors, but when Keras computes the "mse" loss, it computes the mean over all 784 pixels, rather than the sum. So, the reconstruction loss is 784 times smaller than we need it to be. We could

define a custom loss to compute the sum rather than the mean, but it is simpler to divide the latent loss by 784 (the final loss will be 784 times smaller than it should be, but this just means that we should use a larger learning rate).

And finally we can compile and fit the autoencoder!

```
variational_ae.compile(loss="mse", optimizer="adam")
history = variational_ae.fit(X_train, X_train, epochs=25, batch_size=128,
                             validation_data=(X_valid, X_valid))
```

Generating Fashion MNIST Images

Now let's use this variational autoencoder to generate images that look like fashion items. All we need to do is sample random codings from a Gaussian distribution and decode them:

```
codings = tf.random.normal(shape=[3 * 7, codings_size])
images = variational_decoder(codings).numpy()
```

Figure 17-12 shows the 12 generated images.

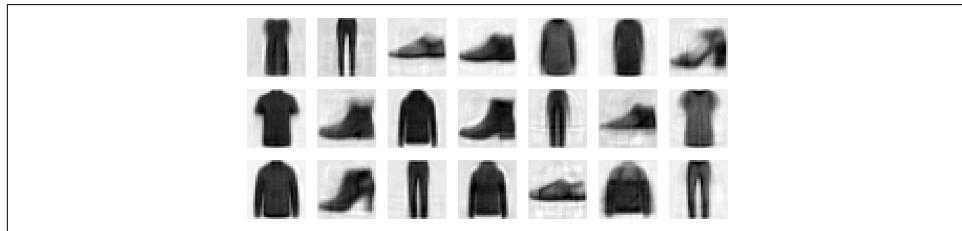


Figure 17-12. Fashion MNIST images generated by the variational autoencoder

The majority of these images look fairly convincing, if a bit too fuzzy. The rest are not great, but don't be too harsh on the autoencoder—it only had a few minutes to learn!

Variational autoencoders make it possible to perform *semantic interpolation*: instead of interpolating between two images at the pixel level, which would look as if the two images were just overlaid, we can interpolate at the codings level. For example, let's take a few codings along an arbitrary line in latent space, and decode them. We get a sequence of images that gradually go from pants to sweaters (see Figure 17-13):

```
codings = np.zeros([7, codings_size])
codings[:, 3] = np.linspace(-0.8, 0.8, 7) # axis 3 looks best in this case
images = variational_decoder(codings).numpy()
```

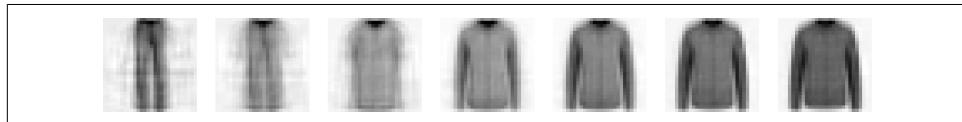


Figure 17-13. Semantic interpolation

Let's now turn our attention to GANs: they are harder to train, but when you manage to get them to work, they produce pretty amazing images.

Generative Adversarial Networks

Generative adversarial networks were proposed in a [2014 paper](#)⁹ by Ian Goodfellow et al., and although the idea got researchers excited almost instantly, it took a few years to overcome some of the difficulties of training GANs. Like many great ideas, it seems simple in hindsight: make neural networks compete against each other in the hope that this competition will push them to excel. As shown in [Figure 17-14](#), a GAN is composed of two neural networks:

Generator

Takes a random distribution as input (typically Gaussian) and outputs some data—typically, an image. You can think of the random inputs as the latent representations (i.e., codings) of the image to be generated. So, as you can see, the generator offers the same functionality as a decoder in a variational autoencoder, and it can be used in the same way to generate new images: just feed it some Gaussian noise, and it outputs a brand-new image. However, it is trained very differently, as we will soon see.

Discriminator

Takes either a fake image from the generator or a real image from the training set as input, and must guess whether the input image is fake or real.

⁹ Ian Goodfellow et al., “Generative Adversarial Nets,” *Proceedings of the 27th International Conference on Neural Information Processing Systems 2* (2014): 2672–2680.

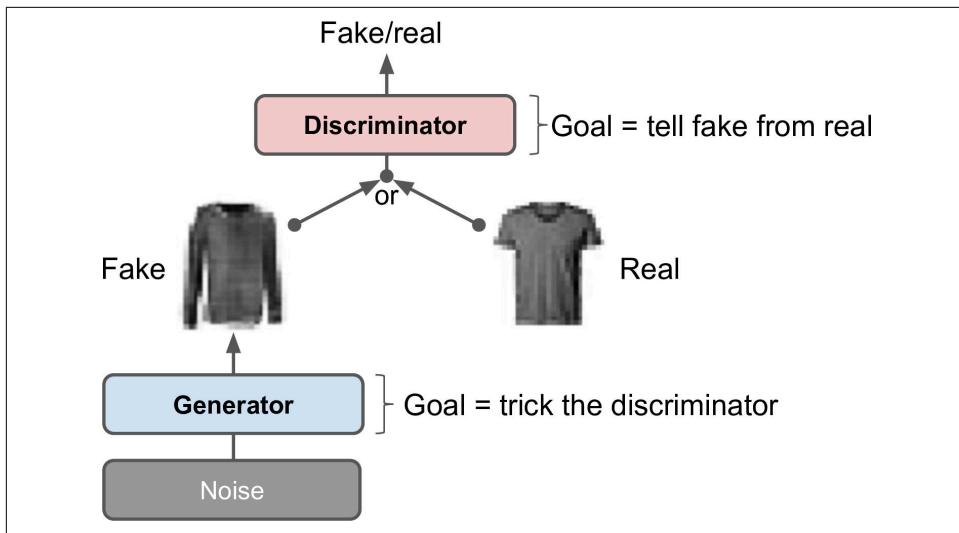


Figure 17-14. A generative adversarial network

During training, the generator and the discriminator have opposite goals: the discriminator tries to tell fake images from real images, while the generator tries to produce images that look real enough to trick the discriminator. Because the GAN is composed of two networks with different objectives, it cannot be trained like a regular neural network. Each training iteration is divided into two phases:

- In the first phase, we train the discriminator. A batch of real images is sampled from the training set and is completed with an equal number of fake images produced by the generator. The labels are set to 0 for fake images and 1 for real images, and the discriminator is trained on this labeled batch for one step, using the binary cross-entropy loss. Importantly, backpropagation only optimizes the weights of the discriminator during this phase.
- In the second phase, we train the generator. We first use it to produce another batch of fake images, and once again the discriminator is used to tell whether the images are fake or real. This time we do not add real images in the batch, and all the labels are set to 1 (real): in other words, we want the generator to produce images that the discriminator will (wrongly) believe to be real! Crucially, the weights of the discriminator are frozen during this step, so backpropagation only affects the weights of the generator.



The generator never actually sees any real images, yet it gradually learns to produce convincing fake images! All it gets is the gradients flowing back through the discriminator. Fortunately, the better the discriminator gets, the more information about the real images is contained in these secondhand gradients, so the generator can make significant progress.

Let's go ahead and build a simple GAN for Fashion MNIST.

First, we need to build the generator and the discriminator. The generator is similar to an autoencoder's decoder, and the discriminator is a regular binary classifier: it takes an image as input and ends with a Dense layer containing a single unit and using the sigmoid activation function. For the second phase of each training iteration, we also need the full GAN model containing the generator followed by the discriminator:

```
codings_size = 30

Dense = tf.keras.layers.Dense
generator = tf.keras.Sequential([
    Dense(100, activation="relu", kernel_initializer="he_normal"),
    Dense(150, activation="relu", kernel_initializer="he_normal"),
    Dense(28 * 28, activation="sigmoid"),
    tf.keras.layers.Reshape([28, 28])
])
discriminator = tf.keras.Sequential([
    tf.keras.layers.Flatten(),
    Dense(150, activation="relu", kernel_initializer="he_normal"),
    Dense(100, activation="relu", kernel_initializer="he_normal"),
    Dense(1, activation="sigmoid")
])
gan = tf.keras.Sequential([generator, discriminator])
```

Next, we need to compile these models. As the discriminator is a binary classifier, we can naturally use the binary cross-entropy loss. The gan model is also a binary classifier, so it can use the binary cross-entropy loss as well. However, the generator will only be trained through the gan model, so we do not need to compile it at all. Importantly, the discriminator should not be trained during the second phase, so we make it non-trainable before compiling the gan model:

```
discriminator.compile(loss="binary_crossentropy", optimizer="rmsprop")
discriminator.trainable = False
gan.compile(loss="binary_crossentropy", optimizer="rmsprop")
```



The `trainable` attribute is taken into account by Keras only when compiling a model, so after running this code, the discriminator is trainable if we call its `fit()` method or its `train_on_batch()` method (which we will be using), while it is *not* trainable when we call these methods on the `gan` model.

Since the training loop is unusual, we cannot use the regular `fit()` method. Instead, we will write a custom training loop. For this, we first need to create a `Dataset` to iterate through the images:

```
batch_size = 32
dataset = tf.data.Dataset.from_tensor_slices(X_train).shuffle(buffer_size=1000)
dataset = dataset.batch(batch_size, drop_remainder=True).prefetch(1)
```

We are now ready to write the training loop. Let's wrap it in a `train_gan()` function:

```
def train_gan(gan, dataset, batch_size, codings_size, n_epochs):
    generator, discriminator = gan.layers
    for epoch in range(n_epochs):
        for X_batch in dataset:
            # phase 1 - training the discriminator
            noise = tf.random.normal(shape=[batch_size, codings_size])
            generated_images = generator(noise)
            X_fake_and_real = tf.concat([generated_images, X_batch], axis=0)
            y1 = tf.constant([[0.]] * batch_size + [[1.]] * batch_size)
            discriminator.train_on_batch(X_fake_and_real, y1)
            # phase 2 - training the generator
            noise = tf.random.normal(shape=[batch_size, codings_size])
            y2 = tf.constant([[1.]] * batch_size)
            gan.train_on_batch(noise, y2)

train_gan(gan, dataset, batch_size, codings_size, n_epochs=50)
```

As discussed earlier, you can see the two phases at each iteration:

- In phase one we feed Gaussian noise to the generator to produce fake images, and we complete this batch by concatenating an equal number of real images. The targets `y1` are set to 0 for fake images and 1 for real images. Then we train the discriminator on this batch. Remember that the discriminator is trainable in this phase, but we are not touching the generator.
- In phase two, we feed the GAN some Gaussian noise. Its generator will start by producing fake images, then the discriminator will try to guess whether these images are fake or real. In this phase, we are trying to improve the generator, which means that we want the discriminator to fail: this is why the targets `y2` are all set to 1, although the images are fake. In this phase, the discriminator is *not* trainable, so the only part of the `gan` model that will improve is the generator.

That's it! After training, you can randomly sample some codings from a Gaussian distribution, and feed them to the generator to produce new images:

```
codings = tf.random.normal(shape=[batch_size, codings_size])
generated_images = generator.predict(codings)
```

If you display the generated images (see [Figure 17-15](#)), you will see that at the end of the first epoch, they already start to look like (very noisy) Fashion MNIST images.



Figure 17-15. Images generated by the GAN after one epoch of training

Unfortunately, the images never really get much better than that, and you may even find epochs where the GAN seems to be forgetting what it learned. Why is that? Well, it turns out that training a GAN can be challenging. Let's see why.

The Difficulties of Training GANs

During training, the generator and the discriminator constantly try to outsmart each other, in a zero-sum game. As training advances, the game may end up in a state that game theorists call a *Nash equilibrium*, named after the mathematician John Nash: this is when no player would be better off changing their own strategy, assuming the other players do not change theirs. For example, a Nash equilibrium is reached when everyone drives on the left side of the road: no driver would be better off being the only one to switch sides. Of course, there is a second possible Nash equilibrium: when everyone drives on the *right* side of the road. Different initial states and dynamics may lead to one equilibrium or the other. In this example, there is a single optimal strategy once an equilibrium is reached (i.e., driving on the same side as everyone else), but a Nash equilibrium can involve multiple competing strategies (e.g., a predator chases its prey, the prey tries to escape, and neither would be better off changing their strategy).

So how does this apply to GANs? Well, the authors of the paper demonstrated that a GAN can only reach a single Nash equilibrium: that's when the generator produces perfectly realistic images, and the discriminator is forced to guess (50% real, 50% fake). This fact is very encouraging: it would seem that you just need to train the GAN for long enough, and it will eventually reach this equilibrium, giving you a perfect generator. Unfortunately, it's not that simple: nothing guarantees that the equilibrium will ever be reached.

The biggest difficulty is called *mode collapse*: this is when the generator's outputs gradually become less diverse. How can this happen? Suppose that the generator gets better at producing convincing shoes than any other class. It will fool the discriminator a bit more with shoes, and this will encourage it to produce even more images of shoes. Gradually, it will forget how to produce anything else. Meanwhile, the only fake images that the discriminator will see will be shoes, so it will also forget how to discriminate fake images of other classes. Eventually, when the discriminator manages to discriminate the fake shoes from the real ones, the generator will be forced to move to another class. It may then become good at shirts, forgetting about shoes, and the discriminator will follow. The GAN may gradually cycle across a few classes, never really becoming very good at any of them.

Moreover, because the generator and the discriminator are constantly pushing against each other, their parameters may end up oscillating and becoming unstable. Training may begin properly, then suddenly diverge for no apparent reason, due to these instabilities. And since many factors affect these complex dynamics, GANs are very sensitive to the hyperparameters: you may have to spend a lot of effort fine-tuning them. In fact, that's why I used RMSProp rather than Nadam when compiling the models: when using Nadam, I ran into a severe mode collapse.

These problems have kept researchers very busy since 2014: many papers were published on this topic, some proposing new cost functions¹⁰ (though a 2018 paper¹¹ by Google researchers questions their efficiency) or techniques to stabilize training or to avoid the mode collapse issue. For example, a popular technique called *experience replay* consists in storing the images produced by the generator at each iteration in a replay buffer (gradually dropping older generated images) and training the discriminator using real images plus fake images drawn from this buffer (rather than just fake images produced by the current generator). This reduces the chances that the discriminator will overfit the latest generator's outputs. Another common technique is called *mini-batch discrimination*: it measures how similar images are across the batch and provides this statistic to the discriminator, so it can easily reject a whole

¹⁰ For a nice comparison of the main GAN losses, check out this great GitHub project by Hwalsuk Lee.

¹¹ Mario Lucic et al., "Are GANs Created Equal? A Large-Scale Study," *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (2018): 698–707.

batch of fake images that lack diversity. This encourages the generator to produce a greater variety of images, reducing the chance of mode collapse. Other papers simply propose specific architectures that happen to perform well.

In short, this is still a very active field of research, and the dynamics of GANs are still not perfectly understood. But the good news is that great progress has been made, and some of the results are truly astounding! So let's look at some of the most successful architectures, starting with deep convolutional GANs, which were the state of the art just a few years ago. Then we will look at two more recent (and more complex) architectures.

Deep Convolutional GANs

The original GAN paper in 2014 experimented with convolutional layers, but only tried to generate small images. Soon after, many researchers tried to build GANs based on deeper convolutional nets for larger images. This proved to be tricky, as training was very unstable, but Alec Radford et al. finally succeeded in late 2015, after experimenting with many different architectures and hyperparameters. They called their architecture *deep convolutional GANs* (DCGANs).¹² Here are the main guidelines they proposed for building stable convolutional GANs:

- Replace any pooling layers with strided convolutions (in the discriminator) and transposed convolutions (in the generator).
- Use Batch Normalization in both the generator and the discriminator, except in the generator's output layer and the discriminator's input layer.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in the generator for all layers except the output layer, which should use tanh.
- Use leaky ReLU activation in the discriminator for all layers.

These guidelines will work in many cases, but not always, so you may still need to experiment with different hyperparameters. In fact, just changing the random seed and training the exact same model again will sometimes work. Here is a small DCGAN that works reasonably well with Fashion MNIST:

```
codings_size = 100

generator = tf.keras.Sequential([
    tf.keras.layers.Dense(7 * 7 * 128),
    tf.keras.layers.Reshape([7, 7, 128]),
```

¹² Alec Radford et al., “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks,” arXiv preprint arXiv:1511.06434 (2015).

```

        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.Conv2DTranspose(64, kernel_size=5, strides=2,
                                      padding="same", activation="relu"),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.Conv2DTranspose(1, kernel_size=5, strides=2,
                                      padding="same", activation="tanh"),
    ])
discriminator = tf.keras.Sequential([
    tf.keras.layers.Conv2D(64, kernel_size=5, strides=2, padding="same",
                          activation=tf.keras.layers.LeakyReLU(0.2)),
    tf.keras.layers.Dropout(0.4),
    tf.keras.layers.Conv2D(128, kernel_size=5, strides=2, padding="same",
                          activation=tf.keras.layers.LeakyReLU(0.2)),
    tf.keras.layers.Dropout(0.4),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(1, activation="sigmoid")
])
gan = tf.keras.Sequential([generator, discriminator])

```

The generator takes codings of size 100, and it projects them to 6272 dimensions ($7 \times 7 \times 128$), and reshapes the result to get a $7 \times 7 \times 128$ tensor. This tensor is batch normalized and fed to a transposed convolutional layer with a stride of 2, which upsamples it from 7×7 to 14×14 and reduces its depth from 128 to 64. The result is batch normalized again and fed to another transposed convolutional layer with a stride of 2, which upsamples it from 14×14 to 28×28 and reduces the depth from 64 to 1. This layer uses the tanh activation function, so the outputs will range from -1 to 1. For this reason, before training the GAN, we need to rescale the training set to that same range. We also need to reshape it to add the channel dimension:

```
X_train_dcgan = X_train.reshape(-1, 28, 28, 1) * 2. - 1. # reshape and rescale
```

The discriminator looks much like a regular CNN for binary classification, except instead of using max pooling layers to downsample the image, we use strided convolutions (`strides=2`). Also note that we use the leaky ReLU activation function.

Overall, we respected the DCGAN guidelines, except we replaced the `BatchNormalization` layers in the discriminator with `Dropout` layers, otherwise training was unstable in this case. Feel free to tweak this architecture: you will see how sensitive it is to the hyperparameters, especially the relative learning rates of the two networks.

Lastly, to build the dataset, then compile and train this model, we use the same code as earlier. After 50 epochs of training, the generator produces images like those shown in [Figure 17-16](#). It's still not perfect, but many of these images are pretty convincing.



Figure 17-16. Images generated by the DCGAN after 50 epochs of training

If you scale up this architecture and train it on a large dataset of faces, you can get fairly realistic images. In fact, DCGANs can learn quite meaningful latent representations, as you can see in [Figure 17-17](#): many images were generated, and nine of them were picked manually (top left), including three representing men with glasses, three men without glasses, and three women without glasses. For each of these categories, the codings that were used to generate the images were averaged, and an image was generated based on the resulting mean codings (lower left). In short, each of the three lower-left images represents the mean of the three images located above it. But this is not a simple mean computed at the pixel level (this would result in three overlapping faces), it is a mean computed in the latent space, so the images still look like normal faces. Amazingly, if you compute men with glasses, minus men without glasses, plus women without glasses—where each term corresponds to one of the mean codings—and you generate the image that corresponds to this coding, you get the image at the center of the 3×3 grid of faces on the right: a woman with glasses! The eight other images around it were generated based on the same vector plus a bit of noise, to illustrate the semantic interpolation capabilities of DCGANs. Being able to do arithmetic on faces feels like science fiction!

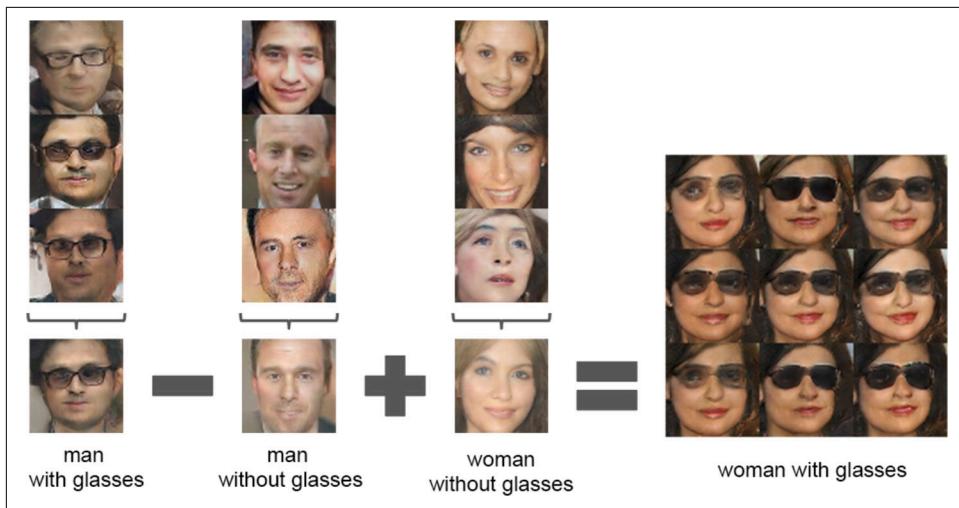


Figure 17-17. Vector arithmetic for visual concepts (part of figure 7 from the DCGAN paper)¹³



If you add each image's class as an extra input to both the generator and the discriminator, they will both learn what each class looks like, and thus you will be able to control the class of each image produced by the generator. This is called a *conditional GAN*¹⁴ (CGAN).

DCGANs aren't perfect, though. For example, when you try to generate very large images using DCGANs, you often end up with locally convincing features but overall inconsistencies, such as shirts with one sleeve much longer than the other, different earings, or eyes looking in opposite directions. How can you fix this?

Progressive Growing of GANs

An important technique was proposed in a [2018 paper](#)¹⁵ by Nvidia researchers Tero Karras et al.: they suggested generating small images at the beginning of training, then gradually adding convolutional layers to both the generator and the discriminator to produce larger and larger images (4×4 , 8×8 , 16×16 , ..., 512×512 , $1,024 \times 1,024$). This approach resembles greedy layer-wise training of stacked autoencoders.

¹³ Reproduced with the kind authorization of the authors.

¹⁴ Mehdi Mirza and Simon Osindero, "Conditional Generative Adversarial Nets," arXiv preprint arXiv:1411.1784 (2014).

¹⁵ Tero Karras et al., "Progressive Growing of GANs for Improved Quality, Stability, and Variation," *Proceedings of the International Conference on Learning Representations* (2018).

The extra layers get added at the end of the generator and at the beginning of the discriminator, and previously trained layers remain trainable.

For example, when growing the generator's outputs from 4×4 to 8×8 (see [Figure 17-18](#)), an upsampling layer (using nearest neighbor filtering) is added to the existing convolutional layer ("Conv 1") to produce 8×8 feature maps. These are fed to the new convolutional layer ("Conv 2"), which in turn feeds into a new output convolutional layer. To avoid breaking the trained weights of "Conv 1", we gradually fade-in the two new convolutional layers (represented with dashed lines in [Figure 17-18](#)) and fade-out the original output layer. To do this, the final outputs are a weighted sum of the new outputs (with weight α) and the original outputs (with weight $1 - \alpha$), slowly increasing α from 0 to 1. A similar fade-in/fade-out technique is used when a new convolutional layer is added to the discriminator (followed by an average pooling layer for downsampling). Note that all convolutional layers use "same" padding and strides of 1, so they preserve the height and width of their inputs. This includes the original convolutional layer, so it now produces 8×8 outputs (since its inputs are now 8×8). Lastly, the output layers use kernel size 1. They just project their inputs down to the desired number of color channels (typically 3).

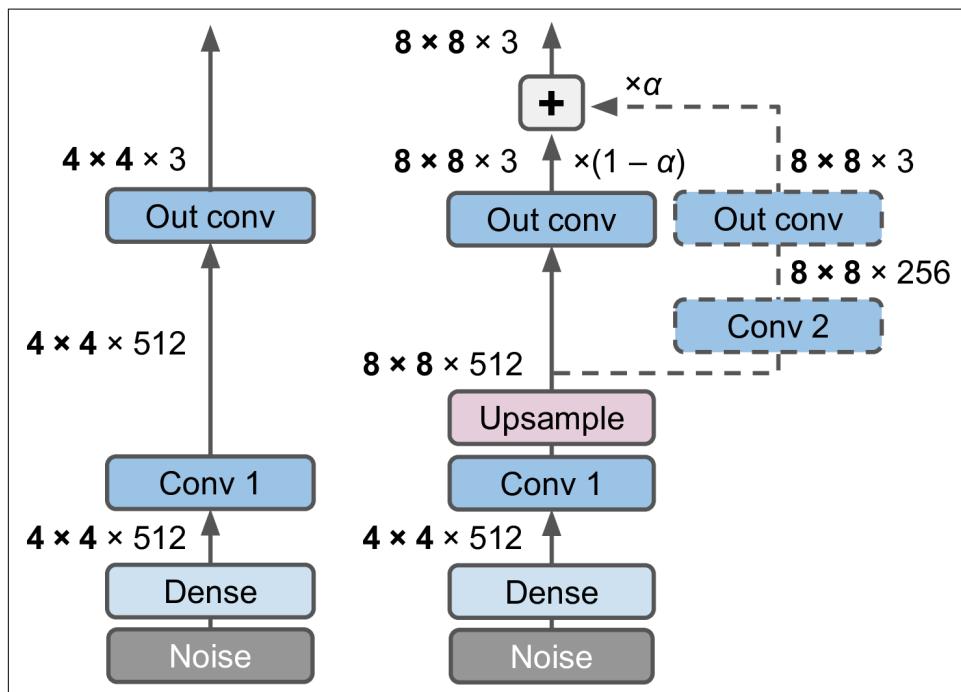


Figure 17-18. Progressively growing GAN: a GAN generator outputs 4×4 color images (left); we extend it to output 8×8 images (right)

The paper also introduced several other techniques aimed at increasing the diversity of the outputs (to avoid mode collapse) and making training more stable:

Minibatch standard deviation layer

Added near the end of the discriminator. For each position in the inputs, it computes the standard deviation across all channels and all instances in the batch ($S = \text{tf.math.reduce_std(inputs, axis=[0, -1])}$). These standard deviations are then averaged across all points to get a single value ($v = \text{tf.reduce_mean}(S)$). Finally, an extra feature map is added to each instance in the batch and filled with the computed value ($\text{tf.concat}([\text{inputs}, \text{tf.fill}([\text{batch_size}, \text{height}, \text{width}, 1], v)], \text{axis}=-1)$). How does this help? Well, if the generator produces images with little variety, then there will be a small standard deviation across feature maps in the discriminator. Thanks to this layer, the discriminator will have easy access to this statistic, making it less likely to be fooled by a generator that produces too little diversity. This will encourage the generator to produce more diverse outputs, reducing the risk of mode collapse.

Equalized learning rate

Initializes all weights using a Gaussian distribution with mean 0 and standard deviation 1 rather than using He initialization. However, the weights are scaled down at runtime (i.e., every time the layer is executed) by the same factor as in He initialization: they are divided by $\sqrt{2/n_{\text{inputs}}}$, where n_{inputs} is the number of inputs to the layer. The paper demonstrated that this technique significantly improved the GAN's performance when using RMSProp, Adam, or other adaptive gradient optimizers. Indeed, these optimizers normalize the gradient updates by their estimated standard deviation (see [Chapter 11](#)), so parameters that have a larger dynamic range¹⁶ will take longer to train, while parameters with a small dynamic range may be updated too quickly, leading to instabilities. By rescaling the weights as part of the model itself rather than just rescaling them upon initialization, this approach ensures that the dynamic range is the same for all parameters, throughout training, so they all learn at the same speed. This both speeds up and stabilizes training.

Pixelwise normalization layer

Added after each convolutional layer in the generator. It normalizes each activation based on all the activations in the same image and at the same location, but across all channels (dividing by the square root of the mean squared activation). In TensorFlow code, this is `inputs / tf.sqrt(tf.reduce_mean(tf.square(X), axis=-1, keepdims=True) + 1e-8)` (the smoothing term `1e-8` is needed to

¹⁶ The dynamic range of a variable is the ratio between the highest and the lowest value it may take.

avoid division by zero). This technique avoids explosions in the activations due to excessive competition between the generator and the discriminator.

The combination of all these techniques allowed the authors to generate **extremely convincing high-definition images of faces**. But what exactly do we call “convincing”? Evaluation is one of the big challenges when working with GANs: although it is possible to automatically evaluate the diversity of the generated images, judging their quality is a much trickier and subjective task. One technique is to use human raters, but this is costly and time-consuming. So the authors proposed to measure the similarity between the local image structure of the generated images and the training images, considering every scale. This idea led them to another groundbreaking innovation: StyleGANs.

StyleGANs

The state of the art in high-resolution image generation was advanced once again by the same Nvidia team in a [2018 paper¹⁷](#) that introduced the popular StyleGAN architecture. The authors used *style transfer* techniques in the generator to ensure that the generated images have the same local structure as the training images, at every scale, greatly improving the quality of the generated images. The discriminator and the loss function were not modified, only the generator. A StyleGAN generator is composed of two networks (see [Figure 17-19](#)):

Mapping network

An eight-layer MLP that maps the latent representations \mathbf{z} (i.e., the codings) to a vector \mathbf{w} . This vector is then sent through multiple *affine transformations* (i.e., Dense layers with no activation functions, represented by the “A” boxes in [Figure 17-19](#)), which produces multiple vectors. These vectors control the style of the generated image at different levels, from fine-grained texture (e.g., hair color) to high-level features (e.g., adult or child). In short, the mapping network maps the codings to multiple style vectors.

Synthesis network

Responsible for generating the images. It has a constant learned input (to be clear, this input will be constant *after* training, but *during* training it keeps getting tweaked by backpropagation). It processes this input through multiple convolutional and upsampling layers, as earlier, but there are two twists: first, some noise is added to the input and to all the outputs of the convolutional layers (before the activation function). Second, each noise layer is followed by an *Adaptive Instance Normalization* (AdaIN) layer: it standardizes each feature map independently (by

¹⁷ Tero Karras et al., “A Style-Based Generator Architecture for Generative Adversarial Networks,” arXiv preprint arXiv:1812.04948 (2018).

subtracting the feature map's mean and dividing by its standard deviation), then it uses the style vector to determine the scale and offset of each feature map (the style vector contains one scale and one bias term for each feature map).

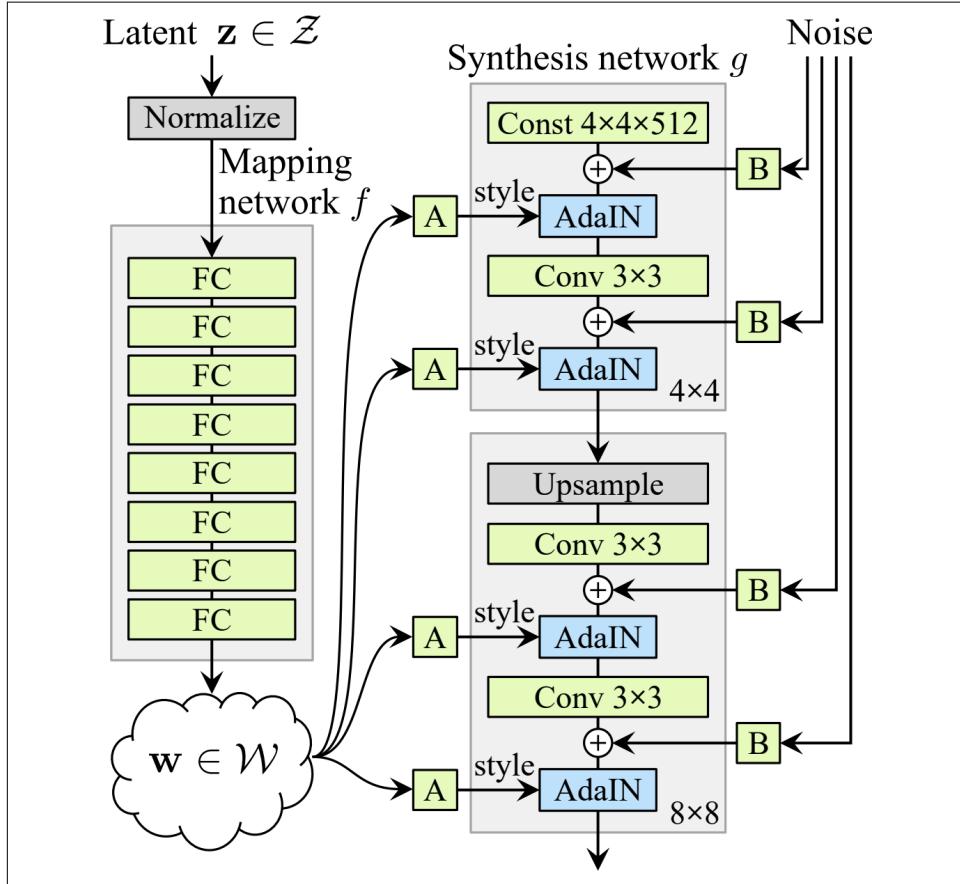


Figure 17-19. StyleGAN's generator architecture (part of figure 1 from the StyleGAN paper)¹⁸

The idea of adding noise independently from the codings is very important. Some parts of an image are quite random, such as the exact position of each freckle or hair. In earlier GANs, this randomness had to either come from the codings or be some pseudorandom noise produced by the generator itself. If it came from the codings, it meant that the generator had to dedicate a significant portion of the codings' representational power to store noise: this is quite wasteful. Moreover, the noise had

¹⁸ Reproduced with the kind authorization of the authors.

to be able to flow through the network and reach the final layers of the generator: this seems like an unnecessary constraint that probably slowed down training. And finally, some visual artifacts may appear because the same noise was used at different levels. If instead the generator tried to produce its own pseudorandom noise, this noise might not look very convincing, leading to more visual artifacts. Plus, part of the generator's weights would be dedicated to generating pseudorandom noise, which again seems wasteful. By adding extra noise inputs, all these issues are avoided; the GAN is able to use the provided noise to add the right amount of stochasticity to each part of the image.

The added noise is different for each level. Each noise input consists of a single feature map full of Gaussian noise, which is broadcast to all feature maps (of the given level) and scaled using learned per-feature scaling factors (this is represented by the “B” boxes in [Figure 17-19](#)) before it is added.

Finally, StyleGAN uses a technique called *mixing regularization* (or *style mixing*), where a percentage of the generated images are produced using two different codings. Specifically, the codings c_1 and c_2 are sent through the mapping network, giving two style vectors w_1 and w_2 . Then the synthesis network generates an image based on the styles w_1 for the first levels and the styles w_2 for the remaining levels. The cutoff level is picked randomly. This prevents the network from assuming that styles at adjacent levels are correlated, which in turn encourages locality in the GAN, meaning that each style vector only affects a limited number of traits in the generated image.

There is such a wide variety of GANs out there that it would require a whole book to cover them all. Hopefully this introduction has given you the main ideas, and most importantly the desire to learn more. Go ahead and implement your own GAN, and do not get discouraged if it has trouble learning at first: unfortunately, this is normal, and it will require quite a bit of patience before it works, but the result is worth it. If you're struggling with an implementation detail, there are plenty of Keras or TensorFlow implementations that you can look at. In fact, if all you want is to get some amazing results quickly, then you can just use a pretrained model (e.g., there are pretrained StyleGAN models available for Keras).

Now that we've examined autoencoders and GANs, let's look at one last type of architecture: diffusion models.

Diffusion Models

The ideas behind diffusion models have been around for many years, but they were first formalized in their modern form in a [2015 paper](#)¹⁹ by Jascha Sohl-Dickstein et al. from Stanford University and UC Berkeley. They applied tools from thermodynamics

¹⁹ Jascha Sohl-Dickstein et al., “Deep Unsupervised Learning using Nonequilibrium Thermodynamics” (2015).

to model a diffusion process, similar to a drop of milk diffusing in a cup of tea. The core idea is to train a model to learn the reverse process: start from the completely mixed state, and gradually “unmix” the milk from the tea. Using this idea, they obtained promising results in image generation, but since GANs produced more convincing images back then, diffusion models did not get as much attention.

Fast forward five years, a [2020 paper²⁰](#) by Jonathan Ho et al., also from UC Berkeley, managed to build a diffusion model called a *Denoising Diffusion Probabilistic Model* (DDPM) capable of generating highly realistic images. A few months later, a [2021 paper²¹](#) by OpenAI researchers Alex Nichol and Prafulla Dhariwal analyzed the DDPM architecture and proposed several improvements that allowed DDPMs to finally beat GANs: not only are DDPMs much easier to train than GANs, but the generated images are more diverse and even higher quality. The main downside of DDPMs, as we will see, is that they take a very long time to generate images, compared to GANs or VAEs.

So how exactly does a DDPM work? Well suppose you start with a picture of a cat (see [Figure 17-20](#)), noted \mathbf{x}_0 , and at each time step t you add a little bit of Gaussian noise to the image, with mean 0 and variance β_t . This noise is independent for each pixel: we call it *isotropic*. You first obtain the image \mathbf{x}_1 , then \mathbf{x}_2 , and so on, until the cat is completely hidden by the noise, impossible to see. The last time step is noted T . In the original DDPM paper, the authors used $T = 1,000$, and they scheduled the variance β_t in such a way that the cat signal fades linearly between time steps 0 and T . In the improved DDPM paper, T was bumped up to 4,000, and the variance schedule was tweaked to change more slowly at the beginning and at the end. In short, we’re gradually drowning the cat in noise: this is called the *forward process*.

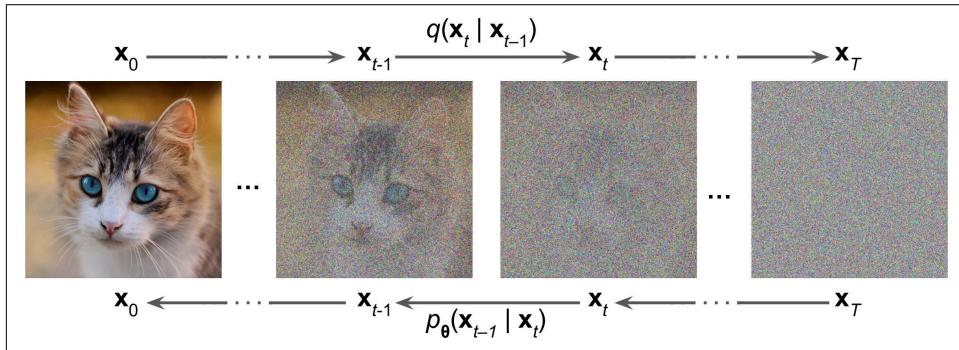


Figure 17-20. The forward process q and reverse process p

²⁰ Jonathan Ho et al., “Denoising Diffusion Probabilistic Models” (2020).

²¹ Alex Nichol and Prafulla Dhariwal, “Improved Denoising Diffusion Probabilistic Models” (2021).

As we add more and more Gaussian noise in the forward process, the distribution of pixel values becomes more and more Gaussian. One important detail I left out is that the pixel values get rescaled slightly at each step, by a factor of $\sqrt{1 - \beta_t}$. This ensures that the mean of the pixel values gradually approaches 0, since the scaling factor is a bit smaller than 1 (imagine repeatedly multiplying a number by 0.99). It also ensures that the variance will gradually converge to 1. This is because the standard deviation of the pixel values also gets scaled by $\sqrt{1 - \beta_t}$, so the variance gets scaled by $1 - \beta_t$ (i.e., the square of the scaling factor). But the variance cannot shrink to 0 since we're adding Gaussian noise with variance β_t at each step. And since variances add up when you sum Gaussian distributions, you can see that the variance can only converge to $1 - \beta_t + \beta_t = 1$.

The forward diffusion process is summarized in [Equation 17-5](#). This equation won't teach you anything new about the forward process, but it's useful to understand this type of mathematical notation, as it's often used in ML papers. This equation defines the probability distribution q of \mathbf{x}_t given \mathbf{x}_{t-1} as a Gaussian distribution with mean \mathbf{x}_{t-1} times the scaling factor, and with a covariance matrix equal to $\beta_t \mathbf{I}$. This is the identity matrix \mathbf{I} multiplied by β_t , which means that the noise is isotropic with variance β_t .

Equation 17-5. Probability distribution q of the forward diffusion process

$$q(\mathbf{x}_t | \mathbf{x}_{t-1}) = \mathcal{N}(\sqrt{1 - \beta_t} \mathbf{x}_{t-1}, \beta_t \mathbf{I})$$

Interestingly, there's a shortcut for the forward process: it's possible to sample an image \mathbf{x}_t given \mathbf{x}_0 without having to first compute $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_{t-1}$. Indeed, since the sum of multiple Gaussian distributions is also a Gaussian distribution, all the noise can be added in just one shot using [Equation 17-6](#). This is the equation we will be using, as it is much faster.

Equation 17-6. Shortcut for the forward diffusion process

$$q(\mathbf{x}_t | \mathbf{x}_0) = \mathcal{N}(\sqrt{\bar{\alpha}_t} \mathbf{x}_0, (1 - \bar{\alpha}_t) \mathbf{I})$$

Our goal of course is not to drown cats in noise. On the contrary, we want to create many new cats! We can do so by training a model that can perform the *reverse process*: going from \mathbf{x}_t to \mathbf{x}_{t-1} . We can then use it to remove a tiny bit of noise from an image, and repeat the operation many times until all the noise is gone. If we train the model on a dataset containing many cat images, then we can give it a picture entirely full of Gaussian noise, and the model will gradually make a brand new cat appear.

OK, so let's start coding! The first thing we need to do is to code the forward process. For this, we will first need to implement the variance schedule. How can we control how fast the cat disappears? Initially, 100% of the variance comes from the original

cat image. Then at each time step t , the variance gets multiplied by $1 - \beta_t$, as explained earlier, and noise gets added. So the part of the variance that comes from the initial distribution shrinks by a factor of $1 - \beta_t$ at each step. If we define $\alpha_t = 1 - \beta_t$, then after t time steps, the cat signal will have been multiplied by a factor of $\bar{\alpha}_t = \alpha_1 \times \alpha_2 \times \dots \times \alpha_t = \bar{\alpha}_t = \prod_{i=1}^t \alpha_i$. It's this "cat signal" factor $\bar{\alpha}_t$ that we want to schedule so it shrinks down from 1 to 0 gradually between time steps 0 and T . In the improved DDPM paper, the authors schedule $\bar{\alpha}_t$ according to [Equation 17-7](#). This schedule is represented in [Figure 17-21](#).

Equation 17-7. Variance schedule equations for the forward diffusion process

$$\beta_t = 1 - \frac{\bar{\alpha}_t}{\bar{\alpha}_{t-1}}, \text{ with } \bar{\alpha}_t = \frac{f(t)}{f(0)} \text{ and } f(t) = \cos\left(\frac{t/T + s}{1+s} \cdot \frac{\pi}{2}\right)^2$$

- s is a tiny value which prevents β_t from being too small near $t = 0$. In the paper, the authors used $s = 0.008$.
- β_t is clipped to be no larger than 0.999, to avoid instabilities near $t = T$.

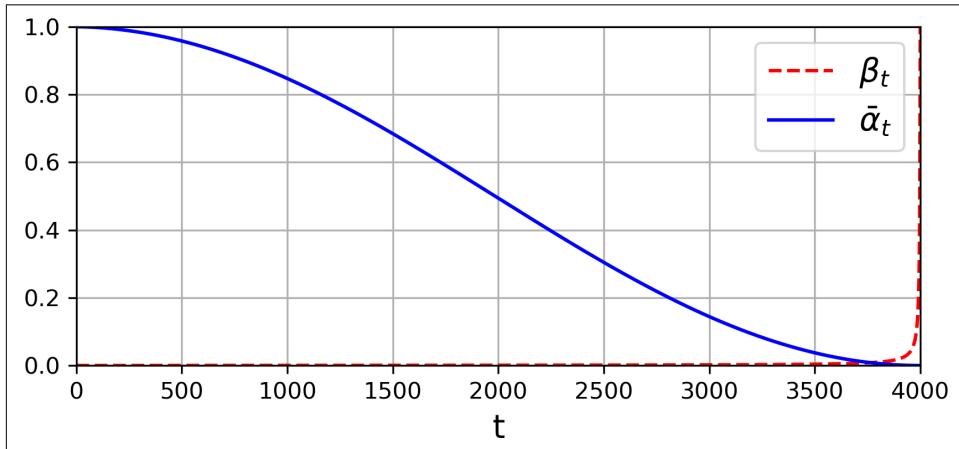


Figure 17-21. Noise variance schedule β_t , and the remaining signal variance $\bar{\alpha}_t$

Let's create a small function to compute α_t , β_t , and $\bar{\alpha}_t$, and call it with $T = 4000$:

```
def variance_schedule(T, s=0.008, max_beta=0.999):
    t = np.arange(T + 1)
    f = np.cos((t / T + s) / (1 + s) * np.pi / 2) ** 2
    alpha = np.clip(f[1:] / f[:-1], 1 - max_beta, 1)
    alpha = np.append(1, alpha).astype(np.float32) # add a0 = 1
    beta = 1 - alpha
    alpha_cumprod = np.cumprod(alpha)
    return alpha, alpha_cumprod, beta # a_t , bar{a}_t , beta_t for t = 0 to T
```

```

T = 4000
alpha, alpha_cumprod, beta = variance_schedule(T)

```

To train our model to reverse the diffusion process, we will need noisy images from different time steps of the forward process. For this, let's create a `prepare_batch()` function that will take a batch of clean images from the dataset and prepare them:

```

def prepare_batch(X):
    X = tf.cast(X[..., tf.newaxis], tf.float32) * 2 - 1 # scale from -1 to +1
    X_shape = tf.shape(X)
    t = tf.random.uniform([X_shape[0]], minval=1, maxval=T + 1, dtype=tf.int32)
    alpha_cm = tf.gather(alpha_cumprod, t)
    alpha_cm = tf.reshape(alpha_cm, [X_shape[0]] + [1] * (len(X_shape) - 1))
    noise = tf.random.normal(X_shape)
    return {
        "X_noisy": alpha_cm ** 0.5 * X + (1 - alpha_cm) ** 0.5 * noise,
        "time": t,
    }, noise

```

Let's go through this code:

- For simplicity we will use Fashion MNIST, so the function must first add a channel axis. It will also help to scale the pixel values from -1 to 1 , so it's closer to the final Gaussian distribution with mean 0 and variance 1 .
- Next, the function creates t , a vector containing a random time step for each image in the batch, between 1 and T .
- Then it uses `tf.gather()` to get the value of `alpha_cumprod` for each of the time steps in the vector t . This gives us the vector `alpha_cm`, containing one value of $\bar{\alpha}_t$ for each image.
- The next line reshapes the `alpha_cm` from [batch size] to [batch size, 1, 1, 1]. This is needed to ensure `alpha_cm` can be broadcasted with the batch `X`.
- Then we generate some Gaussian noise with mean 0 and variance 1 .
- Lastly, we use [Equation 17-6](#) to apply the diffusion process to the images. Note that $x ** 0.5$ is equal to the square root of x . The function returns a tuple containing the inputs and the targets. The inputs are represented as a Python `dict` containing the noisy images and the time steps used to generate them. The target is the Gaussian noise used to generate each image.



With this setup, the model will predict the noise that should be subtracted from the input image to get the original image. But why not predict the original image directly? Well, the authors tried: it simply doesn't work as well.

Now let's create a training dataset and a validation set that will apply the `prepare_batch()` function to every batch. As earlier, `X_train` and `X_valid` contain the Fashion MNIST images with pixel values ranging from 0 to 1.

```
def prepare_dataset(X, batch_size=32, shuffle=False):
    ds = tf.data.Dataset.from_tensor_slices(X)
    if shuffle:
        ds = ds.shuffle(buffer_size=10_000)
    return ds.batch(batch_size).map(prepare_batch).prefetch(1)

train_set = prepare_dataset(X_train, batch_size=32, shuffle=True)
valid_set = prepare_dataset(X_valid, batch_size=32)
```

Now we're ready to build the actual diffusion model itself. It can be any model you want, as long as it takes the noisy images and time steps as inputs, and predicts the noise to subtract from the input images:

```
def build_diffusion_model():
    X_noisy = tf.keras.layers.Input(shape=[28, 28, 1], name="X_noisy")
    time_input = tf.keras.layers.Input(shape=[], dtype=tf.int32, name="time")
    [...] # build the model based on the noisy images and the time steps
    outputs = [...] # predict the noise (same shape as the input images)
    return tf.keras.Model(inputs=[X_noisy, time_input], outputs=[outputs])
```

The DDPM authors used a modified [U-Net architecture](#)²², which has many similarities with the FCN architecture we discussed in [Chapter 14](#) for semantic segmentation: it's a convolutional neural network that gradually downsamples the input images, then gradually upsamples them again, with skip connections crossing over from each level of the downsampling part to the corresponding level in the upsampling part. To take into account the time steps, they encoded them using the same technique as the positional encodings in the Transformer architecture (see [Chapter 16](#)). At every level in the U-Net architecture, they passed these time encodings through Dense layers and fed them to the U-Net. Lastly, they also used Multi-Head Attention layers at various levels. See the notebook for a basic implementation, or <https://hml.info/ddpmcode> for the official implementation: it's based on TF 1.x, which is deprecated, but it's quite readable.

Next we can train the model normally. The authors noted that using the MAE loss worked better than the MSE. You can also use the Huber loss:

```
model = build_diffusion_model()
model.compile(loss=tf.keras.losses.Huber(), optimizer="nadam")
history = model.fit(train_set, validation_data=valid_set, epochs=100)
```

Once the model is trained, you can use it to generate new images. Unfortunately, there's no shortcut in the reverse diffusion process, so you have to sample \mathbf{x}_T ran-

²² Olaf Ronneberger et al., “U-Net: Convolutional Networks for Biomedical Image Segmentation” (2015).

domly from a Gaussian distribution with mean 0 and variance 1, then pass it to the model to predict the noise, subtract it from the image using [Equation 17-8](#), and you get \mathbf{x}_{T-1} . Repeat the process 3,999 more times until you get \mathbf{x}_0 : if all went well, it should look like a regular Fashion MNIST image!

Equation 17-8. Going one step in reverse in the diffusion process

$$\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}_t - \frac{\beta_t}{\sqrt{1 - \bar{\alpha}_t}} \epsilon_{\theta}(\mathbf{x}_t, t) \right) + \sqrt{\beta_t} \mathbf{z}$$

In this equation, $\epsilon_{\theta}(\mathbf{x}_t, t)$ represents the noise predicted by the model given the input image \mathbf{x}_t and the time step t . The θ represents the model parameters. Moreover, \mathbf{z} is Gaussian noise with mean 0 and variance 1. This makes the reverse process stochastic: if you run it multiple times, you will get different images.

Let's write a function that implements this reverse process, and call it to generate a few images:

```
def generate(model, batch_size=32):
    X = tf.random.normal([batch_size, 28, 28, 1])
    for t in range(T, 0, -1):
        noise = (tf.random.normal if t > 1 else tf.zeros)(tf.shape(X))
        X_noisy = model({"X_noisy": X, "time": tf.constant([t] * batch_size)})
        X = (
            1 / alpha[t] ** 0.5
            * (X - beta[t] / (1 - alpha_cumprod[t])) ** 0.5 * X_noisy
            + (1 - alpha[t]) ** 0.5 * noise
        )
    return X

X_gen = generate(model) # generated images
```

This may take a minute or two. That's the main drawback of diffusion models: generating images is slow since the model needs to be called many times. It's possible to make this faster by using a smaller T value, or by using the same model prediction for several steps at a time, but the resulting images may not look as nice. That said, despite this speed limitation, diffusion models do produce high quality and diverse images, as you can see in [Figure 17-22](#).



Figure 17-22. Images generated by the DDPM

Just like Autoencoders and GANs, diffusion models can be applied to other types of inputs, not just images. Moreover, they can also be conditioned to produce the desired class, or even to follow a textual description such as “a dark shirt with long sleeves”. For example, check out the [DiffusionCLIP model](#)²³ The possibilities are endless!

In the next chapter we will move to an entirely different branch of Deep Learning: Deep Reinforcement Learning.

Exercises

1. What are the main tasks that autoencoders are used for?
2. Suppose you want to train a classifier, and you have plenty of unlabeled training data but only a few thousand labeled instances. How can autoencoders help? How would you proceed?
3. If an autoencoder perfectly reconstructs the inputs, is it necessarily a good autoencoder? How can you evaluate the performance of an autoencoder?
4. What are undercomplete and overcomplete autoencoders? What is the main risk of an excessively undercomplete autoencoder? What about the main risk of an overcomplete autoencoder?
5. How do you tie weights in a stacked autoencoder? What is the point of doing so?

²³ Gwanghyun Kim et al., “DiffusionCLIP: Text-Guided Diffusion Models for Robust Image Manipulation” (2021).

6. What is a generative model? Can you name a type of generative autoencoder?
7. What is a GAN? Can you name a few tasks where GANs can shine?
8. What are the main difficulties when training GANs?
9. What are diffusion models good at? What is their main limitation?
10. Try using a denoising autoencoder to pretrain an image classifier. You can use MNIST (the simplest option), or a more complex image dataset such as **CIFAR10** if you want a bigger challenge. Regardless of the dataset you're using, follow these steps:
 - Split the dataset into a training set and a test set. Train a deep denoising autoencoder on the full training set.
 - Check that the images are fairly well reconstructed. Visualize the images that most activate each neuron in the coding layer.
 - Build a classification DNN, reusing the lower layers of the autoencoder. Train it using only 500 images from the training set. Does it perform better with or without pretraining?
11. Train a variational autoencoder on the image dataset of your choice, and use it to generate images. Alternatively, you can try to find an unlabeled dataset that you are interested in and see if you can generate new samples.
12. Train a DCGAN to tackle the image dataset of your choice, and use it to generate images. Add experience replay and see if this helps. Turn it into a conditional GAN where you can control the generated class.

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab3>.

Reinforcement Learning

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 18th chapter of the final book. The GitHub repo is <https://github.com/ageron/handsonml3>.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

Reinforcement Learning (RL) is one of the most exciting fields of Machine Learning today, and also one of the oldest. It has been around since the 1950s, producing many interesting applications over the years.¹ particularly in games (e.g., *TD-Gammon*, a Backgammon-playing program) and in machine control, but seldom making the headline news. But a revolution took place in 2013, when researchers from a British startup called DeepMind demonstrated a system that could learn to play just about any Atari game from scratch,² eventually outperforming humans³ in most of them, using only raw pixels as inputs and without any prior knowledge of the rules of the games.⁴ This was the first of a series of amazing feats, culminating in March 2016

¹ For more details, be sure to check out Richard Sutton and Andrew Barto’s book on RL, *Reinforcement Learning: An Introduction* (MIT Press).

² Volodymyr Mnih et al., “Playing Atari with Deep Reinforcement Learning,” arXiv preprint arXiv:1312.5602 (2013).

³ Volodymyr Mnih et al., “Human-Level Control Through Deep Reinforcement Learning,” *Nature* 518 (2015): 529–533.

with the victory of their system AlphaGo against Lee Sedol, a legendary professional player of the game of Go, and in May 2017 against Ke Jie, the world champion. No program had ever come close to beating a master of this game, let alone the world champion. Today the whole field of RL is boiling with new ideas, with a wide range of applications. DeepMind was bought by Google for over \$500 million in 2014.

So how did DeepMind achieve all this? With hindsight it seems rather simple: they applied the power of Deep Learning to the field of Reinforcement Learning, and it worked beyond their wildest dreams. In this chapter I will first explain what Reinforcement Learning is and what it's good at, then present two of the most important techniques in Deep Reinforcement Learning: *policy gradients* and *deep Q-networks* (DQNs), including a discussion of *Markov decision processes* (MDPs). Let's get started!

Learning to Optimize Rewards

In Reinforcement Learning, a software *agent* makes *observations* and takes *actions* within an *environment*, and in return it receives *rewards* from the environment. Its objective is to learn to act in a way that will maximize its expected rewards over time. If you don't mind a bit of anthropomorphism, you can think of positive rewards as pleasure, and negative rewards as pain (the term "reward" is a bit misleading in this case). In short, the agent acts in the environment and learns by trial and error to maximize its pleasure and minimize its pain.

This is quite a broad setting, which can apply to a wide variety of tasks. Here are a few examples (see [Figure 18-1](#)):

- a. The agent can be the program controlling a robot. In this case, the environment is the real world, the agent observes the environment through a set of *sensors* such as cameras and touch sensors, and its actions consist of sending signals to activate motors. It may be programmed to get positive rewards whenever it approaches the target destination, and negative rewards whenever it wastes time or goes in the wrong direction.
- b. The agent can be the program controlling *Ms. Pac-Man*. In this case, the environment is a simulation of the Atari game, the actions are the nine possible joystick positions (upper left, down, center, and so on), the observations are screenshots, and the rewards are just the game points.
- c. Similarly, the agent can be the program playing a board game such as Go. It only gets a reward if it wins.

⁴ Check out the videos of DeepMind's system learning to play *Space Invaders*, *Breakout*, and other video games at <https://homl.info/dqn3>.

- d. The agent does not have to control a physically (or virtually) moving thing. For example, it can be a smart thermostat, getting positive rewards whenever it is close to the target temperature and saves energy, and negative rewards when humans need to tweak the temperature, so the agent must learn to anticipate human needs.
- e. The agent can observe stock market prices and decide how much to buy or sell every second. Rewards are obviously the monetary gains and losses.

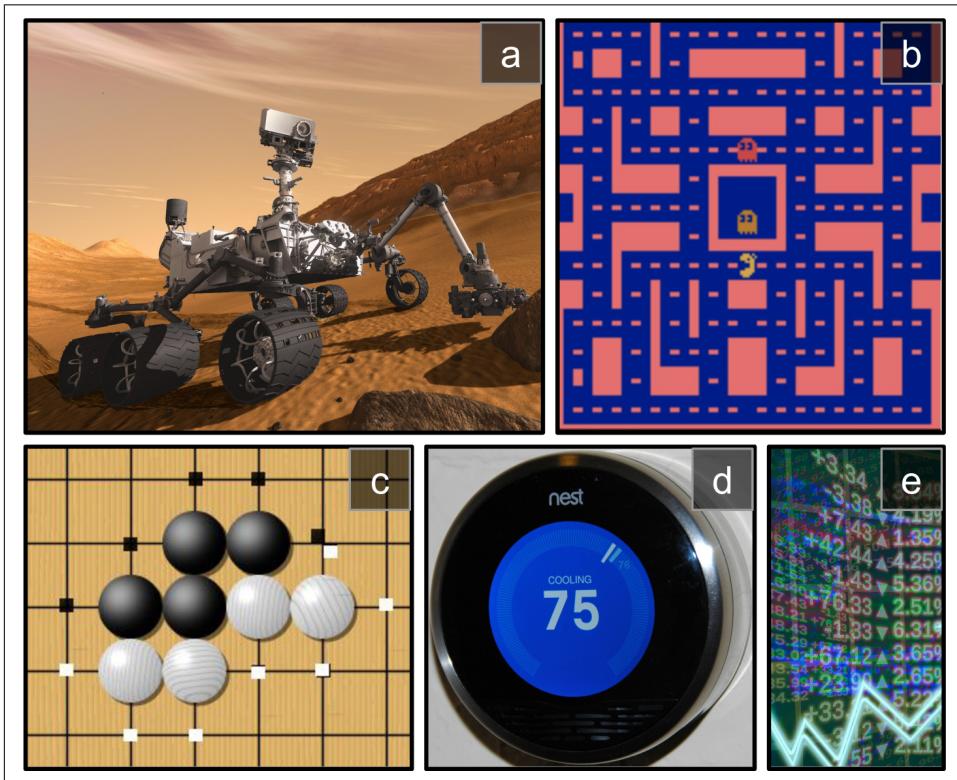


Figure 18-1. Reinforcement Learning examples: (a) robotics, (b) Ms. Pac-Man, (c) Go player, (d) thermostat, (e) automatic trader⁵

Note that there may not be any positive rewards at all; for example, the agent may move around in a maze, getting a negative reward at every time step, so it had better find the exit as quickly as possible! There are many other examples of tasks to

⁵ Images (a), (d), and (e) are in the public domain. Image (b) is a screenshot from the *Ms. Pac-Man* game, copyright Atari (fair use in this chapter). Image (c) is reproduced from Wikipedia, it was created by user Stevertigo and released under Creative Commons BY-SA 2.0.

which Reinforcement Learning is well suited, such as self-driving cars, recommender systems, placing ads on a web page, or controlling where an image classification system should focus its attention.

Policy Search

The algorithm a software agent uses to determine its actions is called its *policy*. The policy could be a neural network taking observations as inputs and outputting the action to take (see Figure 18-2).

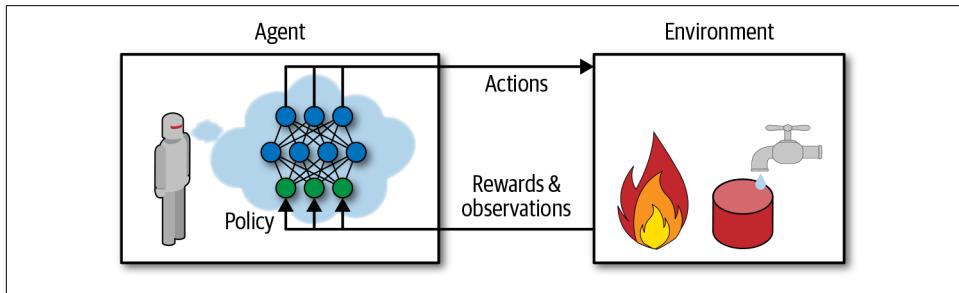


Figure 18-2. Reinforcement Learning using a neural network policy

The policy can be any algorithm you can think of, and it does not have to be deterministic. In fact, in some cases it does not even have to observe the environment! For example, consider a robotic vacuum cleaner whose reward is the amount of dust it picks up in 30 minutes. Its policy could be to move forward with some probability p every second, or randomly rotate left or right with probability $1 - p$. The rotation angle would be a random angle between $-r$ and $+r$. Since this policy involves some randomness, it is called a *stochastic policy*. The robot will have an erratic trajectory, which guarantees that it will eventually get to any place it can reach and pick up all the dust. The question is, how much dust will it pick up in 30 minutes?

How would you train such a robot? There are just two *policy parameters* you can tweak: the probability p and the angle range r . One possible learning algorithm could be to try out many different values for these parameters, and pick the combination that performs best (see Figure 18-3). This is an example of *policy search*, in this case using a brute force approach. When the *policy space* is too large (which is generally the case), finding a good set of parameters this way is like searching for a needle in a gigantic haystack.

Another way to explore the policy space is to use *genetic algorithms*. For example, you could randomly create a first generation of 100 policies and try them out, then

“kill” the 80 worst policies⁶ and make the 20 survivors produce 4 offspring each. An offspring is a copy of its parent⁷ plus some random variation. The surviving policies plus their offspring together constitute the second generation. You can continue to iterate through generations this way until you find a good policy.⁸

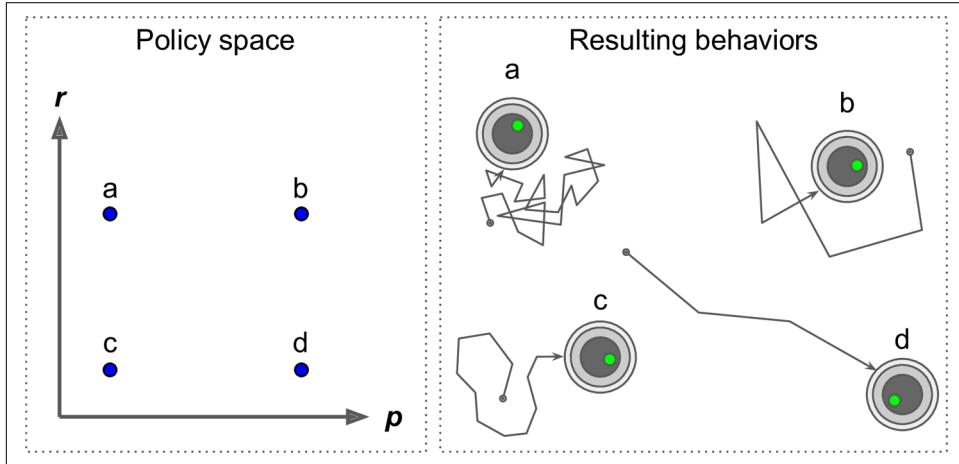


Figure 18-3. Four points in policy space (left) and the agent’s corresponding behavior (right)

Yet another approach is to use optimization techniques, by evaluating the gradients of the rewards with regard to the policy parameters, then tweaking these parameters by following the gradients toward higher rewards.⁹ We will discuss this approach, called *policy gradients* (PG), in more detail later in this chapter. Going back to the vacuum cleaner robot, you could slightly increase p and evaluate whether doing so increases the amount of dust picked up by the robot in 30 minutes; if it does, then increase p some more, or else reduce p . We will implement a popular PG algorithm using TensorFlow, but before we do, we need to create an environment for the agent to live in—so it’s time to introduce OpenAI Gym.

⁶ It is often better to give the poor performers a slight chance of survival, to preserve some diversity in the “gene pool.”

⁷ If there is a single parent, this is called *asexual reproduction*. With two (or more) parents, it is called *sexual reproduction*. An offspring’s genome (in this case a set of policy parameters) is randomly composed of parts of its parents’ genomes.

⁸ One interesting example of a genetic algorithm used for Reinforcement Learning is the *NeuroEvolution of Augmenting Topologies* (NEAT) algorithm.

⁹ This is called *Gradient Ascent*. It’s just like Gradient Descent but in the opposite direction: maximizing instead of minimizing.

Introduction to OpenAI Gym

One of the challenges of Reinforcement Learning is that in order to train an agent, you first need to have a working environment. If you want to program an agent that will learn to play an Atari game, you will need an Atari game simulator. If you want to program a walking robot, then the environment is the real world, and you can directly train your robot in that environment, but this has its limits: if the robot falls off a cliff, you can't just click Undo. You can't speed up time either; adding more computing power won't make the robot move any faster. And it's generally too expensive to train 1,000 robots in parallel. In short, training is hard and slow in the real world, so you generally need a *simulated environment* at least for bootstrap training. For example, you may use a library like **PyBullet** or **MuJoCo** for 3D physics simulation.

OpenAI Gym¹⁰ is a toolkit that provides a wide variety of simulated environments (Atari games, board games, 2D and 3D physical simulations, and so on), so you can train agents, compare them, or develop new RL algorithms.

OpenAI Gym is preinstalled on Colab, but it's an older version so we must replace it with the latest one. We must also install a few of its dependencies. If you are coding on your own machine instead of Colab, and you followed the installation instructions at <https://homl.info/install>, then you can skip this step.

```
# Only run these commands on Colab or Kaggle!
%pip install -q -U gym
%pip install -q -U gym[box2d,atari,accept-rom-license]
!apt update &> /dev/null && apt install -y xvfb &> /dev/null
%pip install -q -U pyvirtualdisplay
```

The first `%pip` command upgrades Gym to the latest version. The `-q` option stands for *quiet*: it makes the output less verbose. The `-U` option stands for *upgrade*. The second `%pip` command installs the libraries required to run the Gym environments based on the Box2D library—a 2D physics engine for games—and based on the Arcade Learning Environment (ALE), which is an emulator for Atari 2600 games. Several Atari game ROMs are downloaded. By running this code you agree with Atari's ROM licenses.

Colab Runtimes are based on the Ubuntu operating system, which uses `apt` as its package manager. The `apt update` command updates the package list, and `&> /dev/null` pipes all the output to the null device: this ignores the output rather than printing it all out. Then we use `apt` to install the `xvfb` package, using the `-y` option to skip the user confirmation step. Xvfb is a display server for *headless servers* (i.e.,

¹⁰ OpenAI is an artificial intelligence research company, funded in part by Elon Musk. Its stated goal is to promote and develop friendly AIs that will benefit humanity (rather than exterminate it).

machines that don't have a screen), such as Colab runtimes or docker containers: this is needed because some Gym environments try to display graphics and they crash if they can't find a display server.

The `pyvirtualdisplay` library is a Python frontend that interacts with Xvfb. It can be used to create a virtual display (e.g., $1,400 \times 900$ pixels) like this (only needed on a headless server):

```
import pyvirtualdisplay

display = pyvirtualdisplay.Display(visible=0, size=(1400, 900)).start()
```

With that, we're ready to use OpenAI Gym. Let's import it and make an environment:

```
import gym

env = gym.make("CartPole-v1")
```

Here, we've created a CartPole environment. This is a 2D simulation in which a cart can be accelerated left or right in order to balance a pole placed on top of it (see Figure 18-4).

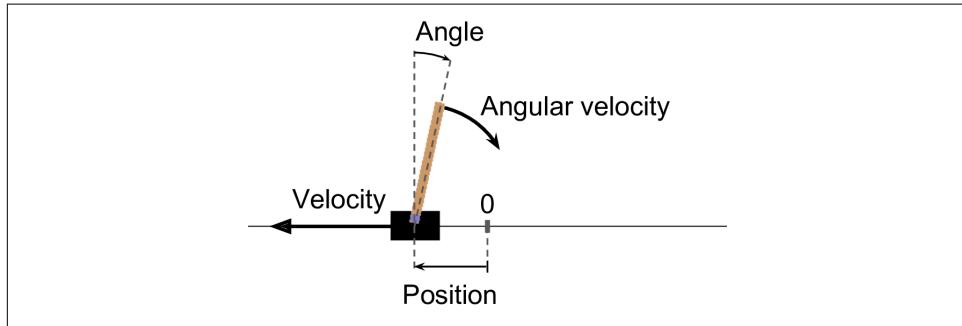


Figure 18-4. The CartPole environment



You can get the list of all available environments by running `gym.envs.registry.all()`.

After the environment is created, you must initialize it using the `reset()` method, optionally specifying a random seed. This returns the first observation. Observations depend on the type of environment. For the CartPole environment, each observation is a 1D NumPy array containing four floats: these floats represent the cart's horizontal position (0.0 = center), its velocity (positive means right), the angle of the pole (0.0 = vertical), and its angular velocity (positive means clockwise):

```
>>> obs = env.reset(seed=42)
>>> obs
array([ 0.0273956 , -0.00611216,  0.03585979,  0.0197368 ], dtype=float32)
```

If you call `env.render()`, Gym will try to display the environment on your screen. However, in Colab and other headless servers, there's no screen, so nothing will happen. Instead, you should set the `mode` argument to "`rgb_array`", to render the environment to a NumPy array:¹¹

```
>>> img = env.render(mode="rgb_array")
>>> img.shape # height, width, channels (3 = Red, Green, Blue)
(400, 600, 3)
```

You can then use Matplotlib's `imshow()` function to display this image, as usual.

Now let's ask the environment what actions are possible:

```
>>> env.action_space
Discrete(2)
```

`Discrete(2)` means that the possible actions are integers 0 and 1, which represent accelerating left (0) or right (1). Other environments may have additional discrete actions, or other kinds of actions (e.g., continuous). Since the pole is leaning toward the right (`obs[2] > 0`), let's accelerate the cart toward the right:

```
>>> action = 1 # accelerate right
>>> obs, reward, done, info = env.step(action)
>>> obs
array([ 0.02727336,  0.18847767,  0.03625453, -0.26141977], dtype=float32)
>>> reward
1.0
>>> done
False
>>> info
{}
```

The `step()` method executes the desired action and returns four values:

obs

This is the new observation. The cart is now moving toward the right (`obs[1] > 0`). The pole is still tilted toward the right (`obs[2] > 0`), but its angular velocity is now negative (`obs[3] < 0`), so it will likely be tilted toward the left after the next step.

reward

In this environment, you get a reward of 1.0 at every step, no matter what you do, so the goal is to keep the episode running for as long as possible.

¹¹ If you are running this code on your own machine, some environments (including CartPole) will also be displayed to screen, even when `mode="rgb_array"`.

done

This value will be `True` when the episode is over. This will happen when the pole tilts too much, or goes off the screen, or after 200 steps (in this last case, you have won). After that, the environment must be reset before it can be used again.

info

This environment-specific dictionary can provide some extra information that you may find useful for debugging or for training. For example, in some games it may indicate how many lives the agent has left.



Once you have finished using an environment, you should call its `close()` method to free resources.

Let's hardcode a simple policy that accelerates left when the pole is leaning toward the left and accelerates right when the pole is leaning toward the right. We will run this policy to see the average rewards it gets over 500 episodes:

```
def basic_policy(obs):
    angle = obs[2]
    return 0 if angle < 0 else 1

totals = []
for episode in range(500):
    episode_rewards = 0
    obs = env.reset(seed=episode)
    for step in range(200):
        action = basic_policy(obs)
        obs, reward, done, info = env.step(action)
        episode_rewards += reward
        if done:
            break
    totals.append(episode_rewards)
```

This code is self-explanatory. Let's look at the result:

```
>>> import numpy as np
>>> np.mean(totals), np.std(totals), min(totals), max(totals)
(41.698, 8.389445512070509, 24.0, 63.0)
```

Even with 500 tries, this policy never managed to keep the pole upright for more than 63 consecutive steps. Not great. If you look at the simulation in the notebook, you will see that the cart oscillates left and right more and more strongly until the pole tilts too much. Let's see if a neural network can come up with a better policy.

Neural Network Policies

Let's create a neural network policy. This neural network will take an observation as input, and it will output the action to be executed, just like the policy we hardcoded earlier. More precisely, it will estimate a probability for each action, and then we will select an action randomly, according to the estimated probabilities (see [Figure 18-5](#)). In the case of the CartPole environment, there are just two possible actions (left or right), so we only need one output neuron. It will output the probability p of action 0 (left), and of course the probability of action 1 (right) will be $1 - p$. For example, if it outputs 0.7, then we will pick action 0 with 70% probability, or action 1 with 30% probability.

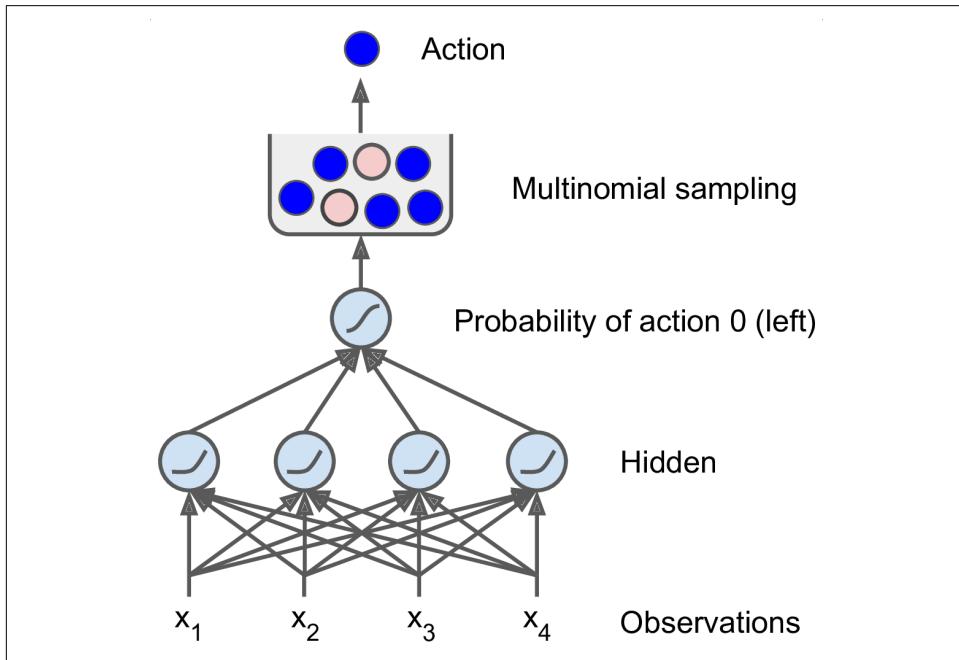


Figure 18-5. Neural network policy

You may wonder why we are picking a random action based on the probabilities given by the neural network, rather than just picking the action with the highest score. This approach lets the agent find the right balance between *exploring* new actions and *exploiting* the actions that are known to work well. Here's an analogy: suppose you go to a restaurant for the first time, and all the dishes look equally appealing, so you randomly pick one. If it turns out to be good, you can increase the probability that you'll order it next time, but you shouldn't increase that probability up to 100%, or else you will never try out the other dishes, some of which may be

even better than the one you tried. This *exploration / exploitation dilemma* is central in Reinforcement Learning.

Also note that in this particular environment, the past actions and observations can safely be ignored, since each observation contains the environment's full state. If there were some hidden state, then you might need to consider past actions and observations as well. For example, if the environment only revealed the position of the cart but not its velocity, you would have to consider not only the current observation but also the previous observation in order to estimate the current velocity. Another example is when the observations are noisy; in that case, you generally want to use the past few observations to estimate the most likely current state. The CartPole problem is thus as simple as can be; the observations are noise-free, and they contain the environment's full state.

Here is the code to build a basic neural network policy using Keras:

```
import tensorflow as tf

model = tf.keras.Sequential([
    tf.keras.layers.Dense(5, activation="relu"),
    tf.keras.layers.Dense(1, activation="sigmoid"),
])
```

We use a `Sequential` model to define the policy network. The number of inputs is the size of the observation space—which in the case of CartPole is 4—and we have just five hidden units because it's a fairly simple task. Finally, we want to output a single probability—the probability of going left—so we have a single output neuron using the sigmoid activation function. If there were more than two possible actions, there would be one output neuron per action, and we would use the softmax activation function instead.

OK, we now have a neural network policy that will take observations and output action probabilities. But how do we train it?

Evaluating Actions: The Credit Assignment Problem

If we knew what the best action was at each step, we could train the neural network as usual, by minimizing the cross entropy between the estimated probability distribution and the target probability distribution. It would just be regular supervised learning. However, in Reinforcement Learning the only guidance the agent gets is through rewards, and rewards are typically sparse and delayed. For example, if the agent manages to balance the pole for 100 steps, how can it know which of the 100 actions it took were good, and which of them were bad? All it knows is that the pole fell after the last action, but surely this last action is not entirely responsible. This is called the *credit assignment problem*: when the agent gets a reward, it is hard for it to

know which actions should get credited (or blamed) for it. Think of a dog that gets rewarded hours after it behaved well; will it understand what it is being rewarded for?

To tackle this problem, a common strategy is to evaluate an action based on the sum of all the rewards that come after it, usually applying a *discount factor* γ (gamma) at each step. This sum of discounted rewards is called the action's *return*. Consider the example in [Figure 18-6](#)). If an agent decides to go right three times in a row and gets +10 reward after the first step, 0 after the second step, and finally -50 after the third step, then assuming we use a discount factor $\gamma = 0.8$, the first action will have a return of $10 + \gamma \times 0 + \gamma^2 \times (-50) = -22$. If the discount factor is close to 0, then future rewards won't count for much compared to immediate rewards. Conversely, if the discount factor is close to 1, then rewards far into the future will count almost as much as immediate rewards. Typical discount factors vary from 0.9 to 0.99. With a discount factor of 0.95, rewards 13 steps into the future count roughly for half as much as immediate rewards (since $0.95^{13} \approx 0.5$), while with a discount factor of 0.99, rewards 69 steps into the future count for half as much as immediate rewards. In the CartPole environment, actions have fairly short-term effects, so choosing a discount factor of 0.95 seems reasonable.

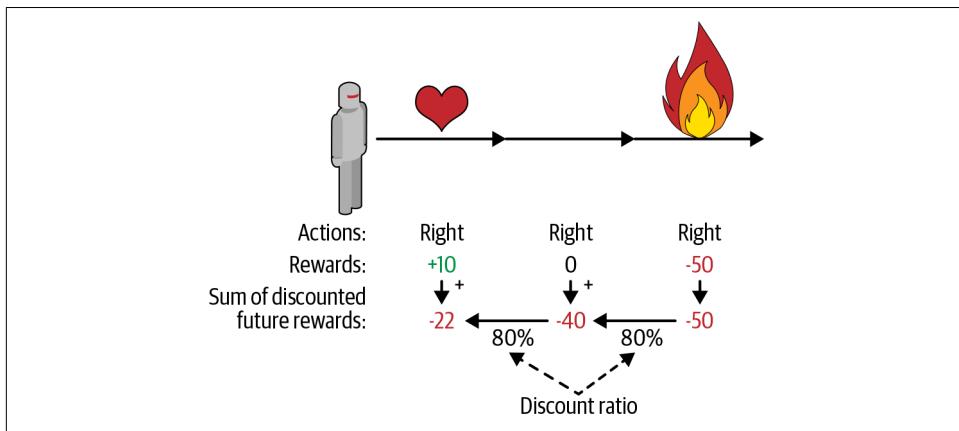


Figure 18-6. Computing an action's return: the sum of discounted future rewards

Of course, a good action may be followed by several bad actions that cause the pole to fall quickly, resulting in the good action getting a low return. Similarly, a good actor may sometimes star in a terrible movie. However, if we play the game enough times, on average good actions will get a higher return than bad ones. We want to estimate how much better or worse an action is, compared to the other possible actions, on average. This is called the *action advantage*. For this, we must run many episodes and normalize all the action returns, by subtracting the mean and dividing by the standard deviation. After that, we can reasonably assume that actions with a negative advantage were bad while actions with a positive advantage were good. OK, now that

we have a way to evaluate each action, we are ready to train our first agent using policy gradients (PG). Let's see how.

Policy Gradients

As discussed earlier, PG algorithms optimize the parameters of a policy by following the gradients toward higher rewards. One popular class of PG algorithms, called *REINFORCE algorithms*, was introduced back in 1992¹² by Ronald Williams. Here is one common variant:

1. First, let the neural network policy play the game several times, and at each step, compute the gradients that would make the chosen action even more likely—but don't apply these gradients yet.
2. Once you have run several episodes, compute each action's advantage, using the method described in the previous section.
3. If an action's advantage is positive, it means that the action was probably good, and you want to apply the gradients computed earlier to make the action even more likely to be chosen in the future. However, if the action's advantage is negative, it means the action was probably bad, and you want to apply the opposite gradients to make this action slightly *less* likely in the future. The solution is to multiply each gradient vector by the corresponding action's advantage.
4. Finally, compute the mean of all the resulting gradient vectors, and use it to perform a Gradient Descent step.

Let's use Keras to implement this algorithm. We will train the neural network policy we built earlier so that it learns to balance the pole on the cart. First, we need a function that will play one step. We will pretend for now that whatever action it takes is the right one so that we can compute the loss and its gradients. These gradients will just be saved for a while, and we will modify them later depending on how good or bad the action turned out to be.

```
def play_one_step(env, obs, model, loss_fn):  
    with tf.GradientTape() as tape:  
        left_proba = model(obs[np.newaxis])  
        action = (tf.random.uniform([1, 1]) > left_proba)  
        y_target = tf.constant([[1.]]) - tf.cast(action, tf.float32)  
        loss = tf.reduce_mean(loss_fn(y_target, left_proba))  
  
        grads = tape.gradient(loss, model.trainable_variables)  
        obs, reward, done, info = env.step(int(action))  
    return obs, reward, done, grads
```

¹² Ronald J. Williams, "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning," *Machine Learning* 8 (1992) : 229–256.

Let's walk though this function:

- Within the `GradientTape` block (see [Chapter 12](#)), we start by calling the model, giving it a single observation. We reshape the observation so it becomes a batch containing a single instance, as the model expects a batch. This outputs the probability of going left.
- Next, we sample a random float between 0 and 1, and we check whether it is greater than `left_proba`. The action will be `False` with probability `left_proba`, or `True` with probability `1 - left_proba`. Once we cast this Boolean to an integer, the action will be 0 (left) or 1 (right) with the appropriate probabilities.
- Next, we define the target probability of going left: it is 1 minus the action (cast to a float). If the action is 0 (left), then the target probability of going left will be 1. If the action is 1 (right), then the target probability will be 0.
- Then we compute the loss using the given loss function, and we use the tape to compute the gradient of the loss with regard to the model's trainable variables. Again, these gradients will be tweaked later, before we apply them, depending on how good or bad the action turned out to be.
- Finally, we play the selected action, and we return the new observation, the reward, whether the episode is ended or not, and of course the gradients that we just computed.

Now let's create another function that will rely on the `play_one_step()` function to play multiple episodes, returning all the rewards and gradients for each episode and each step:

```
def play_multiple_episodes(env, n_episodes, n_max_steps, model, loss_fn):  
    all_rewards = []  
    all_grads = []  
    for episode in range(n_episodes):  
        current_rewards = []  
        current_grads = []  
        obs = env.reset()  
        for step in range(n_max_steps):  
            obs, reward, done, grads = play_one_step(env, obs, model, loss_fn)  
            current_rewards.append(reward)  
            current_grads.append(grads)  
            if done:  
                break  
  
        all_rewards.append(current_rewards)  
        all_grads.append(current_grads)  
  
    return all_rewards, all_grads
```

This code returns a list of reward lists: one reward list per episode, containing one reward per step. It also returns a list of gradient lists: one gradient list per episode,

each containing one tuple of gradients per step and each tuple containing one gradient tensor per trainable variable.

The algorithm will use the `play_multiple_episodes()` function to play the game several times (e.g., 10 times), then it will go back and look at all the rewards, discount them, and normalize them. To do that, we need a couple more functions: the first will compute the sum of future discounted rewards at each step, and the second will normalize all these discounted rewards (i.e., the returns) across many episodes by subtracting the mean and dividing by the standard deviation:

```
def discount_rewards(rewards, discount_factor):
    discounted = np.array(rewards)
    for step in range(len(rewards) - 2, -1, -1):
        discounted[step] += discounted[step + 1] * discount_factor
    return discounted

def discount_and_normalize_rewards(all_rewards, discount_factor):
    all_discounted_rewards = [discount_rewards(rewards, discount_factor)
                              for rewards in all_rewards]
    flat_rewards = np.concatenate(all_discounted_rewards)
    reward_mean = flat_rewards.mean()
    reward_std = flat_rewards.std()
    return [(discounted_rewards - reward_mean) / reward_std
            for discounted_rewards in all_discounted_rewards]
```

Let's check that this works:

```
>>> discount_rewards([10, 0, -50], discount_factor=0.8)
array([-22, -40, -50])
>>> discount_and_normalize_rewards([[10, 0, -50], [10, 20]],
...                                discount_factor=0.8)
...
[array([-0.28435071, -0.86597718, -1.18910299]),
 array([1.26665318, 1.07277777])]
```

The call to `discount_rewards()` returns exactly what we expect (see Figure 18-6). You can verify that the function `discount_and_normalize_rewards()` does indeed return the normalized action advantages for each action in both episodes. Notice that the first episode was much worse than the second, so its normalized advantages are all negative; all actions from the first episode would be considered bad, and conversely all actions from the second episode would be considered good.

We are almost ready to run the algorithm! Now let's define the hyperparameters. We will run 150 training iterations, playing 10 episodes per iteration, and each episode will last at most 200 steps. We will use a discount factor of 0.95:

```
n_iterations = 150
n_episodes_per_update = 10
n_max_steps = 200
discount_factor = 0.95
```

We also need an optimizer and the loss function. A regular Nadam optimizer with learning rate 0.01 will do just fine, and we will use the binary cross-entropy loss function because we are training a binary classifier (there are two possible actions: left or right):

```
optimizer = tf.keras.optimizers.Nadam(learning_rate=0.01)
loss_fn = tf.keras.losses.binary_crossentropy
```

We are now ready to build and run the training loop!

```
for iteration in range(n_iterations):
    all_rewards, all_grads = play_multiple_episodes(
        env, n_episodes_per_update, n_max_steps, model, loss_fn)
    all_final_rewards = discount_and_normalize_rewards(all_rewards,
                                                       discount_factor)

    all_mean_grads = []
    for var_index in range(len(model.trainable_variables)):
        mean_grads = tf.reduce_mean([
            final_reward * all_grads[episode_index][step][var_index]
            for episode_index, final_rewards in enumerate(all_final_rewards)
            for step, final_reward in enumerate(final_rewards)], axis=0)
        all_mean_grads.append(mean_grads)

optimizer.apply_gradients(zip(all_mean_grads, model.trainable_variables))
```

Let's walk through this code:

- At each training iteration, this loop calls the `play_multiple_episodes()` function, which plays 10 episodes and returns the rewards and gradients for each step in each episode.
- Then we call the `discount_and_normalize_rewards()` function to compute each action's normalized advantage, called the `final_reward` in this code. This provides a measure of how good or bad each action actually was, in hindsight.
- Next, we go through each trainable variable, and for each of them we compute the weighted mean of the gradients for that variable over all episodes and all steps, weighted by the `final_reward`.
- Finally, we apply these mean gradients using the optimizer: the model's trainable variables will be tweaked, and hopefully the policy will be a bit better.

And we're done! This code will train the neural network policy, and it will successfully learn to balance the pole on the cart. The mean reward per episode will get very close to 200. By default, that's the maximum for this environment. Success!



Researchers try to find algorithms that work well even when the agent initially knows nothing about the environment. However, unless you are writing a paper, you should not hesitate to inject prior knowledge into the agent, as it will speed up training dramatically. For example, since you know that the pole should be as vertical as possible, you could add negative rewards proportional to the pole's angle. This will make the rewards much less sparse and speed up training. Also, if you already have a reasonably good policy (e.g., hardcoded), you may want to train the neural network to imitate it before using policy gradients to improve it.

The simple policy gradients algorithm we just trained solved the CartPole task, but it would not scale well to larger and more complex tasks. Indeed, it is highly *sample inefficient*, meaning it needs to explore the game for a very long time before it can make significant progress. This is due to the fact that it must run multiple episodes to estimate the advantage of each action, as we have seen. However, it is the foundation of more powerful algorithms, such as *Actor-Critic* algorithms (which we will discuss briefly at the end of this chapter).

We will now look at another popular family of algorithms. Whereas PG algorithms directly try to optimize the policy to increase rewards, the algorithms we will look at now are less direct: the agent learns to estimate the expected return for each state, or for each action in each state, then it uses this knowledge to decide how to act. To understand these algorithms, we must first introduce *Markov decision processes*.

Markov Decision Processes

In the early 20th century, the mathematician Andrey Markov studied stochastic processes with no memory, called *Markov chains*. Such a process has a fixed number of states, and it randomly evolves from one state to another at each step. The probability for it to evolve from a state s to a state s' is fixed, and it depends only on the pair (s, s') , not on past states. This is why we say that the system has no memory.

Figure 18-7 shows an example of a Markov chain with four states.

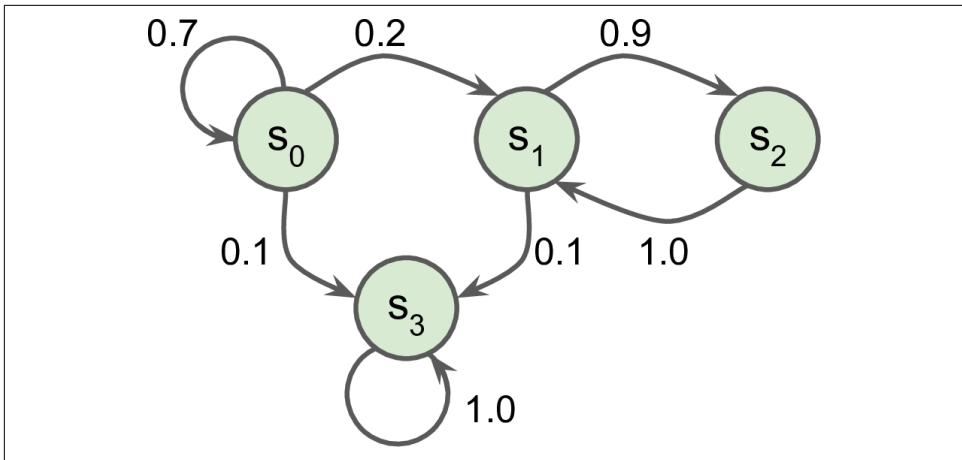


Figure 18-7. Example of a Markov chain

Suppose that the process starts in state s_0 , and there is a 70% chance that it will remain in that state at the next step. Eventually it is bound to leave that state and never come back because no other state points back to s_0 . If it goes to state s_1 , it will then most likely go to state s_2 (90% probability), then immediately back to state s_1 (with 100% probability). It may alternate a number of times between these two states, but eventually it will fall into state s_3 and remain there forever, since there's no way out: this is called a *terminal state*. Markov chains can have very different dynamics, and they are heavily used in thermodynamics, chemistry, statistics, and much more.

Markov decision processes were first described in the 1950s by Richard Bellman.¹³ They resemble Markov chains but with a twist: at each step, an agent can choose one of several possible actions, and the transition probabilities depend on the chosen action. Moreover, some state transitions return some reward (positive or negative), and the agent's goal is to find a policy that will maximize reward over time.

For example, the MDP represented in Figure 18-8 has three states (represented by circles) and up to three possible discrete actions at each step (represented by diamonds).

¹³ Richard Bellman, “A Markovian Decision Process,” *Journal of Mathematics and Mechanics* 6, no. 5 (1957): 679–684.

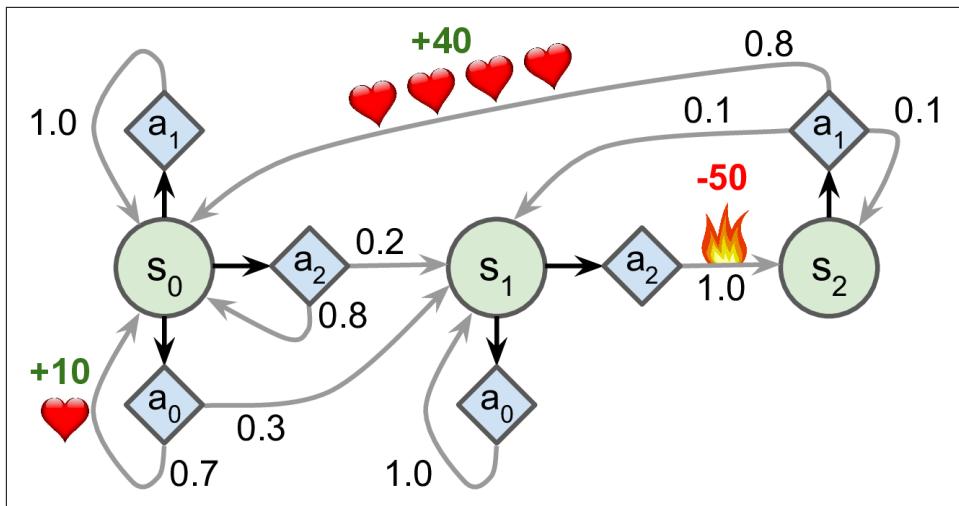


Figure 18-8. Example of a Markov decision process

If it starts in state s_0 , the agent can choose between actions a_0 , a_1 , or a_2 . If it chooses action a_1 , it just remains in state s_0 with certainty, and without any reward. It can thus decide to stay there forever if it wants to. But if it chooses action a_0 , it has a 70% probability of gaining a reward of +10 and remaining in state s_0 . It can then try again and again to gain as much reward as possible, but at one point it is going to end up instead in state s_1 . In state s_1 it has only two possible actions: a_0 or a_2 . It can choose to stay put by repeatedly choosing action a_0 , or it can choose to move on to state s_2 and get a negative reward of -50 (ouch). In state s_2 it has no choice but to take action a_1 , which will most likely lead it back to state s_0 , gaining a reward of +40 on the way. You get the picture. By looking at this MDP, can you guess which strategy will gain the most reward over time? In state s_0 it is clear that action a_0 is the best option, and in state s_2 the agent has no choice but to take action a_1 , but in state s_1 it is not obvious whether the agent should stay put (a_0) or go through the fire (a_2).

Bellman found a way to estimate the *optimal state value* of any state s , noted $V^*(s)$, which is the sum of all discounted future rewards the agent can expect on average after it reaches a state s , assuming it acts optimally. He showed that if the agent acts optimally, then the *Bellman Optimality Equation* applies (see [Equation 18-1](#)). This recursive equation says that if the agent acts optimally, then the optimal value of the current state is equal to the reward it will get on average after taking one optimal

action, plus the expected optimal value of all possible next states that this action can lead to.

Equation 18-1. Bellman Optimality Equation

$$V^*(s) = \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma \cdot V^*(s')] \quad \text{for all } s$$

In this equation:

- $T(s, a, s')$ is the transition probability from state s to state s' , given that the agent chose action a . For example, in [Figure 18-8](#), $T(s_2, a_1, s_0) = 0.8$.
- $R(s, a, s')$ is the reward that the agent gets when it goes from state s to state s' , given that the agent chose action a . For example, in [Figure 18-8](#), $R(s_2, a_1, s_0) = +40$.
- γ is the discount factor.

This equation leads directly to an algorithm that can precisely estimate the optimal state value of every possible state: you first initialize all the state value estimates to zero, and then you iteratively update them using the *Value Iteration* algorithm (see [Equation 18-2](#)). A remarkable result is that, given enough time, these estimates are guaranteed to converge to the optimal state values, corresponding to the optimal policy.

Equation 18-2. Value Iteration algorithm

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma \cdot V_k(s')] \quad \text{for all } s$$

In this equation, $V_k(s)$ is the estimated value of state s at the k^{th} iteration of the algorithm.



This algorithm is an example of *Dynamic Programming*, which breaks down a complex problem into tractable subproblems that can be tackled iteratively.

Knowing the optimal state values can be useful, in particular to evaluate a policy, but it does not give us the optimal policy for the agent. Luckily, Bellman found a very similar algorithm to estimate the optimal *state-action values*, generally called *Q-Values* (Quality Values). The optimal Q-Value of the state-action pair (s, a) , noted $Q^*(s, a)$, is the sum of discounted future rewards the agent can expect on average

after it reaches the state s and chooses action a , but before it sees the outcome of this action, assuming it acts optimally after that action.

Here is how it works: once again, you start by initializing all the Q-Value estimates to zero, then you update them using the *Q-Value Iteration* algorithm (see [Equation 18-3](#)).

Equation 18-3. Q-Value Iteration algorithm

$$Q_{k+1}(s, a) \leftarrow \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma \cdot \max_{a'} Q_k(s', a') \right] \quad \text{for all } (s, a)$$

Once you have the optimal Q-Values, defining the optimal policy, noted $\pi^*(s)$, is trivial: when the agent is in state s , it should choose the action with the highest Q-Value for that state: $\pi^*(s) = \operatorname{argmax}_a Q^*(s, a)$.

Let's apply this algorithm to the MDP represented in [Figure 18-8](#). First, we need to define the MDP:

```
transition_probabilities = [  # shape=[s, a, s']
    [[[0.7, 0.3, 0.0], [1.0, 0.0, 0.0], [0.8, 0.2, 0.0]],
     [[0.0, 1.0, 0.0], None, [0.0, 0.0, 1.0]],
     [None, [0.8, 0.1, 0.1], None]
    ]
    rewards = [  # shape=[s, a, s']
        [[+10, 0, 0], [0, 0, 0], [0, 0, 0]],
        [[0, 0, 0], [0, 0, 0], [0, 0, -50]],
        [[0, 0, 0], [+40, 0, 0], [0, 0, 0]]
    ]
    possible_actions = [[0, 1, 2], [0, 2], [1]]
]
```

For example, to know the transition probability of going from s_2 to s_0 after playing action a_1 , we will look up `transition_probabilities[2][1][0]` (which is 0.8). Similarly, to get the corresponding reward, we will look up `rewards[2][1][0]` (which is +40). And to get the list of possible actions in s_2 , we will look up `possible_actions[2]` (in this case, only action a_1 is possible). Next, we must initialize all the Q-Values to 0 (except for the impossible actions, for which we set the Q-Values to $-\infty$):

```
Q_values = np.full((3, 3), -np.inf) # -np.inf for impossible actions
for state, actions in enumerate(possible_actions):
    Q_values[state, actions] = 0.0 # for all possible actions
```

Now let's run the Q-Value Iteration algorithm. It applies [Equation 18-3](#) repeatedly, to all Q-Values, for every state and every possible action:

```
gamma = 0.90 # the discount factor
```

```

for iteration in range(50):
    Q_prev = Q_values.copy()
    for s in range(3):
        for a in possible_actions[s]:
            Q_values[s, a] = np.sum([
                transition_probabilities[s][a][sp]
                * (rewards[s][a][sp] + gamma * Q_prev[sp].max())
            for sp in range(3)])

```

That's it! The resulting Q-Values look like this:

```

>>> Q_values
array([[18.91891892, 17.02702702, 13.62162162],
       [ 0.          , -inf, -4.87971488],
       [-inf, 50.13365013, -inf]])

```

For example, when the agent is in state s_0 and it chooses action a_1 , the expected sum of discounted future rewards is approximately 17.0.

For each state, let's look at the action that has the highest Q-Value:

```

>>> Q_values.argmax(axis=1) # optimal action for each state
array([0, 0, 1])

```

This gives us the optimal policy for this MDP, when using a discount factor of 0.90: in state s_0 choose action a_0 ; in state s_1 choose action a_0 (i.e., stay put); and in state s_2 choose action a_1 (the only possible action). Interestingly, if we increase the discount factor to 0.95, the optimal policy changes: in state s_1 the best action becomes a_2 (go through the fire!). This makes sense because the more you value future rewards, the more you are willing to put up with some pain now for the promise of future bliss.

Temporal Difference Learning

Reinforcement Learning problems with discrete actions can often be modeled as Markov decision processes, but the agent initially has no idea what the transition probabilities are (it does not know $T(s, a, s')$), and it does not know what the rewards are going to be either (it does not know $R(s, a, s')$). It must experience each state and each transition at least once to know the rewards, and it must experience them multiple times if it is to have a reasonable estimate of the transition probabilities.

The *Temporal Difference Learning* (TD Learning) algorithm is very similar to the Value Iteration algorithm, but tweaked to take into account the fact that the agent has only partial knowledge of the MDP. In general we assume that the agent initially knows only the possible states and actions, and nothing more. The agent uses an *exploration policy*—for example, a purely random policy—to explore the MDP, and

as it progresses, the TD Learning algorithm updates the estimates of the state values based on the transitions and rewards that are actually observed (see [Equation 18-4](#)).

Equation 18-4. TD Learning algorithm

$$V_{k+1}(s) \leftarrow (1 - \alpha)V_k(s) + \alpha(r + \gamma \cdot V_k(s'))$$

or, equivalently:

$$V_{k+1}(s) \leftarrow V_k(s) + \alpha \cdot \delta_k(s, r, s')$$

$$\text{with } \delta_k(s, r, s') = r + \gamma \cdot V_k(s') - V_k(s)$$

In this equation:

- α is the learning rate (e.g., 0.01).
- $r + \gamma \cdot V_k(s')$ is called the *TD target*.
- $\delta_k(s, r, s')$ is called the *TD error*.

A more concise way of writing the first form of this equation is to use the notation $a \xleftarrow{\alpha} b$, which means $a_{k+1} \leftarrow (1 - \alpha) \cdot a_k + \alpha \cdot b_k$. So, the first line of [Equation 18-4](#) can be rewritten like this: $V(s) \xleftarrow{\alpha} r + \gamma \cdot V(s')$.



TD Learning has many similarities with Stochastic Gradient Descent, in particular the fact that it handles one sample at a time. Moreover, just like Stochastic GD, it can only truly converge if you gradually reduce the learning rate, otherwise it will keep bouncing around the optimum Q-Values.

For each state s , this algorithm keeps track of a running average of the immediate rewards the agent gets upon leaving that state, plus the rewards it expects to get later, assuming it acts optimally.

Q-Learning

Similarly, the Q-Learning algorithm is an adaptation of the Q-Value Iteration algorithm to the situation where the transition probabilities and the rewards are initially unknown (see [Equation 18-5](#)). Q-Learning works by watching an agent play (e.g., randomly) and gradually improving its estimates of the Q-Values. Once it has

accurate Q-Value estimates (or close enough), then the optimal policy is just choosing the action that has the highest Q-Value (i.e., the greedy policy).

Equation 18-5. Q-Learning algorithm

$$Q(s, a) \leftarrow r + \gamma \cdot \max_{a'} Q(s', a')$$

For each state-action pair (s, a) , this algorithm keeps track of a running average of the rewards r the agent gets upon leaving the state s with action a , plus the sum of discounted future rewards it expects to get. To estimate this sum, we take the maximum of the Q-Value estimates for the next state s' , since we assume that the target policy would act optimally from then on.

Let's implement the Q-Learning algorithm. First, we will need to make an agent explore the environment. For this, we need a step function so that the agent can execute one action and get the resulting state and reward:

```
def step(state, action):
    probas = transition_probabilities[state][action]
    next_state = np.random.choice([0, 1, 2], p=probas)
    reward = rewards[state][action][next_state]
    return next_state, reward
```

Now let's implement the agent's exploration policy. Since the state space is pretty small, a simple random policy will be sufficient. If we run the algorithm for long enough, the agent will visit every state many times, and it will also try every possible action many times:

```
def exploration_policy(state):
    return np.random.choice(possible_actions[state])
```

Next, after we initialize the Q-Values just like earlier, we are ready to run the Q-Learning algorithm with learning rate decay (using power scheduling, introduced in Chapter 11):

```
alpha0 = 0.05 # initial learning rate
decay = 0.005 # learning rate decay
gamma = 0.90 # discount factor
state = 0 # initial state

for iteration in range(10_000):
    action = exploration_policy(state)
    next_state, reward = step(state, action)
    next_value = Q_values[next_state].max() # greedy policy at the next step
    alpha = alpha0 / (1 + iteration * decay)
    Q_values[state, action] *= 1 - alpha
    Q_values[state, action] += alpha * (reward + gamma * next_value)
    state = next_state
```

This algorithm will converge to the optimal Q-Values, but it will take many iterations, and possibly quite a lot of hyperparameter tuning. As you can see in [Figure 18-9](#), the Q-Value Iteration algorithm (left) converges very quickly, in fewer than 20 iterations, while the Q-Learning algorithm (right) takes about 8,000 iterations to converge. Obviously, not knowing the transition probabilities or the rewards makes finding the optimal policy significantly harder!

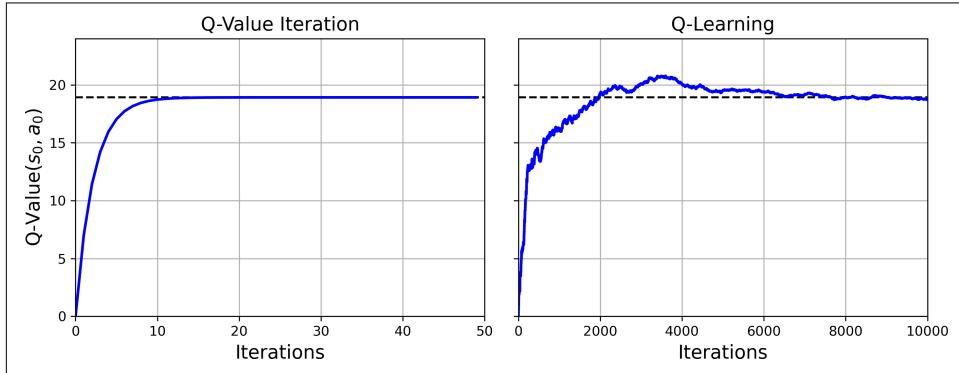


Figure 18-9. The Q-Value Iteration algorithm (left) versus the Q-Learning algorithm (right)

The Q-Learning algorithm is called an *off-policy* algorithm because the policy being trained is not necessarily the one used during training. For example, in the code we just ran, the policy being executed (the exploration policy) was completely random, while the policy being trained was never used. After training, the optimal policy corresponds to systematically choosing the action with the highest Q-Value. Conversely, the Policy Gradients algorithm is an *on-policy* algorithm: it explores the world using the policy being trained. It is somewhat surprising that Q-Learning is capable of learning the optimal policy by just watching an agent act randomly. Imagine learning to play golf when your teacher is a blindfolded monkey. Can we do better?

Exploration Policies

Of course, Q-Learning can work only if the exploration policy explores the MDP thoroughly enough. Although a purely random policy is guaranteed to eventually visit every state and every transition many times, it may take an extremely long time to do so. Therefore, a better option is to use the *ϵ -greedy policy* (ϵ is epsilon): at each step it acts randomly with probability ϵ , or greedily with probability $1-\epsilon$ (i.e., choosing the action with the highest Q-Value). The advantage of the ϵ -greedy policy (compared to a completely random policy) is that it will spend more and more time exploring the interesting parts of the environment, as the Q-Value estimates get better and better, while still spending some time visiting unknown regions of the MDP. It is

quite common to start with a high value for ε (e.g., 1.0) and then gradually reduce it (e.g., down to 0.05).

Alternatively, rather than relying only on chance for exploration, another approach is to encourage the exploration policy to try actions that it has not tried much before. This can be implemented as a bonus added to the Q-Value estimates, as shown in [Equation 18-6](#).

Equation 18-6. Q-Learning using an exploration function

$$Q(s, a) \leftarrow r + \gamma \cdot \max_{a'} f(Q(s', a'), N(s', a'))$$

In this equation:

- $N(s', a')$ counts the number of times the action a' was chosen in state s' .
- $f(Q, N)$ is an *exploration function*, such as $f(Q, N) = Q + \kappa/(1 + N)$, where κ is a curiosity hyperparameter that measures how much the agent is attracted to the unknown.

Approximate Q-Learning and Deep Q-Learning

The main problem with Q-Learning is that it does not scale well to large (or even medium) MDPs with many states and actions. For example, suppose you wanted to use Q-Learning to train an agent to play *Ms. Pac-Man* (see [Figure 18-1](#)). There are about 150 pellets that Ms. Pac-Man can eat, each of which can be present or absent (i.e., already eaten). So, the number of possible states is greater than $2^{150} \approx 10^{45}$. And if you add all the possible combinations of positions for all the ghosts and Ms. Pac-Man, the number of possible states becomes larger than the number of atoms in our planet, so there's absolutely no way you can keep track of an estimate for every single Q-Value.

The solution is to find a function $Q_\theta(s, a)$ that approximates the Q-Value of any state-action pair (s, a) using a manageable number of parameters (given by the parameter vector θ). This is called *Approximate Q-Learning*. For years it was recommended to use linear combinations of handcrafted features extracted from the state (e.g., distance of the closest ghosts, their directions, and so on) to estimate Q-Values, but in 2013, [DeepMind](#) showed that using deep neural networks can work much better, especially for complex problems, and it does not require any feature engineering. A DNN used to estimate Q-Values is called a *Deep Q-Network* (DQN), and using a DQN for Approximate Q-Learning is called *Deep Q-Learning*.

Now, how can we train a DQN? Well, consider the approximate Q-Value computed by the DQN for a given state-action pair (s, a) . Thanks to Bellman, we know we want

this approximate Q-Value to be as close as possible to the reward r that we actually observe after playing action a in state s , plus the discounted value of playing optimally from then on. To estimate this sum of future discounted rewards, we can just execute the DQN on the next state s' , for all possible actions a' . We get an approximate future Q-Value for each possible action. We then pick the highest (since we assume we will be playing optimally) and discount it, and this gives us an estimate of the sum of future discounted rewards. By summing the reward r and the future discounted value estimate, we get a target Q-Value $y(s, a)$ for the state-action pair (s, a) , as shown in [Equation 18-7](#).

Equation 18-7. Target Q-Value

$$y(s, a) = r + \gamma \cdot \max_{a'} Q_\theta(s', a')$$

With this target Q-Value, we can run a training step using any Gradient Descent algorithm. Specifically, we generally try to minimize the squared error between the estimated Q-Value $Q_\theta(s, a)$ and the target Q-Value $y(s, a)$, or the Huber loss to reduce the algorithm's sensitivity to large errors. And that's the Deep Q-Learning algorithm! Let's see how to implement it to solve the CartPole environment.

Implementing Deep Q-Learning

The first thing we need is a Deep Q-Network. In theory, you need a neural net that takes a state-action pair as input, and outputs an approximate Q-Value. However, in practice it's much more efficient to use a neural net that takes only a state as input, and outputs one approximate Q-Value for each possible action. To solve the CartPole environment, we do not need a very complicated neural net; a couple of hidden layers will do:

```
input_shape = [4] # == env.observation_space.shape
n_outputs = 2 # == env.action_space.n

model = tf.keras.Sequential([
    tf.keras.layers.Dense(32, activation="elu", input_shape=input_shape),
    tf.keras.layers.Dense(32, activation="elu"),
    tf.keras.layers.Dense(n_outputs)
])
```

To select an action using this DQN, we pick the action with the largest predicted Q-Value. To ensure that the agent explores the environment, we will use an ϵ -greedy policy (i.e., we will choose a random action with probability ϵ):

```
def epsilon_greedy_policy(state, epsilon=0):
    if np.random.rand() < epsilon:
        return np.random.randint(n_outputs) # random action
```

```

    else:
        Q_values = model.predict(state[np.newaxis])[0]
        return Q_values.argmax() # optimal action according to the DQN

```

Instead of training the DQN based only on the latest experiences, we will store all experiences in a *replay buffer* (or *replay memory*), and we will sample a random training batch from it at each training iteration. This helps reduce the correlations between the experiences in a training batch, which tremendously helps training. For this, we will just use a double-ended queue (`deque`):

```

from collections import deque

replay_buffer = deque(maxlen=2000)

```



A `deque` is a queue for which elements can be efficiently added or removed on both ends. Inserting and deleting items from the ends of the queue is very fast, but random access can be slow when the queue gets long. If you need a very large replay buffer, you should use a circular buffer instead (see the notebook for an implementation). Or check out [DeepMind's Reverb library](#).

Each experience will be composed of five elements: a state s , the action a that the agent took, the resulting reward r , the next state s' it reached, and finally a boolean indicating whether the episode ended at that point (`done`). We will need a small function to sample a random batch of experiences from the replay buffer. It will return five NumPy arrays corresponding to the five experience elements:

```

def sample_experiences(batch_size):
    indices = np.random.randint(len(replay_buffer), size=batch_size)
    batch = [replay_buffer[index] for index in indices]
    states, actions, rewards, next_states, dones = [
        np.array([experience[field_index] for experience in batch])
        for field_index in range(5)
    ]
    return states, actions, rewards, next_states, dones

```

Let's also create a function that will play a single step using the ϵ -greedy policy, then store the resulting experience in the replay buffer:

```

def play_one_step(env, state, epsilon):
    action = epsilon_greedy_policy(state, epsilon)
    next_state, reward, done, info = env.step(action)
    replay_buffer.append((state, action, reward, next_state, done))
    return next_state, reward, done, info

```

Finally, let's create one last function that will sample a batch of experiences from the replay buffer and train the DQN by performing a single Gradient Descent step on this batch:

```

batch_size = 32
discount_factor = 0.95
optimizer = tf.keras.optimizers.Nadam(learning_rate=1e-2)
loss_fn = tf.keras.losses.mean_squared_error

def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones = experiences
    next_Q_values = model.predict(next_states)
    max_next_Q_values = next_Q_values.max(axis=1)
    target_Q_values = (rewards +
                        (1 - dones) * discount_factor * max_next_Q_values)
    target_Q_values = target_Q_values.reshape(-1, 1)
    mask = tf.one_hot(actions, n_outputs)
    with tf.GradientTape() as tape:
        all_Q_values = model(states)
        Q_values = tf.reduce_sum(all_Q_values * mask, axis=1, keepdims=True)
        loss = tf.reduce_mean(loss_fn(target_Q_values, Q_values))

    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))

```

Let's go through this code:

- First we define some hyperparameters, and we create the optimizer and the loss function.
- Then we create the `training_step()` function. It starts by sampling a batch of experiences, then it uses the DQN to predict the Q-Value for each possible action in each experience's next state. Since we assume that the agent will be playing optimally, we only keep the maximum Q-Value for each next state. Next, we use [Equation 18-7](#) to compute the target Q-Value for each experience's state-action pair.
- Next, we want to use the DQN to compute the Q-Value for each experienced state-action pair. However, the DQN will also output the Q-Values for the other possible actions, not just for the action that was actually chosen by the agent. So we need to mask out all the Q-Values we do not need. The `tf.one_hot()` function makes it possible to convert an array of action indices into such a mask. For example, if the first three experiences contain actions 1, 1, 0, respectively, then the mask will start with `[[0, 1], [0, 1], [1, 0], ...]`. We can then multiply the DQN's output with this mask, and this will zero out all the Q-Values we do not want. We then sum over axis 1 to get rid of all the zeros, keeping only the Q-Values of the experienced state-action pairs. This gives us the `Q_values` tensor, containing one predicted Q-Value for each experience in the batch.
- Then we compute the loss: it is the mean squared error between the target and predicted Q-Values for the experienced state-action pairs.

- Finally, we perform a Gradient Descent step to minimize the loss with regard to the model's trainable variables.

This was the hardest part. Now training the model is straightforward:

```
for episode in range(600):
    obs = env.reset()
    for step in range(200):
        epsilon = max(1 - episode / 500, 0.01)
        obs, reward, done, info = play_one_step(env, obs, epsilon)
        if done:
            break

    if episode > 50:
        training_step(batch_size)
```

We run 600 episodes, each for a maximum of 200 steps. At each step, we first compute the `epsilon` value for the ϵ -greedy policy: it will go from 1 down to 0.01, linearly, in a bit under 500 episodes. Then we call the `play_one_step()` function, which will use the ϵ -greedy policy to pick an action, then execute it and record the experience in the replay buffer. If the episode is done, we exit the loop. Finally, if we are past episode 50, we call the `training_step()` function to train the model on one batch sampled from the replay buffer. The reason we play many episodes without training is to give the replay buffer some time to fill up (if we don't wait enough, then there will not be enough diversity in the replay buffer). And that's it, we just implemented the Deep Q-Learning algorithm!

Figure 18-10 shows the total rewards the agent got during each episode.

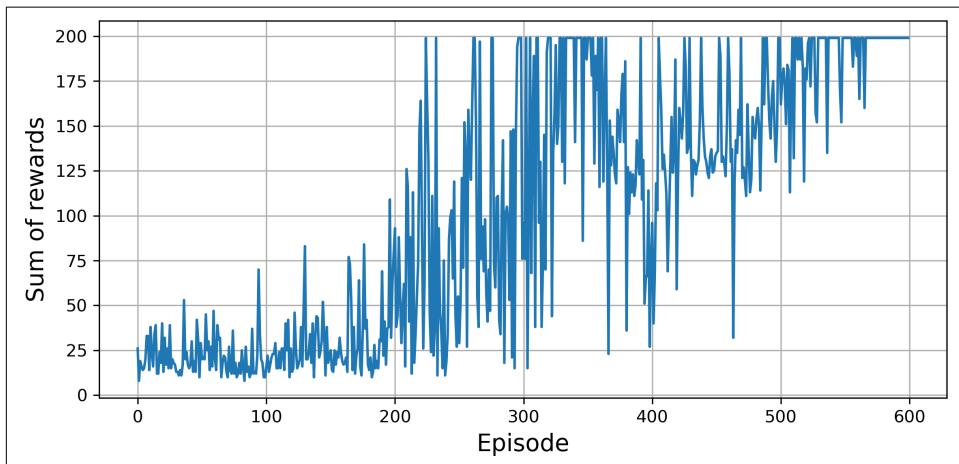


Figure 18-10. Learning curve of the Deep Q-Learning algorithm

As you can see, the algorithm took a while to start learning anything, in part because ϵ was very high at the beginning. Then its progress was erratic: it first reached the max reward around episode 220, but it immediately dropped, then bounced up and down a few times, and when it looked like it had finally stabilized near the max reward, around episode 320, its score soon dropped down dramatically! This is called *catastrophic forgetting*, and it is one of the big problems facing virtually all RL algorithms: as the agent explores the environment, it updates its policy, but what it learns in one part of the environment may break what it learned earlier in other parts of the environment. The experiences are quite correlated, and the learning environment keeps changing—this is not ideal for Gradient Descent! If you increase the size of the replay buffer, the algorithm will be less subject to this problem. Tuning the learning rate may also help. But the truth is, Reinforcement Learning is hard: training is often unstable, and you may need to try many hyperparameter values and random seeds before you find a combination that works well. For example, if you try changing the activation function from "elu" to "relu", the performance will be much lower.



Reinforcement Learning is notoriously difficult, largely because of the training instabilities and the huge sensitivity to the choice of hyperparameter values and random seeds.¹⁴ As the researcher Andrej Karpathy put it: "[Supervised learning] wants to work. [...] RL must be forced to work." You will need time, patience, perseverance, and perhaps a bit of luck too. This is a major reason RL is not as widely adopted as regular Deep Learning (e.g., convolutional nets). But there are a few real-world applications, beyond AlphaGo and Atari games: for example, Google uses RL to optimize its datacenter costs, and it is used in some robotics applications, for hyperparameter tuning, and in recommender systems.

You might wonder why we didn't plot the loss. It turns out that loss is a poor indicator of the model's performance. The loss might go down, yet the agent might perform worse (e.g., this can happen when the agent gets stuck in one small region of the environment, and the DQN starts overfitting this region). Conversely, the loss could go up, yet the agent might perform better (e.g., if the DQN was underestimating the Q-Values, and it starts correctly increasing its predictions, the agent will likely perform better, getting more rewards, but the loss might increase because the DQN also sets the targets, which will be larger too). So it's preferable to plot the rewards.

The basic Deep Q-Learning algorithm we've been using so far would be too unstable to learn to play Atari games. So how did DeepMind do it? Well, they tweaked the algorithm!

¹⁴ A great [2018 post](#) by Alex Irpan nicely lays out RL's biggest difficulties and limitations.

Deep Q-Learning Variants

Let's look at a few variants of the Deep Q-Learning algorithm that can stabilize and speed up training.

Fixed Q-Value Targets

In the basic Deep Q-Learning algorithm, the model is used both to make predictions and to set its own targets. This can lead to a situation analogous to a dog chasing its own tail. This feedback loop can make the network unstable: it can diverge, oscillate, freeze, and so on. To solve this problem, in their 2013 paper the DeepMind researchers used two DQNs instead of one: the first is the *online model*, which learns at each step and is used to move the agent around, and the other is the *target model* used only to define the targets. The target model is just a clone of the online model:

```
target = tf.keras.models.clone_model(model) # clone the model's architecture  
target.set_weights(model.get_weights()) # copy the weights
```

Then, in the `training_step()` function, we just need to change one line to use the target model instead of the online model when computing the Q-Values of the next states:

```
next_Q_values = target.predict(next_states)
```

Finally, in the training loop, we must copy the weights of the online model to the target model, at regular intervals (e.g., every 50 episodes):

```
if episode % 50 == 0:  
    target.set_weights(model.get_weights())
```

Since the target model is updated much less often than the online model, the Q-Value targets are more stable, the feedback loop we discussed earlier is dampened, and its effects are less severe. This approach was one of DeepMind researchers' main contributions in their 2013 paper, allowing agents to learn to play Atari games from raw pixels. To stabilize training, they used a tiny learning rate of 0.00025, they updated the target model only every 10,000 steps (instead of 50), and they used a very large replay buffer of 1 million experiences. They decreased `epsilon` very slowly, from 1 to 0.1 in 1 million steps, and they let the algorithm run for 50 million steps. Moreover, their DQN was a deep convolutional net.

Now let's take a look at another DQN variant that managed to beat the state of the art once more.

Double DQN

In a [2015 paper](#),¹⁵ DeepMind researchers tweaked their DQN algorithm, increasing its performance and somewhat stabilizing training. They called this variant *Double DQN*. The update was based on the observation that the target network is prone to overestimating Q-Values. Indeed, suppose all actions are equally good: the Q-Values estimated by the target model should be identical, but since they are approximations, some may be slightly greater than others, by pure chance. The target model will always select the largest Q-Value, which will be slightly greater than the mean Q-Value, most likely overestimating the true Q-Value (a bit like counting the height of the tallest random wave when measuring the depth of a pool). To fix this, they proposed using the online model instead of the target model when selecting the best actions for the next states, and using the target model only to estimate the Q-Values for these best actions. Here is the updated `training_step()` function:

```
def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones = experiences
    next_Q_values = model.predict(next_states) # not target.predict(...)
    best_next_actions = next_Q_values.argmax(axis=1)
    next_mask = tf.one_hot(best_next_actions, n_outputs).numpy()
    max_next_Q_values = (target.predict(next_states) * next_mask).sum(axis=1)
    [...] # the rest is the same as earlier
```

Just a few months later, another improvement to the DQN algorithm was proposed.

Prioritized Experience Replay

Instead of sampling experiences *uniformly* from the replay buffer, why not sample important experiences more frequently? This idea is called *importance sampling* (IS) or *prioritized experience replay* (PER), and it was introduced in a [2015 paper](#)¹⁶ by DeepMind researchers (once again!).

More specifically, experiences are considered “important” if they are likely to lead to fast learning progress. But how can we estimate this? One reasonable approach is to measure the magnitude of the TD error $\delta = r + \gamma \cdot V(s') - V(s)$. A large TD error indicates that a transition (s, a, s') is very surprising, and thus probably worth learning from.¹⁷ When an experience is recorded in the replay buffer, its priority is set to a very large value, to ensure that it gets sampled at least once. However, once it is sampled (and every time it is sampled), the TD error δ is computed, and this experience’s

¹⁵ Hado van Hasselt et al., “Deep Reinforcement Learning with Double Q-Learning,” *Proceedings of the 30th AAAI Conference on Artificial Intelligence* (2015): 2094–2100.

¹⁶ Tom Schaul et al., “Prioritized Experience Replay,” arXiv preprint arXiv:1511.05952 (2015).

¹⁷ It could also just be that the rewards are noisy, in which case there are better methods for estimating an experience’s importance (see the paper for some examples).

priority is set to $p = |\delta|$ (plus a small constant to ensure that every experience has a non-zero probability of being sampled). The probability P of sampling an experience with priority p is proportional to p^ζ , where ζ is a hyperparameter that controls how greedy we want importance sampling to be: when $\zeta = 0$, we just get uniform sampling, and when $\zeta = 1$, we get full-blown importance sampling. In the paper, the authors used $\zeta = 0.6$, but the optimal value will depend on the task.

There's one catch, though: since the samples will be biased toward important experiences, we must compensate for this bias during training by downweighting the experiences according to their importance, or else the model will just overfit the important experiences. To be clear, we want important experiences to be sampled more often, but this also means we must give them a lower weight during training. To do this, we define each experience's training weight as $w = (n P)^{-\beta}$, where n is the number of experiences in the replay buffer, and β is a hyperparameter that controls how much we want to compensate for the importance sampling bias (0 means not at all, while 1 means entirely). In the paper, the authors used $\beta = 0.4$ at the beginning of training and linearly increased it to $\beta = 1$ by the end of training. Again, the optimal value will depend on the task, but if you increase one, you will usually want to increase the other as well.

Now let's look at one last important variant of the DQN algorithm.

Dueling DQN

The *Dueling DQN* algorithm (DDQN, not to be confused with Double DQN, although both techniques can easily be combined) was introduced in yet another [2015 paper¹⁸](#) by DeepMind researchers. To understand how it works, we must first note that the Q-Value of a state-action pair (s, a) can be expressed as $Q(s, a) = V(s) + A(s, a)$, where $V(s)$ is the value of state s and $A(s, a)$ is the *advantage* of taking the action a in state s , compared to all other possible actions in that state. Moreover, the value of a state is equal to the Q-Value of the best action a^* for that state (since we assume the optimal policy will pick the best action), so $V(s) = Q(s, a^*)$, which implies that $A(s, a^*) = 0$. In a Dueling DQN, the model estimates both the value of the state and the advantage of each possible action. Since the best action should have an advantage of 0, the model subtracts the maximum predicted advantage from all predicted advantages. Here is a simple Dueling DQN model, implemented using the Functional API:

```
input_states = tf.keras.layers.Input(shape=[4])
hidden1 = tf.keras.layers.Dense(32, activation="elu")(input_states)
hidden2 = tf.keras.layers.Dense(32, activation="elu")(hidden1)
```

¹⁸ Ziyu Wang et al., “Dueling Network Architectures for Deep Reinforcement Learning,” arXiv preprint arXiv:1511.06581 (2015).

```

state_values = tf.keras.layers.Dense(1)(hidden2)
raw_advantages = tf.keras.layers.Dense(n_outputs)(hidden2)
advantages = raw_advantages - tf.reduce_max(raw_advantages, axis=1,
                                             keepdims=True)
Q_values = state_values + advantages
model = tf.keras.Model(inputs=[input_states], outputs=[Q_values])

```

The rest of the algorithm is just the same as earlier. In fact, you can build a Double Dueling DQN and combine it with prioritized experience replay! More generally, many RL techniques can be combined, as DeepMind demonstrated in a [2017 paper](#).¹⁹ The paper's authors combined six different techniques into an agent called *Rainbow*, which largely outperformed the state of the art.

As you can see, Deep Reinforcement Learning is a fast growing field and there's much more to discover!

Overview of Some Popular RL Algorithms

Before we close this chapter let's take a brief look at a few other popular algorithms.

*AlphaGo*²⁰

AlphaGo uses a variant of *Monte Carlo Tree Search* (MCTS) based on deep neural networks to beat human champions at the game of Go. MCTS was invented back in 1949 by Nicholas Metropolis and Stanislaw Ulam. It selects the best move to play after running many simulations: it repeatedly explores the search tree starting from the current position, and spends more time on the most promising branches. When it reaches a node that it hasn't visited before, it plays randomly until the game ends, and it updates its estimates for each visited node—excluding the random moves—increasing or decreasing each estimate depending on the final outcome. AlphaGo is based on the same principle, but it uses a policy network to select moves, rather than playing randomly. This policy net is trained using Policy Gradients. The original algorithm actually involves 3 more neural networks, and it's more complicated, but it was later simplified in the [AlphaGo Zero paper](#)²¹, which uses a single neural network to both select moves and to evaluate game states. The [AlphaZero paper](#)²² generalized this algorithm, making it capable of tackling not only the game of Go, but also chess and shogi (Japanese chess). Lastly, the [MuZero paper](#)²³ continued to improve upon this algorithm,

¹⁹ Matteo Hessel et al., “Rainbow: Combining Improvements in Deep Reinforcement Learning,” arXiv preprint arXiv:1710.02298 (2017): 3215–3222.

²⁰ David Silver et al., “Mastering the game of Go with deep neural networks and tree search” (2016).

²¹ David Silver et al., “Mastering the game of Go without human knowledge” (2017).

²² David Silver et al., “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm” (2017).

outperforming the previous iterations even though the agent starts out without even knowing the rules of the game!

Actor-Critic algorithms

Actor-Critics are a family of RL algorithms that combine Policy Gradients with Deep Q-Networks. An Actor-Critic agent contains two neural networks: a policy net and a DQN. The DQN is trained normally, by learning from the agent's experiences. The policy net learns differently (and much faster) than in regular PG: instead of estimating the value of each action by going through multiple episodes, then summing the future discounted rewards for each action, and finally normalizing them, the agent (actor) relies on the action values estimated by the DQN (critic). It's a bit like an athlete (the agent) learning with the help of a coach (the DQN).

Asynchronous Advantage Actor-Critic²⁴ (A3C)

An important Actor-Critic variant introduced by DeepMind researchers in 2016, where multiple agents learn in parallel, exploring different copies of the environment. At regular intervals, but asynchronously (hence the name), each agent pushes some weight updates to a master network, then it pulls the latest weights from that network. Each agent thus contributes to improving the master network and benefits from what the other agents have learned. Moreover, instead of estimating the Q-Values, the DQN estimates the advantage of each action (hence the second A in the name), which stabilizes training.

Advantage Actor-Critic (A2C)

A variant of the A3C algorithm that removes the asynchronicity. All model updates are synchronous, so gradient updates are performed over larger batches, which allows the model to better utilize the power of the GPU.

Soft Actor-Critic²⁵ (SAC)

An Actor-Critic variant proposed in 2018 by Tuomas Haarnoja and other UC Berkeley researchers. It learns not only rewards, but also to maximize the entropy of its actions. In other words, it tries to be as unpredictable as possible while still getting as many rewards as possible. This encourages the agent to explore the environment, which speeds up training, and makes it less likely to repeatedly execute the same action when the DQN produces imperfect estimates. This

²³ Julian Schrittwieser et al., "Mastering Atari, Go, chess and shogi by planning with a learned model" (2019).

²⁴ Volodymyr Mnih et al., "Asynchronous Methods for Deep Reinforcement Learning," *Proceedings of the 33rd International Conference on Machine Learning* (2016): 1928–1937.

²⁵ Tuomas Haarnoja et al., "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor," *Proceedings of the 35th International Conference on Machine Learning* (2018): 1856–1865.

algorithm has demonstrated an amazing sample efficiency (contrary to all the previous algorithms, which learn very slowly).

Proximal Policy Optimization²⁶ (PPO)

An algorithm based on A2C that clips the loss function to avoid excessively large weight updates (which often lead to training instabilities). PPO is a simplification of the previous *Trust Region Policy Optimization²⁷* (TRPO) algorithm, also by John Schulman and other OpenAI researchers. OpenAI made the news in April 2019 with their AI called OpenAI Five, based on the PPO algorithm, which defeated the world champions at the multiplayer game *Dota 2*.

Curiosity-based exploration²⁸

A recurring problem in RL is the sparsity of the rewards, which makes learning very slow and inefficient. Deepak Pathak and other UC Berkeley researchers have proposed an exciting way to tackle this issue: why not ignore the rewards, and just make the agent extremely curious to explore the environment? The rewards thus become intrinsic to the agent, rather than coming from the environment. Similarly, stimulating curiosity in a child is more likely to give good results than purely rewarding the child for getting good grades. How does this work? The agent continuously tries to predict the outcome of its actions, and it seeks situations where the outcome does not match its predictions. In other words, it wants to be surprised. If the outcome is predictable (boring), it goes elsewhere. However, if the outcome is unpredictable but the agent notices that it has no control over it, it also gets bored after a while. With only curiosity, the authors succeeded in training an agent at many video games: even though the agent gets no penalty for losing, the game starts over, which is boring so it learns to avoid it.

Open-Ended Learning

The objective of *Open-Ended Learning* (OEL) is to train agents capable of endlessly learning new and interesting tasks, typically generated procedurally. We're not there yet, but there has been some amazing progress over the last few years. For example, a [2019 paper²⁹](#) by a team of researchers from Uber AI introduced the *POET algorithm*. It generates multiple simulated 2D environments with bumps and holes, and it trains one agent per environment: the agent's goal is to walk as fast as possible while avoiding the obstacles. The algorithm

²⁶ John Schulman et al., “Proximal Policy Optimization Algorithms,” arXiv preprint arXiv:1707.06347 (2017).

²⁷ John Schulman et al., “Trust Region Policy Optimization,” *Proceedings of the 32nd International Conference on Machine Learning* (2015): 1889–1897.

²⁸ Deepak Pathak et al., “Curiosity-Driven Exploration by Self-Supervised Prediction,” *Proceedings of the 34th International Conference on Machine Learning* (2017): 2778–2787.

²⁹ Rui Wang et al., “Paired Open-Ended Trailblazer (POET): Endlessly Generating Increasingly Complex and Diverse Learning Environments and Their Solutions” (2019).

starts out with simple environments, but they gradually get harder over time: this is called *curriculum learning*. Moreover, although each agent is only trained within one environment, it must regularly compete against other agents, across all environments. In each environment, the winner is copied over and it replaces the agent that was there before. This way, knowledge is regularly transferred across environments, and the most adaptable agents are selected. In the end, the agents are much better walkers than agents trained on a single task, and they can tackle much harder environments. Of course, this principle can be applied to other environments and tasks as well. If you're interested in OEL, make sure to check out the Enhanced POET paper, as well as DeepMind's [2021 paper³⁰](#) on this topic.



If you'd like to learn more about Reinforcement Learning, check out the book [Reinforcement Learning, Industrial Applications of Intelligent Agents](#) by Phil Winder (O'Reilly, 2020).

We covered many topics in this chapter: Policy Gradients, Markov chains, Markov decision processes, Q-Learning, Approximate Q-Learning, and Deep Q-Learning and its main variants (fixed Q-Value targets, Double DQN, Dueling DQN, and prioritized experience replay), and finally we took a quick look at a few other popular algorithms. Reinforcement Learning is a huge and exciting field, with new ideas and algorithms popping out every day, so I hope this chapter sparked your curiosity: there is a whole world to explore!

Exercises

1. How would you define Reinforcement Learning? How is it different from regular supervised or unsupervised learning?
2. Can you think of three possible applications of RL that were not mentioned in this chapter? For each of them, what is the environment? What is the agent? What are some possible actions? What are the rewards?
3. What is the discount factor? Can the optimal policy change if you modify the discount factor?
4. How do you measure the performance of a Reinforcement Learning agent?
5. What is the credit assignment problem? When does it occur? How can you alleviate it?

³⁰ OEL Team, “Open-Ended Learning Leads to Generally Capable Agents” (2021).

6. What is the point of using a replay buffer?
7. What is an off-policy RL algorithm?
8. Use policy gradients to solve OpenAI Gym's LunarLander-v2 environment.
9. Use a Double Dueling DQN to train an agent that can achieve a superhuman level at the famous Atari Breakout game ("ALE/Breakout-v5"). The observations are images. To simplify the task, you should convert them to grayscale (i.e., average over the channels axis), crop them and downsample them, so they're just large enough to play, but not more. An individual image does not tell you which way the ball and the paddles are going, so you should merge two or three consecutive images to form each state. Lastly, the DQN should be composed mostly of convolutional layers.
10. If you have about \$100 to spare, you can purchase a Raspberry Pi 3 plus some cheap robotics components, install TensorFlow on the Pi, and go wild! For an example, check out this [fun post](#) by Lukas Biewald, or take a look at GoPiGo or BrickPi. Start with simple goals, like making the robot turn around to find the brightest angle (if it has a light sensor) or the closest object (if it has a sonar sensor), and move in that direction. Then you can start using Deep Learning: for example, if the robot has a camera, you can try to implement an object detection algorithm so it detects people and moves toward them. You can also try to use RL to make the agent learn on its own how to use the motors to achieve that goal. Have fun!

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab3>.

Training and Deploying TensorFlow Models at Scale

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 19th chapter of the final book. The GitHub repo is <https://github.com/ageron/handsonml3>.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mcronin@oreilly.com.

Once you have a beautiful model that makes amazing predictions, what do you do with it? Well, you need to put it in production! This could be as simple as running the model on a batch of data and perhaps writing a script that runs this model every night. However, it is often much more involved. Various parts of your infrastructure may need to use this model on live data, in which case you probably want to wrap your model in a web service: this way, any part of your infrastructure can query your model at any time using a simple REST API (or some other protocol), as we discussed in [Chapter 2](#). But as time passes, you need to regularly retrain your model on fresh data and push the updated version to production. You must handle model versioning, gracefully transition from one model to the next, possibly roll back to the previous model in case of problems, and perhaps run multiple different models in parallel to

perform *A/B experiments*.¹ If your product becomes successful, your service may start to get plenty of *queries per second* (QPS), and it must scale up to support the load. A great solution to scale up your service, as we will see in this chapter, is to use TF Serving, either on your own hardware infrastructure or via a cloud service such as Google Vertex AI². It will take care of efficiently serving your model, handle graceful model transitions, and more. If you use the cloud platform, you will also get many extra features, such as powerful monitoring tools.

Moreover, if you have a lot of training data, and compute-intensive models, then training time may be prohibitively long. If your product needs to adapt to changes quickly, then a long training time can be a showstopper (e.g., think of a news recommendation system promoting news from last week). Perhaps even more importantly, a long training time will prevent you from experimenting with new ideas. In Machine Learning (as in many other fields), it is hard to know in advance which ideas will work, so you should try out as many as possible, as fast as possible. One way to speed up training is to use hardware accelerators such as GPUs or TPUs. To go even faster, you can train a model across multiple machines, each equipped with multiple hardware accelerators. TensorFlow's simple yet powerful Distribution Strategies API makes this easy, as we will see.

In this chapter we will look at how to deploy models, first using TF Serving, then using Vertex AI. We will also take a quick look at deploying models to mobile apps, embedded devices, and web apps. Then we will discuss how to speed up computations using GPUs and how to train models across multiple devices and servers using the Distribution Strategies API. Lastly, we will see how to train models and fine-tune their hyperparameters at scale using Vertex AI. That's a lot of topics to discuss, so let's dive in!

Serving a TensorFlow Model

Once you have trained a TensorFlow model, you can easily use it in any Python code: if it's a Keras model, just call its `predict()` method! But as your infrastructure grows, there comes a point where it is preferable to wrap your model in a small service whose sole role is to make predictions and have the rest of the infrastructure query it (e.g., via a REST or gRPC API).³ This decouples your model from the rest of the

¹ An A/B experiment consists in testing two different versions of your product on different subsets of users in order to check which version works best and get other insights.

² Google AI Platform (formerly known as Google ML Engine) and Google AutoML merged in 2021 to form Google Vertex AI.

³ A REST (or RESTful) API is an API that uses standard HTTP verbs, such as GET, POST, PUT, and DELETE, and uses JSON inputs and outputs. The gRPC protocol is more complex but more efficient. Data is exchanged using protocol buffers (see [Chapter 13](#)).

infrastructure, making it possible to easily switch model versions or scale the service up as needed (independently from the rest of your infrastructure), perform A/B experiments, and ensure that all your software components rely on the same model versions. It also simplifies testing and development, and more. You could create your own microservice using any technology you want (e.g., using the Flask library), but why reinvent the wheel when you can just use TF Serving?

Using TensorFlow Serving

TF Serving is a very efficient, battle-tested model server, written in C++. It can sustain a high load, serve multiple versions of your models and watch a model repository to automatically deploy the latest versions, and more (see [Figure 19-1](#)).

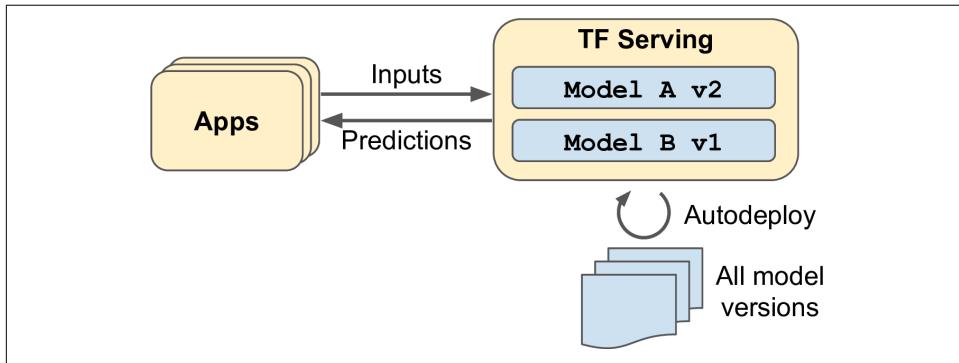


Figure 19-1. TF Serving can serve multiple models and automatically deploy the latest version of each model

So let's suppose you have trained an MNIST model using Keras, and you want to deploy it to TF Serving. The first thing you have to do is export this model to the SavedModel format, introduced in [Chapter 10](#).

Exporting SavedModels

You already know how to save the model: just call `model.save()`. Now to version the model, you just need to create a subdirectory for each model version. Easy!

```
from pathlib import Path
import tensorflow as tf

X_train, X_valid, X_test = [...] # load and split the MNIST dataset
model = [...] # build & train an MNIST model (also handles image preprocessing)

model_name = "my_mnist_model"
model_version = "0001"
model_path = Path(model_name) / model_version
model.save(model_path, save_format="tf")
```

It's usually a good idea to include all the preprocessing layers in the final model you export so that it can ingest data in its natural form once it is deployed to production. This avoids having to take care of preprocessing separately within the application that uses the model. Bundling the preprocessing steps within the model also makes it simpler to update them later on and limits the risk of mismatch between a model and the preprocessing steps it requires.



Since a SavedModel saves the computation graph, it can only be used with models that are based exclusively on TensorFlow operations, excluding the `tf.py_function()` operation, which wraps arbitrary Python code.

TensorFlow comes with a small `saved_model_cli` command-line interface to inspect SavedModels. Let use it to inspect our exported model:

```
$ saved_model_cli show --dir my_mnist_model/0001  
The given SavedModel contains the following tag-sets:  
'serve'
```

What does this output mean? Well, a SavedModel contains one or more *metagraphs*. A metagraph is a computation graph plus some function signature definitions, including their input and output names, types, and shapes. Each metagraph is identified by a set of tags. For example, you may want to have a metagraph containing the full computation graph, including the training operations: you would typically tag this one as "train". And you might have another metagraph containing a pruned computation graph with only the prediction operations, including some GPU-specific operations: this one may be tagged as "serve", "gpu". And you may want to have other metagraphs as well. This can be done using TensorFlow's low-level [SavedModel API](#). However, when you save a Keras model using its `save()` method, it saves a single metagraph tagged as "serve". Let's inspect this "server" tag-set:

```
$ saved_model_cli show --dir 0001/my_mnist_model --tag_set serve  
The given SavedModel MetaGraphDef contains SignatureDefs with these keys:  
SignatureDef key: "__saved_model_init_op"  
SignatureDef key: "serving_default"
```

This metagraph contains two signature definitions: an initialization function called "`__saved_model_init_op`", which you do not need to worry about, and a default serving function called "`serving_default`". When saving a Keras model, the default serving function is the model's `call()` method, which makes predictions, as you already know. Let's get more details about this serving function:

```
$ saved_model_cli show --dir 0001/my_mnist_model --tag_set serve \  
--signature_def serving_default  
The given SavedModel SignatureDef contains the following input(s):  
inputs['flatten_input'] tensor_info:
```

```
  dtype: DT_UINT8
  shape: (-1, 28, 28)
  name: serving_default_flatten_input:0
The given SavedModel SignatureDef contains the following output(s):
  outputs['dense_1'] tensor_info:
    dtype: DT_FLOAT
    shape: (-1, 10)
    name: StatefulPartitionedCall:0
Method name is: tensorflow/serving/predict
```

Note that the function's input is named "flatten_input", and the output is named "dense_1". These correspond to the Keras model's input and output layer names. You can also see the type and shape of the input and output data. Looks good!

Now that you have a SavedModel, the next step is to install TF Serving.

Installing and Starting TensorFlow Serving

There are many ways to install TF Serving: using the system's package manager, using a Docker image,⁴ installing from source, and more. Since Colab runs on Ubuntu, we can use Ubuntu's apt package manager like this:

```
url = "https://storage.googleapis.com/tensorflow-serving-apt"
src = "stable tensorflow-model-server tensorflow-model-server-universal"
!echo 'deb {url} {src}' > /etc/apt/sources.list.d/tensorflow-serving.list
!curl '{url}/tensorflow-serving.release.pub.gpg' | apt-key add -
!apt update -q && apt-get install -y tensorflow-model-server
%pip install -q -U tensorflow-serving-api
```

This code starts by adding TensorFlow's package repository to Ubuntu's list of package sources. Then it downloads TensorFlow's public GPG key and adds it to the package manager's key list so it can verify TensorFlow's package signatures. Next it uses apt to install the tensorflow-model-server package. Lastly, it installs the tensorflow-serving-api library, which we will need to communicate with the server.

Next, we want to start the server. The command will require the absolute path of the base model directory (i.e., the path to `my_mnist_model`, not `0001`), so let's save that to the MODEL_DIR environment variable:

⁴ If you are not familiar with Docker, it allows you to easily download a set of applications packaged in a *Docker image* (including all their dependencies and usually some good default configuration) and then run them on your system using a *Docker engine*. When you run an image, the engine creates a *Docker container* that keeps the applications well isolated from your own system—but you can give it some limited access if you want. It is similar to a virtual machine, but much faster and lighter, as the container relies directly on the host's kernel. This means that the image does not need to include or run its own kernel.

```
import os  
  
os.environ["MODEL_DIR"] = str(model_path.parent.absolute())
```

Now let's start the server:

```
%%bash --bg  
tensorflow_model_server \  
  --port=8500 \  
  --rest_api_port=8501 \  
  --model_name=my_mnist_model \  
  --model_base_path="${MODEL_DIR}" >my_server.log 2>&1
```

In Jupyter or Colab, the `%%bash --bg` magic command executes the cell as a bash script, running it in the background. The `>my_server.log 2>&1` part redirects the standard output and standard error to the `my_server.log` file. And that's it! TF Serving is now running in the background, and its logs are saved to `my_server.log`. It loaded our MNIST model (version 1), and it is now waiting for gRPC and REST requests, respectively on ports 8500 and 8501.

Running TF Serving in a Docker Container

If you are running the notebook on your own machine, and you have installed [Docker](#), you can run `docker pull tensorflow/serving` in a terminal to download the TensorFlow Serving image. The TensorFlow team highly recommends this installation method because it is simple, it will not mess with your system, and it offers high performance.⁵ To start the server inside a Docker container, you can run the following command in a terminal:

```
$ docker run -it --rm -v "/path/to/my_mnist_model:/models/my_mnist_model" \  
  -p 8500:8500 -p 8501:8501 -e MODEL_NAME=my_mnist_model tensorflow/serving
```

Here is what all these command-line options mean:

`-it`

Makes the container interactive (so you can press Ctrl-C to stop it) and displays the server's output.

`--rm`

Deletes the container when you stop it: no need to clutter your machine with interrupted containers. However, it does not delete the image.

`-p 8500:8500`

Makes the Docker engine forward the host's TCP port 8500 to the container's TCP port 8500. By default, TF Serving uses this port to serve the gRPC API.

⁵ There are also GPU images available, and other installation options. For more details, please check out the official [installation instructions](#).

```
-p 8501:8501
Forwards the host's TCP port 8501 to the container's TCP port 8501. The Docker image is configured to use this port by default to serve the REST API.

-v "/path/to/my_mnist_model:/models/my_mnist_model"
Makes the host's my_mnist_model directory available to the container at the path /models/mnist_model. You must replace /path/to/my_mnist_model with the absolute path of this directory. On Windows, remember to use \ instead of / in the host path, but not in the container path (since the container runs on Linux).

-e MODEL_NAME=my_mnist_model
Sets the container's MODEL_NAME environment variable, so TF Serving knows which model to serve. By default, it will look for models in the /models directory, and it will automatically serve the latest version it finds.

tensorflow/serving
This is the name of the image to run.
```

Now that the server is up and running, let's query it, first using the REST API, then the gRPC API.

Querying TF Serving through the REST API

Let's start by creating the query. It must contain the name of the function signature you want to call, and of course the input data. Since the request must use the JSON format, we have to convert the input images from a NumPy array to a Python list:

```
import json

X_new = X_test[:3] # pretend we have 3 new digit images to classify
request_json = json.dumps({
    "signature_name": "serving_default",
    "instances": X_new.tolist(),
})
```

Note that the JSON format is 100% text-based. The request string looks like this:

```
>>> request_json
'{"signature_name": "serving_default", "instances": [[[0, 0, 0, 0, ... ]]]}'
```

Now let's send this request to TF Serving via an HTTP POST request. This can be done using the `requests` library (it is not part of Python's standard library, but it is preinstalled on Colab):

```
import requests

server_url = "http://localhost:8501/v1/models/my_mnist_model:predict"
response = requests.post(server_url, data=request_json)
```

```
response.raise_for_status() # raise an exception in case of error
response = response.json()
```

If all goes well, the response should be a dictionary containing a single "predictions" key. The corresponding value is the list of predictions. This list is a Python list, so let's convert it to a NumPy array and round the floats it contains to the second decimal:

```
>>> import numpy as np
>>> y_proba = np.array(response["predictions"])
>>> y_proba.round(2)
array([[0. , 0. , 0. , 0. , 0. , 0. , 0. , 1. , 0. , 0. ],
       [0. , 0. , 0.99, 0.01, 0. , 0. , 0. , 0. , 0. , 0. ],
       [0. , 0.97, 0.01, 0. , 0. , 0. , 0. , 0.01, 0. , 0. ]])
```

Hurray, we have the predictions! The model is close to 100% confident that the first image is a 7, 99% confident that the second image is a 2, and 97% confident that the third image is a 1. That's correct.

The REST API is nice and simple, and it works well when the input and output data are not too large. Moreover, just about any client application can make REST queries without additional dependencies, whereas other protocols are not always so readily available. However, it is based on JSON, which is text-based and fairly verbose. For example, we had to convert the NumPy array to a Python list, and every float ended up represented as a string. This is very inefficient, both in terms of serialization/deserialization time—to convert all the floats to strings and back—and in terms of payload size: many floats end up being represented using over 15 characters, which translates to over 120 bits for 32-bit floats! This will result in high latency and bandwidth usage when transferring large NumPy arrays.⁶ So let's see how to use gRPC instead.



When transferring large amounts of data, or when latency is important, it is much better to use the gRPC API, if the client supports it, as it uses a compact binary format and an efficient communication protocol based on HTTP/2 framing.

Querying TF Serving through the gRPC API

The gRPC API expects a serialized `PredictRequest` protocol buffer as input, and it outputs a serialized `PredictResponse` protocol buffer. These protos are part of the `tensorflow-serving-api` library, which we installed earlier. First, let's create the request:

⁶ To be fair, this can be mitigated by serializing the data first and encoding it to Base64 before creating the REST request. Moreover, REST requests can be compressed using gzip, which reduces the payload size significantly.

```

from tensorflow_serving.apis.predict_pb2 import PredictRequest

request = PredictRequest()
request.model_spec.name = model_name
request.model_spec.signature_name = "serving_default"
input_name = model.input_names[0] # == "flatten_input"
request.inputs[input_name].CopyFrom(tf.make_tensor_proto(X_new))

```

This code creates a `PredictRequest` protocol buffer and fills in the required fields, including the model name (defined earlier), the signature name of the function we want to call, and finally the input data, in the form of a `Tensor` protocol buffer. The `tf.make_tensor_proto()` function creates a `Tensor` protocol buffer based on the given tensor or NumPy array, in this case `X_new`.

Next, we'll send the request to the server and get its response. For this, we will need the `grpcio` library, which is preinstalled in Colab:

```

import grpc
from tensorflow_serving.apis import prediction_service_pb2_grpc

channel = grpc.insecure_channel('localhost:8500')
predict_service = prediction_service_pb2_grpc.PredictionServiceStub(channel)
response = predict_service.Predict(request, timeout=10.0)

```

The code is quite straightforward: after the imports, we create a gRPC communication channel to `localhost` on TCP port 8500, then we create a gRPC service over this channel and use it to send a request, with a 10-second timeout. Note that the call is synchronous: it will block until it receives the response or when the timeout period expires. In this example the channel is insecure (no encryption, no authentication), but gRPC and TensorFlow Serving also support secure channels over SSL/TLS.

Next, let's convert the `PredictResponse` protocol buffer to a tensor:

```

output_name = model.output_names[0] # == "dense_1"
outputs_proto = response.outputs[output_name]
y_proba = tf.make_ndarray(outputs_proto)

```

If you run this code and print `y_proba.round(2)`, you will get the exact same estimated class probabilities as earlier. And that's all there is to it: in just a few lines of code, you can now access your TensorFlow model remotely, using either REST or gRPC.

Deploying a new model version

Now let's create a new model version and export a `SavedModel`, this time to the `my_mnist_model/0002` directory:

```

model = [...] # build and train a new MNIST model version

model_version = "0002"
model_path = Path(model_name) / model_version
model.save(model_path, save_format="tf")

```

At regular intervals (the delay is configurable), TensorFlow Serving checks the model directory for new model versions. If it finds one, it automatically handles the transition gracefully: by default, it answers pending requests (if any) with the previous model version, while handling new requests with the new version. As soon as every pending request has been answered, the previous model version is unloaded. You can see this at work in the TensorFlow Serving logs (in `my_server.log`):

```
[...]
Reading SavedModel from: /models/my_mnist_model/0002
Reading meta graph with tags { serve }
[...]
Successfully loaded servable version {name: my_mnist_model version: 2}
Quiescing servable version {name: my_mnist_model version: 1}
Done quiescing servable version {name: my_mnist_model version: 1}
Unloading servable version {name: my_mnist_model version: 1}
```



If the SavedModel contains some example instances in the `assets/extras` directory, you can configure TF Serving to run the new model on these instances before starting to use it to serve requests. This is called *model warmup*: it will ensure that everything is properly loaded, avoiding long response times for the first requests.

This approach offers a smooth transition, but it may use too much RAM—especially GPU RAM, which is generally the most limited. In this case, you can configure TF Serving so that it handles all pending requests with the previous model version and unloads it before loading and using the new model version. This configuration will avoid having two model versions loaded at the same time, but the service will be unavailable for a short period.

As you can see, TF Serving makes it straightforward to deploy new models. Moreover, if you discover that version 2 does not work as well as you expected, then rolling back to version 1 is as simple as removing the `my_mnist_model/0002` directory.



Another great feature of TF Serving is its automatic batching capability, which you can activate using the `--enable_batching` option upon startup. When TF Serving receives multiple requests within a short period of time (the delay is configurable), it will automatically batch them together before using the model. This offers a significant performance boost by leveraging the power of the GPU. Once the model returns the predictions, TF Serving dispatches each prediction to the right client. You can trade a bit of latency for a greater throughput by increasing the batching delay (see the `--batching_parameters_file` option).

If you expect to get many queries per second, you will want to deploy TF Serving on multiple servers and load-balance the queries (see [Figure 19-2](#)). This will require deploying and managing many TF Serving containers across these servers. One way to handle that is to use a tool such as [Kubernetes](#), which is an open source system for simplifying container orchestration across many servers. If you do not want to purchase, maintain, and upgrade all the hardware infrastructure, you will want to use virtual machines on a cloud platform such as Amazon AWS, Microsoft Azure, Google Cloud Platform, IBM Cloud, Alibaba Cloud, Oracle Cloud, or some other Platform-as-a-Service (PaaS). Managing all the virtual machines, handling container orchestration (even with the help of Kubernetes), taking care of TF Serving configuration, tuning and monitoring—all of this can be a full-time job. Fortunately, some service providers can take care of all this for you. In this chapter we will use Vertex AI because it's the only platform with TPUs today, it supports TensorFlow 2, Scikit-Learn, and XGBoost, and it offers a nice suite of AI services. But there are several other providers in this space, such as Amazon AWS SageMaker and Microsoft AI Platform, which are capable of serving TensorFlow models as well, so make sure to check them out too.

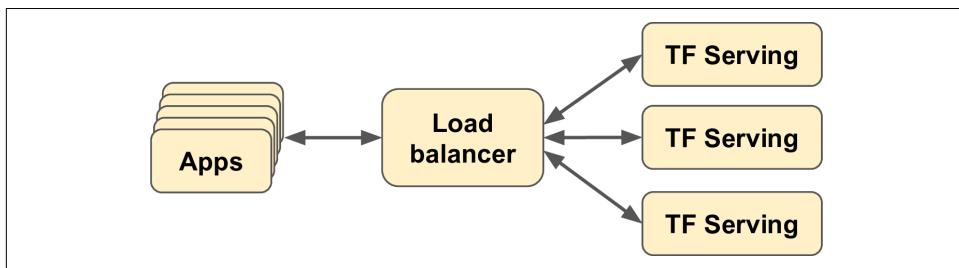


Figure 19-2. Scaling up TF Serving with load balancing

Now let's see how to serve our wonderful MNIST model on the cloud!

Creating a Prediction Service on Vertex AI

Vertex AI is a platform within Google Cloud Platform (GCP) that offers a wide range of AI-related tools and services: you can upload your datasets, get humans to label them, store commonly used features in a feature store and use them for training or in production, train models across many GPU or TPU servers, with automatic hyperparameter tuning or model architecture search (AutoML), manage your trained models, use them to make *batch predictions* on large amounts of data, schedule multiple jobs for your data workflows, serve your models via REST or gRPC at scale, and experiment with your data and models within a hosted Jupyter environment called the *Workbench*. There's also a *Matching Engine* service that lets you compare vectors very efficiently (i.e., approximate nearest neighbors). GCP also includes other AI services, such as APIs for computer vision, translation, speech-to-text, and more.

Before we start, there's a little bit of setup to take care of:

1. Log in to your Google account, and then go to the [Google Cloud Platform \(GCP\) console](#) (see [Figure 19-3](#)). If you don't have a Google account, you'll have to create one.
2. If it is your first time using GCP, you will have to read and accept the terms and conditions. New users are offered a free trial, including \$300 worth of GCP credit that you can use over the course of 90 days (as of May 2022). You will only need a small portion of that to pay for the services you will use in this chapter. Upon signing up for the free trial, you will still need to create a payment profile and enter your credit card number: it is used for verification purposes—probably to avoid people using the free trial multiple times—but you will not be billed for the first \$300, and after that you will only be charged if you opt-in by upgrading to a paid account.

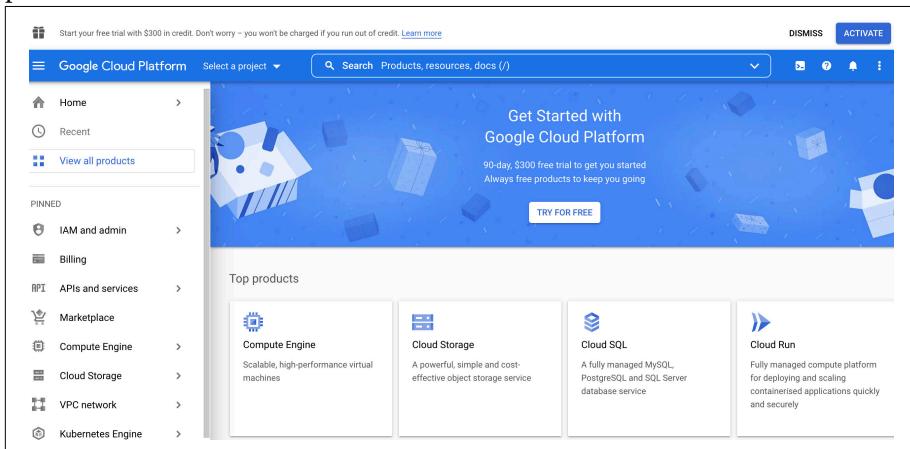


Figure 19-3. Google Cloud Platform console

3. If you have used GCP before and your free trial has expired, then the services you will use in this chapter will cost you some money. It should not be too much, especially if you remember to turn off the services when you do not need them anymore. Make sure you understand and agree to the pricing conditions before you run any service. I hereby decline any responsibility if services end up costing more than you expected! Also make sure your billing account is active. To check, open the \equiv navigation menu on the top left and click Billing, then make sure you have set up a payment method and that the billing account is active.
4. Every resource in GCP belongs to a project. This includes all the virtual machines you may use, the files you store, and the training jobs you run. When you create an account, GCP automatically creates a project for you, called "My First Project". If you want, you can change its display name by going to the

project settings: in the \equiv navigation menu, select *IAM & admin* \rightarrow *Settings*, change the project's display name, and click *SAVE*. Note that the project also has a unique ID and number. You can choose the project ID when you create a project, but you cannot change it later. The project number is automatically generated and cannot be changed. If you want to create a new project, click the project name at the top of the page, then click *NEW PROJECT* and enter the project name. You can also click *EDIT* to set the project ID. Make sure billing is active for this new project so that service fees can be billed (to your free credits if any).



Always set an alarm to remind yourself to turn services off when you know you will only need them for a few hours, or else you might leave them running for days or months, incurring potentially significant costs.

5. Now that you have a GCP account, a project, and billing is activated, you must activate the APIs you need. In the \equiv navigation menu, open *APIs and services*, and make sure the Cloud Storage API is enabled. If needed, click $+ \text{ENABLE APIS AND SERVICES}$, find Cloud Storage, and enable it. Also enable the Vertex AI API.

We could continue to do everything via the GCP console, but I recommend using Python instead: this way you can write scripts to automate just about anything you want with GCP, it's often more convenient than clicking your way through menus and forms, especially for common tasks.

Google Cloud CLI and Shell

Google Cloud's Command Line Interface (CLI) includes the `gcloud` command, which lets you control almost everything in GCP, and `gsutil`, which lets you interact with Google Cloud Storage. This CLI is preinstalled in Colab: all you need to do is authenticate using `google.auth.authenticate_user()`, and you're good to go. For example, `!gcloud config list` will display the configuration.

GCP also offers a preconfigured shell environment called the Google Cloud Shell, which you can use directly in your web browser; it runs on a free Linux VM (Debian), with the Google Cloud SDK already preinstalled and configured for you, no need to authenticate. The Cloud Shell is available anywhere in GCP: just click the *Activate Cloud Shell* icon at the top right of the page (see [Figure 19-4](#)).

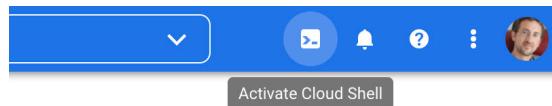


Figure 19-4. Activating the Google Cloud Shell

If you prefer to [install the CLI on your machine](#), then after installation you need to initialize it by running `gcloud init`: follow the instructions to log in to GCP and grant access to your GCP resources, select the default GCP project you want to use (if you have more than one), and the default region where you want your jobs to run.

The first thing you need to do before you can use any GCP service is to authenticate. The simplest solution when using Colab is to execute the following code:

```
from google.colab import auth  
  
auth.authenticate_user()
```

The authentication process is based on [OAuth 2.0](#): a popup window will ask you to confirm that you want the Colab notebook to access your Google credentials. If you accept, you must then select the same Google account you used for GCP. Then you will be asked to confirm that you agree to give Colab full access to all your data on Google Drive and in GCP. If you allow access, only the current notebook will have access, and only until the Colab Runtime expires. Obviously, you should only accept this if you trust the code in the notebook.



If you are *not* working with the official notebooks from <https://github.com/ageron/handson-ml3>, then you should be extra careful: if the notebook's author is mischievous, they could include code to do whatever they want with your data.

Authentication and authorization on GCP

In general, using OAuth 2.0 authentication is only recommended when an application must access the user's personal data or resources from another application, on the user's behalf. For example, some applications allow the user to save data to their Google Drive, but for that the application first needs the user to authenticate with Google and allow access to Google Drive. In general, the application will only ask for the level of access it needs, it won't be an unlimited access: for example, the application will only request access to Google Drive, not GMail or any other Google service. Moreover, the authorization usually expires after a while, and it can always be revoked.

When an application needs to access a service on GCP on its own behalf, not on behalf of the user, then it should generally use a *service account*. For example, if you build a website that needs to send prediction requests to a Vertex AI endpoint, then the website will be accessing the service on its own behalf. There's no data or resource that it needs to access on the user's Google account. In fact, many users of the website will not even have a Google account. For this scenario, you first need to create a *service account*. Select *IAM & admin* → *Service accounts* in the GCP console's \equiv navigation menu (or use the search box), then click + *CREATE SERVICE ACCOUNT*, fill in the first page of the form (service account name, ID, description), and click *CREATE AND CONTINUE*. Next, you must give this account some access rights. Select the *Vertex AI user* role: this will allow the service account to make predictions, and use other Vertex AI services, but nothing else. Click *CONTINUE*. You could now optionally grant some users access to the service account: this is useful when your GCP user account is part of an organization, and you wish to authorize other users in the organization to deploy applications that will be based on this service account, or to manage the service account itself. Next, click *DONE*.

Once you have created a service account, your application must authenticate as that service account. There are several ways to do that. If your application is hosted on GCP, for example if you are coding a website hosted on Google Compute Engine, then the simplest and safest solution is to attach the service account to the GCP resource that hosts your website, such as a VM instance or a Google App Engine service. This can be done when creating the GCP resource, by selecting the service account in the *Identity and API access* section. Some resources, such as VM instances, also let you attach the service account after the VM instance is created: you must stop it and edit its settings. In any case, once a service account is attached to a VM instance, or any other GCP resource running your code, GCP's client libraries (discussed shortly) will automatically authenticate as the chosen service account, with no extra step needed.

If your application is hosted using Kubernetes, then you should use Google's Workload Identity service to map the right service account to each Kubernetes service account. And if your application is not hosted on GCP, for example if you are just running the Jupyter notebook on your own machine, then you can either use the Workload Identity Federation service (that's the safest but hardest option), or just generate an access key for your service account, save it to a JSON file, and point the `GOOGLE_APPLICATION_CREDENTIALS` environment variable to it so your client application can access it. You can manage access keys by clicking on the service account you just created, and then opening the *KEYS* tab. Make sure to keep the key file secret: it's like a password for the service account.

For more details on setting up authentication and authorization so your application can access GCP services, please check out the [documentation](#).

Now let's create a Google Cloud Storage bucket to store our SavedModels (a GCS *bucket* is a container for your data). For this we will use the `google-cloud-storage` library, which is preinstalled in Colab. We first create a `Client` object, which will serve as the interface with GCS, then we use it to create the bucket:

```
from google.cloud import storage

project_id = "my_project" # change this to your project ID
bucket_name = "my_bucket" # change this to a unique bucket name
location = "us-central1"

storage_client = storage.Client(project=project_id)
bucket = storage_client.create_bucket(bucket_name, location=location)
```



If you want to reuse an existing bucket, replace the last line with `bucket = storage_client.bucket(bucket_name)`. Make sure `location` is set to the bucket's region.

GCS uses a single worldwide namespace for buckets, so simple names like "machine-learning" will most likely not be available. Make sure the bucket name conforms to DNS naming conventions, as it may be used in DNS records. Moreover, bucket names are public, so do not put anything private in the name. It is common to use your domain name, your company name, or your project ID as a prefix to ensure uniqueness, or simply use a random number as part of the name.

You can change the region if you want, but be sure to choose one that supports GPUs. Also, you may want to consider the fact that prices vary greatly between regions, some regions produce much more CO₂ than others, some regions do not support all services, and using a single-region bucket improves performance. See [Google Cloud's list of regions](#) and [Vertex AI's documentation on locations](#) for more details. If you are unsure, it might be best to stick with "us-central1".

Next, let's upload the `my_mnist_model` directory to the new bucket. Files in GCS are called *blobs* (or *objects*), and under the hood they are all just placed in the bucket without any directory structure. Blob names can be arbitrary unicode strings, and they can even contain forward slashes /. The GCP console and other tools use these slashes to give the illusion that there are directories. So when we upload the `my_mnist_model` directory, we only care about the files, not the directories:

```
def upload_directory(bucket, dirpath):
    dirpath = Path(dirpath)
    for filepath in dirpath.glob("**/*"):
        if filepath.is_file():
            blob = bucket.blob(filepath.relative_to(dirpath.parent).as_posix())
            blob.upload_from_filename(filepath)
```

```
upload_directory(bucket, "my_mnist_model")
```

This function works fine now, but it would be very slow if there were many files to upload. It's not too hard to speed it up tremendously by multithreading it (see the notebook for an implementation). Alternatively, if you have the Google Cloud CLI, then you can use following command instead:

```
!gsutil -m cp -r my_mnist_model gs://[bucket_name]/
```

Next, let's tell Vertex AI about our MNIST model. To communicate with Vertex AI, we can use the `google-cloud-aiplatform` library (it still uses the old AI Platform name instead of Vertex AI). It's not preinstalled in Colab, so we need to install it. After that, we can import the library and initialize it—just to specify some default values for the project ID and the location—then we can create a new Vertex AI Model: we specify a display name, the GCS path to our model (in this case the version 0001), and we also specify the URL of the Docker container we want Vertex AI to use to run this model. If you visit that URL and navigate up one level, you will find other containers you can use. This one supports TensorFlow 2.8 with a GPU.

```
from google.cloud import aiplatform

server_image = "gcr.io/cloud-aiplatform/prediction/tf2-gpu.2-8:latest"

aiplatform.init(project=project_id, location=location)
mnist_model = aiplatform.Model.upload(
    display_name="mnist",
    artifact_uri=f"gs://[bucket_name]/my_mnist_model/0001",
    serving_container_image_uri=server_image,
)
```

Now let's deploy this model so we can query it via a gRPC or REST API to make predictions. For this we first need to create an *endpoint*. This is what client applications connect to when they want to access a service. Then we need to deploy our model to this endpoint:

```
endpoint = aiplatform.Endpoint.create(display_name="mnist-endpoint")

endpoint.deploy(
    mnist_model,
    min_replica_count=1,
    max_replica_count=5,
    machine_type="n1-standard-4",
    accelerator_type="NVIDIA_TESLA_K80",
    accelerator_count=1
)
```

This code may take a few minutes to run, because Vertex AI needs to setup a virtual machine. In this example, we use a fairly basic machine of type `n1-standard-4` (see <https://cloud.google.com/compute/docs/machine-types> for other types). We also use a basic GPU of type

NVIDIA_TESLA_K80 (see <https://homl.info/accelerators> for other types). If you selected another region than "us-central1", then you may need to change the machine type or the accelerator type to values that are supported in that region (e.g., not all regions have Nvidia Tesla K80 GPUs).



Google Cloud Platform enforces various GPU quotas, both worldwide and per region: you cannot create thousands of GPU nodes without prior authorization from Google. To check your quotas, open *IAM & admin* → *Quotas* in the GCP console. If some quotas are too low (e.g., if you need more GPUs in a particular region), you can ask for them to be increased, it often takes about 48 hours.

Vertex AI will initially spawn the minimum number of compute nodes (just one in this case), and whenever the number of queries per second (QPS) becomes too high, it will spawn more nodes, up to the maximum number you defined (five in this case), and it will load-balance the queries between them. If the QPS goes down for a while, Vertex AI will stop the extra compute nodes automatically. The cost is therefore directly linked to the load, as well as the type of machine and accelerator(s) you selected and the amount of data you store on GCS. This pricing model is great for occasional users and for services with important usage spikes. It's also ideal for startups: the price remains low until the startup actually starts up.

Congratulations, you have deployed your first model to the cloud! Now let's query this prediction service.

```
response = endpoint.predict(instances=X_new.tolist())
```

The images we want to classify must first be converted to a Python list, as we did earlier when we sent requests to TF Serving using the REST API. The response object contains the predictions, represented as a Python list of lists of floats. Let's round them to two decimal places, and convert them to a NumPy array:

```
>>> import numpy as np
>>> np.round(response.predictions, 2)
array([[0. , 0. , 0. , 0. , 0. , 0. , 0. , 1. , 0. , 0. ],
       [0. , 0. , 0.99, 0.01, 0. , 0. , 0. , 0. , 0. , 0. ],
       [0. , 0.97, 0.01, 0. , 0. , 0. , 0. , 0.01, 0. , 0. ]])
```

Yes! We get the exact same predictions as earlier. We now have a nice prediction service running on the cloud that we can query from anywhere securely, and which can automatically scale up or down, depending on the number of QPS. When you are done using the endpoint, don't forget to delete it, to avoid paying for nothing:

```
endpoint.undeploy_all() # undeploy all models from the endpoint
endpoint.delete()
```

Now let's see how to run a job on Vertex AI to make predictions on a potentially very large batch of data.

Running Batch Prediction Jobs on Vertex AI

If we have a large number of predictions to make, then instead of calling our prediction service repeatedly, we can ask Vertex AI to run a prediction job for us. This does not require an endpoint, only a model. For example, let's run a prediction job on the first 100 images of the test set, using our MNIST model. For this, we first need to prepare the batch and upload it to GCS. One way to do this is to create a file containing one instance per line, each formatted as a JSON value—this format is called *JSON Lines*—then pass this file to Vertex AI. So let's create a JSON Lines file in a new directory, then upload this directory to GCS:

```
batch_path = Path("my_mnist_batch")
batch_path.mkdir(exist_ok=True)
with open(batch_path / "my_mnist_batch.jsonl", "w") as jsonl_file:
    for image in X_test[:100].tolist():
        jsonl_file.write(json.dumps(image))
        jsonl_file.write("\n")

upload_directory(bucket, batch_path)
```

Next we're ready to launch the prediction job, specifying the job's name, the type and number of machines and accelerators to use, the GCS path to the JSON Lines file we just created, and the path to the GCS directory where Vertex AI will save the model's predictions:

```
batch_prediction_job = mnist_model.batch_predict(
    job_display_name="my_batch_prediction_job",
    machine_type="n1-standard-4",
    starting_replica_count=1,
    max_replica_count=5,
    accelerator_type="NVIDIA_TESLA_K80",
    accelerator_count=1,
    gcs_source=[f"gs://{bucket_name}/{batch_path.name}/my_mnist_batch.jsonl"],
    gcs_destination_prefix=f"gs://{bucket_name}/my_mnist_predictions/",
    sync=True # set to False if you don't want to wait for completion
)
```



For large batches, you can split the inputs into multiple JSON Lines files, and list them all via the `gcs_source` argument.

This will take a few minutes, mostly to spawn the compute nodes on Vertex AI. Once this command completes, the predictions are available in a set of files named some-

thing like `prediction.results-00001-of-00002`. These files use the JSON Lines format by default, and each value is a dictionary containing an instance and its corresponding prediction (i.e., 10 probabilities). The instances are listed in the same order as the inputs. The job also outputs `prediction-errors*` files, which can be useful for debugging if something goes wrong. You can iterate through all these output files using `batch_prediction_job.iter_outputs()`. So let's go through all the predictions and store them in a `y_probas` array:

```
y_probas = []
for blob in batch_prediction_job.iter_outputs():
    if "prediction.results" in blob.name:
        for line in blob.download_as_text().splitlines():
            y_proba = json.loads(line)["prediction"]
            y_probas.append(y_proba)
```

Let's see how good these predictions are:

```
>>> y_pred = np.argmax(y_probas, axis=1)
>>> accuracy = np.sum(y_pred == y_test[:100]) / 100
0.98
```

Nice, 98% accuracy!

The JSON Lines format is the default, but when dealing with large instances such as images, it is too verbose. Luckily, the `batch_predict()` method accepts an `instances_format` argument which lets you choose another format if you want. It defaults to "jsonl", but you can change it to "csv", "tf-record", "tf-record-gzip", "bigquery", or "file-list". For example, if you set it to "file-list", then the `gcs_source` argument should point to a text file containing one input file path per line, for example pointing to PNG image files. Vertex AI will read these files as binary, encode them using Base64, and pass the resulting byte strings to the model. This means that you must add a preprocessing layer in your model to parse the Base64 string, using `tf.io.decode_base64()`. If the files are images, you must then parse the result using a function like `tf.io.decode_image()` or `tf.io.decode_png()`, as discussed in [Chapter 13](#).

When you're finished using the model, you can delete it if you want, by running `mnist_model.delete()`. You can also delete the directories you created in your GCS bucket, optionally the bucket itself (if it's empty) and the batch prediction job:

```
for prefix in ["my_mnist_model/", "my_mnist_batch/", "my_mnist_predictions/"]:
    blobs = bucket.list_blobs(prefix=prefix)
    for blob in blobs:
        blob.delete()

bucket.delete() # if the bucket is empty
batch_prediction_job.delete()
```

OK, you now know how to deploy a model to Vertex AI, create a prediction service, and run batch prediction jobs. But what if you want to deploy your model to a mobile app instead? Or to an embedded device, such as a heating control system, a fitness tracker, or a self-driving car?

Deploying a Model to a Mobile or Embedded Device

Machine Learning models are not limited to running on big centralized servers with multiple GPUs: they can run closer to the source of data (this is called *edge computing*), for example in the user's mobile device, or in an embedded device. There are many benefits to decentralizing the computations and moving them toward the edge: first, it allows the device to be smart even when it's not connected to the Internet. It reduces latency by not having to send data to a remote server. It also reduces the load on the servers. And it may improve privacy, since the user's data can stay on the device.

However, deploying models to the edge has its downsides too. The device's computing resources are generally tiny compared to a beefy multi-GPU server. A large model may not fit in the device, it may use too much RAM and CPU, and it may take too long to download. As a result, the application may become unresponsive, and the device may heat up and quickly run out of battery. To avoid all this, you need to make a lightweight and efficient model, without sacrificing too much of its accuracy. The [TFLite](#) library provides several tools⁷ to help you deploy your models to the edge, with three main objectives:

- Reduce the model size, to shorten download time and reduce RAM usage.
- Reduce the amount of computations needed for each prediction, to reduce latency, battery usage, and heating.
- Adapt the model to device-specific constraints.

To reduce the model size, TFLite's model converter can take a SavedModel and compress it to a much lighter format based on [FlatBuffers](#). This is an efficient cross-platform serialization library (a bit like protocol buffers) initially created by Google for gaming. It is designed so you can load FlatBuffers straight to RAM without any preprocessing: this reduces the loading time and memory footprint. Once the model is loaded into a mobile or embedded device, the TFLite interpreter will execute it to make predictions. Here is how you can convert a SavedModel to a FlatBuffer and save it to a `.tflite` file:

```
converter = tf.lite.TFLiteConverter.from_saved_model(str(model_path))
tflite_model = converter.convert()
```

⁷ Also check out TensorFlow's [Graph Transform Tool](#) for modifying and optimizing computational graphs.

```
with open("myConvertedSavedModel.tflite", "wb") as f:  
    f.write(tflite_model)
```



You can also save a Keras model directly to a FlatBuffer using `tf.lite.TFLiteConverter.from_keras_model(model)`.

The converter also optimizes the model, both to shrink it and to reduce its latency. It prunes all the operations that are not needed to make predictions (such as training operations), and it optimizes computations whenever possible; for example, $3 \times a + 4 \times a + 5 \times a$ will be converted to $12 \times a$. It also tries to fuse operations whenever possible. For example, Batch Normalization layers end up folded into the previous layer's addition and multiplication operations, whenever possible. To get a good idea of how much TFLite can optimize a model, download one of the [pretrained TFLite models](#), such as Inception_V1_quant (click on `tflite.pb`), unzip the archive, then open the excellent [Netron graph visualization tool](#) and upload the `.pb` file to view the original model. It's a big, complex graph, right? Next, open the optimized `.tflite` model and marvel at its beauty!

Another way you can reduce the model size—other than simply using smaller neural network architectures—is by using smaller bit-widths: for example, if you use half-floats (16 bits) rather than regular floats (32 bits), the model size will shrink by a factor of 2, at the cost of a (generally small) accuracy drop. Moreover, training will be faster, and you will use roughly half the amount of GPU RAM.

TFLite's converter can go further than that, by quantizing the model weights down to fixed-point, 8-bit integers! This leads to a fourfold size reduction compared to using 32-bit floats. The simplest approach is called *post-training quantization*: it just quantizes the weights after training, using a fairly basic but efficient symmetrical quantization technique. It finds the maximum absolute weight value, m , then it maps the floating-point range $-m$ to $+m$ to the fixed-point (integer) range -127 to +127. For example (see [Figure 19-5](#)), if the weights range from -1.5 to +0.8, then the bytes -127, 0, and +127 will correspond to the floats -1.5, 0.0, and +1.5, respectively. Note that 0.0 always maps to 0 when using symmetrical quantization. Also note that the byte values +68 to +127 will not be used in this example, since they map to floats greater than +0.8.

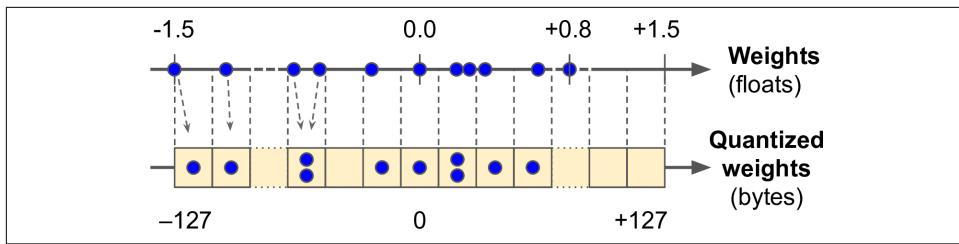


Figure 19-5. From 32-bit floats to 8-bit integers, using symmetrical quantization

To perform this post-training quantization, simply add `DEFAULT` to the list of converter optimizations before calling the `convert()` method:

```
converter.optimizations = [tf.lite.Optimize.DEFAULT]
```

This technique dramatically reduces the model's size, which makes it much faster to download, and uses less storage space. At runtime the quantized weights get converted back to floats before they are used. These recovered floats are not perfectly identical to the original floats, but not too far off, so the accuracy loss is usually acceptable. To avoid recomputing the float values all the time, which would severely slow down the model, TFLite caches them: unfortunately, this means that this technique does not reduce RAM usage, and it doesn't speed up the model either. It's mostly useful to reduce the application's size.

The most effective way to reduce latency and power consumption is to also quantize the activations so that the computations can be done entirely with integers, without the need for any floating-point operations. Even when using the same bit-width (e.g., 32-bit integers instead of 32-bit floats), integer computations use less CPU cycles, consume less energy, and produce less heat. And if you also reduce the bit-width (e.g., down to 8-bit integers), you can get huge speedups. Moreover, some neural network accelerator devices—such as Google's Edge TPU—can only process integers, so full quantization of both weights and activations is compulsory. This can be done post-training; it requires a calibration step to find the maximum absolute value of the activations, so you need to provide a representative sample of training data to TFLite (it does not need to be huge), and it will process the data through the model and measure the activation statistics required for quantization. This step is typically fast.

The main problem with quantization is that it loses a bit of accuracy: it is similar to adding noise to the weights and activations. If the accuracy drop is too severe, then you may need to use *quantization-aware training*. This means adding fake quantization operations to the model so it can learn to ignore the quantization noise during training; the final weights will then be more robust to quantization. Moreover, the calibration step can be taken care of automatically during training, which simplifies the whole process.

I have explained the core concepts of TFLite, but going all the way to coding a mobile or embedded application requires a dedicated book. Fortunately, some exist: if you want to learn more about building TensorFlow applications for mobile and embedded devices, check out the O'Reilly book *TinyML: Machine Learning with TensorFlow on Arduino and Ultra-Low Power Micro-Controllers* (O'Reilly, 2019), by Pete Warden (former lead of the TFLite team) and Daniel Situnayake. You can also read *AI and Machine Learning for On-Device Development* (O'Reilly, 2021), by Laurence Moroney.

Now what if you want to use your model in a website, running directly in the user's browser?

Running a Model in a Web Page

Running your Machine Learning model on the client side—in the user's browser rather than on the server side—can be useful in many scenarios, such as:

- When your web application is often used in situations where the user's connectivity is intermittent or slow (e.g., a website for hikers), so running the model directly on the client side is the only way to make your website reliable.
- When you need the model's responses to be as fast as possible (e.g., for an online game). Removing the need to query the server to make predictions will definitely reduce the latency and make the website much more responsive.
- When your web service makes predictions based on some private user data, and you want to protect the user's privacy by making the predictions on the client side so that the private data never has to leave the user's machine.

For all these scenarios, you can use the [TensorFlow.js JavaScript library](#). This library can load a TFLite model and make predictions directly in the user's browser. For example, the following Javascript module imports the TFJS library, downloads a pre-trained MobileNet model, and uses this model to classify an image and log the predictions. You can play with the code at <https://hml.info/tfjscode>, using glitch.com, a website that lets you build web apps in your browser for free. Click the `bsp}PREVIEW` button in the lower right corner of the page to see the code in action.

```
import "https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest";
import "https://cdn.jsdelivr.net/npm/@tensorflow-models/mobilenet@1.0.0";

const image = document.getElementById("image");

mobilenet.load().then(model => {
  model.classify(image).then(predictions => {
    for (var i = 0; i < predictions.length; i++) {
      let className = predictions[i].className
      let proba = (predictions[i].probability * 100).toFixed(1)
      console.log(className + " : " + proba + "%");
    }
  })
})
```

```
        }
    });
});
```

It's even possible to turn this website into a *Progressive Web App* (PWA): this is a website that respects a number of criteria⁸ that allow it to be viewed in any browser, and even installed as a standalone app on a mobile device. For example, try visiting <https://homl.info/tfjswpa> on a mobile device: most modern browsers will ask you whether you would like to add TFJS Demo to your home screen. If you accept, you will see a new icon in your list of applications. Clicking this icon will load the TFJS Demo website inside its own window, just like a regular mobile app. A PWA can even be configured to work offline, by using a *service worker*: this is a JavaScript module that runs in its own separate thread in the browser and intercepts network requests, allowing it to cache resources so the PWA can run faster, or even entirely offline. It can also deliver push messages, run tasks in the background, and more. PWAs allow you to manage a single code base for the Web and for mobile devices. They also make it easier to ensure that all users run the same version of your application. You can play with this TFJS Demo's PWA code on glitch.com at <https://homl.info/wpacode>.



Check out many more demos of Machine Learning models running in your browser at <https://tensorflow.org/js/demos>.

TFJS also supports training a model directly in your Web browser! And it's actually pretty fast. If your computer has a GPU card, then TFJS can generally use it, even if it's not an Nvidia card. Indeed, TFJS will use WebGL when it's available, and since modern Web browsers generally support a wide range of GPU cards, TFJS actually supports more GPU cards than regular TensorFlow (which only supports Nvidia cards).

Training a model in a user's Web browser can be especially useful to guarantee that this user's data remains private. A model can be trained centrally, and then fine-tuned locally, in the browser, based on that user's data. If you're interested in this topic, check out [federated learning](#).

Once again, doing justice to this topic requires a whole book. If you want to learn more about TensorFlow.js, check out the book [Practical Deep Learning for Cloud, Mobile, and Edge](#) (O'Reilly, 2019), by Anirudh Koul, Siddha Ganju, and Meher Kasam, or [Learning TensorFlow.js](#) (O'Reilly, 2021), by Gant Laborde.

⁸ For example, a PWA must include icons of various sizes for different mobile devices, it must be served via HTTPS, it must include a manifest file containing metadata such as the name of the app and the background color.

Now that we've seen how to deploy TensorFlow models to TF Serving, or to the cloud with Vertex AI, or to mobile and embedded devices using TFLite, or to the Web browser using TFJS, let's discuss how to use GPUs to speed up computations.

Using GPUs to Speed Up Computations

In [Chapter 11](#) we saw several techniques that can considerably speed up training: better weight initialization, sophisticated optimizers, and so on. But even with all of these techniques, training a large neural network on a single machine with a single CPU can take hours, days, or even weeks, depending on the task. Thanks to GPUs, this training time can be reduced down to minutes or hours. Not only does this save an enormous amount of time, but it also means that you can experiment with various models much more easily, and frequently retrain your models on fresh data.

In the previous chapters, we used GPU-enabled Runtimes on Google Colab: all we had to do was to select *Runtime → Change runtime type* from the menu, and choose the GPU accelerator type. TensorFlow automatically detected the GPU and used it to speed up computations, the code was exactly the same as without a GPU. And in this chapter we saw how to deploy your models to Vertex AI on multiple GPU-enabled compute nodes: it was just a matter of selecting the right GPU-enabled Docker image when creating the Vertex AI Model, and selecting the desired GPU type when calling `endpoint.deploy()`, that's all. But what if you want to buy your own GPU? And what if you want to distribute the computations across the CPU and multiple GPU devices on a single machine (see [Figure 19-6](#))? This is what we will discuss now, then later in this chapter we will discuss how to distribute computations across multiple servers.

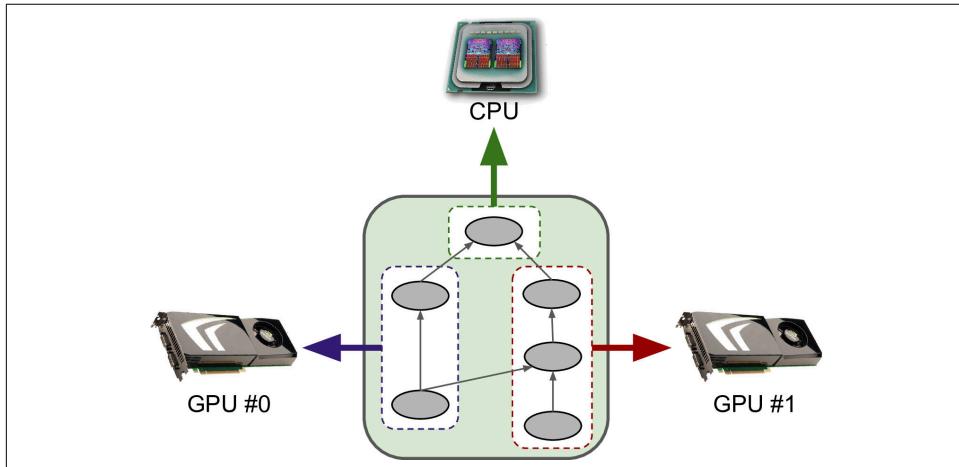


Figure 19-6. Executing a TensorFlow graph across multiple devices in parallel

Getting Your Own GPU

If you know that you'll be using a GPU heavily and for a long period of time, then buying your own can make financial sense. You may also want to train your models locally because you do not want to upload your data to the cloud. Or perhaps you just want to buy a GPU card for gaming, and you'd like to use it for Deep Learning as well.

If you decide to purchase a GPU card, then take some time to make the right choice. You will need to consider the amount of RAM you will need for your tasks (e.g., typically at least 10 GB for image processing or NLP), the bandwidth (i.e., how fast you can send data in and out of the GPU), the number of cores, the cooling system, etc. Tim Dettmers wrote an [excellent blog post](#) to help you choose: I encourage you to read it carefully. At the time of this writing, TensorFlow only supports [Nvidia cards with CUDA Compute Capability 3.5+](#) (as well as Google's TPUs, of course), but it may extend its support to other manufacturers, so make sure to check [TensorFlow's documentation](#) to see what devices are supported today.

If you go for an Nvidia GPU card, you will need to install the appropriate Nvidia drivers and several Nvidia libraries.⁹ These include the *Compute Unified Device Architecture* library (CUDA) Toolkit, which allows developers to use CUDA-enabled GPUs for all sorts of computations (not just graphics acceleration), and the *CUDA Deep Neural Network* library (cuDNN), a GPU-accelerated library of common DNN computations such as activation layers, normalization, forward and backward convolutions, and pooling (see [Chapter 14](#)). cuDNN is part of Nvidia's Deep Learning SDK. Note that you will need to create an Nvidia developer account in order to download it. TensorFlow uses CUDA and cuDNN to control the GPU cards and accelerate computations (see [Figure 19-7](#)).

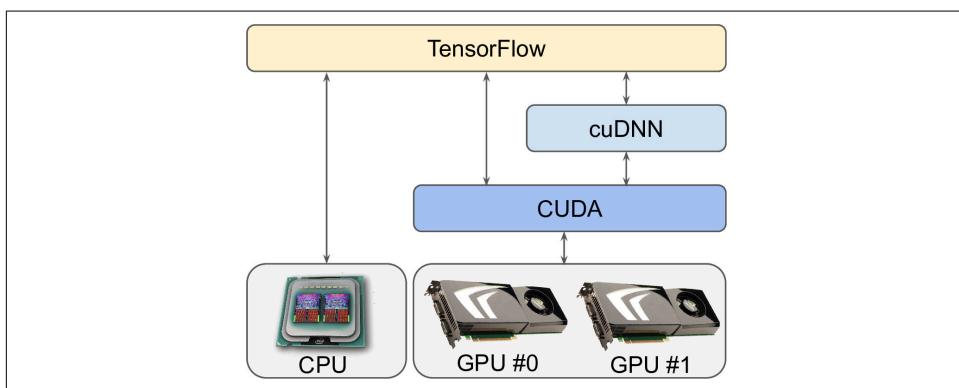


Figure 19-7. TensorFlow uses CUDA and cuDNN to control GPUs and boost DNNs

⁹ Please check the TensorFlow docs for detailed and up-to-date installation instructions, as they change quite often.

Once you have installed the GPU card(s) and all the required drivers and libraries, you can use the `nvidia-smi` command to check that everything is properly installed. This command lists the available GPU cards, as well as all the processes running on each card. In this example, it's an Nvidia Tesla T4 GPU card with about 15 GB of available RAM, and there are no processes currently running on it.

```
$ nvidia-smi
Sun Apr 10 04:52:10 2022
+
| NVIDIA-SMI 460.32.03     Driver Version: 460.32.03     CUDA Version: 11.2  |
+-----+-----+-----+-----+-----+-----+-----+-----+
| GPU  Name      Persistence-M | Bus-Id      Disp.A  | Volatile Uncorr. ECC  | | |
| Fan  Temp  Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M.  |
|          |             |              |             |          MIG M.   |
+-----+-----+-----+-----+-----+-----+-----+-----+
|  0  Tesla T4           Off  | 00000000:00:04.0 Off  |                  0 | | |
| N/A   34C    P8    9W /  70W |      3MiB / 15109MiB |      0%     Default |
|          |             |              |             |          N/A   |
+-----+-----+-----+-----+-----+-----+-----+-----+
+
+-----+-----+-----+-----+-----+-----+-----+-----+
| Processes:
| GPU  GI  CI      PID  Type  Process name          GPU Memory  |
|          ID  ID                 Usage          |
+-----+-----+-----+-----+-----+-----+-----+-----+
| No running processes found
+-----+-----+-----+-----+-----+-----+-----+-----+
```

To check that TensorFlow actually sees your GPU, run the following command and make sure the result is not empty:

```
>>> physical_gpus = tf.config.list_physical_devices("GPU")
>>> physical_gpus
[PhysicalDevice(name='/physical device:GPU:0', device type='GPU')]
```

Managing the GPU RAM

By default TensorFlow automatically grabs almost all the RAM in all available GPUs the first time you run a computation. It does this to limit GPU RAM fragmentation. This means that if you try to start a second TensorFlow program (or any program that requires the GPU), it will quickly run out of RAM. This does not happen as often as you might think, as you will most often have a single TensorFlow program running on a machine: usually a training script, a TF Serving node, or a Jupyter notebook. If you need to run multiple programs for some reason (e.g., to train two different models in parallel on the same machine), then you will need to split the GPU RAM between these processes more evenly.

If you have multiple GPU cards on your machine, a simple solution is to assign each of them to a single process. To do this, you can set the `CUDA_VISIBLE_DEVICES`

environment variable so that each process only sees the appropriate GPU card(s). Also set the CUDA_DEVICE_ORDER environment variable to PCI_BUS_ID to ensure that each ID always refers to the same GPU card. For example, if you have four GPU cards, you could start two programs, assigning two GPUs to each of them, by executing commands like the following in two separate terminal windows:

```
$ CUDA_DEVICE_ORDER=PCI_BUS_ID CUDA_VISIBLE_DEVICES=0,1 python3 program_1.py  
# and in another terminal:  
$ CUDA_DEVICE_ORDER=PCI_BUS_ID CUDA_VISIBLE_DEVICES=3,2 python3 program_2.py
```

Program 1 will then only see GPU cards 0 and 1, named "/gpu:0" and "/gpu:1" respectively in TensorFlow, and program 2 will only see GPU cards 2 and 3, named "/gpu:1" and "/gpu:0" respectively (note the order). Everything will work fine (see Figure 19-8). Of course, you can also define these environment variables in Python by setting `os.environ["CUDA_DEVICE_ORDER"]` and `os.environ["CUDA_VISIBLE_DEVICES"]`, as long as you do so before using TensorFlow.

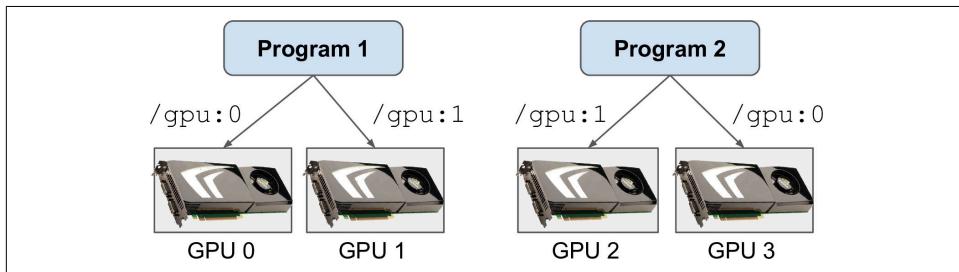


Figure 19-8. Each program gets two GPUs

Another option is to tell TensorFlow to grab only a specific amount of GPU RAM. This must be done immediately after importing TensorFlow. For example, to make TensorFlow grab only 2 GiB of RAM on each GPU, you must create a *logical GPU device* (sometimes called a *virtual GPU device*) for each physical GPU device and set its memory limit to 2 GiB (i.e., 2,048 MiB):

```
for gpu in physical_gpus:  
    tf.config.set_logical_device_configuration(  
        gpu,  
        [tf.config.LogicalDeviceConfiguration(memory_limit=2048)])
```

Let's suppose you have four GPUs, each with at least 4 GiB of RAM: in this case, two programs like this one can run in parallel, each using all four GPU cards (see Figure 19-9). If you run the `nvidia-smi` command while both programs are running, you should see that each process holds 2 GiB of RAM on each card.

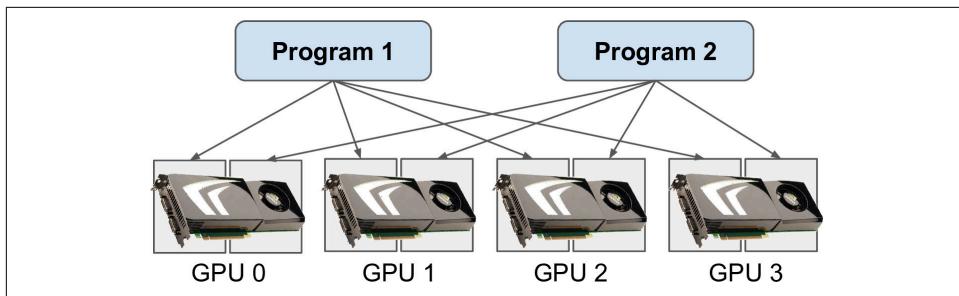


Figure 19-9. Each program gets all four GPUs, but with only 2 GiB of RAM on each GPU

Yet another option is to tell TensorFlow to grab memory only when it needs it. Again, this must be done immediately after importing TensorFlow:

```
for gpu in physical_gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
```

Another way to do this is to set the `TF_FORCE_GPU_ALLOW_GROWTH` environment variable to `true`. With this option, TensorFlow will never release memory once it has grabbed it (again, to avoid memory fragmentation), except of course when the program ends. It can be harder to guarantee deterministic behavior using this option (e.g., one program may crash because another program's memory usage went through the roof), so in production you'll probably want to stick with one of the previous options. However, there are some cases where it is very useful: for example, when you use a machine to run multiple Jupyter notebooks, several of which use TensorFlow. The `TF_FORCE_GPU_ALLOW_GROWTH` environment variable is set to `true` in Colab Runtimes.

Lastly, in some cases you may want to split a GPU into two or more *logical devices*. For example, this is useful if you only have one physical GPU—like in a Colab Runtime—but you want to test a multi-GPU algorithm. The following code splits GPU #0 into two logical devices, with 2 GiB of RAM each (again, this must be done immediately after importing TensorFlow):

```
tf.config.set_logical_device_configuration(
    physical_gpus[0],
    [tf.config.LogicalDeviceConfiguration(memory_limit=2048),
     tf.config.LogicalDeviceConfiguration(memory_limit=2048])
)
```

These two logical devices are called "`/gpu:0`" and "`/gpu:1`", and you can use them as if they were two normal GPUs. You can list all logical devices like this:

```
>>> logical_gpus = tf.config.list_logical_devices("GPU")
>>> logical_gpus
```

```
[LogicalDevice(name='/device:GPU:0', device_type='GPU'),  
 LogicalDevice(name='/device:GPU:1', device_type='GPU')]
```

Now let's see how TensorFlow decides which devices it should use to place variables and execute operations.

Placing Operations and Variables on Devices

Keras and tf.data generally do a good job of placing operations and variables where they belong, but you can also place operations and variables manually on each device, if you want more control:

- You generally want to place the data preprocessing operations on the CPU, and place the neural network operations on the GPUs.
- GPUs usually have a fairly limited communication bandwidth, so it is important to avoid unnecessary data transfers in and out of the GPUs.
- Adding more CPU RAM to a machine is simple and fairly cheap, so there's usually plenty of it, whereas the GPU RAM is baked into the GPU: it is an expensive and thus limited resource, so if a variable is not needed in the next few training steps, it should probably be placed on the CPU (e.g., datasets generally belong on the CPU).

By default, all variables and all operations will be placed on the first GPU (the one named "/gpu:0"), except for variables and operations that don't have a GPU kernel:¹⁰ these are placed on the CPU (always named "/cpu:0"). A tensor or variable's `device` attribute tells you which device it was placed on:¹¹

```
>>> a = tf.Variable([1., 2., 3.]) # float32 variable goes to the GPU  
>>> a.device  
'/job:localhost/replica:0/task:0/device:GPU:0'  
>>> b = tf.Variable([1, 2, 3]) # int32 variable goes to the CPU  
>>> b.device  
'/job:localhost/replica:0/task:0/device:CPU:0'
```

You can safely ignore the prefix `/job:localhost/replica:0/task:0` for now, we will discuss jobs, replicas, and tasks later in this chapter. As you can see, the first variable was placed on GPU #0, which is the default device. However, the second variable was placed on the CPU: this is because there are no GPU kernels for integer variables, or for operations involving integer tensors, so TensorFlow fell back to the CPU.

¹⁰ As we saw in [Chapter 12](#), a kernel is an operation's implementation for a specific data type and device type. For example, there is a GPU kernel for the `float32 tf.matmul()` operation, but there is no GPU kernel for `int32 tf.matmul()`, only a CPU kernel.

¹¹ You can also use `tf.debugging.set_log_device_placement(True)` to log all device placements.

If you want to place an operation on a different device than the default one, use a `tf.device()` context:

```
>>> with tf.device("/cpu:0"):
...     c = tf.Variable([1., 2., 3.])
...
>>> c.device
'/job:localhost/replica:0/task:0/device:CPU:0'
```



The CPU is always treated as a single device ("`/cpu:0`"), even if your machine has multiple CPU cores. Any operation placed on the CPU may run in parallel across multiple cores if it has a multithreaded kernel.

If you explicitly try to place an operation or variable on a device that does not exist or for which there is no kernel, then TensorFlow will silently fallback to the device it would have chosen by default. This is useful when you want to be able to run the same code on different machines that don't have the same number of GPUs. However, you can run `tf.config.set_soft_device_placement(False)` if you prefer to get an exception.

Now how exactly does TensorFlow execute operations across multiple devices?

Parallel Execution Across Multiple Devices

As we saw in [Chapter 12](#), one of the benefits of using TF Functions is parallelism. Let's look at this a bit more closely. When TensorFlow runs a TF Function, it starts by analyzing its graph to find the list of operations that need to be evaluated, and it counts how many dependencies each of them has. TensorFlow then adds each operation with zero dependencies (i.e., each source operation) to the evaluation queue of this operation's device (see [Figure 19-10](#)). Once an operation has been evaluated, the dependency counter of each operation that depends on it is decremented. Once an operation's dependency counter reaches zero, it is pushed to the evaluation queue of its device. And once all the outputs have been computed, they are returned.

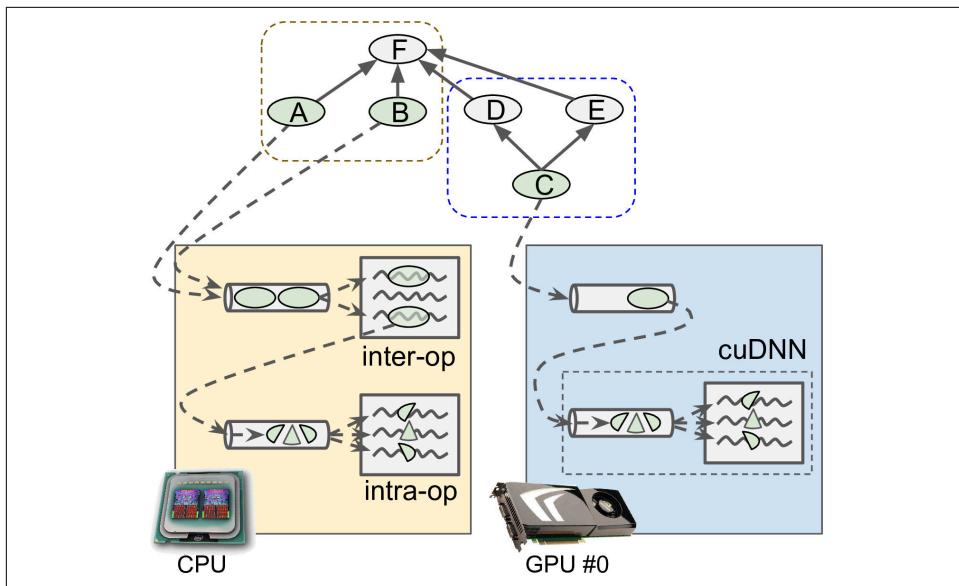


Figure 19-10. Parallelized execution of a TensorFlow graph

Operations in the CPU's evaluation queue are dispatched to a thread pool called the *inter-op thread pool*. If the CPU has multiple cores, then these operations will effectively be evaluated in parallel. Some operations have multithreaded CPU kernels: these kernels split their tasks into multiple suboperations, which are placed in another evaluation queue and dispatched to a second thread pool called the *intra-op thread pool* (shared by all multithreaded CPU kernels). In short, multiple operations and suboperations may be evaluated in parallel on different CPU cores.

For the GPU, things are a bit simpler. Operations in a GPU's evaluation queue are evaluated sequentially. However, most operations have multithreaded GPU kernels, typically implemented by libraries that TensorFlow depends on, such as CUDA and cuDNN. These implementations have their own thread pools, and they typically exploit as many GPU threads as they can (which is the reason why there is no need for an inter-op thread pool in GPUs: each operation already floods most GPU threads).

For example, in [Figure 19-10](#), operations A, B, and C are source ops, so they can immediately be evaluated. Operations A and B are placed on the CPU, so they are sent to the CPU's evaluation queue, then they are dispatched to the inter-op thread pool and immediately evaluated in parallel. Operation A happens to have a multithreaded kernel; its computations are split into three parts, which are executed in parallel by the intra-op thread pool. Operation C goes to GPU #0's evaluation queue, and in this example its GPU kernel happens to use cuDNN, which manages its own intra-op thread pool and runs the operation across many GPU threads in parallel.

Suppose C finishes first. The dependency counters of D and E are decremented and they reach zero, so both operations are pushed to GPU #0's evaluation queue, and they are executed sequentially. Note that C only gets evaluated once, even though both D and E depend on it. Suppose B finishes next. Then F's dependency counter is decremented from 4 to 3, and since that's not 0, it does not run yet. Once A, D, and E are finished, then F's dependency counter reaches 0, and it is pushed to the CPU's evaluation queue and evaluated. Finally, TensorFlow returns the requested outputs.

An extra bit of magic that TensorFlow performs is when the TF Function modifies a stateful resource, such as a variable: it ensures that the order of execution matches the order in the code, even if there is no explicit dependency between the statements. For example, if your TF Function contains `v.assign_add(1)` followed by `v.assign(v * 2)`, TensorFlow will ensure that these operations are executed in that order.



You can control the number of threads in the inter-op thread pool by calling `tf.config.threading.set_inter_op_parallelism_threads()`. To set the number of intra-op threads, use `tf.config.threading.set_intra_op_parallelism_threads()`. This is useful if you do not want TensorFlow to use all the CPU cores or if you want it to be single-threaded.¹²

With that, you have all you need to run any operation on any device, and exploit the power of your GPUs! Here are some of the things you could do:

- You could train several models in parallel, each on its own GPU: just write a training script for each model and run them in parallel, setting `CUDA_DEVICE_ORDER` and `CUDA_VISIBLE_DEVICES` so that each script only sees a single GPU device. This is great for hyperparameter tuning, as you can train in parallel multiple models with different hyperparameters. If you have a single machine with two GPUs, and it takes one hour to train one model on one GPU, then training two models in parallel, each on its own dedicated GPU, will take just one hour. Simple!
- You could train a model on a single GPU and perform all the preprocessing in parallel on the CPU, using the dataset's `prefetch()` method¹³ to prepare the next few batches in advance so that they are ready when the GPU needs them (see [Chapter 13](#)).

¹² This can be useful if you want to guarantee perfect reproducibility, as I explain in [this video](#), based on TF 1.

¹³ At the time of this writing it only prefetches the data to the CPU RAM, but you can use `tf.data.experimental.prefetch_to_device()` to make it prefetch the data and push it to the device of your choice so that the GPU does not waste time waiting for the data to be transferred.

- If your model takes two images as input and processes them using two CNNs before joining their outputs,¹⁴ then it will probably run much faster if you place each CNN on a different GPU.
- You can create an efficient ensemble: just place a different trained model on each GPU so that you can get all the predictions much faster to produce the ensemble's final prediction.

But what if you want to speed up training by using multiple GPUs?

Training Models Across Multiple Devices

There are two main approaches to training a single model across multiple devices: *model parallelism*, where the model is split across the devices, and *data parallelism*, where the model is replicated across every device, and each replica is trained on a different subset of the data. Let's look at these two options.

Model Parallelism

So far we have trained each neural network on a single device. What if we want to train a single neural network across multiple devices? This requires chopping the model into separate chunks and running each chunk on a different device. Unfortunately, such model parallelism turns out to be pretty tricky, and it really depends on the architecture of your neural network. For fully connected networks, there is generally not much to be gained from this approach (see [Figure 19-11](#)). Intuitively, it may seem that an easy way to split the model is to place each layer on a different device, but this does not work because each layer needs to wait for the output of the previous layer before it can do anything. So perhaps you can slice it vertically—for example, with the left half of each layer on one device, and the right part on another device? This is slightly better, since both halves of each layer can indeed work in parallel, but the problem is that each half of the next layer requires the output of both halves, so there will be a lot of cross-device communication (represented by the dashed arrows). This is likely to completely cancel out the benefit of the parallel computation, since cross-device communication is slow (and even more so when the devices are located on different machines).

¹⁴ If the two CNNs are identical, then it is called a *Siamese neural network*.

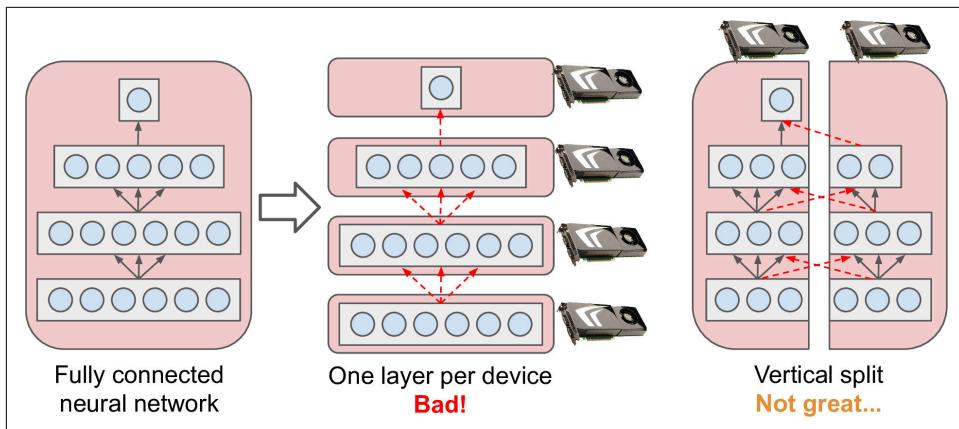


Figure 19-11. Splitting a fully connected neural network

Some neural network architectures, such as convolutional neural networks (see [Chapter 14](#)), contain layers that are only partially connected to the lower layers, so it is much easier to distribute chunks across devices in an efficient way ([Figure 19-12](#)).

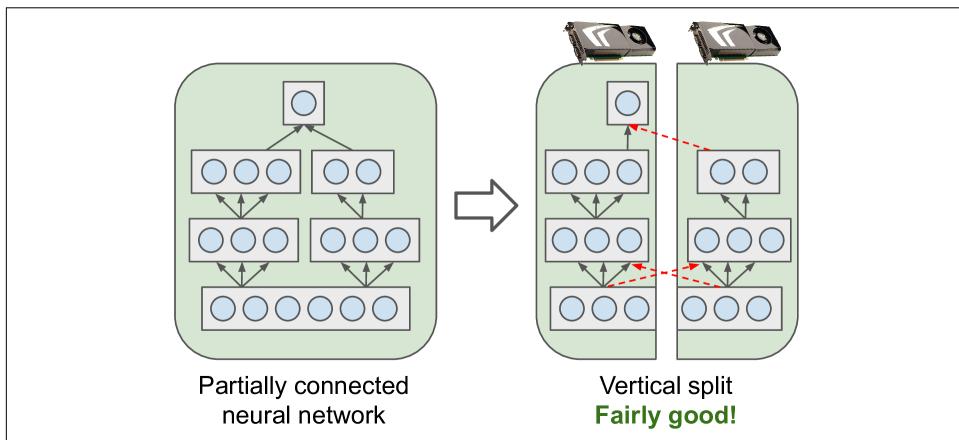


Figure 19-12. Splitting a partially connected neural network

Deep recurrent neural networks (see [Chapter 15](#)) can be split a bit more efficiently across multiple GPUs. If you split the network horizontally by placing each layer on a different device, and you feed the network with an input sequence to process, then at the first time step only one device will be active (working on the sequence's first value), at the second step two will be active (the second layer will be handling the output of the first layer for the first value, while the first layer will be handling the second value), and by the time the signal propagates to the output layer, all devices will be active simultaneously ([Figure 19-13](#)). There is still a lot of cross-device communication going on, but since each cell may be fairly complex, the benefit

of running multiple cells in parallel may (in theory) outweigh the communication penalty. However, in practice a regular stack of LSTM layers running on a single GPU actually runs much faster.

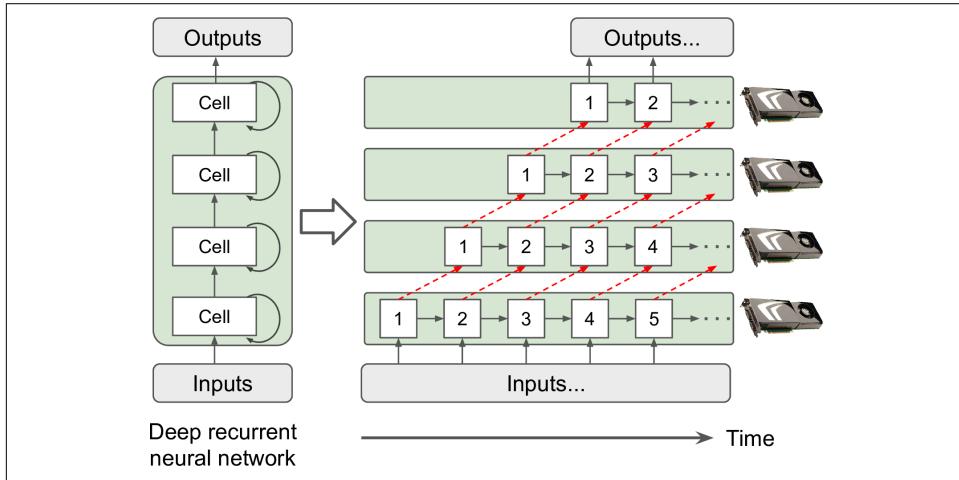


Figure 19-13. Splitting a deep recurrent neural network

In short, model parallelism may speed up running or training some types of neural networks, but not all, and it requires special care and tuning, such as making sure that devices that need to communicate the most run on the same machine.¹⁵ So let's look at a much simpler and generally more efficient option: data parallelism.

Data Parallelism

Another way to parallelize the training of a neural network is to replicate it on every device and run each training step simultaneously on all replicas, using a different mini-batch for each. The gradients computed by each replica are then averaged, and the result is used to update the model parameters. This is called *data parallelism*, or sometimes *Single Program Multiple Data* (SPMD). There are many variants of this idea, so let's look at the most important ones.

Data parallelism using the mirrored strategy

Arguably the simplest approach is to completely mirror all the model parameters across all the GPUs and always apply the exact same parameter updates on every GPU. This way, all replicas always remain perfectly identical. This is called the *mirrored strategy*, and it turns out to be quite efficient, especially when using a single machine (see Figure 19-14).

¹⁵ If you are interested in going further with model parallelism, check out [Mesh TensorFlow](#).

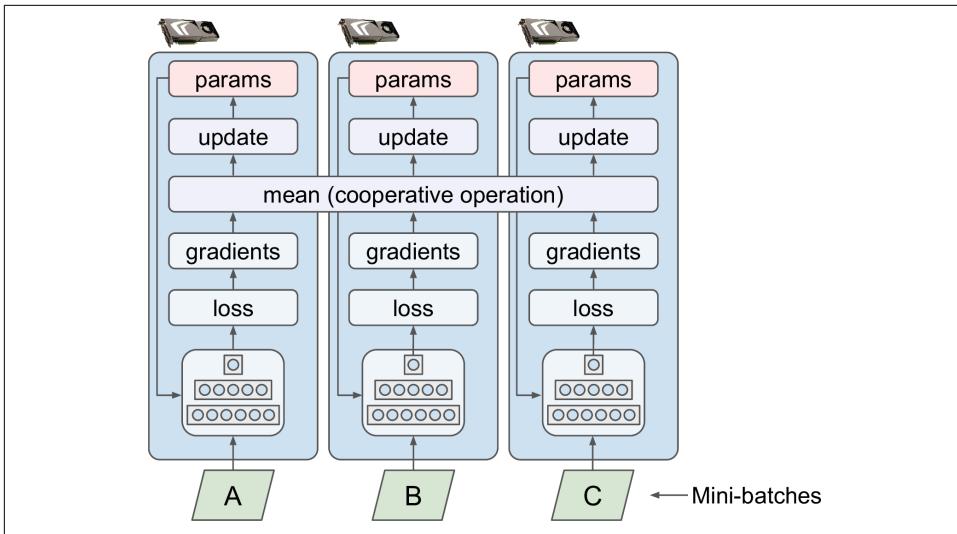


Figure 19-14. Data parallelism using the mirrored strategy

The tricky part when using this approach is to efficiently compute the mean of all the gradients from all the GPUs and distribute the result across all the GPUs. This can be done using an *AllReduce* algorithm, a class of algorithms where multiple nodes collaborate to efficiently perform a *reduce operation* (such as computing the mean, sum, and max), while ensuring that all nodes obtain the same final result. Fortunately, there are off-the-shelf implementations of such algorithms, as we will see.

Data parallelism with centralized parameters

Another approach is to store the model parameters outside of the GPU devices performing the computations (called *workers*), for example on the CPU (see Figure 19-15). In a distributed setup, you may place all the parameters on one or more CPU-only servers called *parameter servers*, whose only role is to host and update the parameters.

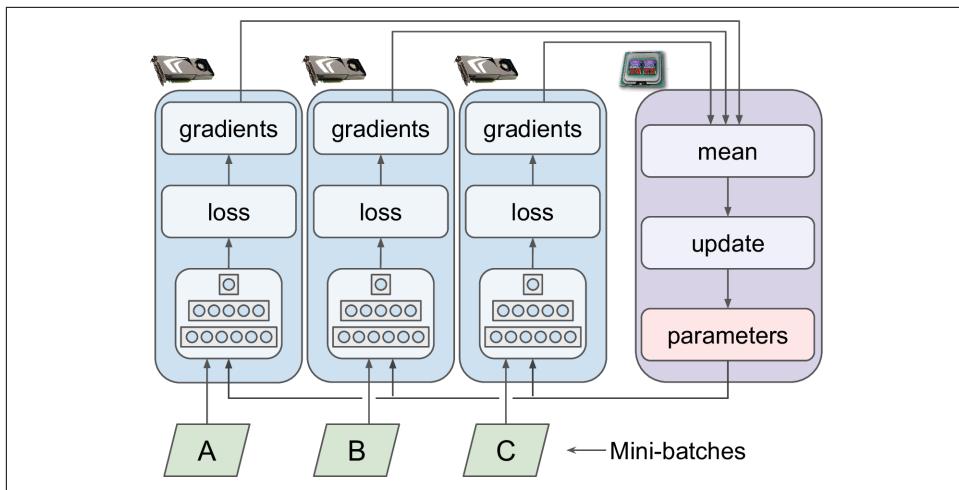


Figure 19-15. Data parallelism with centralized parameters

Whereas the mirrored strategy imposes synchronous weight updates across all GPUs, this centralized approach allows either synchronous or asynchronous updates. Let's see the pros and cons of both options.

Synchronous updates. With *synchronous updates*, the aggregator waits until all gradients are available before it computes the average gradients and passes them to the optimizer, which will update the model parameters. Once a replica has finished computing its gradients, it must wait for the parameters to be updated before it can proceed to the next mini-batch. The downside is that some devices may be slower than others, so the fast devices will have to wait for the slow ones at every step, making the whole process as slow as the slowest device. Moreover, the parameters will be copied to every device almost at the same time (immediately after the gradients are applied), which may saturate the parameter servers' bandwidth.



To reduce the waiting time at each step, you could ignore the gradients from the slowest few replicas (typically $\sim 10\%$). For example, you could run 20 replicas, but only aggregate the gradients from the fastest 18 replicas at each step, and just ignore the gradients from the last 2. As soon as the parameters are updated, the first 18 replicas can start working again immediately, without having to wait for the 2 slowest replicas. This setup is generally described as having 18 replicas plus 2 *spare replicas*.¹⁶

Asynchronous updates. With asynchronous updates, whenever a replica has finished computing the gradients, the gradients are immediately used to update the model parameters. There is no aggregation (it removes the “mean” step in Figure 19-15) and no synchronization. Replicas work independently of the other replicas. Since there is no waiting for the other replicas, this approach runs more training steps per minute. Moreover, although the parameters still need to be copied to every device at every step, this happens at different times for each replica, so the risk of bandwidth saturation is reduced.

Data parallelism with asynchronous updates is an attractive choice because of its simplicity, the absence of synchronization delay, and a better use of the bandwidth. However, although it works reasonably well in practice, it is almost surprising that it works at all! Indeed, by the time a replica has finished computing the gradients based on some parameter values, these parameters will have been updated several times by other replicas (on average $N - 1$ times, if there are N replicas), and there is no guarantee that the computed gradients will still be pointing in the right direction (see Figure 19-16). When gradients are severely out-of-date, they are called *stale gradients*: they can slow down convergence, introducing noise and wobble effects (the learning curve may contain temporary oscillations), or they can even make the training algorithm diverge.

¹⁶ This name is slightly confusing because it sounds like some replicas are special, doing nothing. In reality, all replicas are equivalent: they all work hard to be among the fastest at each training step, and the losers vary at every step (unless some devices are really slower than others). However, it does mean that if one or two servers crash, training will continue just fine.

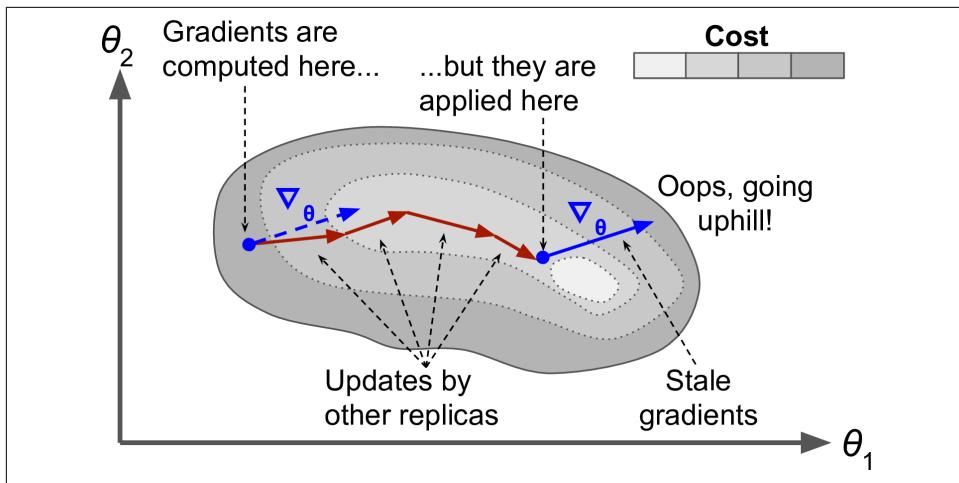


Figure 19-16. Stale gradients when using asynchronous updates

There are a few ways you can reduce the effect of stale gradients:

- Reduce the learning rate.
- Drop stale gradients or scale them down.
- Adjust the mini-batch size.
- Start the first few epochs using just one replica (this is called the *warmup phase*). Stale gradients tend to be more damaging at the beginning of training, when gradients are typically large and the parameters have not settled into a valley of the cost function yet, so different replicas may push the parameters in quite different directions.

A paper published by the Google Brain team in 2016¹⁷ benchmarked various approaches and found that using synchronous updates with a few spare replicas was more efficient than using asynchronous updates, not only converging faster but also producing a better model. However, this is still an active area of research, so you should not rule out asynchronous updates just yet.

Bandwidth saturation

Whether you use synchronous or asynchronous updates, data parallelism with centralized parameters still requires communicating the model parameters from the parameter servers to every replica at the beginning of each training step, and the gradients in the other direction at the end of each training step. Similarly, when using

¹⁷ Jianmin Chen et al., “Revisiting Distributed Synchronous SGD,” arXiv preprint arXiv:1604.00981 (2016).

the mirrored strategy, the gradients produced by each GPU will need to be shared with every other GPU. Unfortunately, there often comes a point where adding an extra GPU will not improve performance at all because the time spent moving the data into and out of GPU RAM (and across the network in a distributed setup) will outweigh the speedup obtained by splitting the computation load. At that point, adding more GPUs will just worsen the bandwidth saturation and actually slow down training.

Saturation is more severe for large dense models, since they have a lot of parameters and gradients to transfer. It is less severe for small models (but the parallelization gain is limited) and for large sparse models, where the gradients are typically mostly zeros and so can be communicated efficiently. Jeff Dean, initiator and lead of the Google Brain project, [reported](#) typical speedups of 25–40× when distributing computations across 50 GPUs for dense models, and a 300× speedup for sparser models trained across 500 GPUs. As you can see, sparse models really do scale better. Here are a few concrete examples:

- Neural machine translation: 6× speedup on 8 GPUs
- Inception/ImageNet: 32× speedup on 50 GPUs
- RankBrain: 300× speedup on 500 GPUs

There is plenty of research going on to alleviate the bandwidth saturation issue, with the goal of allowing training to scale linearly with the number of GPUs available. For example, a [2018 paper¹⁸](#) by a team of researchers from Carnegie Mellon University, Stanford University, and Microsoft Research, proposed a system called *PipeDream* which managed to reduce network communications by over 90%, making it possible to train large models across many machines. They achieved this using a new technique called *pipeline parallelism*, which combines model parallelism and data parallelism: they chop the model into consecutive parts, called *stages*, and train each stage on a different machine. Each stage pulls a mini-batch from its input queue, processes it, then pushes its output to the next stage's input queue, then it immediately starts working on the following mini-batch: this results in an asynchronous pipeline in which all machines work in parallel with very little idle time. During training, each stage alternates one round of forward propagation, and one round of backpropagation (see [Figure 19-17](#)). Each stage pulls a mini-batch from its input queue, process it, sends the outputs to the next stage's input queue, then it pulls one mini-batch of gradients from its gradient queue, it backpropagates these gradients and updates its own model parameters, then it pushes the backpropagated gradients to the previous stage's gradient queue. It then repeats the whole process again and again. Each stage

¹⁸ Aaron Harlap et al., “PipeDream: Fast and Efficient Pipeline Parallel DNN Training” (2018).

can also use regular data parallelism (e.g., using the mirrored strategy), independently from the other stages.

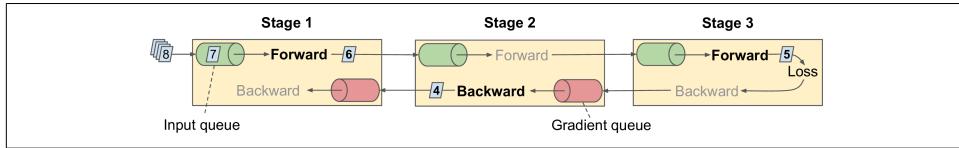


Figure 19-17. PipeDream's pipeline parallelism

However, as it's presented, PipeDream would not work so well. To understand why, consider mini-batch #5 in Figure 19-17: when it went through stage 1 during the forward pass, the gradients #4 had not yet been backpropagated through that stage, but by the time gradients #5 flow back to stage 1, gradients #4 will have been used to update the model parameters, so gradients #5 will be a bit stale. As we have seen, this can degrade training speed and accuracy, and even make it diverge. Actually, the more stages there are, the worse this problem becomes. But the PipeDream authors proposed methods to mitigate this issue: for example, each stage saves weights during forward propagation, and restores them during backpropagation, to ensure that the same weights are used for both the forward pass and the backward pass. This is called *weight stashing*. Thanks to this, PipeDream demonstrated impressive scaling capability, well beyond simple data parallelism.

The latest breakthrough in this field of research was published in a [2022 paper¹⁹](#) by Google researchers: they developed a system called *Pathways* which uses automated model parallelism, *asynchronous gang scheduling*, and other techniques to reach close to 100% hardware utilization across thousands of TPUs! Scheduling means organizing when and where each task must run, and gang scheduling means running related tasks at the same time in parallel and close to each other to reduce the time tasks have to wait for the others' outputs. As we saw in Chapter 16, this system was used to train a massive language model across over 6,000 TPUs, with close to 100% hardware utilization: that's a mindblowing engineering feat.

As of April 2022, Pathways is not public yet, but it's likely that in the near future you will be able to train huge models on Vertex AI using Pathways, or a similar system. In the meantime, to reduce the saturation problem, you probably want to use a few powerful GPUs rather than plenty of weak GPUs, and if you need to train a model across multiple servers, you should group your GPUs on few and very well interconnected servers. You can also try dropping the float precision from 32 bits (`tf.float32`) to 16 bits (`tf.bfloat16`). This will cut in half the amount of data to transfer, often without much impact on the convergence rate or the model's

¹⁹ Paul Barham et al., "Pathways: Asynchronous Distributed Dataflow for ML" (2022).

performance. Lastly, if you are using centralized parameters, you can shard (split) the parameters across multiple parameter servers: adding more parameter servers will reduce the network load on each server and limit the risk of bandwidth saturation.

OK, now we've gone through all the theory, let's actually train a model across multiple GPUs!

Training at Scale Using the Distribution Strategies API

Luckily, TensorFlow comes with a very nice API that takes care of all the complexity of distributing your model across multiple devices and machines: the *Distribution Strategies API*. To train a Keras model across all available GPUs (on a single machine, for now) using data parallelism with the mirrored strategy, just create a `MirroredStrategy` object, call its `scope()` method to get a distribution context, and wrap the creation and compilation of your model inside that context. Then call the model's `fit()` method normally:

```
strategy = tf.distribute.MirroredStrategy()

with strategy.scope():
    model = tf.keras.Sequential([...]) # create a Keras model normally
    model.compile([...]) # compile the model normally

batch_size = 100 # preferably divisible by the number of replicas
model.fit(X_train, y_train, epochs=10,
           validation_data=(X_valid, y_valid), batch_size=batch_size)
```

Under the hood, Keras is distribution-aware, so in this `MirroredStrategy` context it knows that it must replicate all variables and operations across all available GPU devices. If you look at the model's weights, they are of type `MirroredVariable`:

```
>>> type(model.weights[0])
tensorflow.python.distribute.values.MirroredVariable
```

Note that the `fit()` method will automatically split each training batch across all the replicas, so it's preferable to ensure that the batch size is divisible by the number of replicas (i.e., the number of available GPUs), so that all replicas get batches of the same size. And that's all! Training will generally be significantly faster than using a single device, and the code change was really minimal.

Once you have finished training your model, you can use it to make predictions efficiently: call the `predict()` method, and it will automatically split the batch across all replicas, making predictions in parallel. Again, the batch size must be divisible by the number of replicas. If you call the model's `save()` method, it will be saved as a regular model, *not* as a mirrored model with multiple replicas. So when you load it, it will run like a regular model, on a single device: by default on GPU #0, or on the CPU

if there are no GPUs. If you want to load a model and run it on all available devices, you must call `tf.keras.models.load_model()` within a distribution context:

```
with strategy.scope():
    model = tf.keras.models.load_model("my_mirrored_model")
```

If you only want to use a subset of all the available GPU devices, you can pass the list to the `MirroredStrategy`'s constructor:

```
strategy = tf.distribute.MirroredStrategy(devices=["/gpu:0", "/gpu:1"])
```

By default, the `MirroredStrategy` class uses the *NVIDIA Collective Communications Library* (NCCL) for the AllReduce mean operation, but you can change it by setting the `cross_device_ops` argument to an instance of the `tf.distribute.HierarchicalCopyAllReduce` class, or an instance of the `tf.distribute.ReductionToOneDevice` class. The default NCCL option is based on the `tf.distribute.NcclAllReduce` class, which is usually faster, but this depends on the number and types of GPUs, so you may want to give the alternatives a try.²⁰

If you want to try using data parallelism with centralized parameters, replace the `MirroredStrategy` with the `CentralStorageStrategy`:

```
strategy = tf.distribute.experimental.CentralStorageStrategy()
```

You can optionally set the `compute_devices` argument to specify the list of devices you want to use as workers—by default it will use all available GPUs—and you can optionally set the `parameter_device` argument to specify the device you want to store the parameters on—by default it will use the CPU, or the GPU if there is just one.

Now let's see how to train a model across a cluster of TensorFlow servers!

Training a Model on a TensorFlow Cluster

A *TensorFlow cluster* is a group of TensorFlow processes running in parallel, usually on different machines, and talking to each other to complete some work—for example, training or executing a neural network. Each TF process in the cluster is called a *task*, or a *TF server*. It has an IP address, a port, and a type (also called its *role* or its *job*). The type can be either "worker", "chief", "ps" (parameter server), or "evaluator":

- Each *worker* performs computations, usually on a machine with one or more GPUs.

²⁰ For more details on AllReduce algorithms, read this [great post](#) by Yuichiro Ueno, and this page on [scaling with NCCL](#).

- The *chief* performs computations as well (it is a worker), but it also handles extra work such as writing TensorBoard logs or saving checkpoints. There is a single chief in a cluster. If no chief is specified explicitly, then by convention the first worker is the chief.
- A *parameter server* only keeps track of variable values, and it is usually on a CPU-only machine. This type of task is only used with the `ParameterServerStrategy`.
- An *evaluator* obviously takes care of evaluation. This type is not used often, and when it's used, there's usually just one evaluator.

To start a TensorFlow cluster, you must first define its specification. This means defining each task's IP address, TCP port, and type. For example, the following *cluster specification* defines a cluster with three tasks (two workers and one parameter server; see [Figure 19-18](#)). The cluster spec is a dictionary with one key per job, and the values are lists of task addresses (*IP:port*):

```
cluster_spec = {
    "worker": [
        "machine-a.example.com:2222",      # /job:worker/task:0
        "machine-b.example.com:2222"        # /job:worker/task:1
    ],
    "ps": ["machine-a.example.com:2221"]  # /job:ps/task:0
}
```

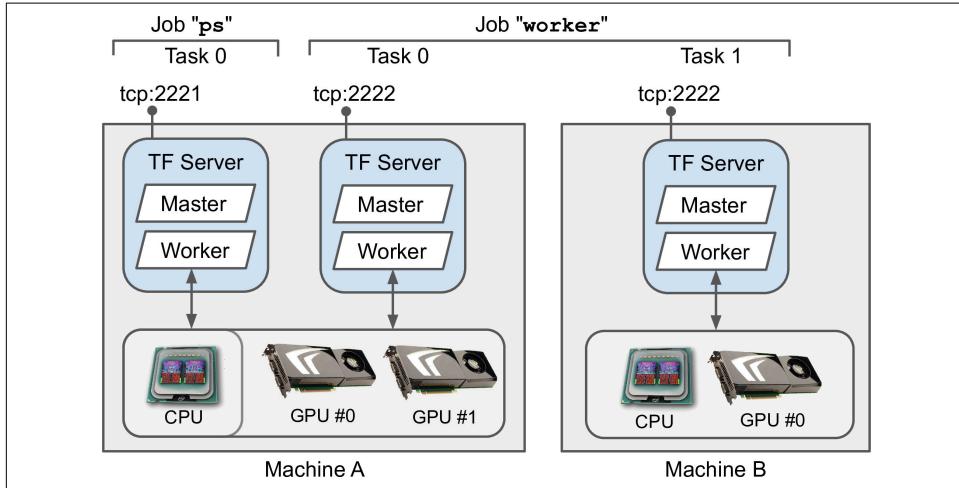


Figure 19-18. TensorFlow cluster

In general there will be a single task per machine, but as this example shows, you can configure multiple tasks on the same machine if you want. In this case, if they share the same GPUs, make sure the RAM is split appropriately, as discussed earlier.



By default, every task in the cluster may communicate with every other task, so make sure to configure your firewall to authorize all communications between these machines on these ports (it's usually simpler if you use the same port on every machine).

When you start a task, you must give it the cluster spec, and you must also tell it what its type and index are (e.g., worker #0). The simplest way to specify everything at once (both the cluster spec and the current task's type and index) is to set the TF_CONFIG environment variable before starting TensorFlow. It must be a JSON-encoded dictionary containing a cluster specification (under the "cluster" key) and the type and index of the current task (under the "task" key). For example, the following TF_CONFIG environment variable uses the cluster we just defined and specifies that the task to start is the worker #0:

```
os.environ["TF_CONFIG"] = json.dumps({
    "cluster": cluster_spec,
    "task": {"type": "worker", "index": 0}
})
```



In general you want to define the TF_CONFIG environment variable outside of Python, so the code does not need to include the current task's type and index (this makes it possible to use the same code across all workers).

Now let's train a model on a cluster! We will start with the mirrored strategy. First, you need to set the TF_CONFIG environment variable appropriately for each task. There should be no parameter server (remove the "ps" key in the cluster spec), and in general you will want a single worker per machine. Make extra sure you set a different task index for each task. Finally, run the following script on every worker:

```
import tempfile
import tensorflow as tf

strategy = tf.distribute.MultiWorkerMirroredStrategy() # at the start!
resolver = tf.distribute.cluster_resolver.TFConfigClusterResolver()
print(f"Starting task {resolver.task_type} #{resolver.task_id}")
[...] # Load and split the MNIST dataset

with strategy.scope():
    model = tf.keras.Sequential([...]) # build the Keras model
    model.compile([...]) # compile the model

model.fit(X_train, y_train, validation_data=(X_valid, y_valid), epochs=10)

if resolver.task_id == 0: # the chief saves the model to the right location
    model.save("my_mnist_multiworker_model", save_format="tf")
```

```
else:  
    tmpdir = tempfile.mkdtemp() # other workers save to a temporary directory  
    model.save(tmpdir, save_format="tf")  
    tf.io.gfile.rmtree(tmpdir) # and we can delete this directory at the end!
```

That's almost the same code we used earlier, except this time we are using the `MultiWorkerMirroredStrategy`. When you start this script on the first workers, they will remain blocked at the AllReduce step, but training will begin as soon as the last worker starts up, and you will see them all advancing at exactly the same rate, since they synchronize at each step.



When using the `MultiWorkerMirroredStrategy`, it's important to ensure that all workers do the same thing, including saving model checkpoints, or writing TensorBoard logs, even though we will only keep what the chief writes. This is because these operations may need to run the AllReduce operations, so all workers must be in sync.

There are two AllReduce implementations for this distribution strategy: a ring AllReduce algorithm based on gRPC for the network communications, and NCCL's implementation. The best algorithm to use depends on the number of workers, the number and types of GPUs, and the network. By default, TensorFlow will apply some heuristics to select the right algorithm for you, but you can force NCCL (or RING) like this:

```
strategy = tf.distribute.MultiWorkerMirroredStrategy(  
    communication_options=tf.distribute.experimental.CommunicationOptions(  
        implementation=tf.distribute.experimental.CollectiveCommunication.NCCL))
```

If you prefer to implement asynchronous data parallelism with parameter servers, change the strategy to `ParameterServerStrategy`, add one or more parameter servers, and configure `TF_CONFIG` appropriately for each task. Note that although the workers will work asynchronously, the replicas on each worker will work synchronously.

Lastly, if you have access to [TPUs on Google Cloud](#)—for example if you use Colab and you set the accelerator type to TPU—then you can create a `TPUStrategy` like this:

```
resolver = tf.distribute.cluster_resolver.TPUClusterResolver()  
tf.tpu.experimental.initialize_tpu_system(resolver)  
strategy = tf.distribute.experimental.TPUStrategy(resolver)
```

This needs to be run right after importing TensorFlow. You can then use this strategy normally.



If you are a researcher, you may be eligible to use TPUs for free; see <https://tensorflow.org/tfrc> for more details.

You can now train models across multiple GPUs and multiple servers: give yourself a pat on the back! If you want to train a very large model, you will need many GPUs, across many servers, which will require either buying a lot of hardware or managing a lot of cloud virtual machines. In many cases, it's less hassle and less expensive to use a cloud service that takes care of provisioning and managing all this infrastructure for you, just when you need it. Let's see how to do that using Vertex AI.

Running Large Training Jobs on Vertex AI

Vertex AI allows you to create custom training jobs with your own training code. In fact, you can use almost the same training code as you would use on your own TF cluster. The main thing you must change is where the chief should save the model, the checkpoints, and the TensorBoard logs. Instead of saving the model to a local directory, the chief must save it to GCS, using the path provided by Vertex AI in the `AIP_MODEL_DIR` environment variable. For the model checkpoints and TensorBoard logs, you should use the paths contained in the `AIP_CHECKPOINT_DIR` and `AIP_TENSORBOARD_LOG_DIR` environment variables, respectively. Of course, you must also make sure that the training data can be accessed from the virtual machines, such as on GCS, or another GCP service like BigQuery, or directly from the Web. Lastly, Vertex AI sets the "chief" task type explicitly, so you should identify the chief using `resolved.task_type == "chief"` instead of `resolved.task_id == 0`.

```
import os
[...] # other imports, create MultiWorkerMirroredStrategy, and resolver

if resolver.task_type == "chief":
    model_dir = os.getenv("AIP_MODEL_DIR") # paths provided by Vertex AI
    tensorboard_log_dir = os.getenv("AIP_TENSORBOARD_LOG_DIR")
    checkpoint_dir = os.getenv("AIP_CHECKPOINT_DIR")
else:
    tmp_dir = Path(tempfile.mkdtemp()) # other workers use a temporary dirs
    model_dir = tmp_dir / "model"
    tensorboard_log_dir = tmp_dir / "logs"
    checkpoint_dir = tmp_dir / "ckpt"

callbacks = [tf.keras.callbacks.TensorBoard(tensorboard_log_dir),
            tf.keras.callbacks.ModelCheckpoint(checkpoint_dir)]
[...] # build and compile using the strategy scope, just like earlier
model.fit(X_train, y_train, validation_data=(X_valid, y_valid), epochs=10,
           callbacks=callbacks)
model.save(model_dir, save_format="tf")
```



If you place the training data on GCS, you can create a `tf.data.TextLineDataset` or `tf.data.TFRecordDataset` to access it: just use the GCS paths as the filenames (e.g., `gs://my_bucket/data/001.csv`). These datasets rely on the `tf.io.gfile` package to access files: it supports both local files and GCS files.

Now we can create a custom training job on Vertex AI, based on this script. We specify the job name, the path to our training script, the Docker image to use for training, the one to use for predictions (after training), any additional Python libraries you may need, and lastly the bucket that Vertex AI should use as a staging directory to store the training script. By default, that's also where the training script will save the trained model, as well as the TensorBoard logs and model checkpoints (if any). Let's create the job:

```
custom_training_job = aiplatform.CustomTrainingJob(  
    display_name="my_custom_training_job",  
    script_path="my_vertex_ai_training_task.py",  
    container_uri="gcr.io/cloud-aiplatform/training/tf-gpu.2-4:latest",  
    model_serving_container_image_uri=server_image,  
    requirements=["gcsfs==2022.3.0"], # not needed, this is just an example  
    staging_bucket=f"gs://{bucket_name}/staging"  
)
```

And now let's run it on two workers, each with 2 GPUs:

```
mnist_model2 = custom_training_job.run(  
    machine_type="n1-standard-4",  
    replica_count=2,  
    accelerator_type="NVIDIA_TESLA_K80",  
    accelerator_count=2,  
)
```

And that's it: Vertex AI will provision the compute nodes you requested (within your quotas), and it will run your training script across them. Once the job is complete, the `run()` method will return a trained model that you can use exactly like the one you created earlier: you can deploy it to an endpoint, or use it to make batch predictions. If anything goes wrong during training, you can view the logs in the GCP console: in the `☰` navigation menu, select *Vertex AI* → *Training*, click on your training job, and click on *VIEW LOGS*. Alternatively, you can click on the *CUSTOM JOBS* tab and copy the job's ID (e.g., 1234), then select *Logging* from the `☰` navigation menu, and query `resource.labels.job_id=1234`.



To visualize the training progress, just start TensorBoard and point its `--logdir` to the GCS path of the logs. It will use *application default credentials*, which you can setup using `gcloud auth application-default login`. Vertex AI also offers hosted TensorBoard servers if you prefer.

If you want to try out a few hyperparameter values, one option is to run multiple jobs. You can pass the hyperparameter values to your script as command line arguments by setting the `args` parameter when calling the `run()` method. Or you can pass them as environment variables using the `environment_variables` parameter.

However, if you want to run a large hyperparameter tuning job on the cloud, a much better option is to use Vertex AI's hyperparameter tuning service. Let's see how.

Hyperparameter Tuning on Vertex AI

Vertex AI's hyperparameter tuning service is based on a Bayesian optimization algorithm, capable of quickly finding optimal combinations of hyperparameters: to use it, you first need to create a training script that accepts hyperparameter values as command line arguments. For example, your script could use the `argparse` standard library like this:

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("--n_hidden", type=int, default=2)
parser.add_argument("--n_neurons", type=int, default=256)
parser.add_argument("--learning_rate", type=float, default=1e-2)
parser.add_argument("--optimizer", default="adam")
args = parser.parse_args()
```

The hyperparameter tuning service will call your script multiple times, each time with different hyperparameter values: each run is called a *trial*, and the set of trials is called a *study*. Your training script must then use the given hyperparameter values to build and compile a model. You can use a mirrored distribution strategy if you want, in case each trial runs on a multi-GPU machine. Then the script can load the dataset and train the model. For example:

```
import tensorflow as tf

def build_model(args):
    with tf.distribute.MirroredStrategy().scope():
        model = tf.keras.Sequential()
        model.add(tf.keras.layers.Flatten(input_shape=[28, 28], dtype=tf.uint8))
        for _ in range(args.n_hidden):
            model.add(tf.keras.layers.Dense(args.n_neurons, activation="relu"))
        model.add(tf.keras.layers.Dense(10, activation="softmax"))
        opt = tf.keras.optimizers.get(args.optimizer)
        opt.learning_rate = args.learning_rate
        model.compile(loss="sparse_categorical_crossentropy", optimizer=opt,
                      metrics=["accuracy"])
    return model

[...] # load the dataset
model = build_model(args)
history = model.fit([...])
```



You can use the `AIP_*` environment variables we discussed earlier to know where to save the checkpoints, the TensorBoard logs, and the final model.

Lastly, the script must report the model's performance back to Vertex AI's hyperparameter tuning service, so it can decide which hyperparameters to try next. For this, you must use the `hypertune` library, which is automatically installed on Vertex AI training VMs:

```
import hypertune

hypertune = hypertune.HyperTune()
hypertune.report_hyperparameter_tuning_metric(
    hyperparameter_metric_tag="accuracy", # name of the reported metric
    metric_value=max(history.history["val_accuracy"]),
    global_step=model.optimizer.iterations.numpy(),
)
```

Now that your training script is ready, you need to define the type of machine you would like to run it on. For this, you must define a custom job, which Vertex AI will use as a template for each trial:

```
trial_job = aiplatform.CustomJob.from_local_script(
    display_name="my_search_trial_job",
    script_path="my_vertex_ai_trial.py", # path to your training script
    container_uri="gcr.io/cloud-aiplatform/training/tf-gpu.2-4:latest",
    staging_bucket=f"gs://{bucket_name}/staging",
    accelerator_type="NVIDIA_TESLA_K80",
    accelerator_count=2, # in this example, each trial will have 2 GPUs
)
```

Finally, you're ready to create and run the hyperparameter tuning job:

```
from google.cloud.aiplatform import hyperparameter_tuning as hpt

hp_job = aiplatform.HyperparameterTuningJob(
    display_name="my_hp_search_job",
    custom_job=trial_job,
    metric_spec={"accuracy": "maximize"},
    parameter_spec={
        "learning_rate": hpt.DoubleParameterSpec(min=1e-3, max=10, scale="log"),
        "n_neurons": hpt.IntegerParameterSpec(min=1, max=300, scale="linear"),
        "n_hidden": hpt.IntegerParameterSpec(min=1, max=10, scale="linear"),
        "optimizer": hpt.CategoricalParameterSpec(["sgd", "adam"]),
    },
    max_trial_count=100,
    parallel_trial_count=20,
)
hp_job.run()
```

Here, we tell Vertex AI to maximize the metric named "accuracy": this name must match the name of the metric reported by the training script. We also define the search space, using a log scale for the learning rate, and a linear scale (i.e., uniform) for the other hyperparameters. The hyperparameter names must match the command line arguments of the training script. Lastly, we set the maximum number of trials to 100, and the maximum number of trials running in parallel to 20. If you increase the number of parallel trials to (say) 60, then the total search time will be reduced significantly, by a factor of up to 3. However, the first 60 trials will be started in parallel, so they will not benefit from the other trials' feedback. Therefore, you should increase the max number of trials to compensate, for example up to about 140.

This will take quite a while. Once the job is completed, you can fetch the trial results using `hp_job.trials`. Each trial result is represented as a protobuf object, containing the hyperparameter values and the resulting metrics. Let's find the best trial:

```
def get_final_metric(trial, metric_id):
    for metric in trial.final_measurement.metrics:
        if metric.metric_id == metric_id:
            return metric.value

trials = hp_job.trials
trial_accuracies = [get_final_metric(trial, "accuracy") for trial in trials]
best_trial = trials[np.argmax(trial_accuracies)]
```

Now let's look at this trial's accuracy, and its hyperparameter values:

```
>>> max(trial_accuracies)
0.977400004863739
>>> best_trial.id
'98'
>>> best_trial.parameters
[parameter_id: "learning_rate" value { number_value: 0.001 },
 parameter_id: "n_hidden" value { number_value: 8.0 },
 parameter_id: "n_neurons" value { number_value: 216.0 },
 parameter_id: "optimizer" value { string_value: "adam" }
]
```

That's it! Now you can get this trial's saved model, optionally train it a bit more, and deploy it to production.



Vertex AI also includes an AutoML service which completely takes care of finding the right model architecture and training it for you. All you need to do is upload your dataset to Vertex AI using a special format that depends on the type of dataset (images, text, tabular, video...), then create an AutoML training job, pointing to the dataset and specifying the maximum number of compute hours you're willing to spend. See the notebook for an example.

Hyperparameter Tuning using Keras Tuner on Vertex AI

Instead of using Vertex AI's hyperparameter tuning service, you can use Keras Tuner (introduced in [Chapter 10](#)) and run it on Vertex AI VMs. Keras Tuner provides a simple way to scale hyperparameter search by distributing it across multiple machines: it only requires setting three environment variables on each machine, then running your regular Keras Tuner code on each machine. You can use the exact same script on all machines. One of the machines acts as the chief (i.e., the oracle), and the others act as workers. Each worker asks the chief which hyperparameter values to try, then the worker trains the model using these hyperparameter values, and finally it reports the model's performance back to the chief, which can then decide which hyperparameter values the worker should try next.

The three environment variables you need to set on each machine are:

- `KERASTUNER_TUNER_ID`: equal to "chief" on the chief machine, or a unique identifier on each worker machine, such as "`worker0`", "`worker1`", etc.
- `KERASTUNER_ORACLE_IP`: the IP address or hostname of the chief machine. The chief itself should generally use "`0.0.0.0`" to listen on every IP address on the machine.
- `KERASTUNER_ORACLE_PORT`: the TCP port that the chief will be listening on.

You can use distributed Keras Tuner on any set of machines. If you want to run it on Vertex AI machines, then you can spawn a regular training job, and just modify the training script to set the environment variables properly before using Keras Tuner. See the notebook for an example.

Now you have all the tools and knowledge you need to create state-of-the-art neural net architectures and train them at scale using various distribution strategies, on your own infrastructure or on the cloud, and then deploy them anywhere. In other words, you now have superpowers: use them well!

Exercises

1. What does a `SavedModel` contain? How do you inspect its content?
2. When should you use TF Serving? What are its main features? What are some tools you can use to deploy it?
3. How do you deploy a model across multiple TF Serving instances?
4. When should you use the gRPC API rather than the REST API to query a model served by TF Serving?

5. What are the different ways TFLite reduces a model's size to make it run on a mobile or embedded device?
6. What is quantization-aware training, and why would you need it?
7. What are model parallelism and data parallelism? Why is the latter generally recommended?
8. When training a model across multiple servers, what distribution strategies can you use? How do you choose which one to use?
9. Train a model (any model you like) and deploy it to TF Serving or Google Vertex AI. Write the client code to query it using the REST API or the gRPC API. Update the model and deploy the new version. Your client code will now query the new version. Roll back to the first version.
10. Train any model across multiple GPUs on the same machine using the `Mirrored Strategy` (if you do not have access to GPUs, you can use Colaboratory with a GPU Runtime and create two logical GPUs). Train the model again using the `CentralStorageStrategy` and compare the training time.
11. Fine-tune a model of your choice on Vertex AI, using either Keras Tuner or Vertex AI's hyperparameter tuning service.

Thank You!

Before we close the last chapter of this book, I would like to thank you for reading it up to the last paragraph. I truly hope that you had as much pleasure reading this book as I had writing it, and that it will be useful for your projects, big or small.

If you find errors, please send feedback. More generally, I would love to know what you think, so please don't hesitate to contact me via O'Reilly, through the [ageron/handson-ml3](#) GitHub project, or on Twitter at @aureliengeron.

Going forward, my best advice to you is to practice and practice: try going through all the exercises (if you have not done so already), play with the notebooks, join Kaggle.com or some other ML community, watch ML courses, read papers, attend conferences, and meet experts. Things move fast so try to keep up to date: several YouTube channels regularly present Deep Learning papers in great detail, in a very approachable way. I particularly recommend channels by Yannic Kilcher, Letitia Parcalabescu, and Xander Steenbrugge. For fascinating ML discussions and higher level insights, make sure to check out ML Street Talk, and Lex Fridman's channel. It also helps tremendously to have a concrete project to work on, whether it is for work or for fun (ideally for both), so if there's anything you have always dreamt of building, give it a shot! Work incrementally; don't shoot for the moon right away, but stay focused on your project and build it piece by piece. It will require patience and

perseverance, but when you have a walking robot, or a working chatbot, or whatever else you fancy to build, it will be immensely rewarding.

My greatest hope is that this book will inspire you to build a wonderful ML application that will benefit all of us! What will it be?

—Aurélien Géron, April, 2022

APPENDIX A

Machine Learning Project Checklist

This checklist can guide you through your Machine Learning projects. There are eight main steps:

1. Frame the problem and look at the big picture.
2. Get the data.
3. Explore the data to gain insights.
4. Prepare the data to better expose the underlying data patterns to Machine Learning algorithms.
5. Explore many different models and shortlist the best ones.
6. Fine-tune your models and combine them into a great solution.
7. Present your solution.
8. Launch, monitor, and maintain your system.

Obviously, you should feel free to adapt this checklist to your needs.

Frame the Problem and Look at the Big Picture

1. Define the objective in business terms.
2. How will your solution be used?
3. What are the current solutions/workarounds (if any)?
4. How should you frame this problem (supervised/unsupervised, online/offline, etc.)?
5. How should performance be measured?
6. Is the performance measure aligned with the business objective?

7. What would be the minimum performance needed to reach the business objective?
8. What are comparable problems? Can you reuse experience or tools?
9. Is human expertise available?
10. How would you solve the problem manually?
11. List the assumptions you (or others) have made so far.
12. Verify assumptions if possible.

Get the Data

Note: automate as much as possible so you can easily get fresh data.

1. List the data you need and how much you need.
2. Find and document where you can get that data.
3. Check how much space it will take.
4. Check legal obligations, and get authorization if necessary.
5. Get access authorizations.
6. Create a workspace (with enough storage space).
7. Get the data.
8. Convert the data to a format you can easily manipulate (without changing the data itself).
9. Ensure sensitive information is deleted or protected (e.g., anonymized).
10. Check the size and type of data (time series, sample, geographical, etc.).
11. Sample a test set, put it aside, and never look at it (no data snooping!).

Explore the Data

Note: try to get insights from a field expert for these steps.

1. Create a copy of the data for exploration (sampling it down to a manageable size if necessary).
2. Create a Jupyter notebook to keep a record of your data exploration.
3. Study each attribute and its characteristics:
 - Name
 - Type (categorical, int/float, bounded/unbounded, text, structured, etc.)
 - % of missing values

- Noisiness and type of noise (stochastic, outliers, rounding errors, etc.)
 - Usefulness for the task
 - Type of distribution (Gaussian, uniform, logarithmic, etc.)
4. For supervised learning tasks, identify the target attribute(s).
 5. Visualize the data.
 6. Study the correlations between attributes.
 7. Study how you would solve the problem manually.
 8. Identify the promising transformations you may want to apply.
 9. Identify extra data that would be useful (go back to “[Get the Data](#)” on page 790).
 10. Document what you have learned.

Prepare the Data

Notes:

- Work on copies of the data (keep the original dataset intact).
 - Write functions for all data transformations you apply, for five reasons:
 - So you can easily prepare the data the next time you get a fresh dataset
 - So you can apply these transformations in future projects
 - To clean and prepare the test set
 - To clean and prepare new data instances once your solution is live
 - To make it easy to treat your preparation choices as hyperparameters
1. Data cleaning:
 - Fix or remove outliers (optional).
 - Fill in missing values (e.g., with zero, mean, median...) or drop their rows (or columns).
 2. Feature selection (optional):
 - Drop the attributes that provide no useful information for the task.
 3. Feature engineering, where appropriate:
 - Discretize continuous features.
 - Decompose features (e.g., categorical, date/time, etc.).
 - Add promising transformations of features (e.g., $\log(x)$, \sqrt{x} , x^2 , etc.).
 - Aggregate features into promising new features.

4. Feature scaling:
 - Standardize or normalize features.

Shortlist Promising Models

Notes:

- If the data is huge, you may want to sample smaller training sets so you can train many different models in a reasonable time (be aware that this penalizes complex models such as large neural nets or Random Forests).
 - Once again, try to automate these steps as much as possible.
1. Train many quick-and-dirty models from different categories (e.g., linear, naive Bayes, SVM, Random Forest, neural net, etc.) using standard parameters.
 2. Measure and compare their performance.
 - For each model, use N -fold cross-validation and compute the mean and standard deviation of the performance measure on the N folds.
 3. Analyze the most significant variables for each algorithm.
 4. Analyze the types of errors the models make.
 - What data would a human have used to avoid these errors?
 5. Perform a quick round of feature selection and engineering.
 6. Perform one or two more quick iterations of the five previous steps.
 7. Shortlist the top three to five most promising models, preferring models that make different types of errors.

Fine-Tune the System

Notes:

- You will want to use as much data as possible for this step, especially as you move toward the end of fine-tuning.
 - As always, automate what you can.
1. Fine-tune the hyperparameters using cross-validation:
 - Treat your data transformation choices as hyperparameters, especially when you are not sure about them (e.g., if you're not sure whether to replace missing values with zeros or with the median value, or to just drop the rows).

- Unless there are very few hyperparameter values to explore, prefer random search over grid search. If training is very long, you may prefer a Bayesian optimization approach (e.g., using Gaussian process priors, [as described by Jasper Snoek et al.](#)).¹
- 2. Try Ensemble methods. Combining your best models will often produce better performance than running them individually.
- 3. Once you are confident about your final model, measure its performance on the test set to estimate the generalization error.



Don't tweak your model after measuring the generalization error: you would just start overfitting the test set.

Present Your Solution

1. Document what you have done.
2. Create a nice presentation.
 - Make sure you highlight the big picture first.
3. Explain why your solution achieves the business objective.
4. Don't forget to present interesting points you noticed along the way.
 - Describe what worked and what did not.
 - List your assumptions and your system's limitations.
5. Ensure your key findings are communicated through beautiful visualizations or easy-to-remember statements (e.g., "the median income is the number-one predictor of housing prices").

Launch!

1. Get your solution ready for production (plug into production data inputs, write unit tests, etc.).
2. Write monitoring code to check your system's live performance at regular intervals and trigger alerts when it drops.
 - Beware of slow degradation: models tend to "rot" as data evolves.

¹ Jasper Snoek et al., "Practical Bayesian Optimization of Machine Learning Algorithms," *Proceedings of the 25th International Conference on Neural Information Processing Systems 2* (2012): 2951–2959.

- Measuring performance may require a human pipeline (e.g., via a crowdsourcing service).
 - Also monitor your inputs' quality (e.g., a malfunctioning sensor sending random values, or another team's output becoming stale). This is particularly important for online learning systems.
3. Retrain your models on a regular basis on fresh data (automate as much as possible).

APPENDIX B

Autodiff

This appendix explains how TensorFlow’s autodifferentiation (autodiff) feature works, and how it compares to other solutions.

Suppose you define a function $f(x, y) = x^2y + y + 2$, and you need its partial derivatives $\partial f / \partial x$ and $\partial f / \partial y$, typically to perform Gradient Descent (or some other optimization algorithm). Your main options are manual differentiation, finite difference approximation, forward-mode autodiff, and reverse-mode autodiff. TensorFlow implements reverse-mode autodiff, but to understand it, it’s useful to look at the other options first. So let’s go through each of them, starting with manual differentiation.

Manual Differentiation

The first approach to compute derivatives is to pick up a pencil and a piece of paper and use your calculus knowledge to derive the appropriate equation. For the function $f(x, y)$ just defined, it is not too hard; you just need to use five rules:

- The derivative of a constant is 0.
- The derivative of λx is λ (where λ is a constant).
- The derivative of x^λ is $\lambda x^{\lambda-1}$, so the derivative of x^2 is $2x$.
- The derivative of a sum of functions is the sum of these functions’ derivatives.
- The derivative of λ times a function is λ times its derivative.

From these rules, you can derive [Equation B-1](#).

Equation B-1. Partial derivatives of $f(x, y)$

$$\frac{\partial f}{\partial x} = \frac{\partial(x^2y)}{\partial x} + \frac{\partial y}{\partial x} + \frac{\partial 2}{\partial x} = y \frac{\partial(x^2)}{\partial x} + 0 + 0 = 2xy$$

$$\frac{\partial f}{\partial y} = \frac{\partial(x^2y)}{\partial y} + \frac{\partial y}{\partial y} + \frac{\partial 2}{\partial y} = x^2 + 1 + 0 = x^2 + 1$$

This approach can become very tedious for more complex functions, and you run the risk of making mistakes. Fortunately, there are other options. Let's look at finite difference approximation now.

Finite Difference Approximation

Recall that the derivative $h'(x_0)$ of a function $h(x)$ at a point x_0 is the slope of the function at that point. More precisely, the derivative is defined as the limit of the slope of a straight line going through this point x_0 and another point x on the function, as x gets infinitely close to x_0 (see [Equation B-2](#)).

Equation B-2. Definition of the derivative of a function $h(x)$ at point x_0

$$\begin{aligned} h'(x_0) &= \lim_{x \rightarrow x_0} \frac{h(x) - h(x_0)}{x - x_0} \\ &= \lim_{\varepsilon \rightarrow 0} \frac{h(x_0 + \varepsilon) - h(x_0)}{\varepsilon} \end{aligned}$$

So, if we wanted to calculate the partial derivative of $f(x, y)$ with regard to x at $x = 3$ and $y = 4$, we could compute $f(3 + \varepsilon, 4) - f(3, 4)$ and divide the result by ε , using a very small value for ε . This type of numerical approximation of the derivative is called a *finite difference approximation*, and this specific equation is called *Newton's difference quotient*. That's exactly what the following code does:

```
def f(x, y):
    return x**2*y + y + 2

def derivative(f, x, y, x_eps, y_eps):
    return (f(x + x_eps, y + y_eps) - f(x, y)) / (x_eps + y_eps)

df_dx = derivative(f, 3, 4, 0.00001, 0)
df_dy = derivative(f, 3, 4, 0, 0.00001)
```

Unfortunately, the result is imprecise (and it gets worse for more complicated functions). The correct results are respectively 24 and 10, but instead we get:

```

>>> df_dx
24.000039999805264
>>> df_dy
10.0000000000331966

```

Notice that to compute both partial derivatives, we have to call `f()` at least three times (we called it four times in the preceding code, but it could be optimized). If there were 1,000 parameters, we would need to call `f()` at least 1,001 times. When you are dealing with large neural networks, this makes finite difference approximation way too inefficient.

However, this method is so simple to implement that it is a great tool to check that the other methods are implemented correctly. For example, if it disagrees with your manually derived function, then your function probably contains a mistake.

So far, we have considered two ways to compute gradients: using manual differentiation and using finite difference approximation. Unfortunately, both were fatally flawed to train a large-scale neural network. So let's turn to autodiff, starting with forward mode.

Forward-Mode Autodiff

Figure B-1 shows how forward-mode autodiff works on an even simpler function, $g(x, y) = 5 + xy$. The graph for that function is represented on the left. After forward-mode autodiff, we get the graph on the right, which represents the partial derivative $\frac{\partial g}{\partial x} = 0 + (0 \times x + y \times 1) = y$ (we could similarly obtain the partial derivative with regard to y).

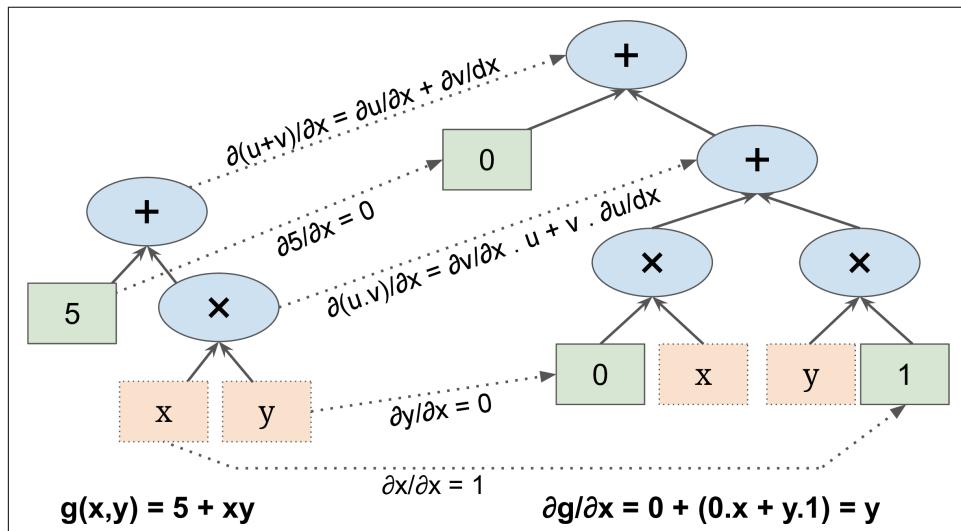


Figure B-1. Forward-mode autodiff

The algorithm will go through the computation graph from the inputs to the outputs (hence the name “forward mode”). It starts by getting the partial derivatives of the leaf nodes. The constant node (5) returns the constant 0, since the derivative of a constant is always 0. The variable x returns the constant 1 since $\partial x / \partial x = 1$, and the variable y returns the constant 0 since $\partial y / \partial x = 0$ (if we were looking for the partial derivative with regard to y , it would be the reverse).

Now we have all we need to move up the graph to the multiplication node in function g . Calculus tells us that the derivative of the product of two functions u and v is $\partial(u \times v) / \partial x = \partial v / \partial x \times u + v \times \partial u / \partial x$. We can therefore construct a large part of the graph on the right, representing $0 \times x + y \times 1$.

Finally, we can go up to the addition node in function g . As mentioned, the derivative of a sum of functions is the sum of these functions’ derivatives. So we just need to create an addition node and connect it to the parts of the graph we have already computed. We get the correct partial derivative: $\partial g / \partial x = 0 + (0 \times x + y \times 1)$.

However, this equation can be simplified (a lot). A few pruning steps can be applied to the computation graph to get rid of all unnecessary operations, and we get a much smaller graph with just one node: $\partial g / \partial x = y$. In this case simplification is fairly easy, but for a more complex function forward-mode autodiff can produce a huge graph that may be tough to simplify and lead to suboptimal performance.

Note that we started with a computation graph, and forward-mode autodiff produced another computation graph. This is called *symbolic differentiation*, and it has two nice features: first, once the computation graph of the derivative has been produced, we can use it as many times as we want to compute the derivatives of the given function for any value of x and y ; second, we can run forward-mode autodiff again on the resulting graph to get second-order derivatives if we ever need to (i.e., derivatives of derivatives). We could even compute third-order derivatives, and so on.

But it is also possible to run forward-mode autodiff without constructing a graph (i.e., numerically, not symbolically), just by computing intermediate results on the fly. One way to do this is to use *dual numbers*, which are weird but fascinating numbers of the form $a + b\epsilon$, where a and b are real numbers and ϵ is an infinitesimal number such that $\epsilon^2 = 0$ (but $\epsilon \neq 0$). You can think of the dual number $42 + 24\epsilon$ as something akin to $42.0000\cdots000024$ with an infinite number of 0s (but of course this is simplified just to give you some idea of what dual numbers are). A dual number is represented in memory as a pair of floats. For example, $42 + 24\epsilon$ is represented by the pair $(42.0, 24.0)$.

Dual numbers can be added, multiplied, and so on, as shown in [Equation B-3](#).

Equation B-3. A few operations with dual numbers

$$\lambda(a + b\epsilon) = \lambda a + \lambda b\epsilon$$

$$(a + b\epsilon) + (c + d\epsilon) = (a + c) + (b + d)\epsilon$$

$$(a + b\epsilon) \times (c + d\epsilon) = ac + (ad + bc)\epsilon + (bd)\epsilon^2 = ac + (ad + bc)\epsilon$$

Most importantly, it can be shown that $h(a + b\epsilon) = h(a) + b \times h'(a)\epsilon$, so computing $h(a + \epsilon)$ gives you both $h(a)$ and the derivative $h'(a)$ in just one shot. [Figure B-2](#) shows that the partial derivative of $f(x, y)$ with regard to x at $x = 3$ and $y = 4$ (which I will write $\partial f / \partial x(3, 4)$) can be computed using dual numbers. All we need to do is compute $f(3 + \epsilon, 4)$; this will output a dual number whose first component is equal to $f(3, 4)$ and whose second component is equal to $\partial f / \partial x(3, 4)$.

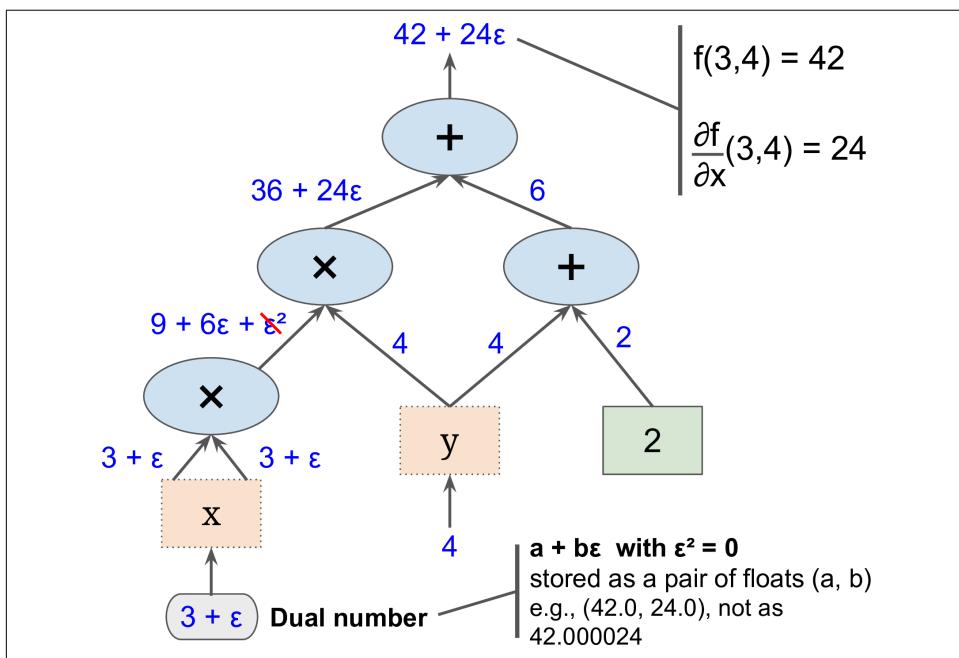


Figure B-2. Forward-mode autodiff using dual numbers

To compute $\partial f / \partial y(3, 4)$ we would have to go through the graph again, but this time with $x = 3$ and $y = 4 + \epsilon$.

So forward-mode autodiff is much more accurate than finite difference approximation, but it suffers from the same major flaw, at least when there are many inputs and few outputs (as is the case when dealing with neural networks): if there were 1,000 parameters, it would require 1,000 passes through the graph to compute all the partial

derivatives. This is where reverse-mode autodiff shines: it can compute all of them in just two passes through the graph. Let's see how.

Reverse-Mode Autodiff

Reverse-mode autodiff is the solution implemented by TensorFlow. It first goes through the graph in the forward direction (i.e., from the inputs to the output) to compute the value of each node. Then it does a second pass, this time in the reverse direction (i.e., from the output to the inputs), to compute all the partial derivatives. The name “reverse mode” comes from this second pass through the graph, where gradients flow in the reverse direction. [Figure B-3](#) represents the second pass. During the first pass, all the node values were computed, starting from $x = 3$ and $y = 4$. You can see those values at the bottom right of each node (e.g., $x \times x = 9$). The nodes are labeled n_1 to n_7 for clarity. The output node is n_7 : $f(3, 4) = n_7 = 42$.

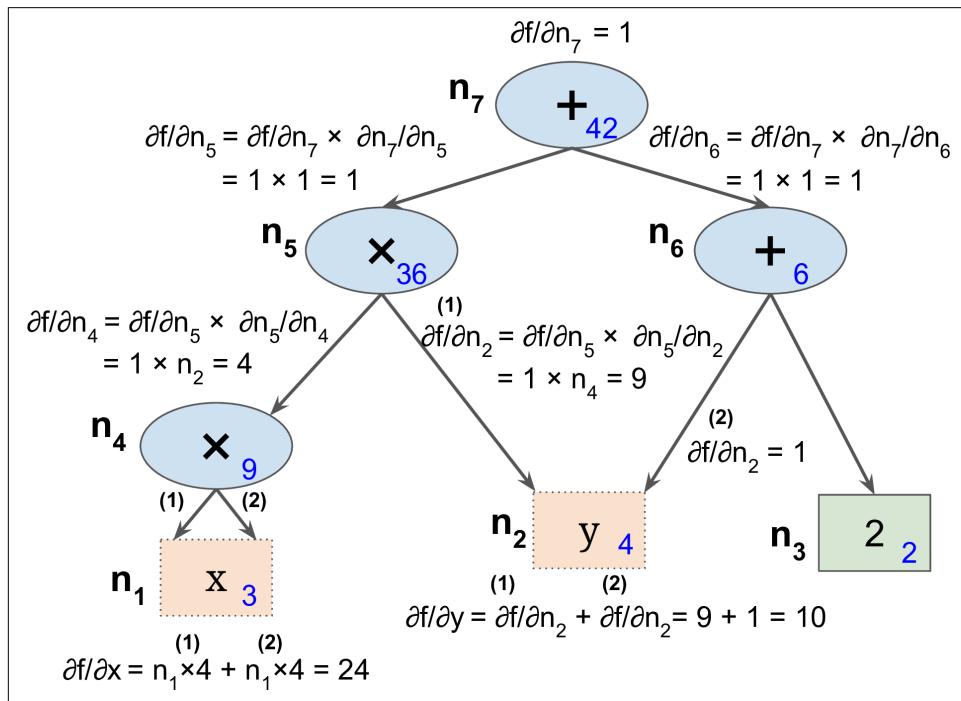


Figure B-3. Reverse-mode autodiff

The idea is to gradually go down the graph, computing the partial derivative of $f(x, y)$ with regard to each consecutive node, until we reach the variable nodes. For this, reverse-mode autodiff relies heavily on the *chain rule*, shown in [Equation B-4](#).

Equation B-4. Chain rule

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial n_i} \times \frac{\partial n_i}{\partial x}$$

Since n_7 is the output node, $f = n_7$, so $\partial f / \partial n_7 = 1$.

Let's continue down the graph to n_5 : how much does f vary when n_5 varies? The answer is $\partial f / \partial n_5 = \partial f / \partial n_7 \times \partial n_7 / \partial n_5$. We already know that $\partial f / \partial n_7 = 1$, so all we need is $\partial n_7 / \partial n_5$. Since n_7 simply performs the sum $n_5 + n_6$, we find that $\partial n_7 / \partial n_5 = 1$, so $\partial f / \partial n_5 = 1 \times 1 = 1$.

Now we can proceed to node n_4 : how much does f vary when n_4 varies? The answer is $\partial f / \partial n_4 = \partial f / \partial n_5 \times \partial n_5 / \partial n_4$. Since $n_5 = n_4 \times n_2$, we find that $\partial n_5 / \partial n_4 = n_2$, so $\partial f / \partial n_4 = 1 \times n_2 = 4$.

The process continues until we reach the bottom of the graph. At that point we will have calculated all the partial derivatives of $f(x, y)$ at the point $x = 3$ and $y = 4$. In this example, we find $\partial f / \partial x = 24$ and $\partial f / \partial y = 10$. Sounds about right!

Reverse-mode autodiff is a very powerful and accurate technique, especially when there are many inputs and few outputs, since it requires only one forward pass plus one reverse pass per output to compute all the partial derivatives for all outputs with regard to all the inputs. When training neural networks, we generally want to minimize the loss, so there is a single output (the loss), and hence only two passes through the graph are needed to compute the gradients. Reverse-mode autodiff can also handle functions that are not entirely differentiable, as long as you ask it to compute the partial derivatives at points that are differentiable.

In [Figure B-3](#), the numerical results are computed on the fly, at each node. However, that's not exactly what TensorFlow does: instead, it creates a new computation graph. In other words, it implements *symbolic* reverse-mode autodiff. This way, the computation graph to compute the gradients of the loss with regard to all the parameters in the neural network only needs to be generated once, and then it can be executed over and over again, whenever the optimizer needs to compute the gradients. Moreover, this makes it possible to compute higher-order derivatives if needed.



If you ever want to implement a new type of low-level TensorFlow operation in C++, and you want to make it compatible with auto-diff, then you will need to provide a function that returns the partial derivatives of the function's outputs with regard to its inputs. For example, suppose you implement a function that computes the square of its input: $f(x) = x^2$. In that case you would need to provide the corresponding derivative function: $f'(x) = 2x$.

APPENDIX C

Special Data Structures

In this appendix we will take a very quick look at the data structures supported by TensorFlow, beyond regular float or integer tensors. This includes strings, ragged tensors, sparse tensors, tensor arrays, sets, and queues.

Strings

Tensors can hold byte strings, which is useful in particular for natural language processing (see [Chapter 16](#)):

```
>>> tf.constant(b"hello world")
<tf.Tensor: shape=(), dtype=string, numpy=b'hello world'>
```

If you try to build a tensor with a Unicode string, TensorFlow automatically encodes it to UTF-8:

```
>>> tf.constant("CAFÉ")
<tf.Tensor: shape=(), dtype=string, numpy=b'CAF\xC3\xA9'>
```

It is also possible to create tensors representing Unicode strings. Just create an array of 32-bit integers, each representing a single Unicode code point:¹

```
>>> u = tf.constant([ord(c) for c in "CAFÉ"])
>>> u
<tf.Tensor: shape=(4,), [...], numpy=array([ 99,  97, 102, 233], dtype=int32)>
```

¹ If you are not familiar with Unicode code points, please check out <https://homl.info/unicode>.



In tensors of type `tf.string`, the string length is not part of the tensor's shape. In other words, strings are considered as atomic values. However, in a Unicode string tensor (i.e., an `int32` tensor), the length of the string *is* part of the tensor's shape.

The `tf.strings` package contains several functions to manipulate string tensors, such as `length()` to count the number of bytes in a byte string (or the number of code points if you set `unit="UTF8_CHAR"`), `unicode_encode()` to convert a Unicode string tensor (i.e., `int32` tensor) to a byte string tensor, and `unicode_decode()` to do the reverse:

```
>>> b = tf.strings.unicode_encode(u, "UTF-8")
>>> b
<tf.Tensor: shape=(), dtype=string, numpy=b'caf\xc3\xaa'>
>>> tf.strings.length(b, unit="UTF8_CHAR")
<tf.Tensor: shape=(), dtype=int32, numpy=4>
>>> tf.strings.unicode_decode(b, "UTF-8")
<tf.Tensor: shape=(4,), [...], numpy=array([ 99,  97, 102, 233], dtype=int32)>
```

You can also manipulate tensors containing multiple strings:

```
>>> p = tf.constant(["Café", "Coffee", "caffè", "咖啡"])
>>> tf.strings.length(p, unit="UTF8_CHAR")
<tf.Tensor: shape=(4,), dtype=int32, numpy=array([4, 6, 5, 2], dtype=int32)>
>>> r = tf.strings.unicode_decode(p, "UTF8")
>>> r
<tf.RaggedTensor [[67, 97, 102, 233], [67, 111, 102, 102, 101, 101], [99, 97,
102, 102, 232], [21654, 21857]]>
```

Notice that the decoded strings are stored in a `RaggedTensor`. What is that?

Ragged Tensors

A ragged tensor is a special kind of tensor that represents a list of arrays of different sizes. More generally, it is a tensor with one or more *ragged dimensions*, meaning dimensions whose slices may have different lengths. In the ragged tensor `r`, the second dimension is a ragged dimension. In all ragged tensors, the first dimension is always a regular dimension (also called a *uniform dimension*).

All the elements of the ragged tensor `r` are regular tensors. For example, let's look at the second element of the ragged tensor:

```
>>> r[1]
<tf.Tensor: [...], numpy=array([ 67, 111, 102, 102, 101, 101], dtype=int32)>
```

The `tf.ragged` package contains several functions to create and manipulate ragged tensors. Let's create a second ragged tensor using `tf.ragged.constant()` and concatenate it with the first ragged tensor, along axis 0:

```
>>> r2 = tf.ragged.constant([[65, 66], [], [67]])
>>> tf.concat([r, r2], axis=0)
<tf.RaggedTensor [[67, 97, 102, 233], [67, 111, 102, 102, 101, 101], [99, 97,
102, 102, 232], [21654, 21857], [65, 66], [], [67]]>
```

The result is not too surprising: the tensors in `r2` were appended after the tensors in `r` along axis 0. But what if we concatenate `r` and another ragged tensor along axis 1?

```
>>> r3 = tf.ragged.constant([[68, 69, 70], [71], [], [72, 73]])
>>> print(tf.concat([r, r3], axis=1))
<tf.RaggedTensor [[67, 97, 102, 233, 68, 69, 70], [67, 111, 102, 102, 101, 101,
71], [99, 97, 102, 102, 232], [21654, 21857, 72, 73]]>
```

This time, notice that the i^{th} tensor in `r` and the i^{th} tensor in `r3` were concatenated. Now that's more unusual, since all of these tensors can have different lengths.

If you call the `to_tensor()` method, the ragged tensor gets converted to a regular tensor, padding shorter tensors with zeros to get tensors of equal lengths (you can change the default value by setting the `default_value` argument):

```
>>> r.to_tensor()
<tf.Tensor: shape=(4, 6), dtype=int32, numpy=
array([[ 67,    97,   102,   233,     0,     0],
       [ 67,   111,   102,   102,   101,   101],
       [ 99,    97,   102,   102,   232,     0],
       [21654, 21857,      0,      0,      0,     0]], dtype=int32)>
```

Many TF operations support ragged tensors. For the full list, see the documentation of the `tf.RaggedTensor` class.

Sparse Tensors

TensorFlow can also efficiently represent *sparse tensors* (i.e., tensors containing mostly zeros). Just create a `tf.SparseTensor`, specifying the indices and values of the nonzero elements and the tensor's shape. The indices must be listed in “reading order” (from left to right, and top to bottom). If you are unsure, just use `tf.sparse.reorder()`. You can convert a sparse tensor to a dense tensor (i.e., a regular tensor) using `tf.sparse.to_dense()`:

```
>>> s = tf.SparseTensor(indices=[[0, 1], [1, 0], [2, 3]],
...                         values=[1., 2., 3.],
...                         dense_shape=[3, 4])
...
>>> tf.sparse.to_dense(s)
<tf.Tensor: shape=(3, 4), dtype=float32, numpy=
array([[0., 1., 0., 0.],
       [2., 0., 0., 0.],
       [0., 0., 0., 3.]], dtype=float32)>
```

Note that sparse tensors do not support as many operations as dense tensors. For example, you can multiply a sparse tensor by any scalar value, and you get a new sparse tensor, but you cannot add a scalar value to a sparse tensor, as this would not return a sparse tensor:

```
>>> s * 42.0
<tensorflow.python.framework.sparse_tensor.SparseTensor at 0x7f84a6749f10>
>>> s + 42.0
[...] TypeError: unsupported operand type(s) for +: 'SparseTensor' and 'float'
```

Tensor Arrays

A `tf.TensorArray` represents a list of tensors. This can be handy in dynamic models containing loops, to accumulate results and later compute some statistics. You can read or write tensors at any location in the array:

```
array = tf.TensorArray(dtype=tf.float32, size=3)
array = array.write(0, tf.constant([1., 2.]))
array = array.write(1, tf.constant([3., 10.]))
array = array.write(2, tf.constant([5., 7.]))
tensor1 = array.read(1) # => returns (and zeros out!) tf.constant([3., 10.])
```

By default, reading an item also replaces it with a tensor of the same shape but full of zeros. You can set `clear_after_read` to `False` if you don't want this.



When you write to the array, you must assign the output back to the array, as shown in this code example. If you don't, although your code will work fine in eager mode, it will break in graph mode (these modes are discussed in [Chapter 12](#)).

By default, a `TensorArray` has a fixed size which is set upon creation. Alternatively, you can set `size=0` and `dynamic_size=True` to let the array grow automatically when needed. However, this will hinder performance, so if you know the `size` in advance, it's better to use a fixed size array. You must also specify the `dtype`, and all elements must have the same shape as the first one written to the array.

You can stack all the items into a regular tensor by calling the `stack()` method:

```
>>> array.stack()
<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[1., 2.],
       [0., 0.],
       [5., 7.]], dtype=float32)>
```

Sets

TensorFlow supports sets of integers or strings (but not floats). It represents sets using regular tensors. For example, the set {1, 5, 9} is just represented as the tensor [[1, 5, 9]]. Note that the tensor must have at least two dimensions, and the sets must be in the last dimension. For example, [[1, 5, 9], [2, 5, 11]] is a tensor holding two independent sets: {1, 5, 9} and {2, 5, 11}.

The `tf.sets` package contains several functions to manipulate sets. For example, let's create two sets and compute their union (the result is a sparse tensor, so we call `to_dense()` to display it):

```
>>> a = tf.constant([[1, 5, 9]])
>>> b = tf.constant([[5, 6, 9, 11]])
>>> u = tf.sets.union(a, b)
>>> u
<tensorflow.python.framework.sparse_tensor.SparseTensor at 0x132b60d30>
>>> tf.sparse.to_dense(u)
<tf.Tensor: [...], numpy=array([[ 1,  5,  6,  9, 11]], dtype=int32)>
```

You can also compute the union of multiple pairs of sets simultaneously. If some sets are shorter than others, you must pad them with a padding value, such as 0:

```
>>> a = tf.constant([[1, 5, 9], [10, 0, 0]])
>>> b = tf.constant([[5, 6, 9, 11], [13, 0, 0, 0]])
>>> u = tf.sets.union(a, b)
>>> tf.sparse.to_dense(u)
<tf.Tensor: [...] numpy=array([[ 1,  5,  6,  9, 11],
   [ 0, 10, 13,  0,  0]], dtype=int32)>
```

If you prefer to use a different padding value, such as -1, then you must set `default_value=-1` when calling `to_dense()`.



The default `default_value` is 0, so when dealing with string sets, you must set the `default_value` (e.g., to an empty string).

Other functions available in `tf.sets` include `difference()`, `intersection()`, and `size()`, which are self-explanatory. If you want to check whether or not a set contains some given values, you can compute the intersection of that set and the values. If you want to add some values to a set, you can compute the union of the set and the values.

Queues

A queue is a data structure to which you can push data records, and later pull them out. TensorFlow implements several types of queues in the `tf.queue` package.

They used to be very important when implementing efficient data loading and pre-processing pipelines, but the tf.data API has essentially rendered them useless (except perhaps in some rare cases) because it is much simpler to use and provides all the tools you need to build efficient pipelines. For the sake of completeness, though, let's take a quick look at them.

The simplest kind of queue is the first-in, first-out (FIFO) queue. To build it, you need to specify the maximum number of records it can contain. Moreover, each record is a tuple of tensors, so you must specify the type of each tensor, and optionally their shapes. For example, the following code example creates a FIFO queue with maximum three records, each containing a tuple with a 32-bit integer and a string. Then it pushes two records to it, looks at the size (which is 2 at this point), and pulls a record out:

```
>>> q = tf.queue.FIFOQueue(3, [tf.int32, tf.string], shapes=[(), ()])
>>> q.enqueue([10, b"windy"])
>>> q.enqueue([15, b"sunny"])
>>> q.size()
<tf.Tensor: shape=(), dtype=int32, numpy=2>
>>> q.dequeue()
[<tf.Tensor: shape=(), dtype=int32, numpy=10>,
 <tf.Tensor: shape=(), dtype=string, numpy=b'windy'>]
```

It is also possible to enqueue and dequeue multiple records at once using `enqueue_many()` and `dequeue_many()` (to use `dequeue_many()`, you must specify the `shapes` argument when you create the queue, as we did above):

```
>>> q.enqueue_many([[13, 16], [b'cloudy', b'rainy']])
>>> q.dequeue_many(3)
[<tf.Tensor: [...], numpy=array([15, 13, 16], dtype=int32)>,
 <tf.Tensor: [...], numpy=array([b'sunny', b'cloudy', b'rainy'], dtype=object)>]
```

Other queue types include:

PaddingFIFOQueue

Same as `FIFOQueue`, but its `dequeue_many()` method supports dequeuing multiple records of different shapes. It automatically pads the shortest records to ensure all the records in the batch have the same shape.

PriorityQueue

A queue that dequeues records in a prioritized order. The priority must be a 64-bit integer included as the first element of each record. Surprisingly, records with a lower priority will be dequeued first. Records with the same priority will be dequeued in FIFO order.

RandomShuffleQueue

A queue whose records are dequeued in random order. This was useful to implement a shuffle buffer before tf.data existed.

If a queue is already full and you try to enqueue another record, the `enqueue*()` method will freeze until a record is dequeued by another thread. Similarly, if a queue is empty and you try to dequeue a record, the `dequeue*()` method will freeze until records are pushed to the queue by another thread.

TensorFlow Graphs

In this appendix, we will explore the graphs generated by TF Functions (see [Chapter 12](#)).

TF Functions and Concrete Functions

TF Functions are polymorphic, meaning they support inputs of different types (and shapes). For example, consider the following `tf_cube()` function:

```
@tf.function
def tf_cube(x):
    return x ** 3
```

Every time you call a TF Function with a new combination of input types or shapes, it generates a new *concrete function*, with its own graph specialized for this particular combination. Such a combination of argument types and shapes is called an *input signature*. If you call the TF Function with an input signature it has already seen before, it will reuse the concrete function it generated earlier. For example, if you call `tf_cube(tf.constant(3.0))`, the TF Function will reuse the same concrete function it used for `tf_cube(tf.constant(2.0))` (for float32 scalar tensors). But it will generate a new concrete function if you call `tf_cube(tf.constant([2.0]))` or `tf_cube(tf.constant([3.0]))` (for float32 tensors of shape [1]), and yet another for `tf_cube(tf.constant([[1.0, 2.0], [3.0, 4.0]]))` (for float32 tensors of shape [2, 2]). You can get the concrete function for a particular combination of inputs by calling the TF Function's `get_concrete_function()` method. It can then be called like a regular function, but it will only support one input signature (in this example, float32 scalar tensors):

```
>>> concrete_function = tf_cube.get_concrete_function(tf.constant(2.0))
>>> concrete_function
<ConcreteFunction tf_cube(x) at 0x7F84411F4250>
```

```
>>> concrete_function(tf.constant(2.0))
<tf.Tensor: shape=(), dtype=float32, numpy=8.0>
```

Figure D-1 shows the `tf_cube()` TF Function, after we called `tf_cube(2)` and `tf_cube(tf.constant(2.0))`: two concrete functions were generated, one for each signature, each with its own optimized *function graph* (`FuncGraph`), and its own *function definition* (`FunctionDef`). A function definition points to the parts of the graph that correspond to the function's inputs and outputs. In each `FuncGraph`, the nodes (ovals) represent operations (e.g., power, constants, or placeholders for arguments like `x`), while the edges (the solid arrows between the operations) represent the tensors that will flow through the graph. The concrete function on the left is specialized for `x = 2`, so TensorFlow managed to simplify it to just output 8 all the time (note that the function definition does not even have an input). The concrete function on the right is specialized for float32 scalar tensors, and it could not be simplified. If we call `tf_cube(tf.constant(5.0))`, the second concrete function will be called, the placeholder operation for `x` will output 5.0, then the power operation will compute $5.0^{**} 3$, so the output will be 125.0.

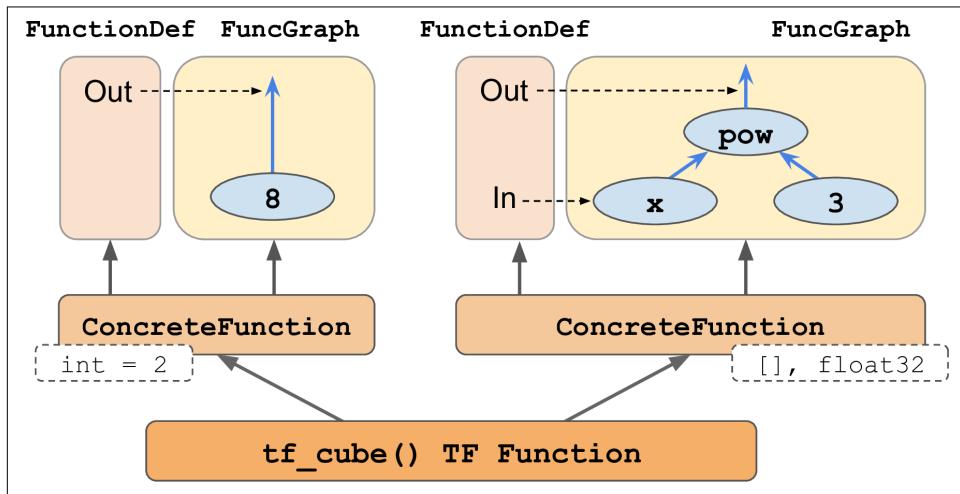


Figure D-1. The `tf_cube()` TF Function, with its `ConcreteFunctions` and their `FunctionGraphs`

The tensors in these graphs are *symbolic tensors*, meaning they don't have an actual value, just a data type, a shape, and a name. They represent the future tensors that will flow through the graph once an actual value is fed to the placeholder `x` and the graph is executed. Symbolic tensors make it possible to specify ahead of time how to connect operations, and they also allow TensorFlow to recursively infer the data types and shapes of all tensors, given the data types and shapes of their inputs.

Now let's continue to peek under the hood, and see how to access function definitions and function graphs and how to explore a graph's operations and tensors.

Exploring Function Definitions and Graphs

You can access a concrete function's computation graph using the `graph` attribute, and get the list of its operations by calling the graph's `get_operations()` method:

```
>>> concrete_function.graph
<tensorflow.python.framework.func_graph.FuncGraph at 0x7f84411f4790>
>>> ops = concrete_function.graph.get_operations()
>>> ops
[<tf.Operation 'x' type=Placeholder>,
 <tf.Operation 'pow/y' type=Const>,
 <tf.Operation 'pow' type=Pow>,
 <tf.Operation 'Identity' type=Identity>]
```

In this example, the first operation represents the input argument `x` (it is called a *placeholder*), the second “operation” represents the constant 3, the third operation represents the power operation (`**`), and the final operation represents the output of this function (it is an identity operation, meaning it will do nothing more than copy the output of the power operation¹). Each operation has a list of input and output tensors that you can easily access using the operation's `inputs` and `outputs` attributes. For example, let's get the list of inputs and outputs of the power operation:

```
>>> pow_op = ops[2]
>>> list(pow_op.inputs)
[<tf.Tensor 'x:0' shape=() dtype=float32>,
 <tf.Tensor 'pow/y:0' shape=() dtype=float32>]
>>> pow_op.outputs
[<tf.Tensor 'pow:0' shape=() dtype=float32>]
```

This computation graph is represented in [Figure D-2](#).

¹ You can safely ignore it—it is only here for technical reasons, to ensure that TF Functions don't leak internal structures.

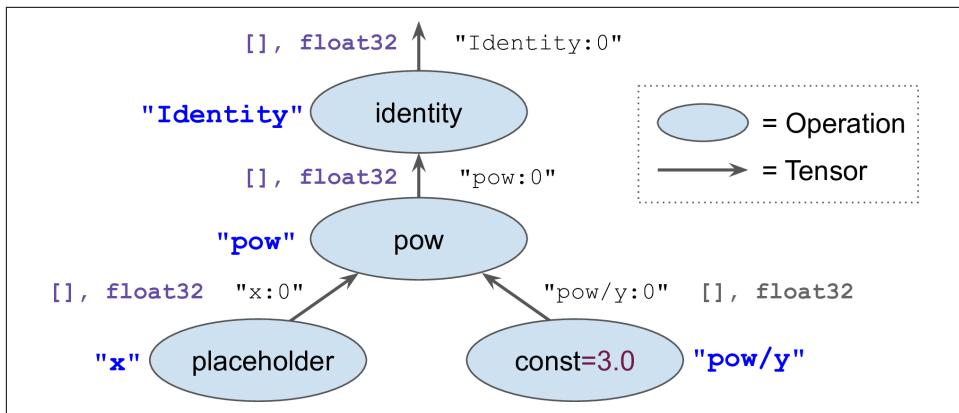


Figure D-2. Example of a computation graph

Note that each operation has a name. It defaults to the name of the operation (e.g., “`pow`”), but you can define it manually when calling the operation (e.g., `tf.pow(x, 3, name="other_name")`). If a name already exists, TensorFlow automatically adds a unique index (e.g., “`pow_1`”, “`pow_2`”, etc.). Each tensor also has a unique name: it is always the name of the operation that outputs this tensor, plus `:0` if it is the operation’s first output, or `:1` if it is the second output, and so on. You can fetch an operation or a tensor by name using the graph’s `get_operation_by_name()` or `get_tensor_by_name()` methods:

```

>>> concrete_function.graph.get_operation_by_name('x')
<tf.Operation 'x' type=Placeholder>
>>> concrete_function.graph.get_tensor_by_name('Identity:0')
<tf.Tensor 'Identity:0' shape=() dtype=float32>

```

The concrete function also contains the function definition (represented as a protocol buffer²), which includes the function’s signature. This signature allows the concrete function to know which placeholders to feed with the input values, and which tensors to return:

```

>>> concrete_function.function_def.signature
name: "__inference_tf_cube_3515903"
input_arg {
    name: "x"
    type: DT_FLOAT
}
output_arg {
    name: "identity"
    type: DT_FLOAT
}

```

² A popular binary format discussed in Chapter 13.

Now let's look more closely at tracing.

A Closer Look at Tracing

Let's tweak the `tf_cube()` function to print its input:

```
@tf.function
def tf_cube(x):
    print(f"x = {x}")
    return x ** 3
```

Now let's call it:

```
>>> result = tf_cube(tf.constant(2.0))
x = Tensor("x:0", shape=(), dtype=float32)
>>> result
<tf.Tensor: shape=(), dtype=float32, numpy=8.0>
```

The `result` looks good, but look at what was printed: `x` is a symbolic tensor! It has a shape and a data type, but no value. Plus it has a name ("`x:0`"). This is because the `print()` function is not a TensorFlow operation, so it will only run when the Python function is traced, which happens in graph mode, with arguments replaced with symbolic tensors (same type and shape, but no value). Since the `print()` function was not captured into the graph, the next times we call `tf_cube()` with float32 scalar tensors, nothing is printed:

```
>>> result = tf_cube(tf.constant(3.0))
>>> result = tf_cube(tf.constant(4.0))
```

But if we call `tf_cube()` with a tensor of a different type or shape, or with a new Python value, the function will be traced again, so the `print()` function will be called:

```
>>> result = tf_cube(2) # new Python value: trace!
x = 2
>>> result = tf_cube(3) # new Python value: trace!
x = 3
>>> result = tf_cube(tf.constant([[1., 2.]])) # New shape: trace!
x = Tensor("x:0", shape=(1, 2), dtype=float32)
>>> result = tf_cube(tf.constant([[3., 4.], [5., 6.]])) # New shape: trace!
x = Tensor("x:0", shape=(None, 2), dtype=float32)
>>> result = tf_cube(tf.constant([[7., 8.], [9., 10.]])) # Same shape: no trace
```



If your function has Python side effects (e.g., it saves some logs to disk), be aware that this code will only run when the function is traced (i.e., every time the TF Function is called with a new input signature). It best to assume that the function may be traced (or not) any time the TF Function is called.

In some cases, you may want to restrict a TF Function to a specific input signature. For example, suppose you know that you will only ever call a TF Function with batches of 28×28 -pixel images, but the batches will have very different sizes. You may not want TensorFlow to generate a different concrete function for each batch size, or count on it to figure out on its own when to use `None`. In this case, you can specify the input signature like this:

```
@tf.function(input_signature=[tf.TensorSpec([None, 28, 28], tf.float32)])
def shrink(images):
    return images[:, ::2, ::2] # drop half the rows and columns
```

This TF Function will accept any float32 tensor of shape $[*, 28, 28]$, and it will reuse the same concrete function every time:

```
img_batch_1 = tf.random.uniform(shape=[100, 28, 28])
img_batch_2 = tf.random.uniform(shape=[50, 28, 28])
preprocessed_images = shrink(img_batch_1) # Works fine, traces the function
preprocessed_images = shrink(img_batch_2) # Works fine, same concrete function
```

However, if you try to call this TF Function with a Python value, or a tensor of an unexpected data type or shape, you will get an exception:

```
img_batch_3 = tf.random.uniform(shape=[2, 2, 2])
preprocessed_images = shrink(img_batch_3) # ValueError! Incompatible inputs
```

Using AutoGraph to Capture Control Flow

If your function contains a simple `for` loop, what do you expect will happen? For example, let's write a function that will add 10 to its input, by just adding 1 10 times:

```
@tf.function
def add_10(x):
    for i in range(10):
        x += 1
    return x
```

It works fine, but when we look at its graph, we find that it does not contain a loop: it just contains 10 addition operations!

```
>>> add_10(tf.constant(0))
<tf.Tensor: shape=(), dtype=int32, numpy=15>
>>> add_10.get_concrete_function(tf.constant(0)).graph.get_operations()
[<tf.Operation 'x' type=Placeholder>, [...],
 <tf.Operation 'add' type=AddV2>, [...],
 <tf.Operation 'add_1' type=AddV2>, [...],
 <tf.Operation 'add_2' type=AddV2>, [...],
 [...]
 <tf.Operation 'add_9' type=AddV2>, [...],
 <tf.Operation 'Identity' type=Identity>]
```

This actually makes sense: when the function got traced, the loop ran 10 times, so the `x += 1` operation was run 10 times, and since it was in graph mode, it recorded this operation 10 times in the graph. You can think of this `for` loop as a “static” loop that gets unrolled when the graph is created.

If you want the graph to contain a “dynamic” loop instead (i.e., one that runs when the graph is executed), you can create one manually using the `tf.while_loop()` operation, but it is not very intuitive (see the “Using AutoGraph to Capture Control Flow” section of the Chapter 12 notebook for an example). Instead, it is much simpler to use TensorFlow’s *AutoGraph* feature, discussed in [Chapter 12](#). AutoGraph is actually activated by default (if you ever need to turn it off, you can pass `autograph=False` to `tf.function()`). So if it is on, why didn’t it capture the `for` loop in the `add_10()` function? Well, it only captures `for` loops that iterate over tensors of `tf.data.Dataset` objects. So you should use `tf.range()`, not `range()`. This is to give you the choice:

- If you use `range()`, the `for` loop will be static, meaning it will only be executed when the function is traced. The loop will be “unrolled” into a set of operations for each iteration, as we saw.
- If you use `tf.range()`, the loop will be dynamic, meaning that it will be included in the graph itself (but it will not run during tracing).

Let’s look at the graph that gets generated if you just replace `range()` with `tf.range()` in the `add_10()` function:

```
>>> add_10.get_concrete_function(tf.constant(0)).graph.get_operations()  
[<tf.Operation 'x' type=Placeholder>, [...],  
 <tf.Operation 'while' type=StatelessWhile>, [...]]
```

As you can see, the graph now contains a `While` loop operation, as if you had called the `tf.while_loop()` function.

Handling Variables and Other Resources in TF Functions

In TensorFlow, variables and other stateful objects, such as queues or datasets, are called *resources*. TF Functions treat them with special care: any operation that reads or updates a resource is considered stateful, and TF Functions ensure that stateful operations are executed in the order they appear (as opposed to stateless operations, which may be run in parallel, so their order of execution is not guaranteed). Moreover, when you pass a resource as an argument to a TF Function, it gets passed by reference, so the function may modify it. For example:

```
counter = tf.Variable(0)  
  
@tf.function
```

```

def increment(counter, c=1):
    return counter.assign_add(c)

increment(counter) # counter is now equal to 1
increment(counter) # counter is now equal to 2

```

If you peek at the function definition, the first argument is marked as a resource:

```

>>> function_def = increment.get_concrete_function(counter).function_def
>>> function_def.signature.input_arg[0]
name: "counter"
type: DT_RESOURCE

```

It is also possible to use a `tf.Variable` defined outside of the function, without explicitly passing it as an argument:

```

counter = tf.Variable(0)

@tf.function
def increment(c=1):
    return counter.assign_add(c)

```

The TF Function will treat this as an implicit first argument, so it will actually end up with the same signature (except for the name of the argument). However, using global variables can quickly become messy, so you should generally wrap variables (and other resources) inside classes. The good news is `@tf.function` works fine with methods too:

```

class Counter:
    def __init__(self):
        self.counter = tf.Variable(0)

    @tf.function
    def increment(self, c=1):
        return self.counter.assign_add(c)

```



Do not use `=`, `+=`, `-=`, or any other Python assignment operator with TF variables. Instead, you must use the `assign()`, `assign_add()`, or `assign_sub()` methods. If you try to use a Python assignment operator, you will get an exception when you call the method.

A good example of this object-oriented approach is, of course, Keras. Let's see how to use TF Functions with Keras.

Using TF Functions with Keras (or Not)

By default, any custom function, layer, or model you use with Keras will automatically be converted to a TF Function; you do not need to do anything at all! However, in some cases you may want to deactivate this automatic conversion—for example, if

your custom code cannot be turned into a TF Function, or if you just want to debug your code (which is much easier in eager mode). To do this, you can simply pass `dynamic=True` when creating the model or any of its layers:

```
model = MyModel(dynamic=True)
```

If your custom model or layer will always be dynamic, you can instead call the base class's constructor with `dynamic=True`:

```
class MyDense(tf.keras.layers.Layer):
    def __init__(self, units, **kwargs):
        super().__init__(dynamic=True, **kwargs)
        [...]
```

Alternatively, you can pass `run_eagerly=True` when calling the `compile()` method:

```
model.compile(loss=my_mse, optimizer="adam", metrics=[my_mae],
               run_eagerly=True)
```

Now you know how TF Functions handle polymorphism (with multiple concrete functions), how graphs are automatically generated using AutoGraph and tracing, what graphs look like, how to explore their symbolic operations and tensors, how to handle variables and resources, and how to use TF Functions with Keras.

Index

About the Author

Aurélien Géron is a Machine Learning consultant and lecturer. A former Googler, he led YouTube's video classification team from 2013 to 2016. He's been a founder of and CTO at a few different companies: Wifirst, a leading wireless ISP in France; Polyconseil, a consulting firm focused on telecoms, media, and strategy; and Kiwisoft, a consulting firm focused on Machine Learning and data privacy.

Before all that he worked as an engineer in a variety of domains: finance (JP Morgan and Société Générale), defense (Canada's DOD), and healthcare (blood transfusion). He also published a few technical books (on C++, WiFi, and internet architectures) and lectured about computer science at a French engineering school.

A few fun facts: he taught his three children to count in binary with their fingers (up to 1,023), he studied microbiology and evolutionary genetics before going into software engineering, and his parachute didn't open on the second jump.

Colophon

The animal on the cover of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow* is the fire salamander (*Salamandra salamandra*), an amphibian found across most of Europe. Its black, glossy skin features large yellow spots on the head and back, signaling the presence of alkaloid toxins. This is a possible source of this amphibian's common name: contact with these toxins (which they can also spray short distances) causes convulsions and hyperventilation. Either the painful poisons or the moistness of the salamander's skin (or both) led to a misguided belief that these creatures not only could survive being placed in fire but could extinguish it as well.

Fire salamanders live in shaded forests, hiding in moist crevices and under logs near the pools or other freshwater bodies that facilitate their breeding. Though they spend most of their lives on land, they give birth to their young in water. They subsist mostly on a diet of insects, spiders, slugs, and worms. Fire salamanders can grow up to a foot in length, and in captivity may live as long as 50 years.

The fire salamander's numbers have been reduced by destruction of their forest habitat and capture for the pet trade, but the greatest threat they face is the susceptibility of their moisture-permeable skin to pollutants and microbes. Since 2014, they have become extinct in parts of the Netherlands and Belgium due to an introduced fungus.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. The cover illustration is by Karen Montgomery, based on an engraving from *Wood's Illustrated Natural History*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.