

III B.Tech I Semester

23 A39501 Applied

23A32501T	INTRODUCTION TO MACHINE LEARNING (Professional Core)	L	T	P	C
		3	0	0	3

COURSE OBJECTIVES:

- To introduce the fundamental concepts of machine learning algorithms and their practical relevance.
- To explore supervised, unsupervised, semi-supervised, and reinforcement learning techniques.
- To equip students with algorithmic and mathematical intuition behind ML models.
- To train students in using tools and frameworks for implementing ML models.
- To enable application of ML algorithms to solve real-world problems in various domains.

COURSE OUTCOMES:

After successful completion of the course, students will be able to:

1. Understand the concepts and types of machine learning algorithms.
2. Design and implement suitable machine learning algorithms for classification, regression, and clustering.
3. Evaluate model performance and perform model tuning.
4. Apply ML algorithms in real-life domains such as NLP, finance, and healthcare.
5. Utilize tools like Scikit-learn, TensorFlow, and PyTorch for practical ML solutions.

UNIT I: Fundamentals of Machine Learning

Introduction to Machine Learning: Definitions, Scope, and Applications, Types of Machine Learning: Supervised, Unsupervised, Semi-Supervised, Reinforcement, Key Concepts: Features, Labels, Training vs Testing Data, Data Splitting, Data Preprocessing: Cleaning, Feature Engineering, Feature Scaling, Over fitting, Under fitting, and Bias-Variance Trade-off, ML Pipeline & Cross-validation Techniques.

UNIT II: Supervised Learning Algorithms

Linear Regression and Ridge/Lasso Regularization, Logistic Regression and its Application in Classification, Decision Trees and Random Forests, Support Vector Machines (SVMs), k-Nearest Neighbors (k-NN), Model Evaluation: Confusion Matrix, Precision, Recall, F1, AUC.

UNIT III: Unsupervised and Ensemble Methods

Clustering Algorithms: k-Means, DBSCAN, Agglomerative, Dimensionality Reduction: PCA, t-SNE, Association Rule Mining: Apriori and FP-Growth, Ensemble Techniques: Bagging, Boosting, and Stacking, Gradient Boosting, AdaBoost, and XGBoost, Evaluation of Unsupervised Models and Visualization.

UNIT IV: Advanced Learning Paradigms

Reinforcement Learning: Markov Decision Process, Q-Learning, Semi-Supervised Learning Techniques, Introduction to Deep Learning: Perceptrons, MLP, Feature Selection & Feature Importance Methods, Model Tuning: Hyperparameter Optimization using Grid Search/Random Search, Explainable AI (XAI) and Model Interpretability Tools.

UNIT V: Applications of Machine Learning

Machine Learning in Healthcare and Diagnostics, Financial Modeling and Fraud Detection, NLP Applications: Text Classification and Sentiment Analysis, Recommender Systems: Collaborative & Content-Based Filtering, Image Classification using Pre-trained Models, Deployment of ML Models using Flask, Streamlit.

TEXTBOOKS:

1. Aurélien Géron, Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, O'Reilly.
2. Ethem Alpaydin, Introduction to Machine Learning, MIT Press.
3. Tom Mitchell, Machine Learning, McGraw-Hill Education.

REFERENCE BOOKS:

1. Kevin Murphy, Machine Learning: A Probabilistic Perspective, MIT Press.
2. Trevor Hastie, Robert Tibshirani, Jerome Friedman, The Elements of Statistical Learning, Springer.
3. Peter Flach, Machine Learning: The Art and Science of Algorithms, Cambridge University Press.
4. François Chollet, Deep Learning with Python, Manning Publications.

Chapter 1. The Machine Learning Landscape

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!

TIP

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 (nospam, 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. This 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.

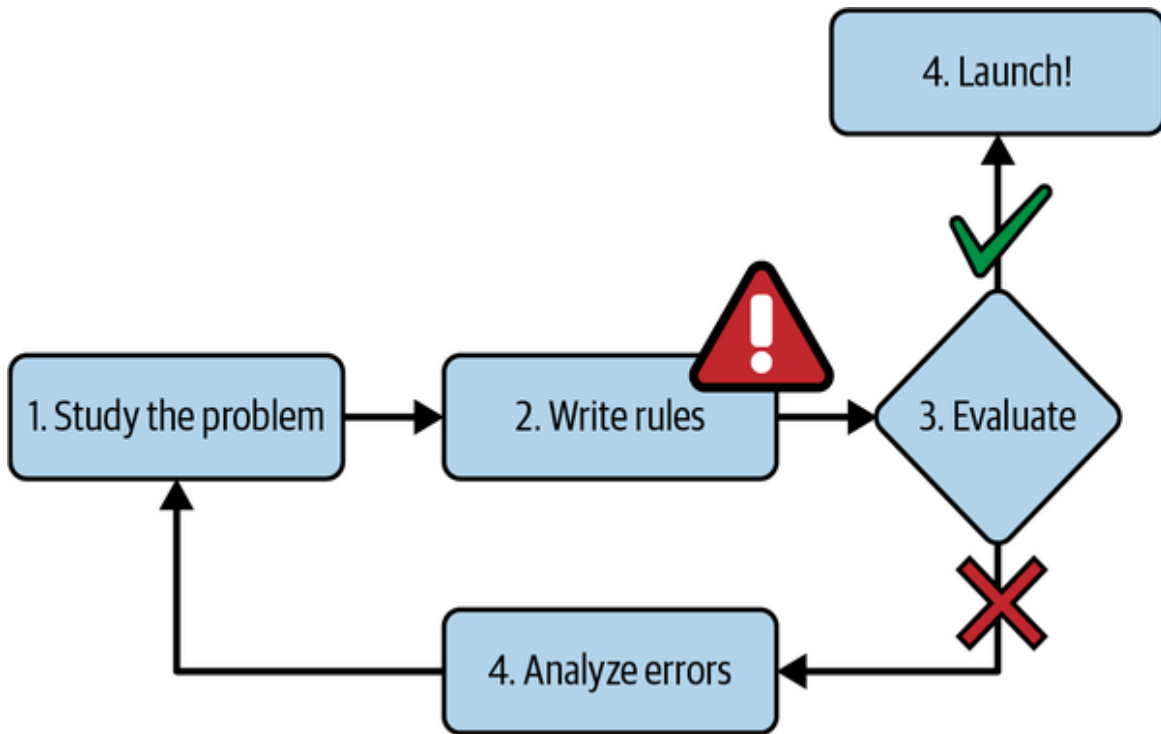


Figure 1-1. The traditional approach

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.

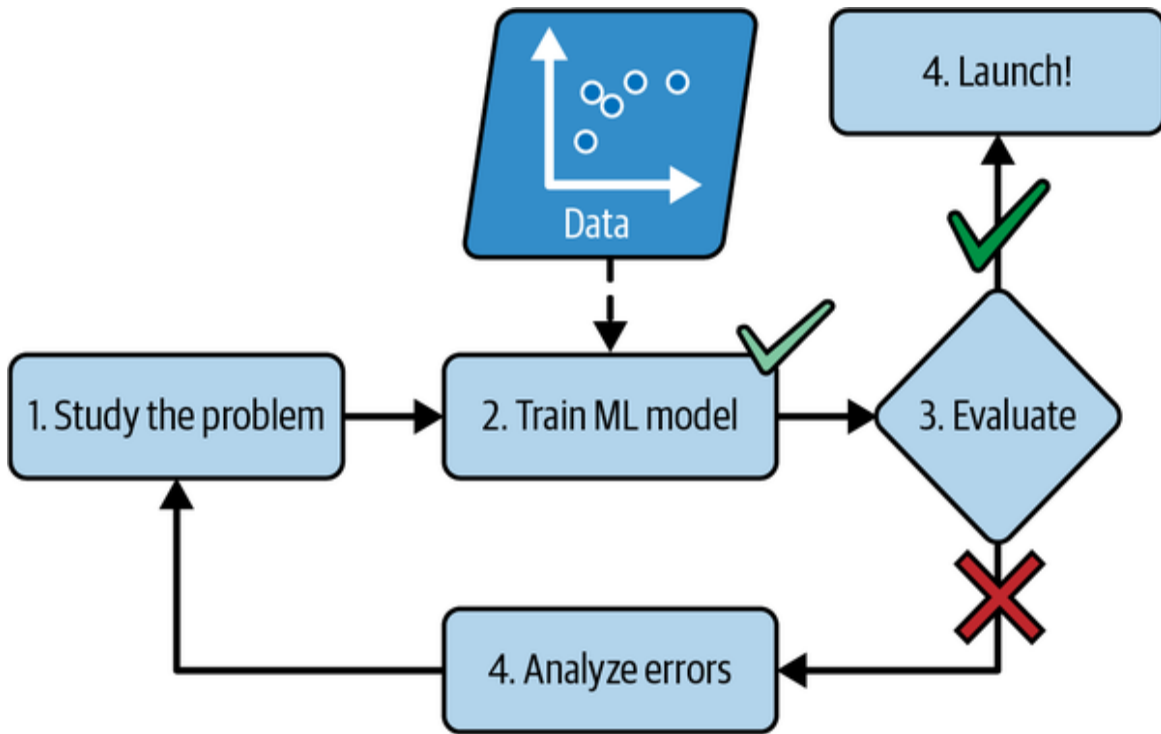


Figure 1-2. The machine learning approach

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).

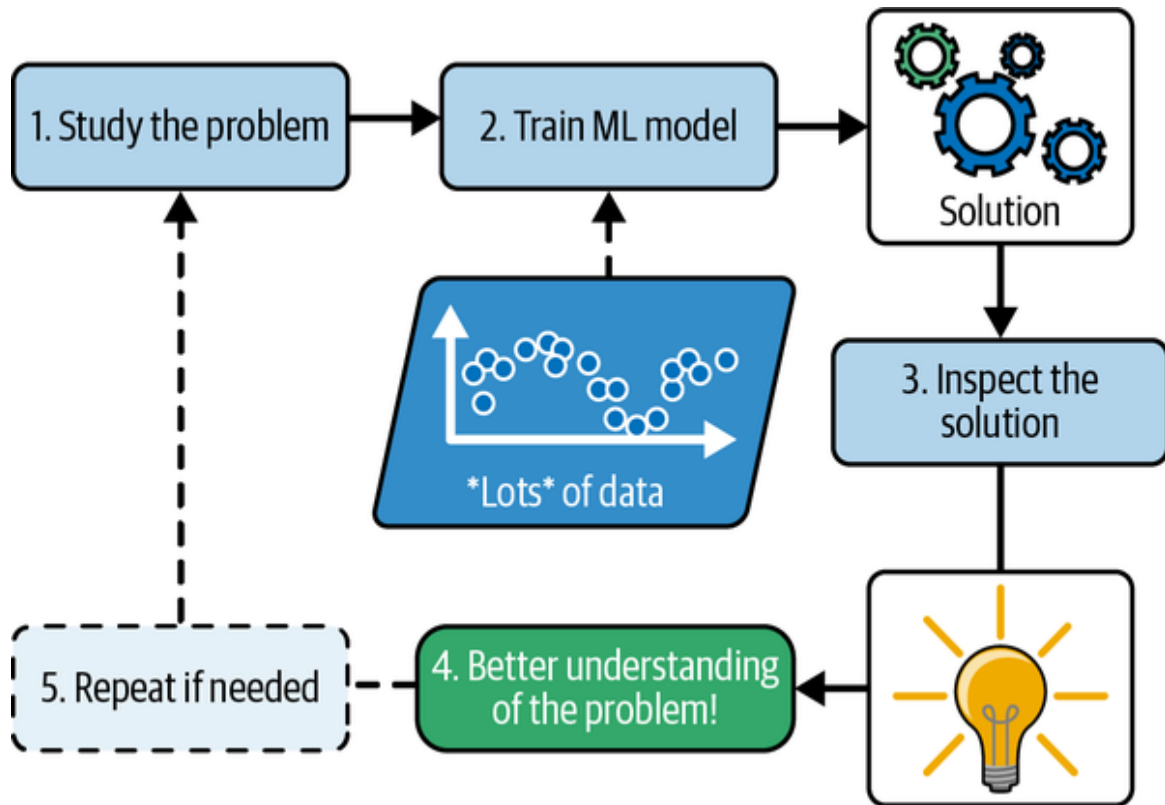


Figure 1-4. Machine learning can help humans learn

To summarize, machine learning is great for:

- Problems for which existing solutions require a lot of fine-tuning or long lists of rules (a 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 support vector machine (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).

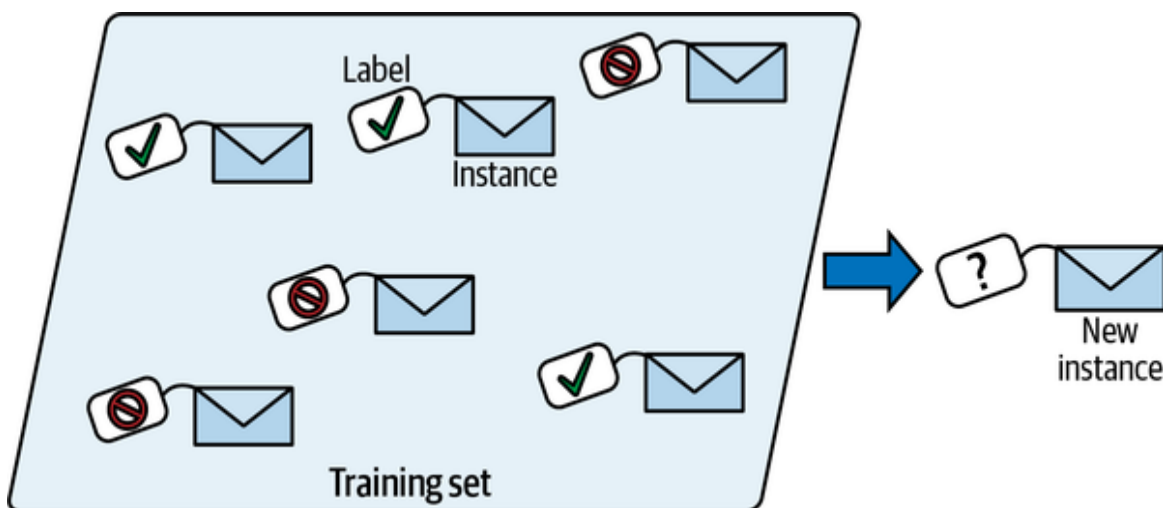


Figure 1-5. A labeled training set for spam classification (an example of supervised learning)

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).

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).

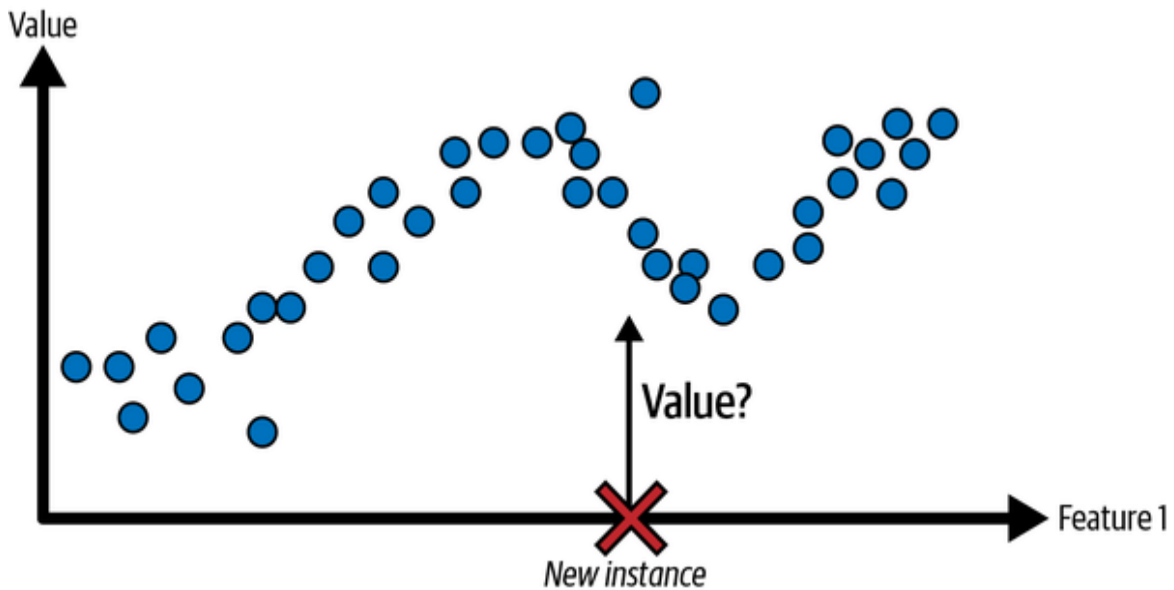


Figure 1-6. A regression problem: predict a value, given an input feature (there are usually multiple input features, and sometimes multiple output values)

NOTE

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”).

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.

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.

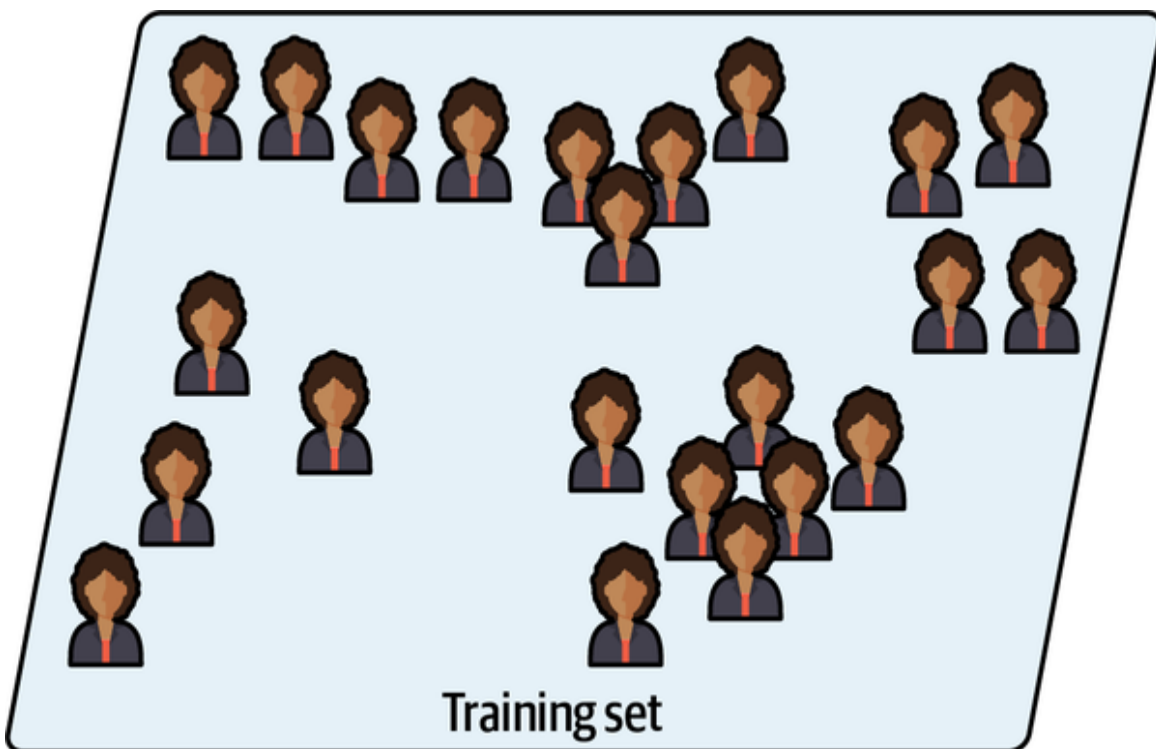


Figure 1-7. An unlabeled training set for unsupervised learning

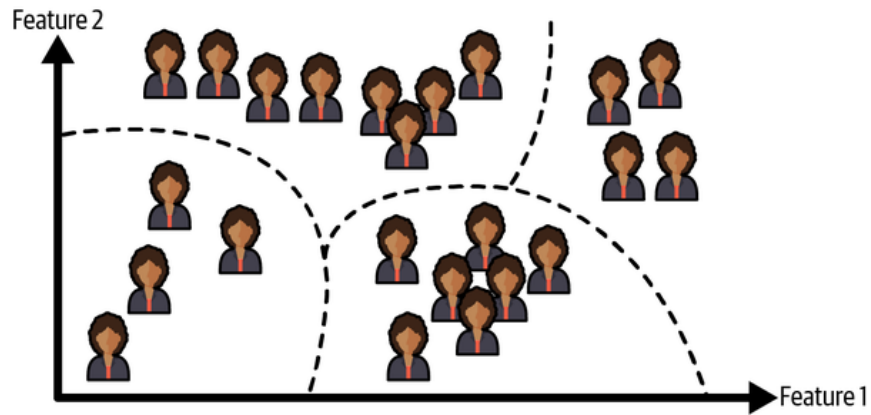


Figure 1-8. Clustering

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.

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*.

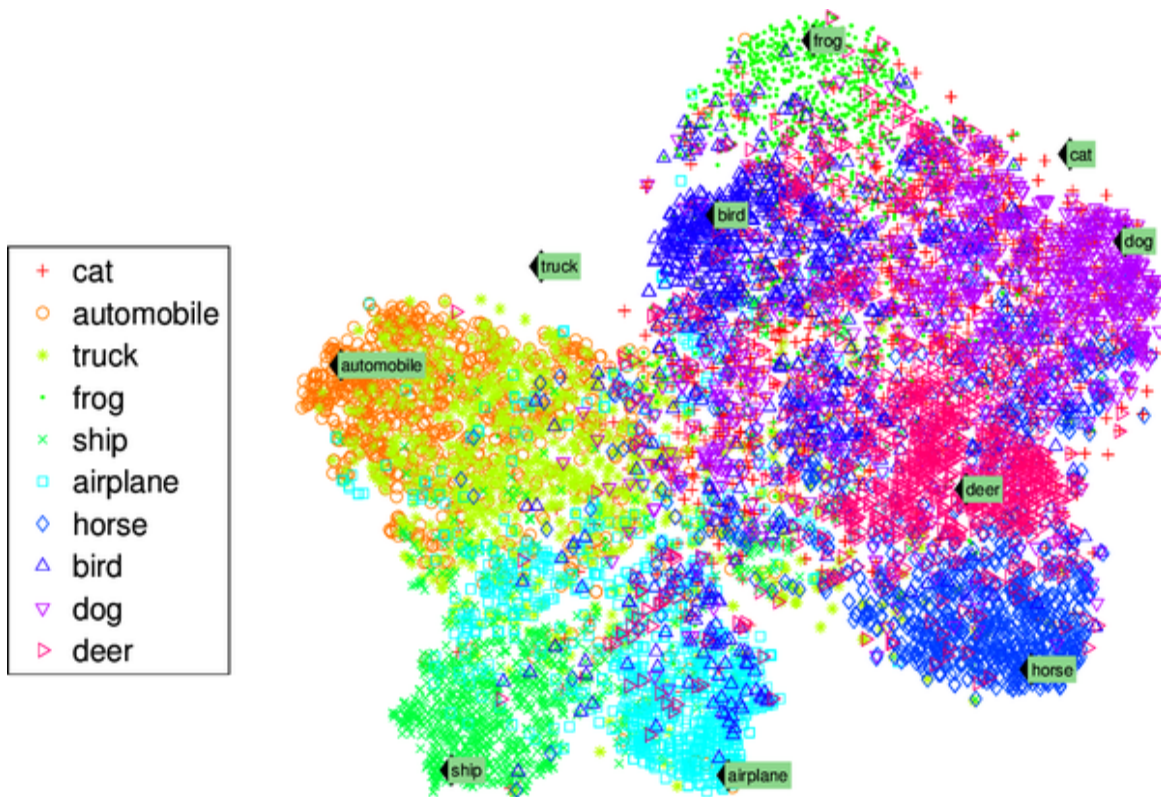


Figure 1-9. Example of a t-SNE visualization highlighting semantic clusters²

TIP

It is often a good idea to try to reduce the number of dimensions in 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 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).

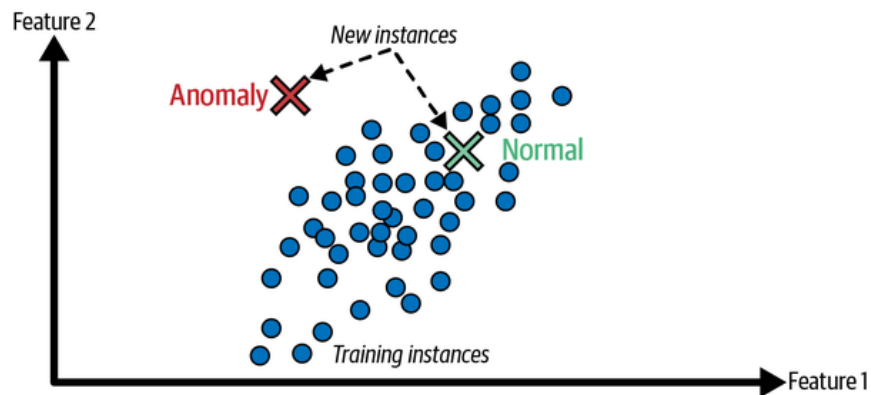


Figure 1-10. Anomaly detection

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).

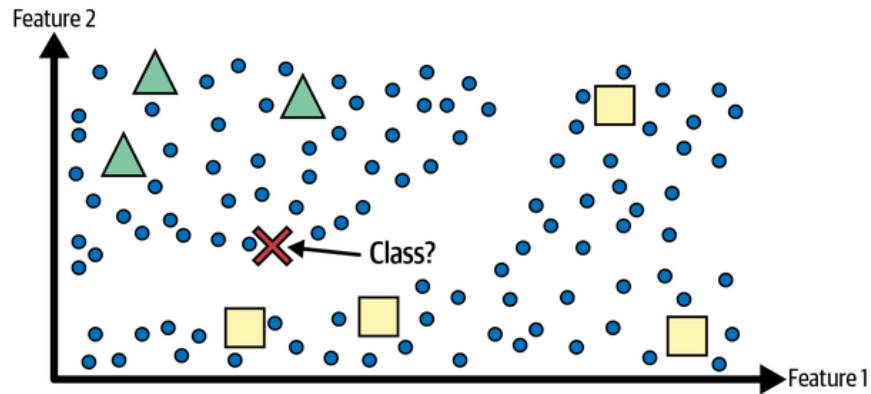


Figure 1-11. Semi-supervised learning with two classes (triangles and squares): the unlabeled examples (circles) help classify a new instance (the cross) into the triangle class rather than the square class, even though it is closer to the labeled squares

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 its 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.

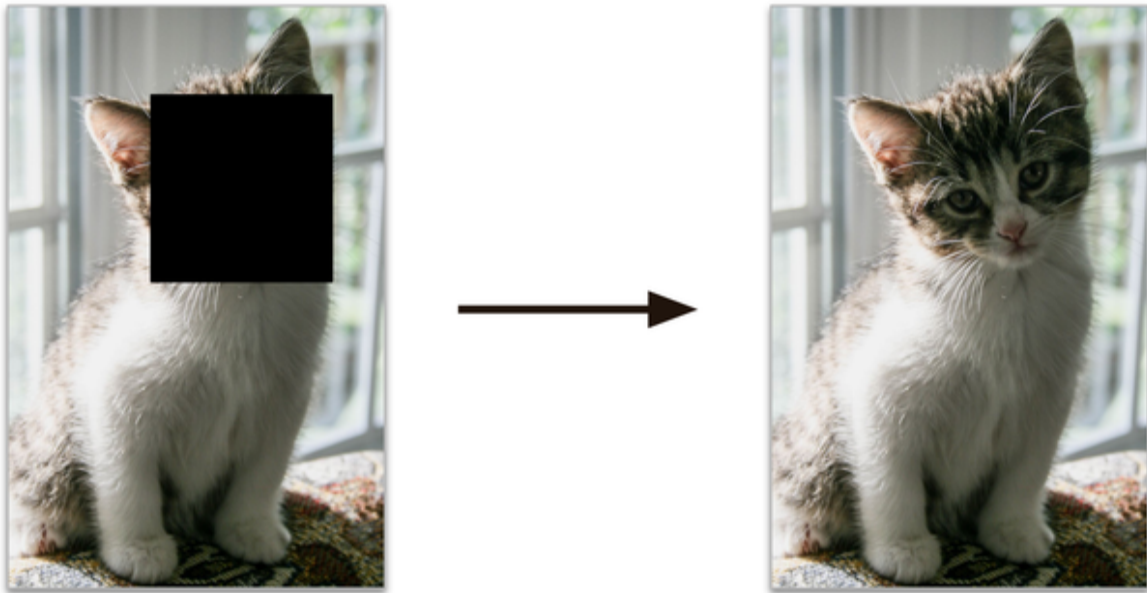


Figure 1-12. Self-supervised learning example: input (left) and target (right)

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'll 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's performing well, it should be able to distinguish different pet species: when it repairs an image of a cat whose face is masked, it must know not to 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.

NOTE

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.

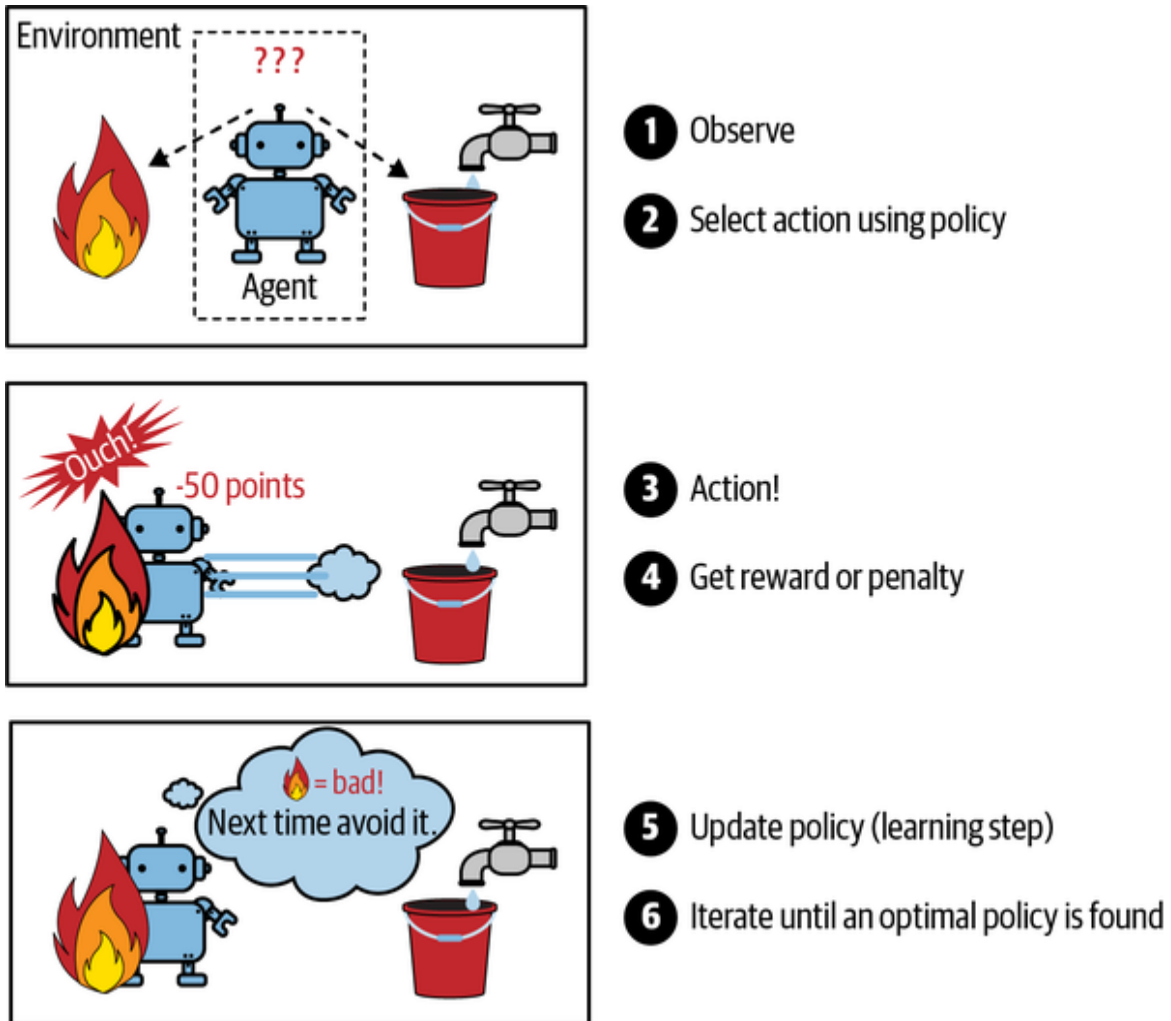


Figure 1-13. Reinforcement learning

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 you will see in the next section, this is called *offline learning*.

Batch Versus Online Learning

Chapter 4. Training Models

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 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. We will explore 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 examine two more models that are commonly used for classification tasks: logistic regression and softmax regression.

WARNING

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:

$$life_satisfaction = \theta_0 + \theta_1 \times GDP_per_capita$$

This model is just a linear function of the input feature `GDP_per_capita`. θ_0 and θ_1 are the model's parameters.

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

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$$

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 $\theta_0x_0 + \theta_1x_1 + \theta_2x_2 + \dots + \theta_nx_n$.

NOTE

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 ([Equation 2-1](#)). Therefore, to train a linear regression model, we need to find the value of $\boldsymbol{\theta}$ 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).

WARNING

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 you 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

$$\text{MSE}(\mathbf{X}, h_{\theta}) = \frac{1}{m} \sum_{i=1}^m \left(\theta^{\top} \mathbf{x}^{(i)} - y^{(i)} \right)^2$$

Most of these notations were presented in [Chapter 2](#) (see “[Notations](#)”). 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

$$\hat{\theta} = (\mathbf{X}^{\top} \mathbf{X})^{-1} \mathbf{X}^{\top} \mathbf{y}$$

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
```

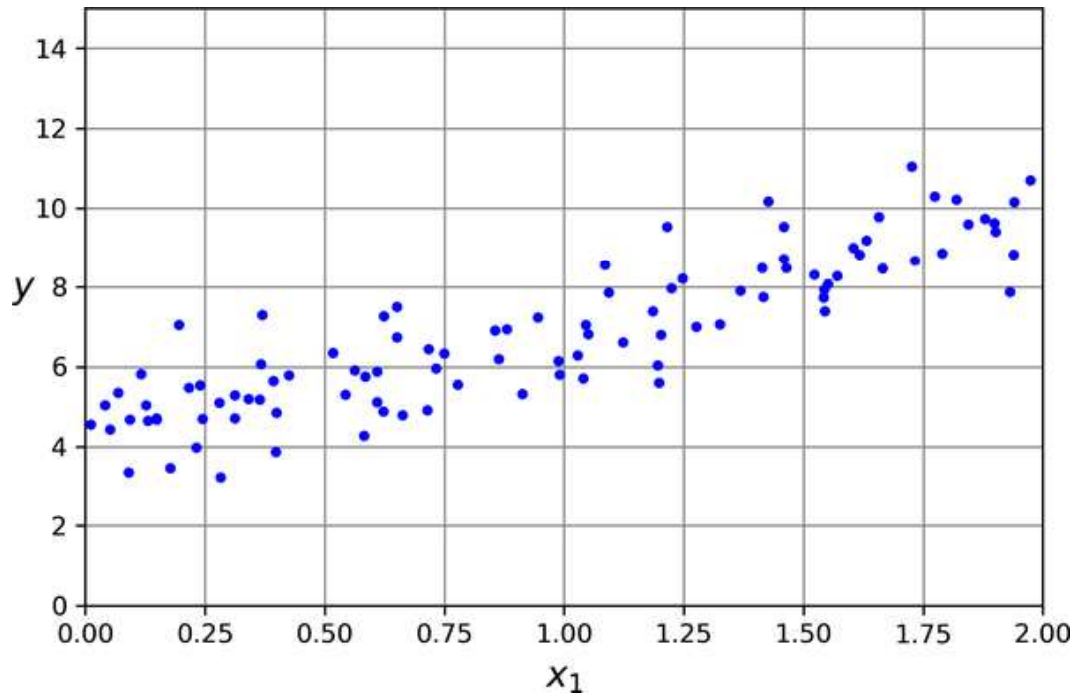


Figure 4-1. A randomly generated linear dataset

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 x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T @ X_b) @ X_b.T @ y
```

NOTE

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, and 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 x0 = 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()
```

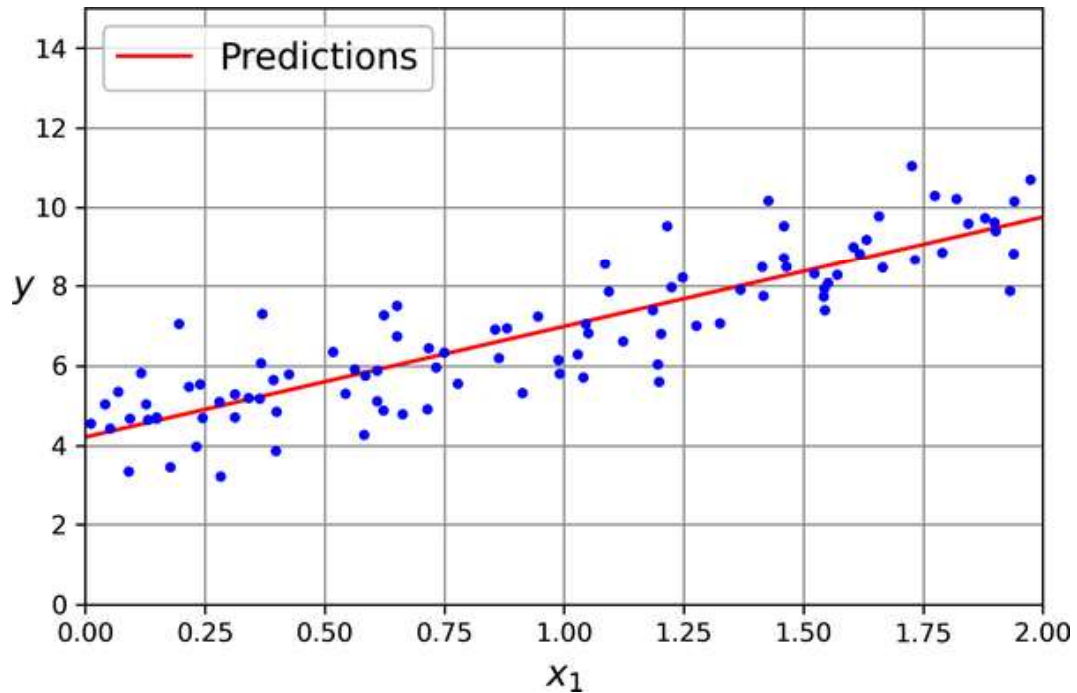


Figure 4-2. Linear regression model predictions

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 $\mathbf{U} \mathbf{\Sigma} \mathbf{V}^\top$ (see `numpy.linalg.svd()`). The pseudoinverse is computed as $\mathbf{X}^+ = \mathbf{V} \mathbf{\Sigma}^+ \mathbf{U}^\top$. To compute the matrix $\mathbf{\Sigma}^+$, the algorithm takes $\mathbf{\Sigma}$ 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.

WARNING

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](#)).

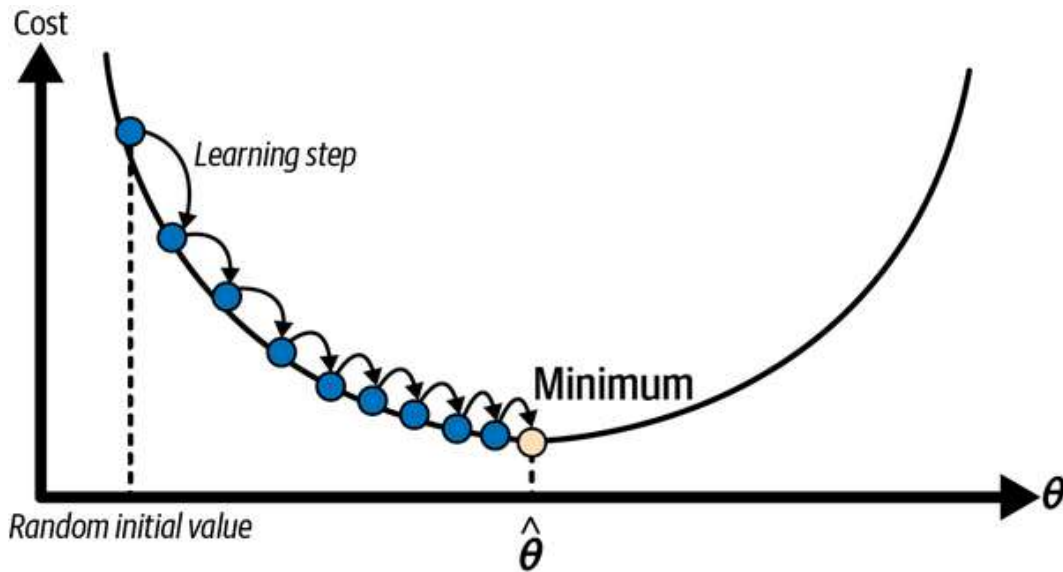


Figure 4-3. In this depiction of gradient descent, the model parameters are initialized randomly and get tweaked repeatedly to minimize the cost function; the learning step size is proportional to the slope of the cost function, so the steps gradually get smaller as the cost approaches the minimum

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).

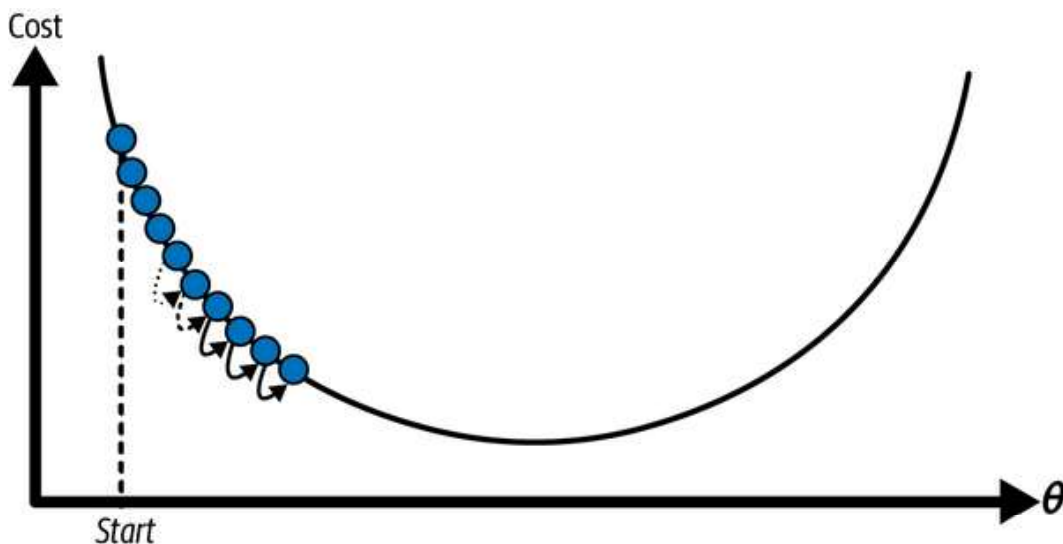


Figure 4-4. Learning rate too small

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).

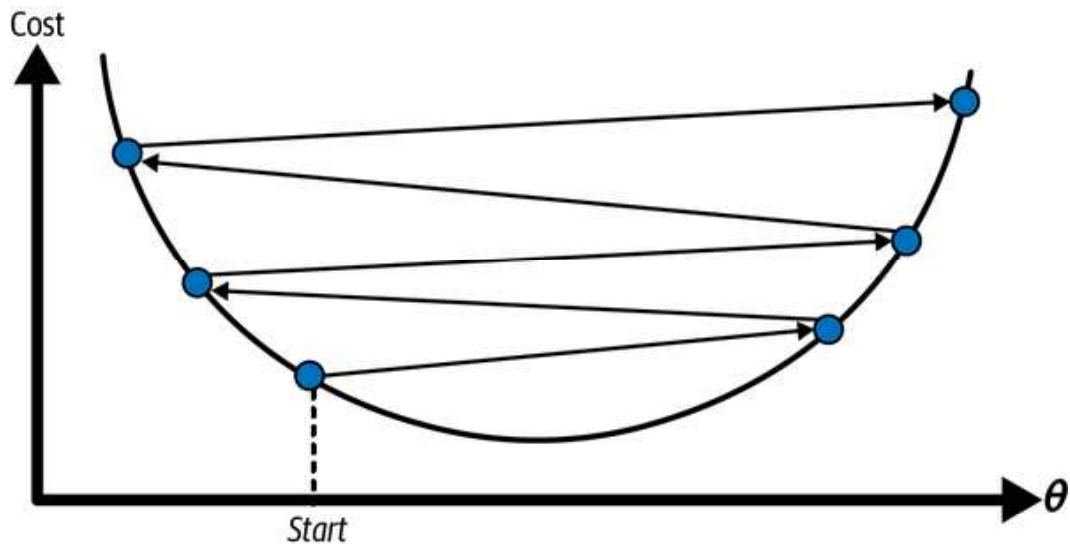


Figure 4-5. Learning rate too high

Additionally, not all cost functions look like nice, regular bowls. There may be holes, ridges, plateaus, and all sorts of irregular terrain, 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.

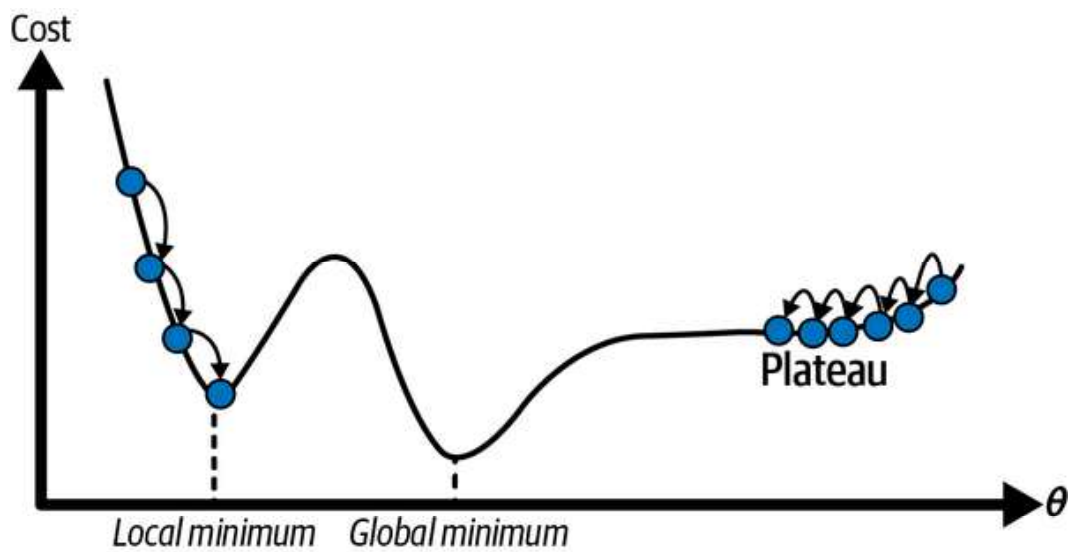


Figure 4-6. Gradient descent pitfalls

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 closely the global minimum (if you wait long enough and if the learning rate is not too high).

While the cost function has the shape of a bowl, 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

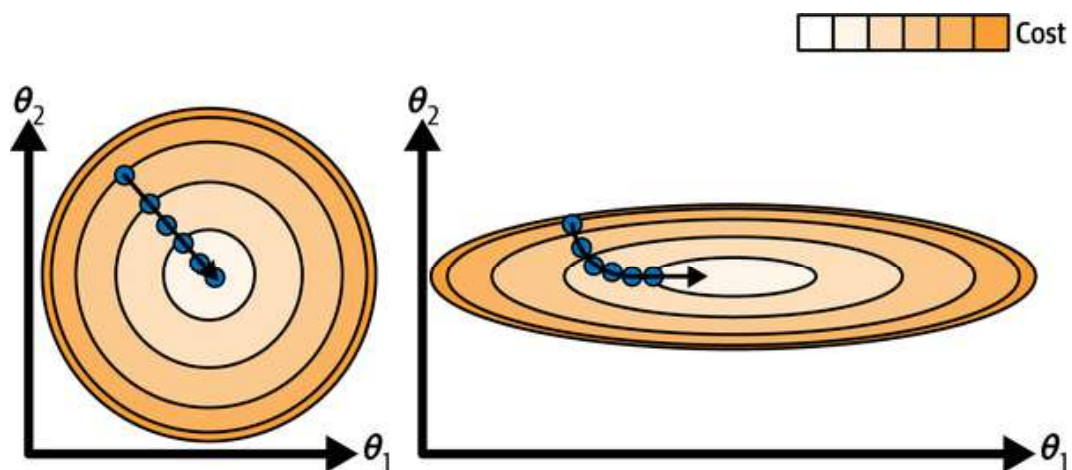


Figure 4-7. Gradient descent with (left) and without (right) feature scaling

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.

WARNING

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 $\partial \text{MSE}(\boldsymbol{\theta}) / \partial \theta_j$.

Equation 4-5. Partial derivatives of the cost function

$$\frac{\partial}{\partial \theta_j} \text{MSE}(\boldsymbol{\theta}) = \frac{2}{m} \sum_{i=1}^m \left(\boldsymbol{\theta}^\top \mathbf{x}^{(i)} - y^{(i)} \right) x_j^{(i)}$$

Instead of computing these partial derivatives individually, you can use [Equation 4-6](#) to compute them all in one go. The gradient vector, noted $\nabla_{\boldsymbol{\theta}} \text{MSE}(\boldsymbol{\theta})$, contains all the partial derivatives of the cost function (one for each model parameter).

Equation 4-6. Gradient vector of the cost function

$$\nabla_{\boldsymbol{\theta}} \text{MSE}(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \text{MSE}(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1} \text{MSE}(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \text{MSE}(\boldsymbol{\theta}) \end{pmatrix} = \frac{2}{m} \mathbf{X}^\top (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$

WARNING

Notice that this formula involves calculations over the full training set 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 (we will look at some 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

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.

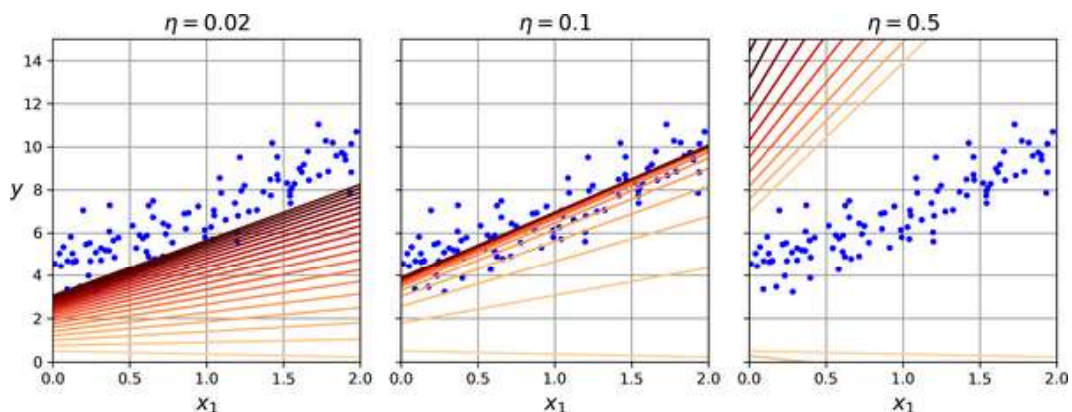


Figure 4-8. Gradient descent with various learning rates

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](#)). Once the algorithm stops, the final parameter values will be good, but not optimal.

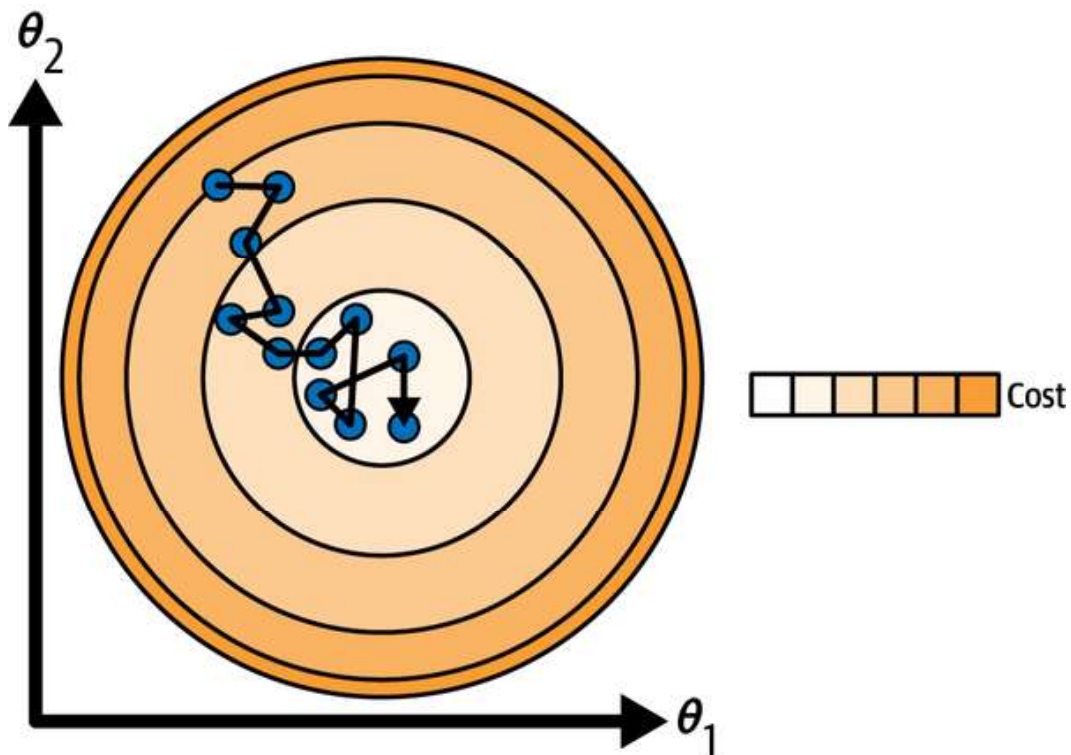


Figure 4-9. With stochastic gradient descent, each training step is much faster but also much more stochastic than when using batch gradient descent

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 by 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).

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.

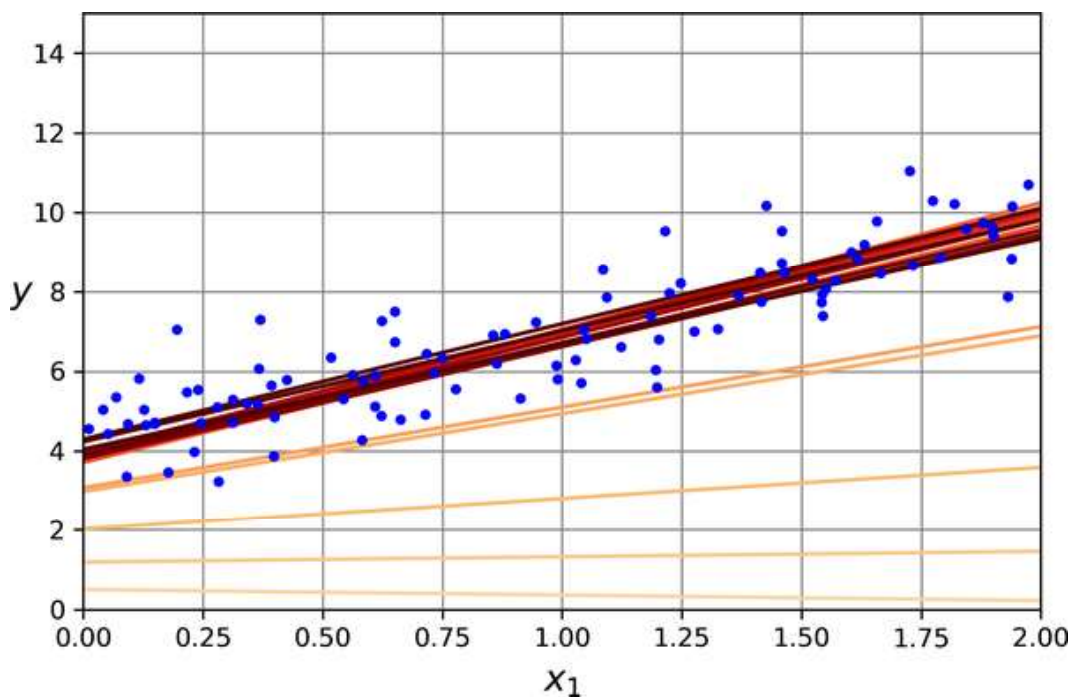


Figure 4-10. The first 20 steps of stochastic gradient descent

WARNING

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]))
```

TIP

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 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.

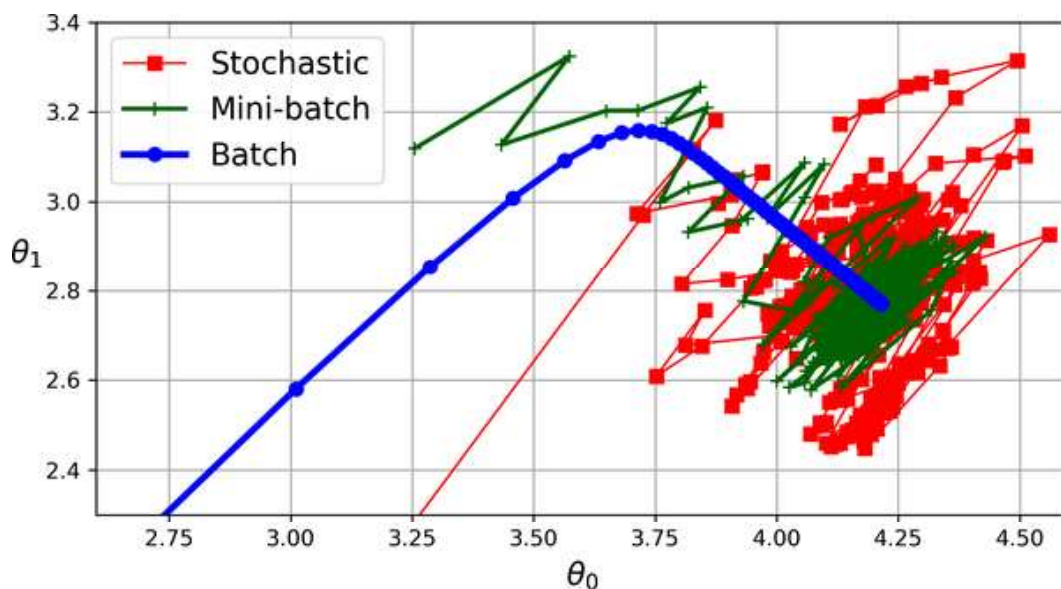


Figure 4-11. Gradient descent paths in parameter space

Table 4-1 compares 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).

Table 4-1. Comparison of algorithms for linear regression

Algorithm	Large m	Out-of-core support	Large n	Hyperparams	Sca
Normal equation	Fast	No	Slow	0	No
SVD	Fast	No	Slow	0	No
Batch GD	Slow	No	Fast	2	Yes
Stochastic GD	Fast	Yes	Fast	≥ 2	Yes
Mini-batch GD	Fast	Yes	Fast	≥ 2	Yes

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, we'll 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)
```

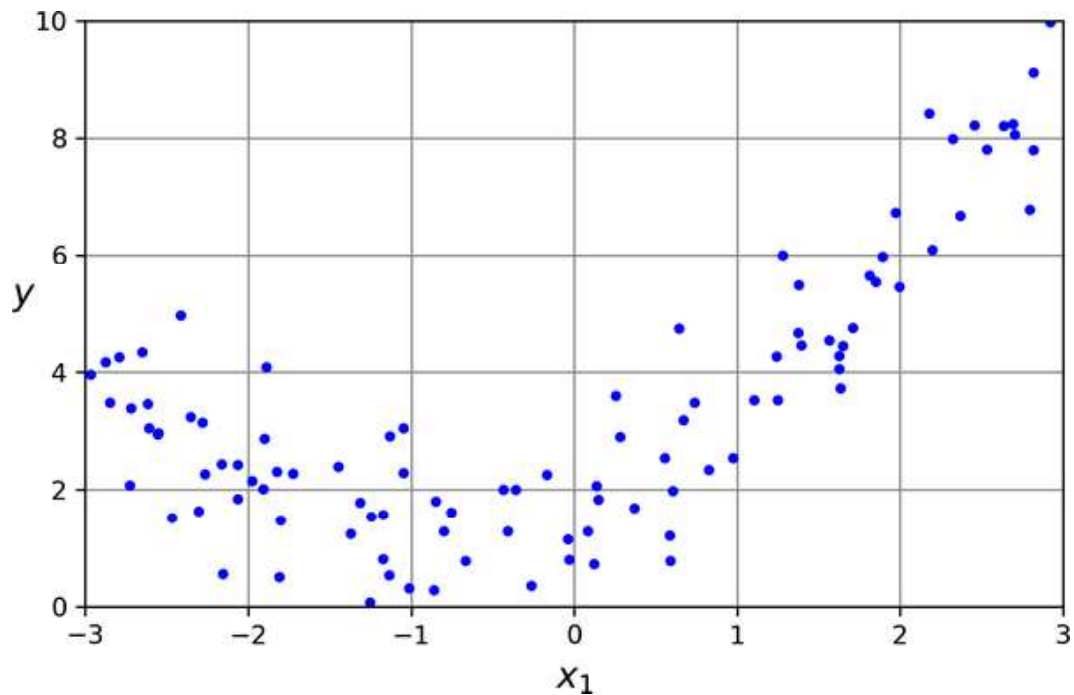


Figure 4-12. Generated nonlinear and noisy dataset

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 we 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]]))
```

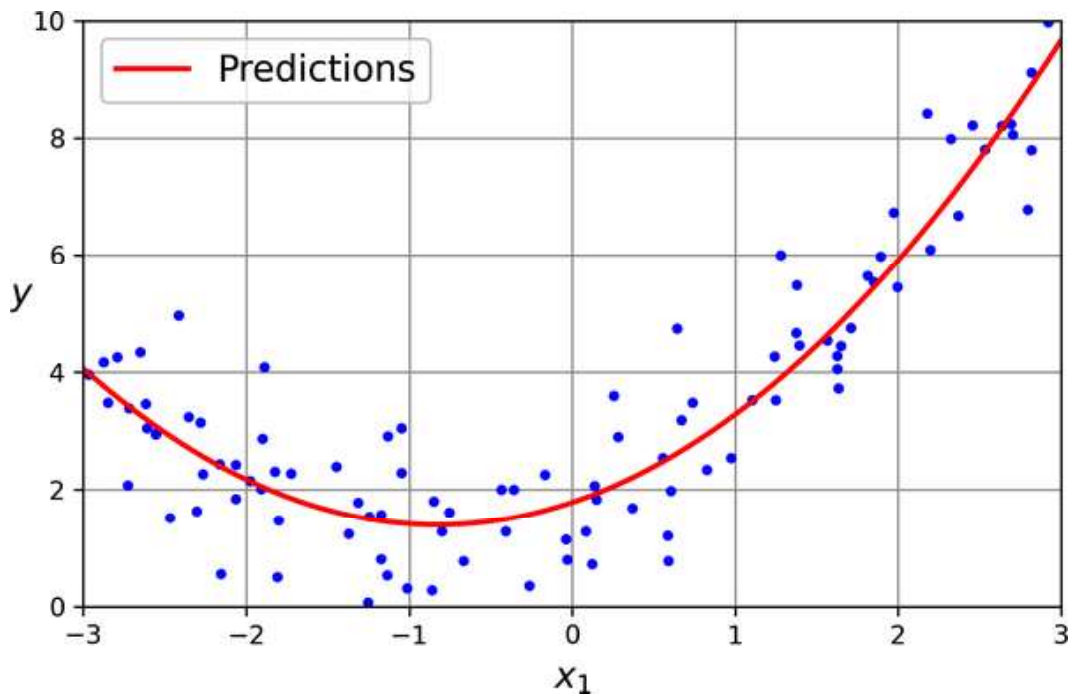


Figure 4-13. Polynomial regression model predictions

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 .

WARNING

`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.

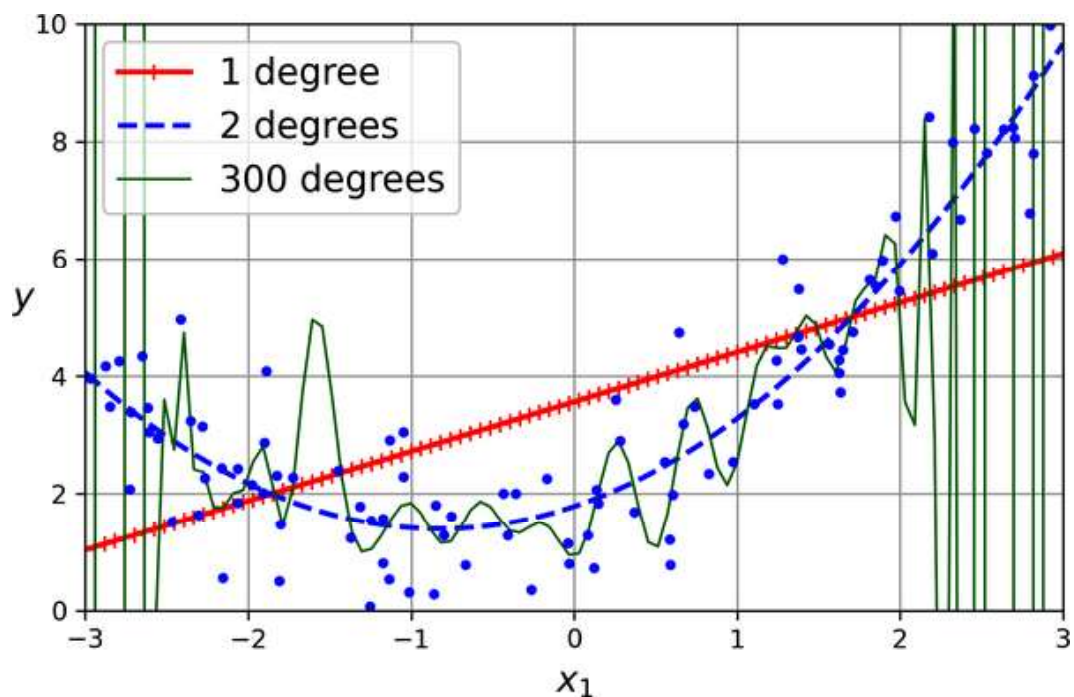


Figure 4-14. High-degree polynomial regression

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*, which 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 it 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_root_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()
```

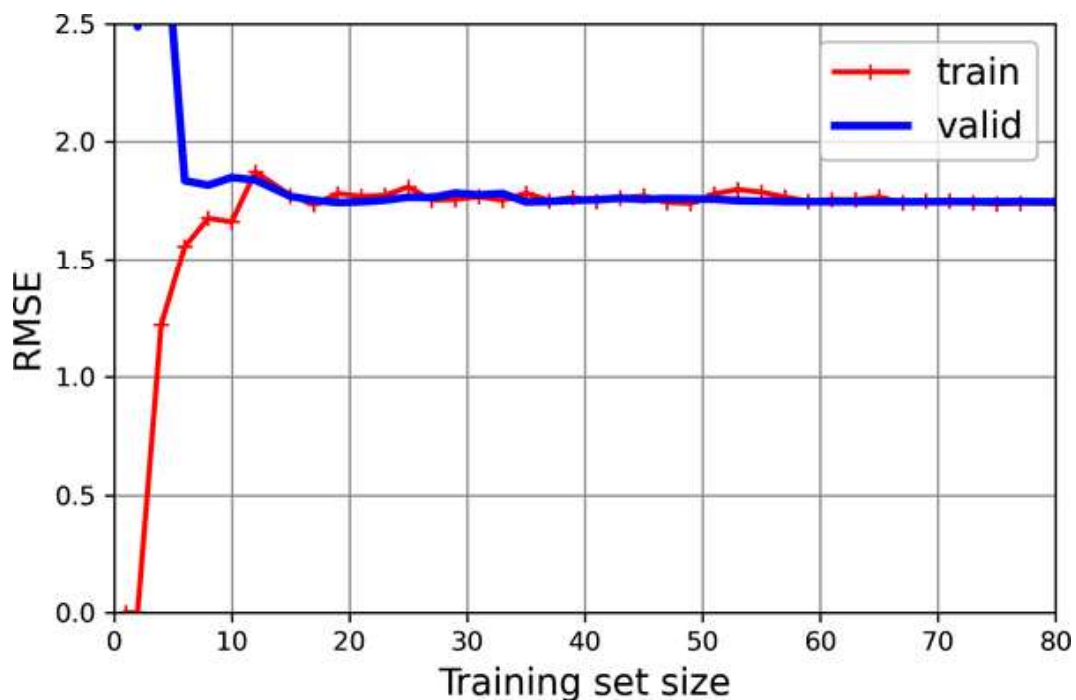


Figure 4-15. Learning curves

This model is underfitting. To 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 of 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.

TIP

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")
[...]
```

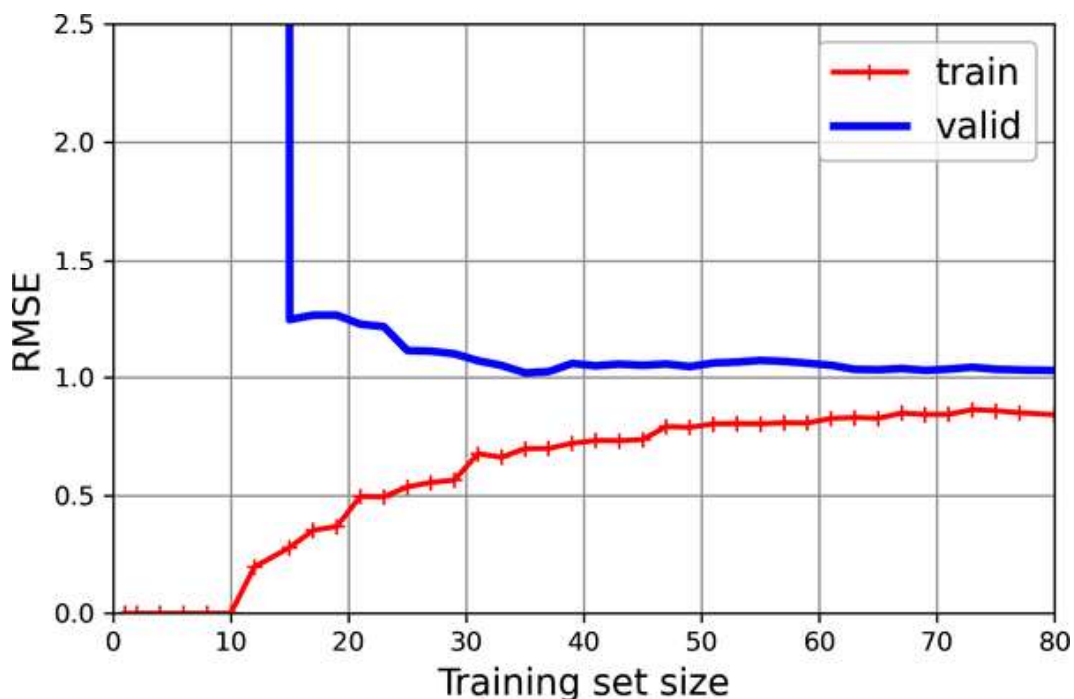


Figure 4-16. Learning curves for the 10th-degree polynomial model

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.

TIP

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 you 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 regression, 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 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](#)).

WARNING

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 that were 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.

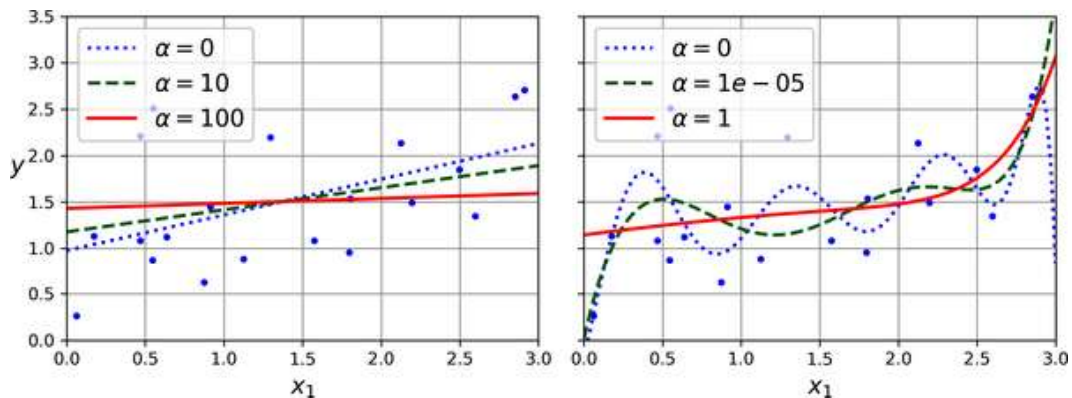


Figure 4-17. Linear (left) and a polynomial (right) models, both with various levels of ridge regularization

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
```

```
>>> 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; that's why we passed `alpha=0.1 / m`, to get the same result as `Ridge(alpha=0.1)`.

TIP

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` and `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 the ridge models with lasso models and uses different α values.

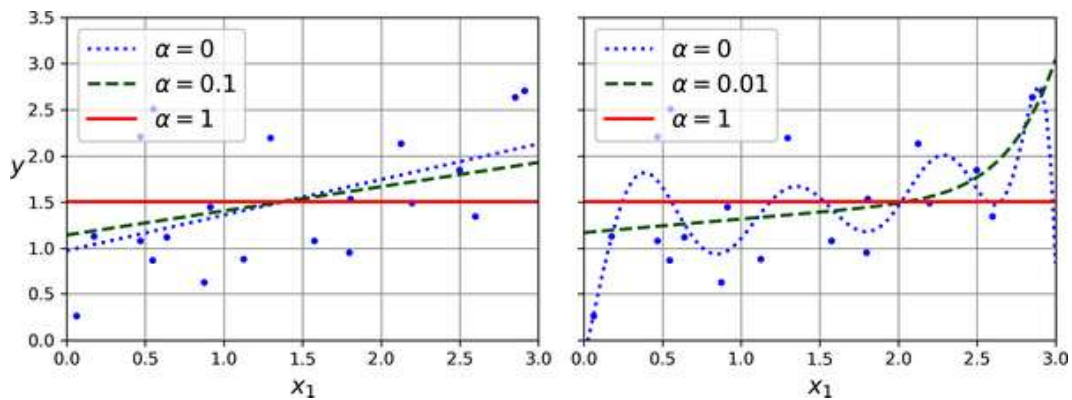


Figure 4-18. Linear (left) and polynomial (right) models, both using various levels of lasso regularization

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* 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 regression'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 $\theta_2 = 0.5$).

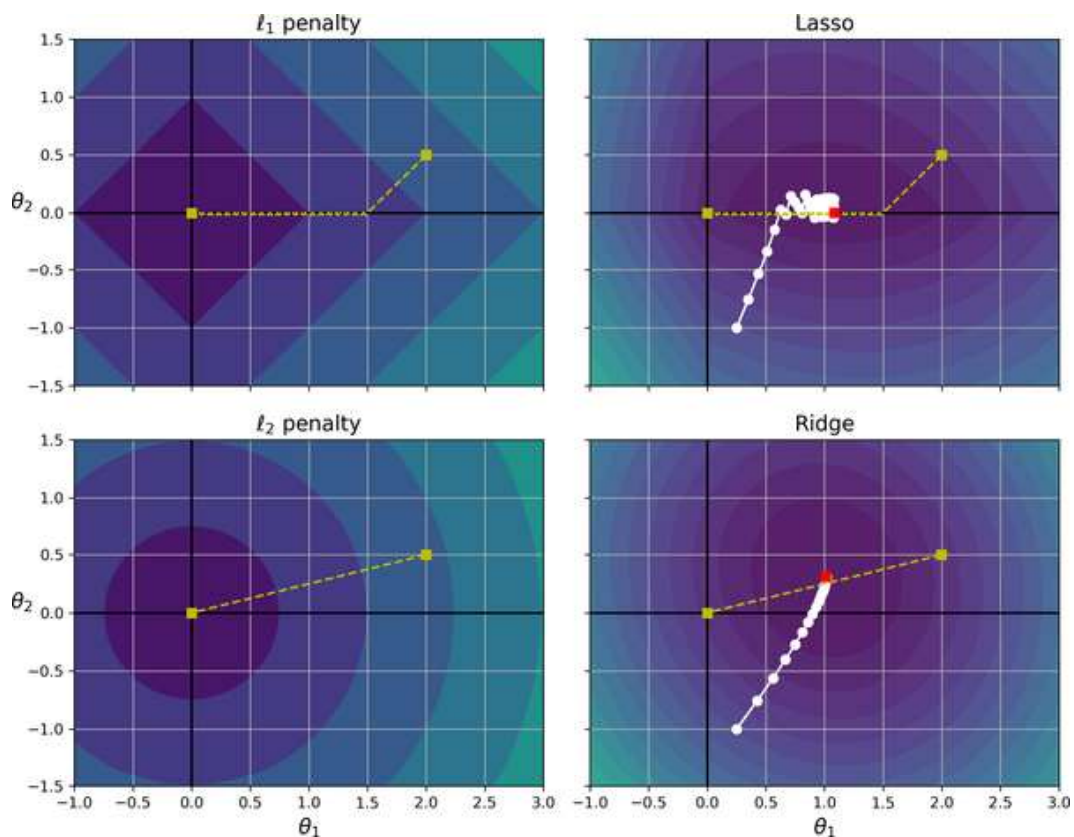


Figure 4-19. Lasso versus ridge regularization

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. This limits the bouncing around, which helps ridge converge faster than lasso regression. 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.

TIP

To keep gradient descent from bouncing around the optimum at the end when using lasso regression, 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 Regression

Elastic net regression 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 (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 elastic net regression, or ridge, lasso, or plain linear regression (i.e., without any regularization)? 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 discussed earlier. In general, elastic net is preferred over lasso because 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”.

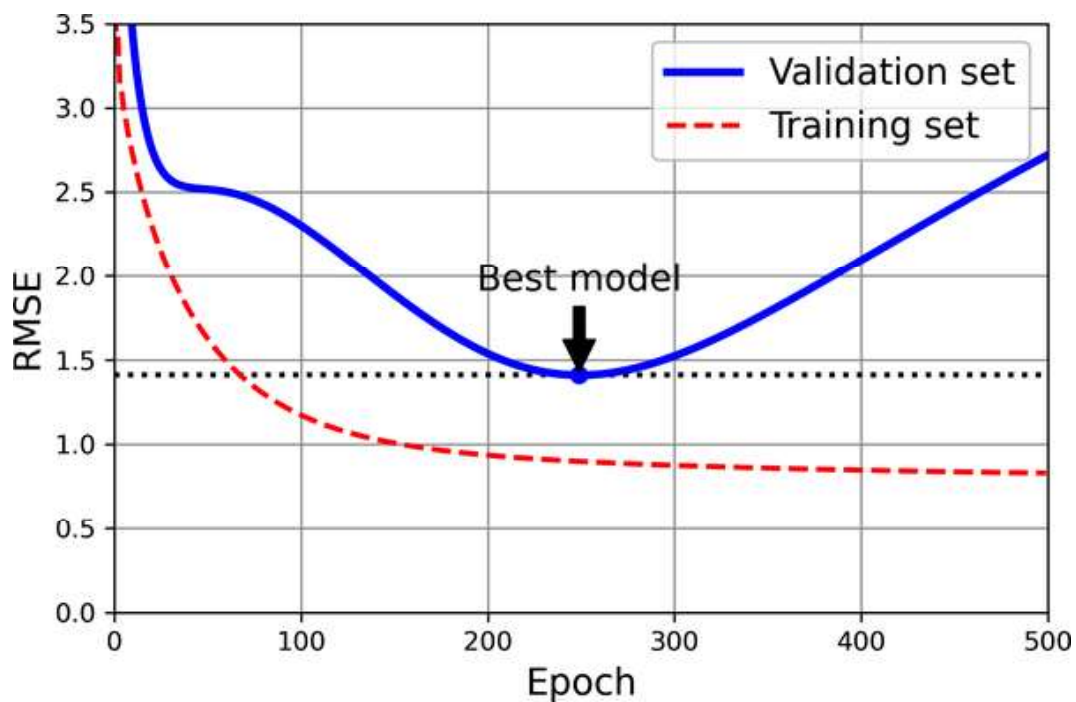


Figure 4-20. Early stopping regularization

TIP

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)

```

This code first adds the polynomial features and scales all the input features, both for the training set and for the validation set (the code assumes that you have split the original 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 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)

$$\hat{p} = h_{\theta}(\mathbf{x}) = \sigma(\theta^T \mathbf{x})$$

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

$$\sigma(t) = \frac{1}{1 + \exp(-t)}$$

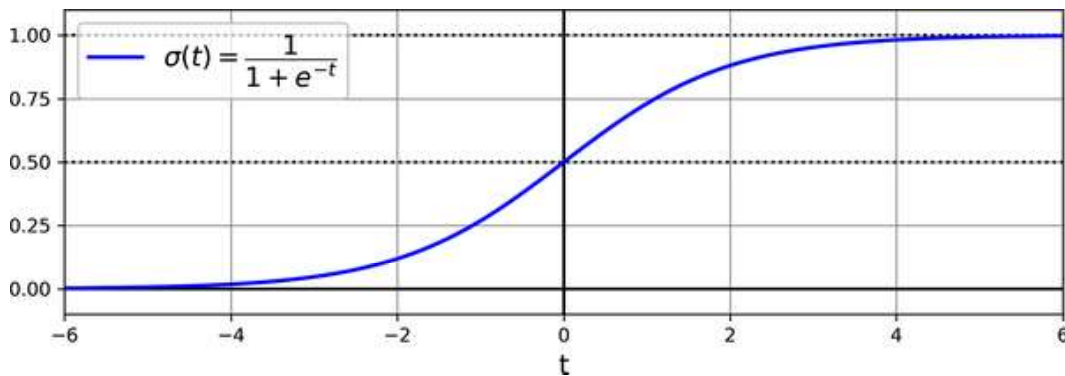


Figure 4-21. Logistic function

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 50% threshold probability

$$\hat{y} = \begin{cases} 0 & \text{if } \hat{p} < 0.5 \\ 1 & \text{if } \hat{p} \geq 0.5 \end{cases}$$

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.

NOTE

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 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 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)

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(\hat{p}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{p}^{(i)})]$$

WARNING

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., if it's 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 θ 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

$$\frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^m \left(\sigma(\boldsymbol{\theta}^\top \mathbf{x}^{(i)}) - y^{(i)} \right) x_j^{(i)}$$

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

We can 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](#)).

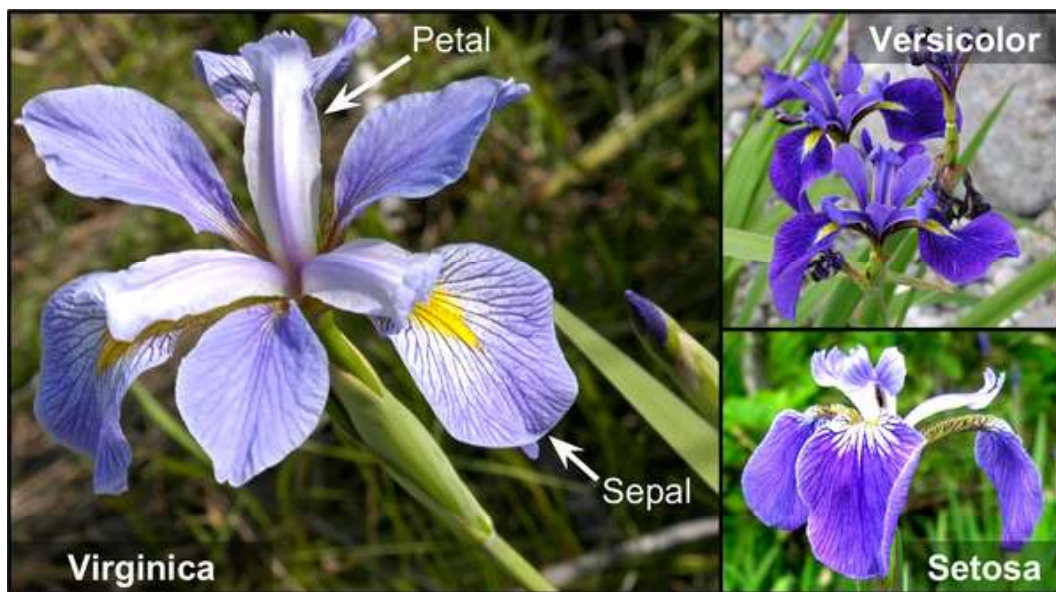


Figure 4-22. Flowers of three iris plant species¹²

Let's try to build a classifier to detect the *Iris virginica* type based only on the petal width feature. The first step is to 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')
```

Next we'll 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()
```

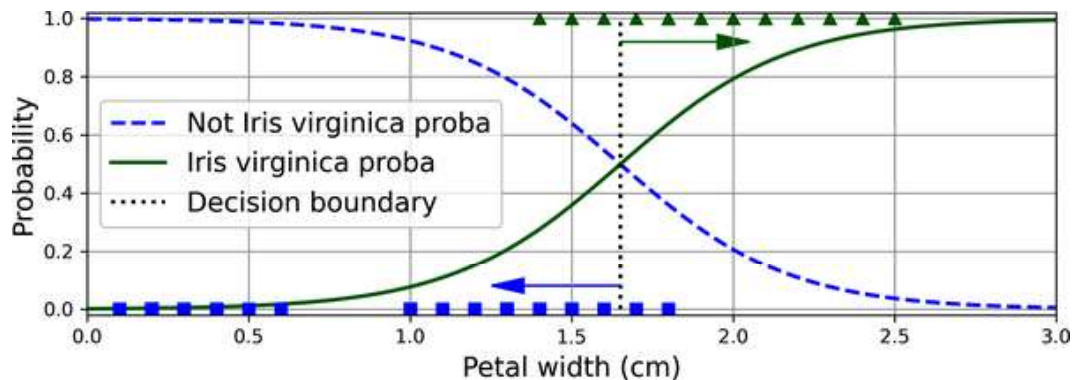


Figure 4-23. Estimated probabilities and decision boundary

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 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 greater 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.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.

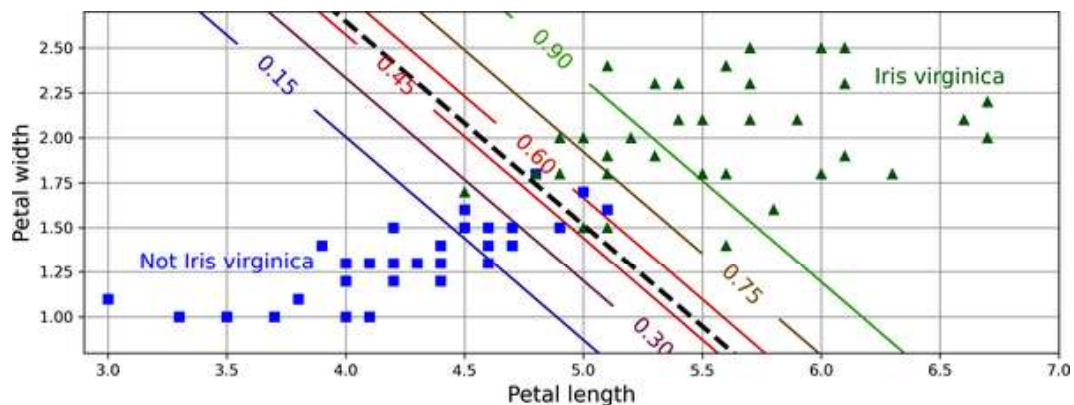


Figure 4-24. Linear decision boundary

NOTE

The hyperparameter controlling the regularization strength of a Scikit-Learn `LogisticRegression` model is not α (as in other linear models), but its inverse: C . The higher the value of C , the less the model is regularized.

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.

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

$$s_k(\mathbf{x}) = \left(\boldsymbol{\theta}^{(k)} \right)^T \mathbf{x}$$

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* Θ .

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

$$\hat{y} = \underset{k}{\operatorname{argmax}} \sigma(\mathbf{s}(\mathbf{x}))_k = \underset{k}{\operatorname{argmax}} s_k(\mathbf{x}) = \underset{k}{\operatorname{argmax}} \left(\left(\boldsymbol{\theta}^{(k)} \right)^\top \mathbf{x} \right)$$

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 $\sigma(\mathbf{s}(\mathbf{x}))_k$.

TIP

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

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log(\hat{p}_k^{(i)})$$

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 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 3 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 assumption is 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` classifier 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 mentioned 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%.

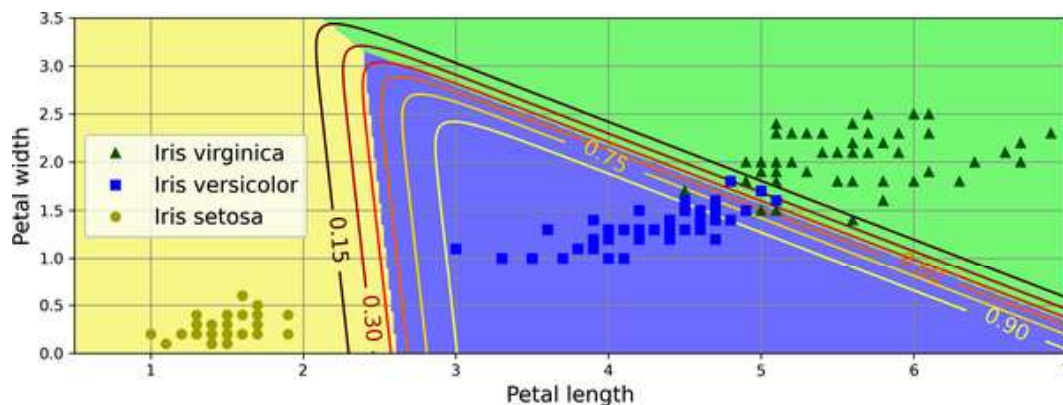


Figure 4-25. Softmax regression decision boundaries

In this chapter, you learned various ways to train linear models, both for regression and for 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 regression?

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>.

-
- 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.
 - 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.
 - 4 Eta (η) is the seventh letter of the Greek alphabet.
 - 5 While the Normal equation can only perform linear regression, the gradient descent algorithms can be used to train many other models, as you'll see.
 - 6 This notion of bias is not to be confused with the bias term of linear models.
 - 7 It is common to use the notation $J(\boldsymbol{\theta})$ for cost functions that don't have a short name; I'll 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](#).
 - 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.
 - 11 You can think of a subgradient vector at a nondifferentiable point as an intermediate vector between the gradient vectors around that point.
 - 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.
 - 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.
 - 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.

Chapter 6. Decision Trees

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 explore 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()` function 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](#).

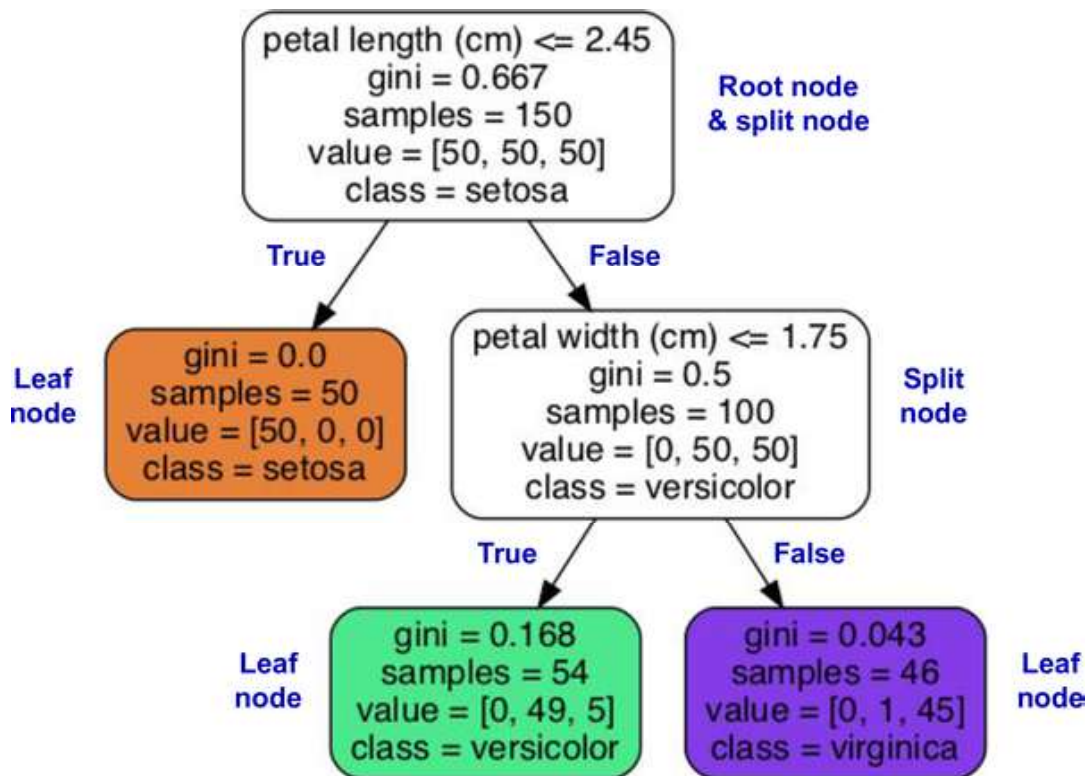


Figure 6-1. Iris decision tree

Making Predictions

Let's see how the tree represented in [Figure 6-1](#) makes predictions. Suppose you find an iris flower 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 again start at the root but now move down to its right child node (depth 1, right). This 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.

NOTE

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" (*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.

NOTE

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).

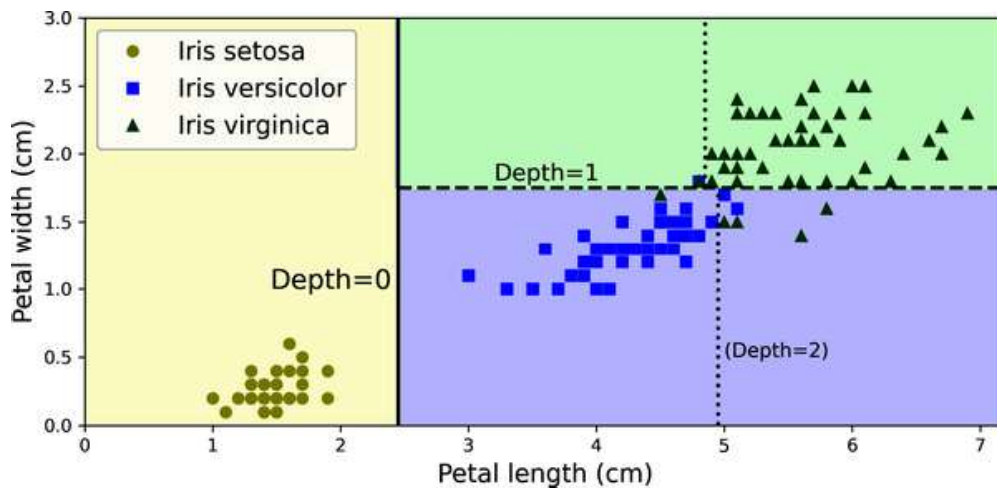


Figure 6-2. Decision tree decision boundaries

TIP

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 [this chapter's notebook](#) for an example.

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 you will see, random forests and 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 in 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.

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`.

WARNING

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` hyperparameter 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 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

$$H_i = - \sum_{\substack{k=1 \\ p_{i,k} > 0}}^n p_{i,k} \log_2(p_{i,k})$$

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

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.

NOTE

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`. Here's the code; [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)
```

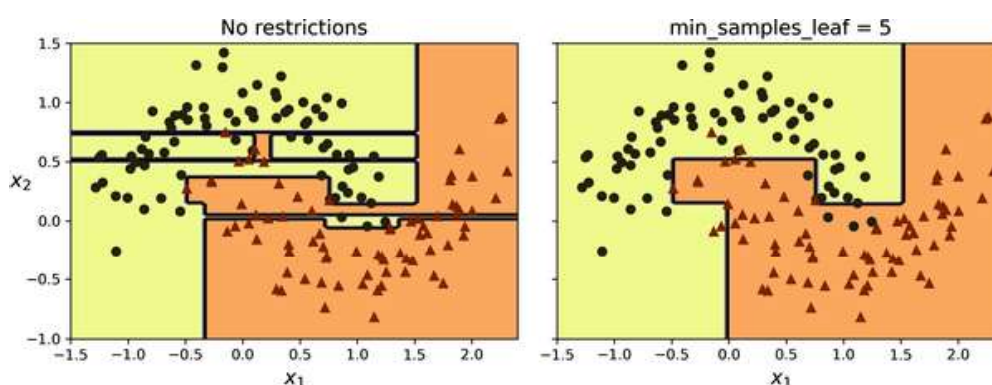


Figure 6-3. Decision boundaries of an unregularized tree (left) and a regularized tree (right)

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](#).

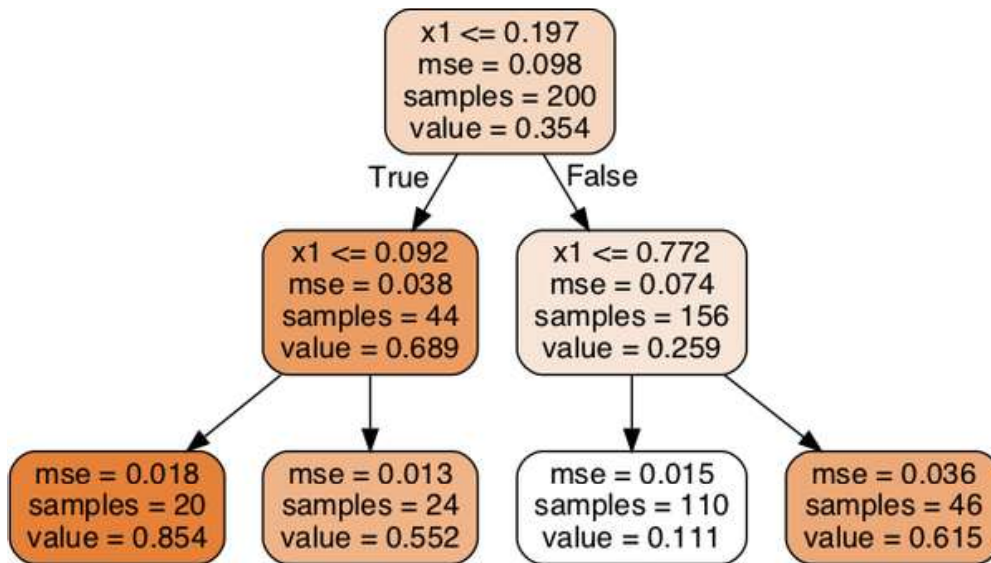


Figure 6-4. A decision tree for regression

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 $value=0.111$. 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.

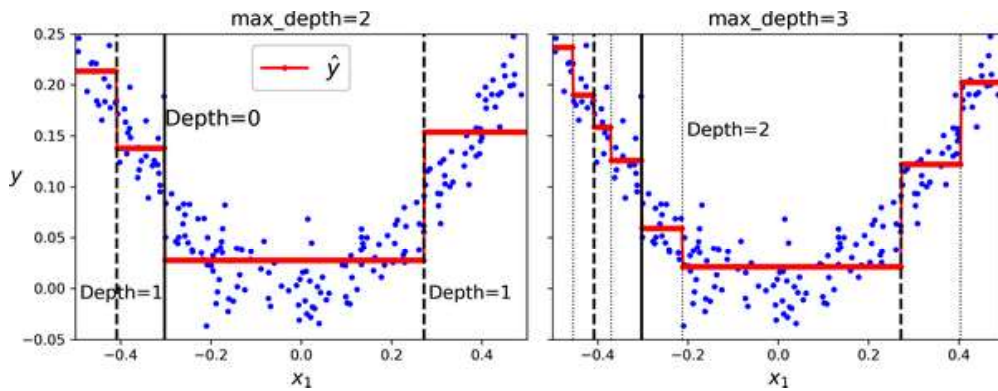


Figure 6-5. Predictions of two decision tree regression models

The CART algorithm works as described 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

$$J(k, t_k) = \frac{m_{\text{left}}}{m} \text{MSE}_{\text{left}} + \frac{m_{\text{right}}}{m} \text{MSE}_{\text{right}} \quad \text{where} \quad \text{MSE}_{\text{node}} = \frac{\sum_{i \in \text{node}} (\hat{y}_{\text{node}} - y^{(i)})^2}{m_{\text{node}}}$$

$$\hat{y}_{\text{node}} = \frac{\sum_{i \in \text{node}} y^{(i)}}{m_{\text{node}}}$$

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.

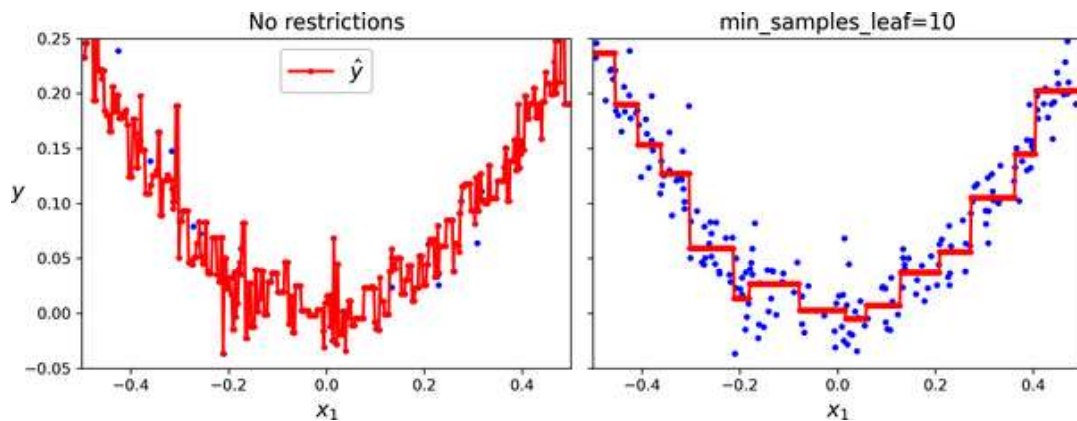


Figure 6-6. Predictions of an unregularized regression tree (left) and a regularized tree (right)

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.

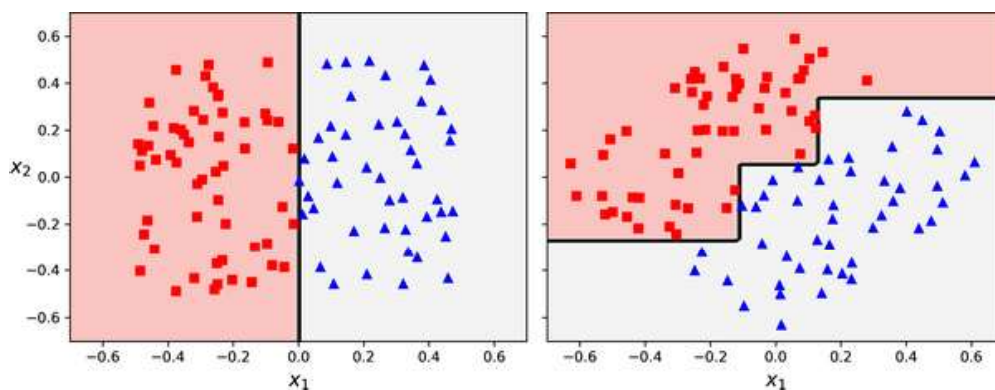


Figure 6-7. Sensitivity to training set rotation

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 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. Here's the code:

```
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)
```

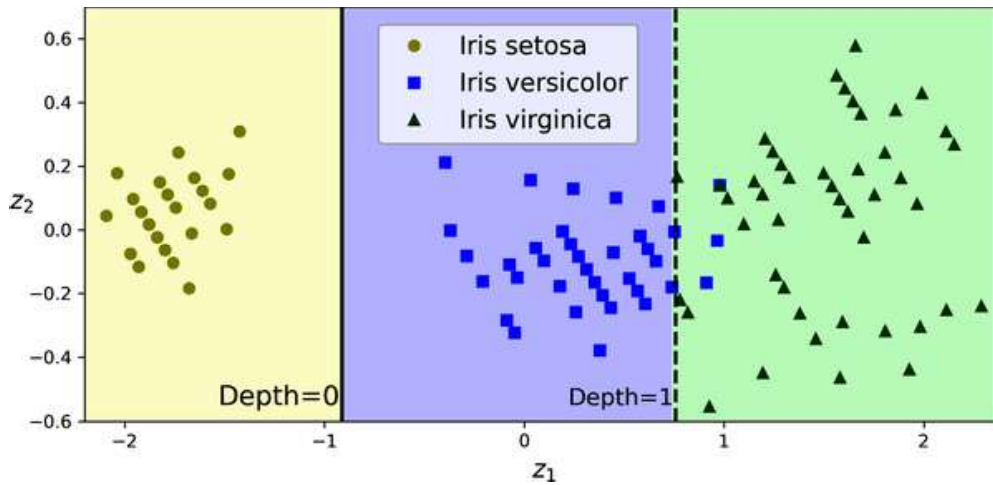


Figure 6-8. A tree's decision boundaries on the scaled and PCA-rotated iris dataset

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).

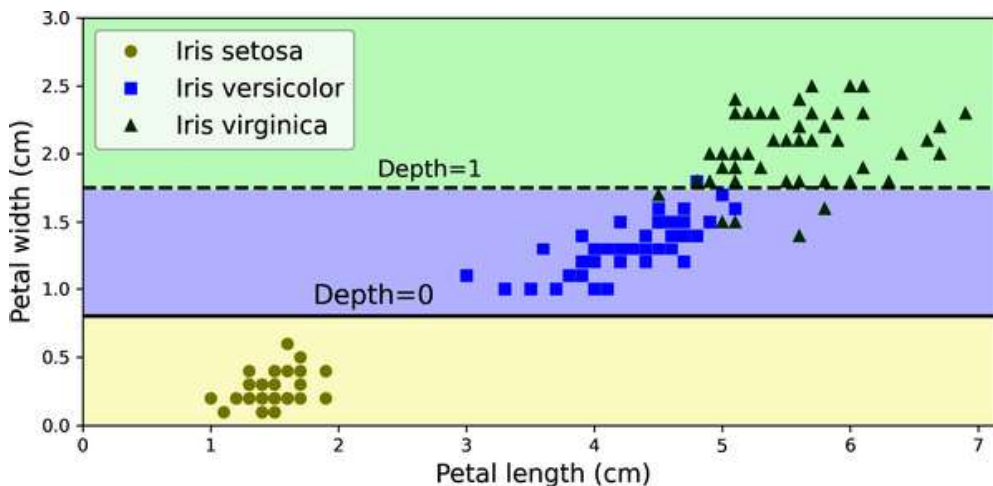


Figure 6-9. Retraining the same model on the same data may produce a very different model

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 you 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 higher than its parent's? Is it *generally* lower/higher, or *always* lower/higher?
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 one million instances, roughly how much time will it take to train another decision tree on a training set containing ten 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>.

-
- 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 $P \neq NP$ (which seems likely), then no polynomial algorithm will ever be found for any NP-complete problem (except perhaps one day on a quantum computer).
 - 2 See Sebastian Raschka's [interesting analysis](#) for more details.

OceanofPDF.com

Chapter 7. Ensemble Learning and Random Forests

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 examine the most popular ensemble methods, including voting classifiers, bagging and pasting ensembles, random forests, and boosting, and stacking ensembles.

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](#)).

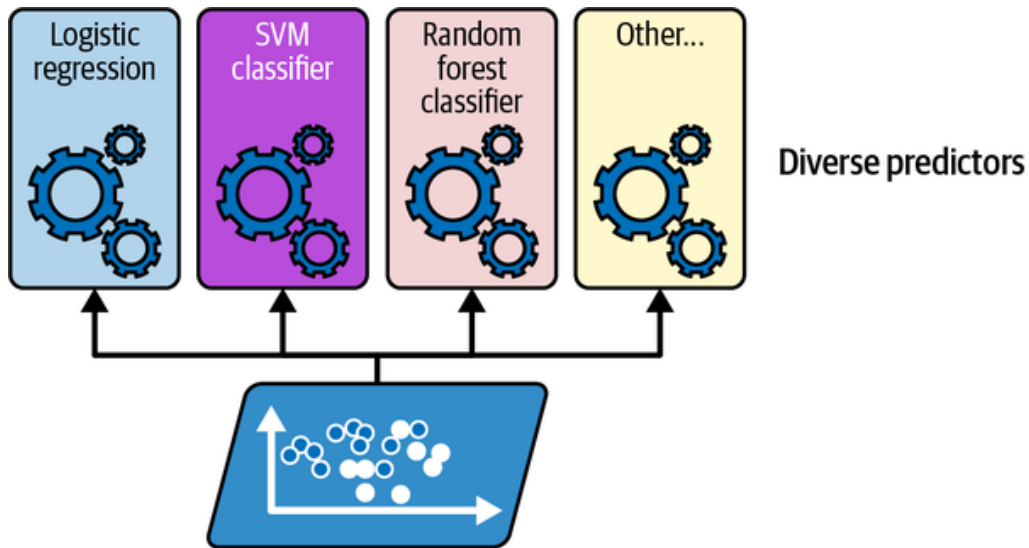


Figure 7-1. Training diverse classifiers

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).

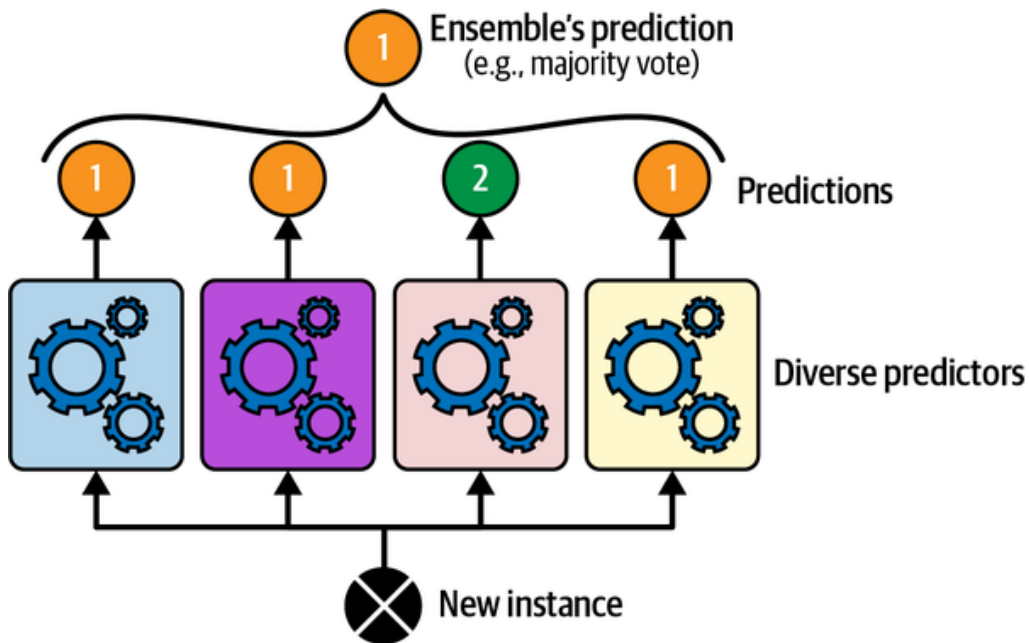


Figure 7-2. Hard voting classifier predictions

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%.

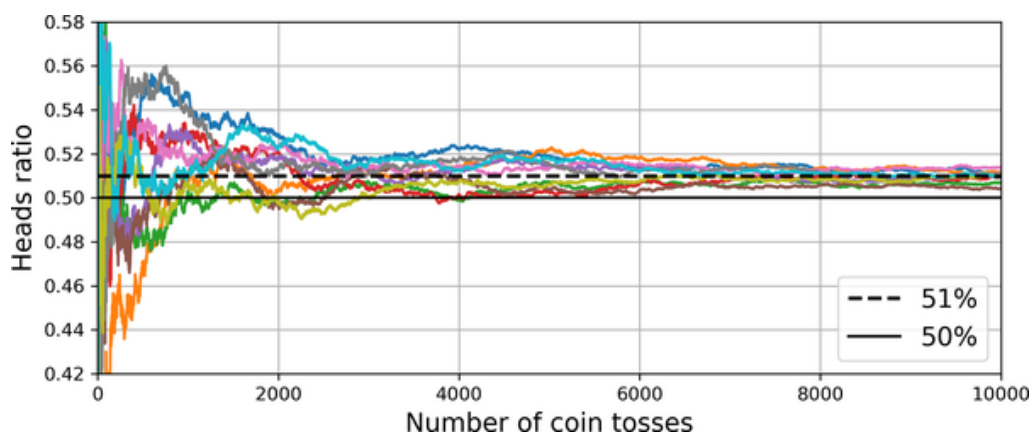


Figure 7-3. The law of large numbers

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.

TIP

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. Let’s try it on the moons dataset (introduced in [Chapter 5](#)). 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. To begin, 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 two out of three 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., if 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](#).

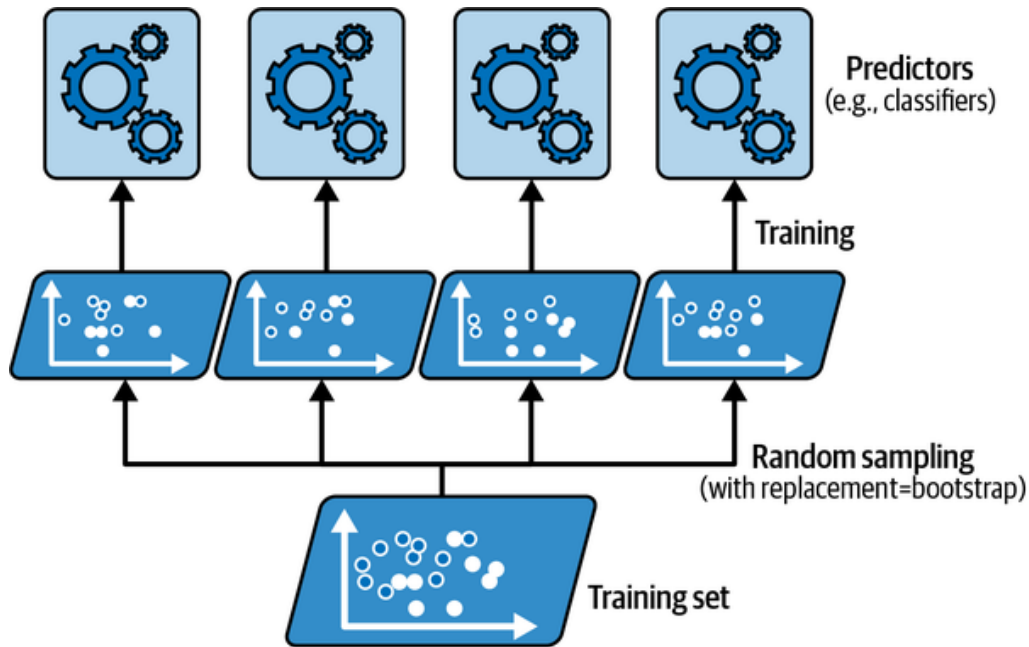


Figure 7-4. Bagging and pasting involve training several predictors on different random samples of the training set

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 with 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: `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)
```

NOTE

A `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 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).

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's generally preferred. But 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.

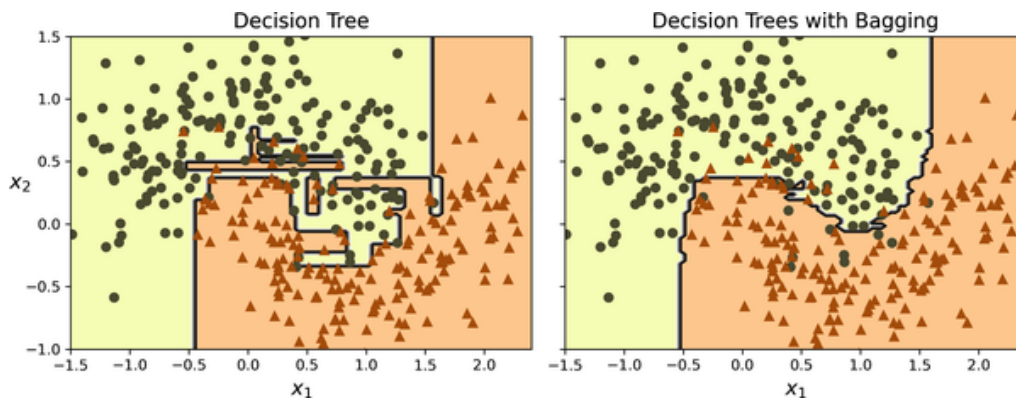


Figure 7-5. A single decision tree (left) versus a bagging ensemble of 500 trees (right)

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).

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, just 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 a 32.4% probability 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 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 leaf nodes, 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`:

```
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*¹² (or *extra-trees* for short) ensemble. Once again, this technique trades more bias for a lower variance. It also makes extra-trees classifiers 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`.

TIP

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 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](#).

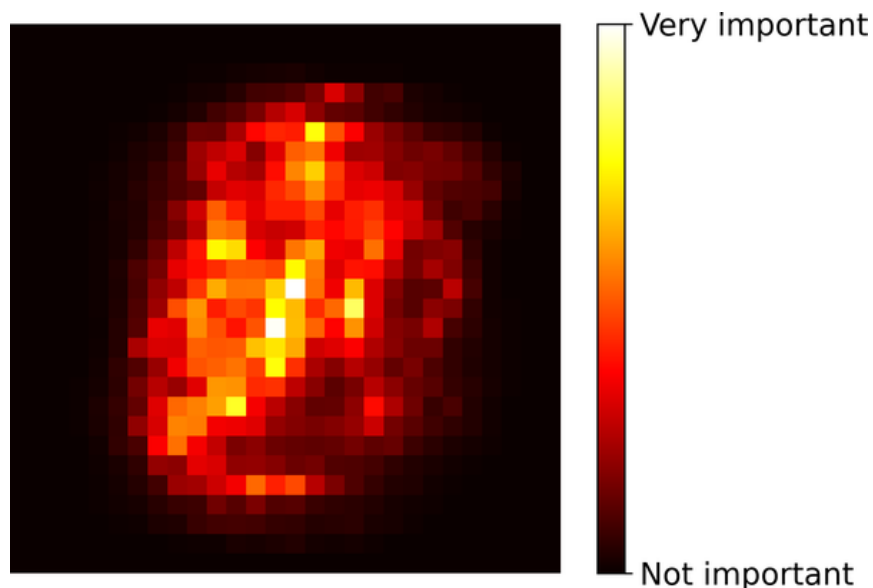


Figure 7-6. MNIST pixel importance (according to a random forest classifier)

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 underfit. 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 7-8](#) 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 kernel).¹⁴ The first classifier gets many instances wrong, so their weights 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.

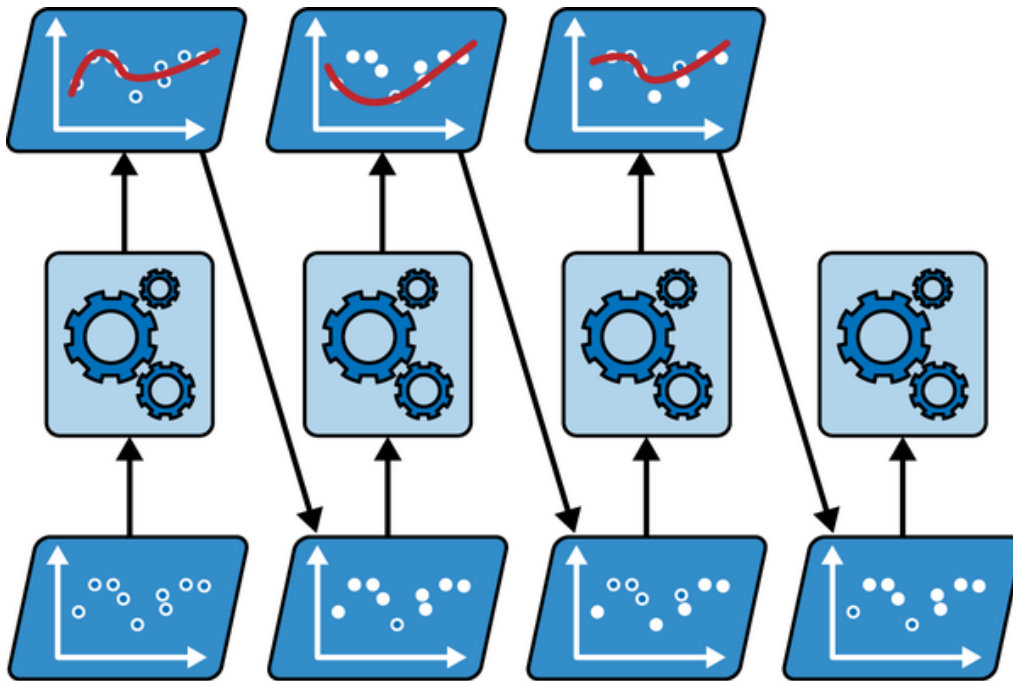


Figure 7-7. AdaBoost sequential training with instance weight updates

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.

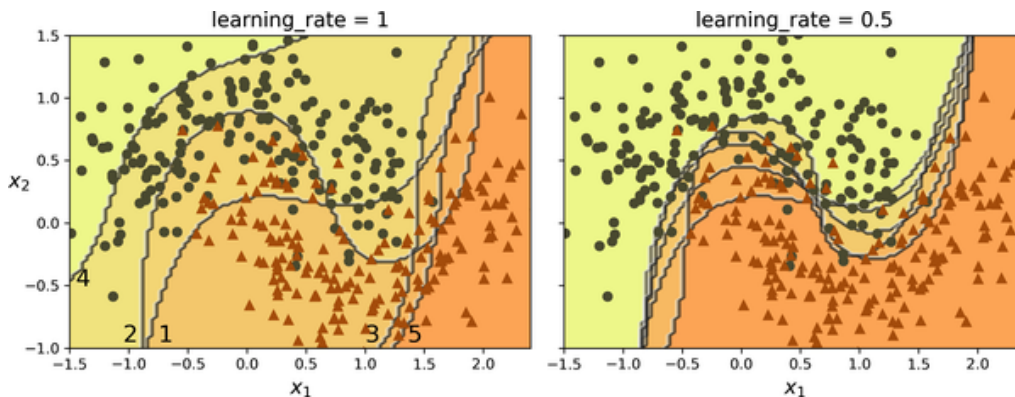


Figure 7-8. Decision boundaries of consecutive predictors

WARNING

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

$$\text{for } i = 1, 2, \dots, m$$
$$w^{(i)} \leftarrow \begin{cases} w^{(i)} & \text{if } \hat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} \exp(\alpha_j) & \text{if } \hat{y}_j^{(i)} \neq y^{(i)} \end{cases}$$

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

$$\hat{y}(\mathbf{x}) = \underset{k}{\operatorname{argmax}} \sum_{j=1}^N \alpha_j \quad \text{where } N \text{ is the number of predictors}$$

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 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)
```

TIP

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 = 3x2 + Gaussian noise

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)

```

And then we'll 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.

You can use Scikit-Learn's `GradientBoostingRegressor` class to train GBRT ensembles more easily (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

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3,

```

learning_rate=1.0, random_state=42)

gbdt.fit(X, y)

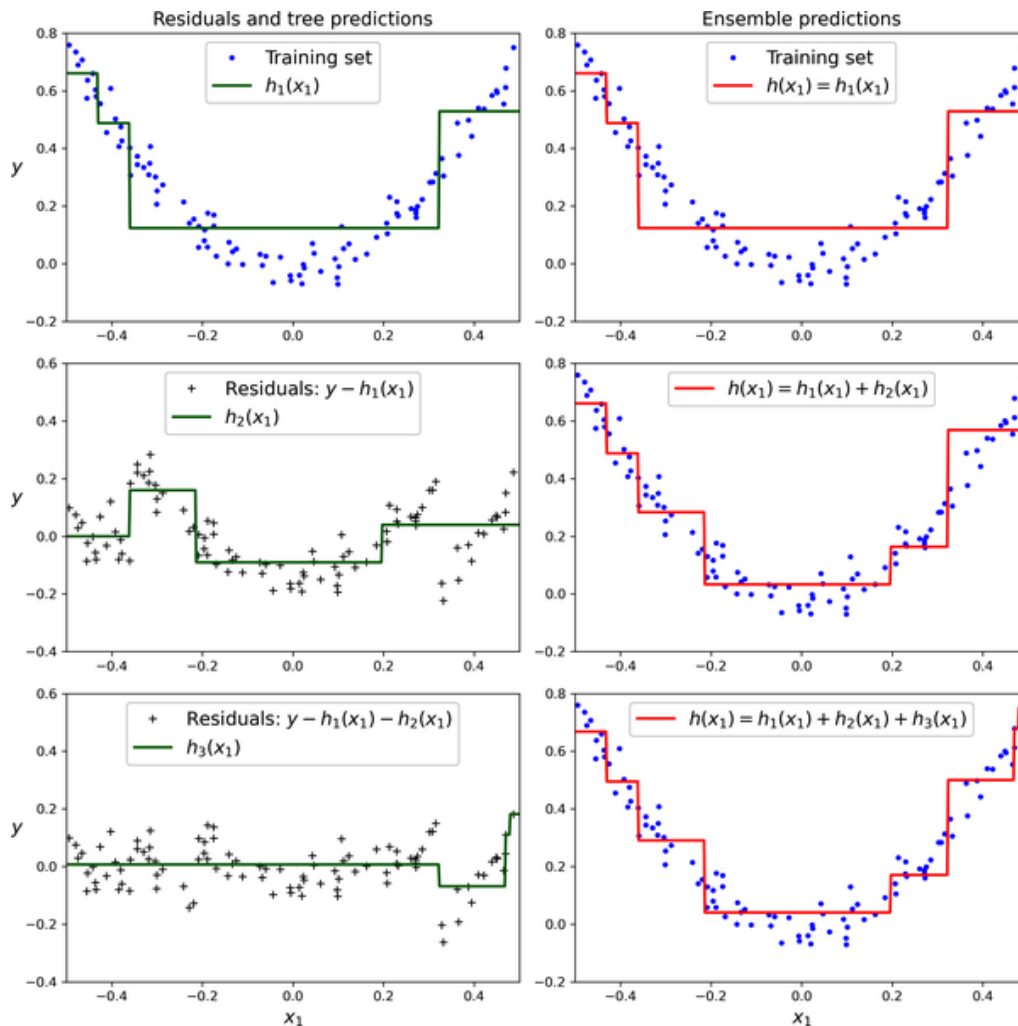


Figure 7-9. In this depiction of gradient boosting, the first predictor (top left) is trained normally, then each consecutive predictor (middle left and lower left) is trained on the previous predictor's residuals; the right column shows the resulting ensemble's predictions

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.

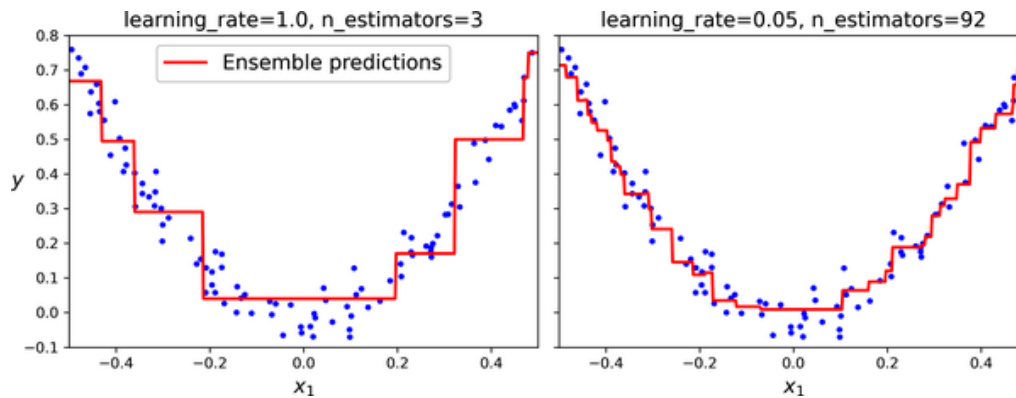


Figure 7-10. GBRT ensembles with not enough predictors (left) and just enough (right)

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:

```
gbrt_best = GradientBoostingRegressor(
    max_depth=2, learning_rate=0.05, n_estimators=500,
    n_iter_no_change=10, random_state=42)
gbrt_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:

```
>>> gbrt_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 input 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 $O(b \times m)$ instead of $O(n \times m \times \log(m))$, where b is the number of bins, 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're similar to `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 always 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, scaler, or a one-hot encoder, so 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.

TIP

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 a variety of random forest algorithms, including 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 (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).

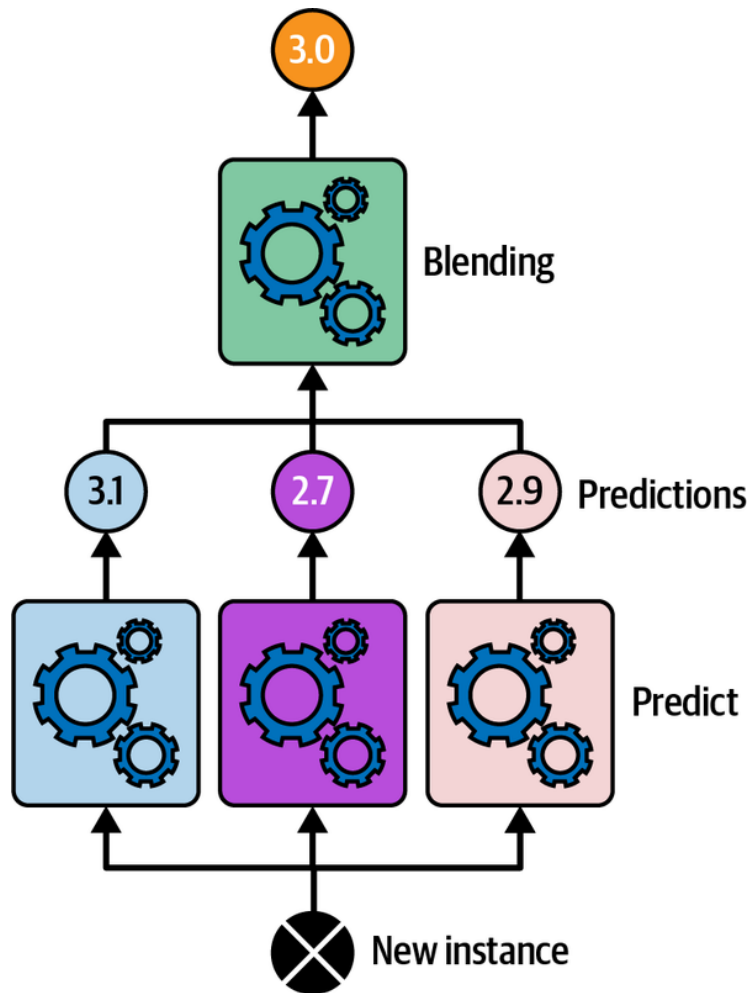


Figure 7-11. Aggregating predictions using a blending predictor

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), and use 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.

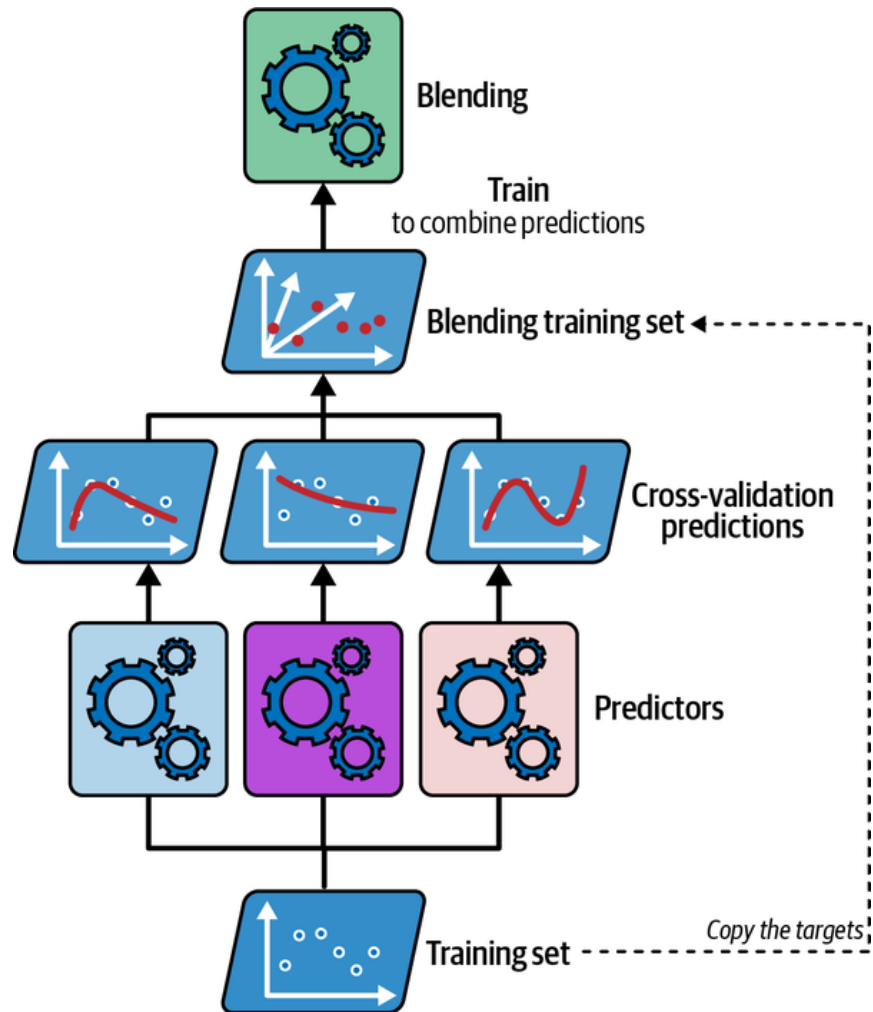


Figure 7-12. Training the blender in a stacking ensemble

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 by doing this, but it will cost you in both training time and system complexity.

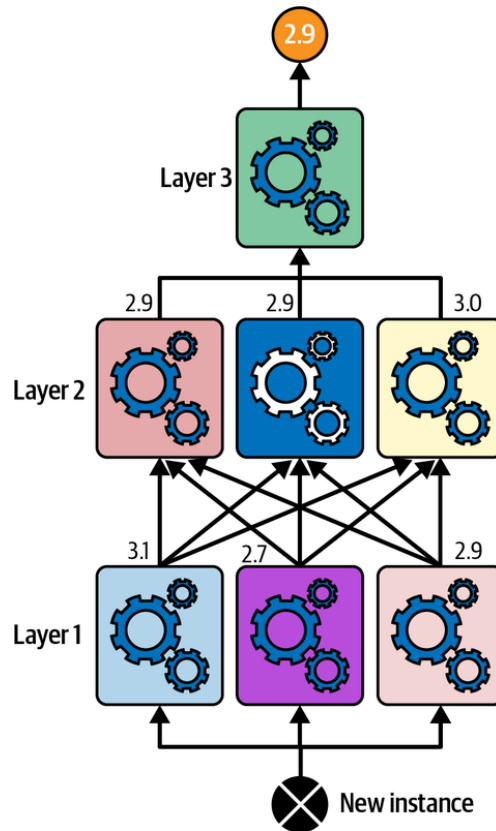


Figure 7-13. Predictions in a multilayer stacking ensemble

Scikit-Learn provides two classes for stacking ensembles: `StackingClassifier` and `StackingRegressor`. For example, we can replace the `VotingClassifier` we 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; if not it will fall back to `decision_function()` or, as a last resort, 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 for 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 ensembles more random than regular random forests? How can this extra randomness help? Are extra-trees classifiers 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 dataset (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>.

-
- 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.
 - 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`.
 - 7 As m grows, this ratio approaches $1 - \exp(-1) \approx 63\%$.
 - 8 Gilles Louppe and Pierre Geurts, “Ensembles on Random Patches”, *Lecture Notes in Computer Science* 7523 (2012): 346–361.
 - 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.
 - 12 Pierre Geurts et al., “Extremely Randomized Trees”, *Machine Learning* 63, no. 1 (2006): 3–42.
 - 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.
 - 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.
 - 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.
 - 18 David H. Wolpert, “Stacked Generalization”, *Neural Networks* 5, no. 2 (1992): 241–259.

Chapter 5. Support Vector Machines

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 you 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*.

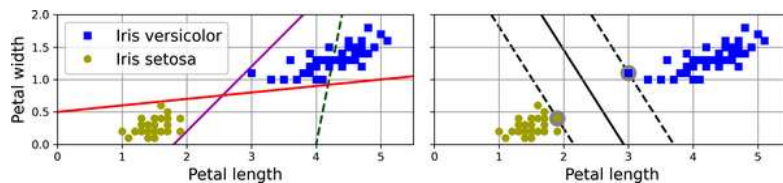


Figure 5-1. Large margin classification

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**).

WARNING

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.

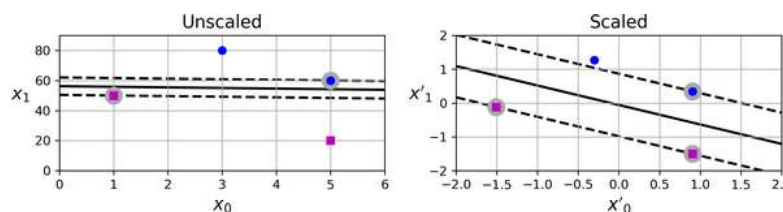


Figure 5-2. Sensitivity to feature scales

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 the model will probably not generalize as well.

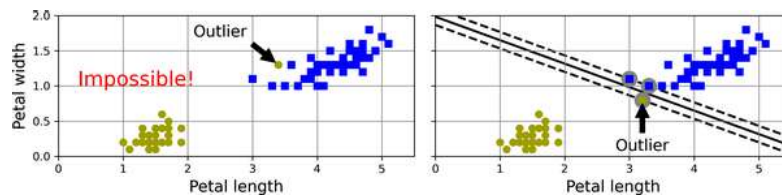


Figure 5-3. Hard margin sensitivity to outliers

To avoid these issues, we need to 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 the regularization hyperparameter 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 $C=1$.

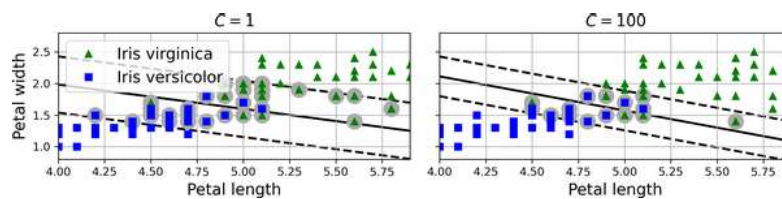


Figure 5-4. Large margin (left) versus fewer margin violations (right)

TIP

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 virginica*, 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 `LogisticRegression`, `LinearSVC` doesn’t 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` hyperparameter 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 we did in [Chapter 4](#)); in some cases this can result in a linearly separable dataset. Consider the lefthand 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.

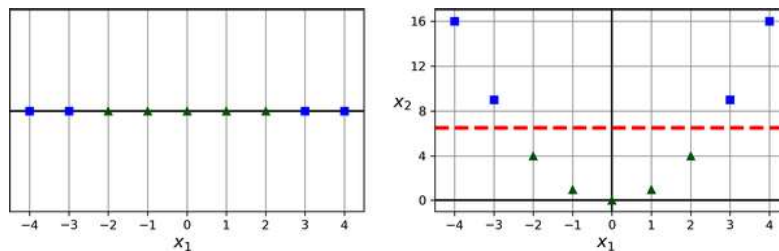


Figure 5-5. Adding features to make a dataset linearly separable

To implement this idea using Scikit-Learn, you can create a pipeline containing a `PolynomialFeatures` transformer (discussed in “[Polynomial Regression](#)”), followed by a `StandardScaler` and a `LinearSVC` classifier. Let’s test this on the moons dataset, 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)
```

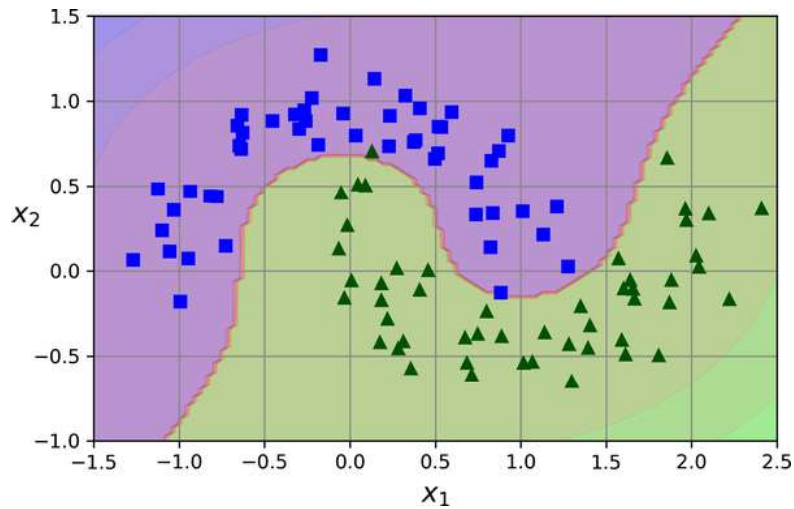


Figure 5-6. Linear SVM classifier using polynomial features

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, 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.

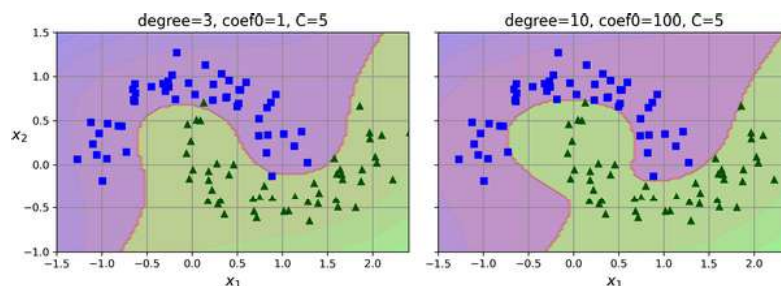


Figure 5-7. SVM classifiers with a polynomial kernel

TIP

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 from earlier and add two landmarks to it at $x_1 = -2$ and $x_1 = 1$ (see the left plot in [Figure 5-8](#)). Next, we'll 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.

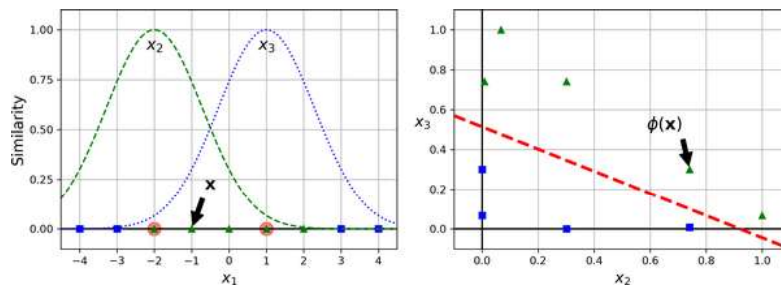


Figure 5-8. Similarity features using the Gaussian RBF

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 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).

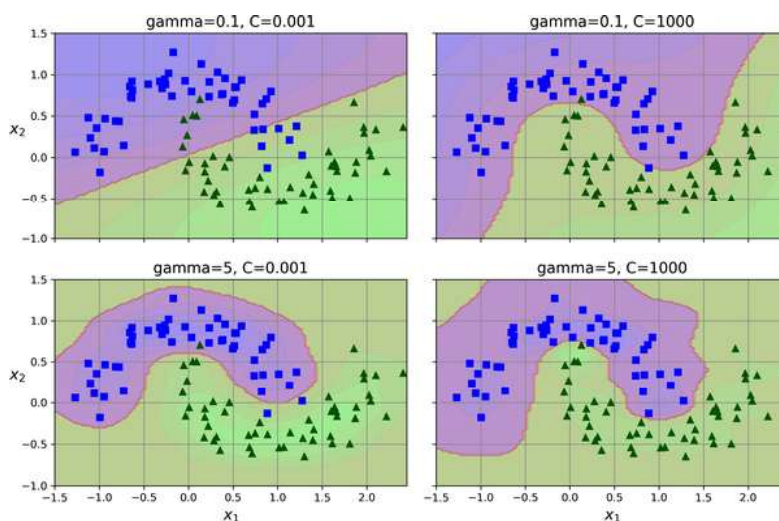


Figure 5-9. SVM classifiers using an RBF kernel

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).

TIP

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), so 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.

The `SGDClassifier` class also performs large margin classification by default, and its hyperparameters—especially the regularization hyperparameters (`alpha` and `penalty`) and the `learning_rate`—can be adjusted to produce similar results as the linear SVMs. 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., for out-of-core learning). Moreover, it scales very well, as its computational complexity is $O(m \times n)$. [Table 5-1](#) compares Scikit-Learn's SVM classification classes.

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 algorithms 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 ($\epsilon = 1.2$).

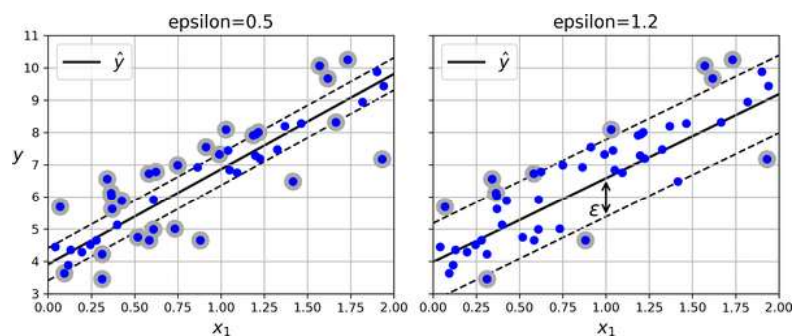


Figure 5-10. SVM regression

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).

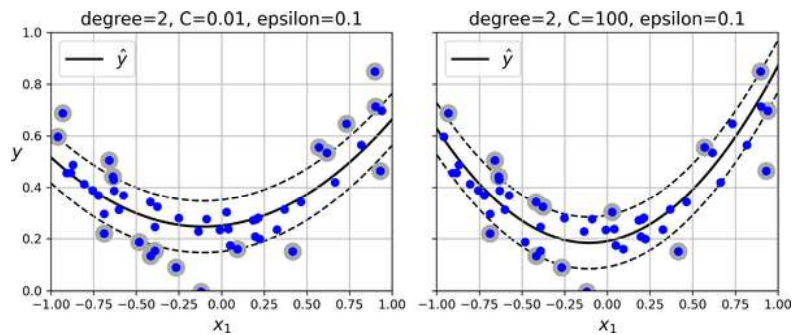


Figure 5-11. SVM regression using a second-degree polynomial kernel

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).

NOTE

SVMs can also be used for novelty detection, as you 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

A linear SVM classifier 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 [LogisticRegression](#) (discussed in [Chapter 4](#)).

NOTE

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? This requires finding the weights vector \mathbf{w} and the bias term b that make the street, or 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 $+1$. In 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. In 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.

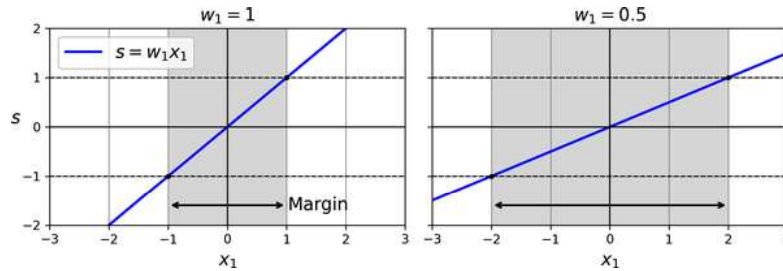


Figure 5-12. A smaller weight vector results in a larger margin

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

$$\begin{aligned} & \underset{\mathbf{w}, b}{\text{minimize}} && \frac{1}{2} \mathbf{w}^\top \mathbf{w} \\ & \text{subject to} && t^{(i)} (\mathbf{w}^\top \mathbf{x}^{(i)} + b) \geq 1 \quad \text{for } i = 1, 2, \dots, m \end{aligned}$$

NOTE

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 trade-off between these two objectives. This gives us the constrained optimization problem in [Equation 5-2](#).

Equation 5-2. Soft margin linear SVM classifier objective

$$\begin{aligned} & \underset{\mathbf{w}, b, \zeta}{\text{minimize}} && \frac{1}{2} \mathbf{w}^\top \mathbf{w} + C \sum_{i=1}^m \zeta^{(i)} \\ & \text{subject to} && t^{(i)} (\mathbf{w}^\top \mathbf{x}^{(i)} + b) \geq 1 - \zeta^{(i)} \quad \text{and} \quad \zeta^{(i)} \geq 0 \quad \text{for } i = 1, 2, \dots, m \end{aligned}$$

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

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 0 if the output s of the decision function ($s = \mathbf{w}^T \mathbf{x} + b$) is greater than 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 0 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.

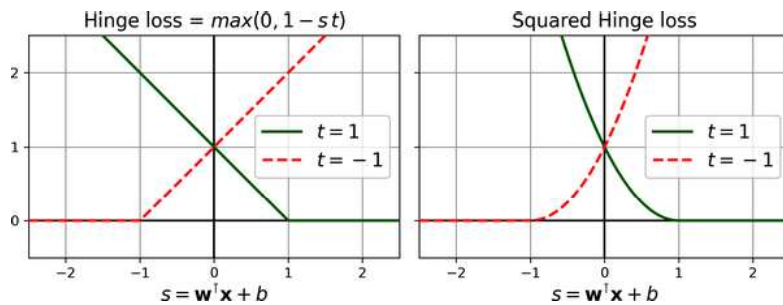


Figure 5-13. The hinge loss (left) and the squared hinge loss (right)

Next, we'll look at 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 [this chapter's notebook](#).

Equation 5-3. Dual form of the linear SVM objective

$$\underset{\alpha}{\text{minimize}} \quad \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)T} \mathbf{x}^{(j)} - \sum_{i=1}^m \alpha^{(i)} \text{subject to } \alpha^{(i)} \geq 0 \text{ for all } i = 1, 2, \dots, m \text{ and } \sum_{i=1}^m \alpha^{(i)} = 1$$

Once you find the vector $\hat{\alpha}$ that minimizes this equation (using a QP solver), use [Equation 5-4](#) to compute the $\hat{\mathbf{w}}$ 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

$$\hat{\mathbf{w}} = \sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} \mathbf{x}^{(i)}$$

$$\hat{b} = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left(t^{(i)} - \hat{\mathbf{w}}^T \mathbf{x}^{(i)} \right)$$

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 problem

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

$$\phi(\mathbf{x}) = \phi\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix}$$

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

$$\begin{aligned} \phi(\mathbf{a})^\top \phi(\mathbf{b}) &= \begin{pmatrix} a_1^2 \\ \sqrt{2}a_1a_2 \\ a_2^2 \end{pmatrix}^\top \begin{pmatrix} b_1^2 \\ \sqrt{2}b_1b_2 \\ b_2^2 \end{pmatrix} = a_1^2b_1^2 + 2a_1b_1a_2b_2 + a_2^2b_2^2 \\ &= (a_1b_1 + a_2b_2)^2 = \left(\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^\top \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}\right)^2 = (\mathbf{a}^\top \mathbf{b})^2 \end{aligned}$$

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 and 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 know about) the transformation ϕ . Equation 5-7 lists some of the most commonly used kernels.

Equation 5-7. Common kernels

Linear:	$K(\mathbf{a}, \mathbf{b}) = \mathbf{a}^\top \mathbf{b}$
Polynomial:	$K(\mathbf{a}, \mathbf{b}) = (\gamma \mathbf{a}^\top \mathbf{b} + r)^d$
Gaussian RBF:	$K(\mathbf{a}, \mathbf{b}) = \exp\left(-\gamma \ \mathbf{a} - \mathbf{b}\ ^2\right)$
Sigmoid:	$K(\mathbf{a}, \mathbf{b}) = \tanh(\gamma \mathbf{a}^\top \mathbf{b} + r)$

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 $K(\mathbf{a}, \mathbf{b}) = \phi(\mathbf{a})^\top \phi(\mathbf{b})$. 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 $\phi(\mathbf{x}^{(i)})$. In fact, $\hat{\mathbf{w}}$ must have the same number of dimensions as $\phi(\mathbf{x}^{(i)})$, which may be huge or even infinite, so you can't compute it. But how can you make predictions without knowing $\hat{\mathbf{w}}$? Well, the good news is that you can plug the formula for $\hat{\mathbf{w}}$ 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

$$\begin{aligned} h_{\hat{\mathbf{w}}, \hat{b}}(\phi(\mathbf{x}^{(n)})) &= \hat{\mathbf{w}}^\top \phi(\mathbf{x}^{(n)}) + \hat{b} = \left(\sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} \phi(\mathbf{x}^{(i)}) \right)^\top \phi(\mathbf{x}^{(n)}) + \hat{b} \\ &= \sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} \left(\phi(\mathbf{x}^{(i)})^\top \phi(\mathbf{x}^{(n)}) \right) + \hat{b} \\ &= \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \hat{\alpha}^{(i)} t^{(i)} K(\mathbf{x}^{(i)}, \mathbf{x}^{(n)}) + \hat{b} \end{aligned}$$

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 \hat{b} (Equation 5-9).

Equation 5-9. Using the kernel trick to compute the bias term

$$\begin{aligned} \hat{b} &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left(t^{(i)} - \hat{\mathbf{w}}^\top \phi(\mathbf{x}^{(i)}) \right) = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left(t^{(i)} - \left(\sum_{j=1}^m \hat{\alpha}^{(j)} t^{(j)} \phi(\mathbf{x}^{(j)}) \right)^\top \phi(\mathbf{x}^{(i)}) \right) \\ &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left(t^{(i)} - \sum_{\substack{j=1 \\ \hat{\alpha}^{(j)} > 0}}^m \hat{\alpha}^{(j)} t^{(j)} K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) \right) \end{aligned}$$

If you are starting to get a headache, that's perfectly normal: it's an unfortunate side effect of the kernel trick.

NOTE

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`, and `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 analyses 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 three 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](#), which you can load 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 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>.

-
- 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).
 - 3 Zeta (ζ) is the sixth letter of the Greek alphabet.
 - 4 To learn more about quadratic programming, you can start by reading Stephen Boyd and Lieven Vandenberghe's book *Convex Optimization* (Cambridge University Press) or watching Richard Brown's [series of video lectures](#).
 - 5 The objective function is convex, and the inequality constraints are continuously differentiable and convex functions.
 - 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 $\mathbf{a}^T \mathbf{b}$. 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.
 - 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.

Chapter 9. Unsupervised Learning Techniques

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 \mathbf{X} , but we do not have the labels \mathbf{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 new trends in 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 subclusters. That said, the dataset has two additional features (sepal length and width) that are 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).

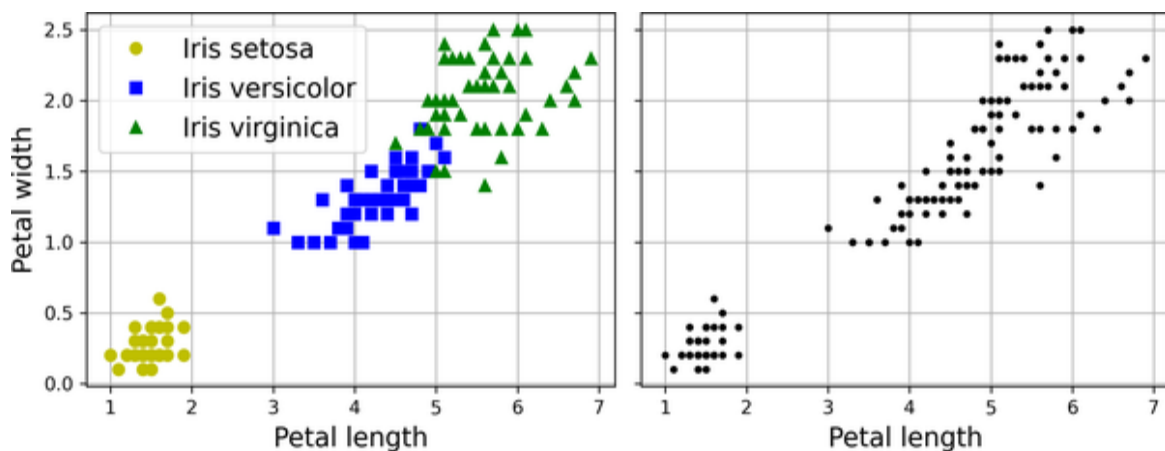


Figure 9-1. Classification (left) versus clustering (right)

Clustering is used in a wide variety of applications, including:

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.

Data analysis

When you analyze a new dataset, it can be helpful to run a clustering algorithm, and then analyze each cluster separately.

Dimensionality reduction

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. The new vector is typically much lower-dimensional than the original feature vector, but it can preserve enough information for further processing.

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.

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.

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.

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'd need to do is use the trained clustering model to find this image's cluster, and you could then simply return all the images from this cluster.

Image segmentation

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 an 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 the Lloyd–Forgy algorithm.

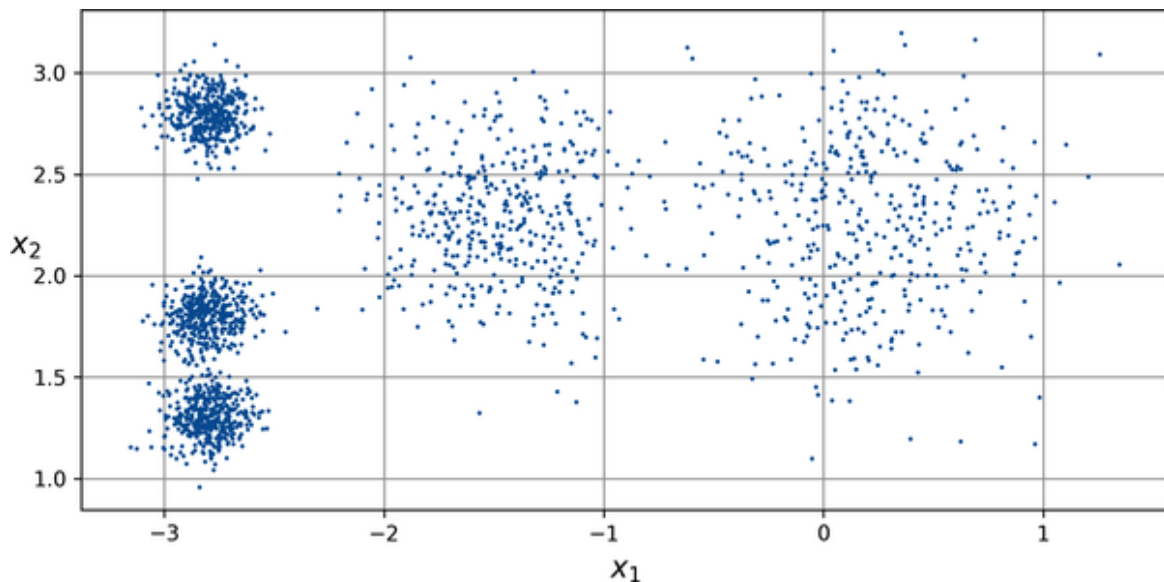


Figure 9-2. An unlabeled dataset composed of five blobs of instances

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; that'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 will be assigned to one of the five clusters. In the context of clustering, an instance's *label* is the index of the cluster to which the algorithm assigns this instance; 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 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.

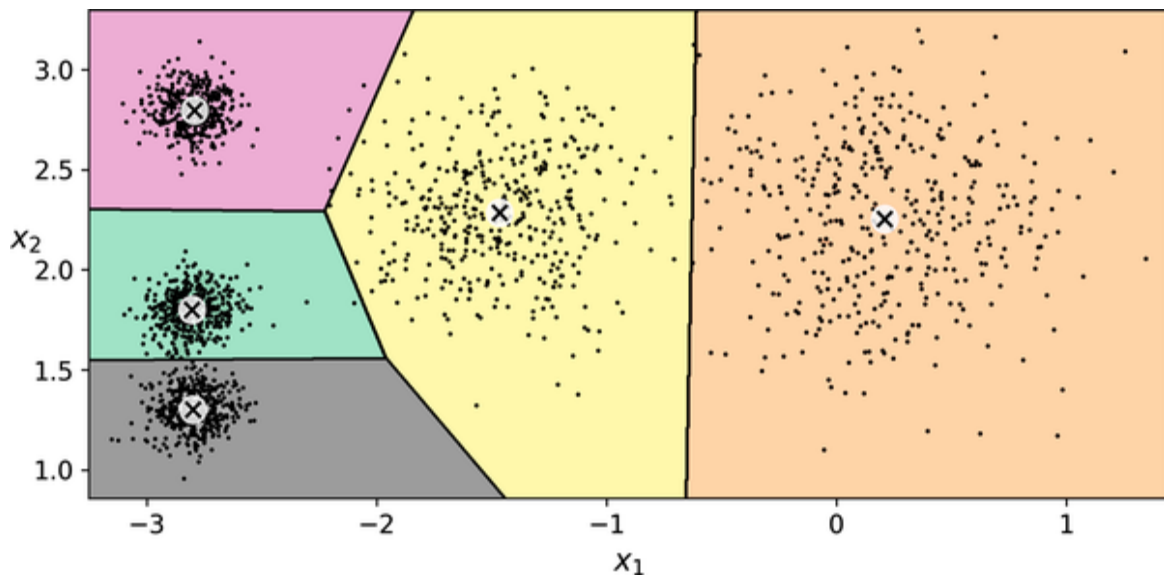


Figure 9-3. *k*-means decision boundaries (Voronoi tessellation)

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 or 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 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? 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 centroids 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.

NOTE

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.

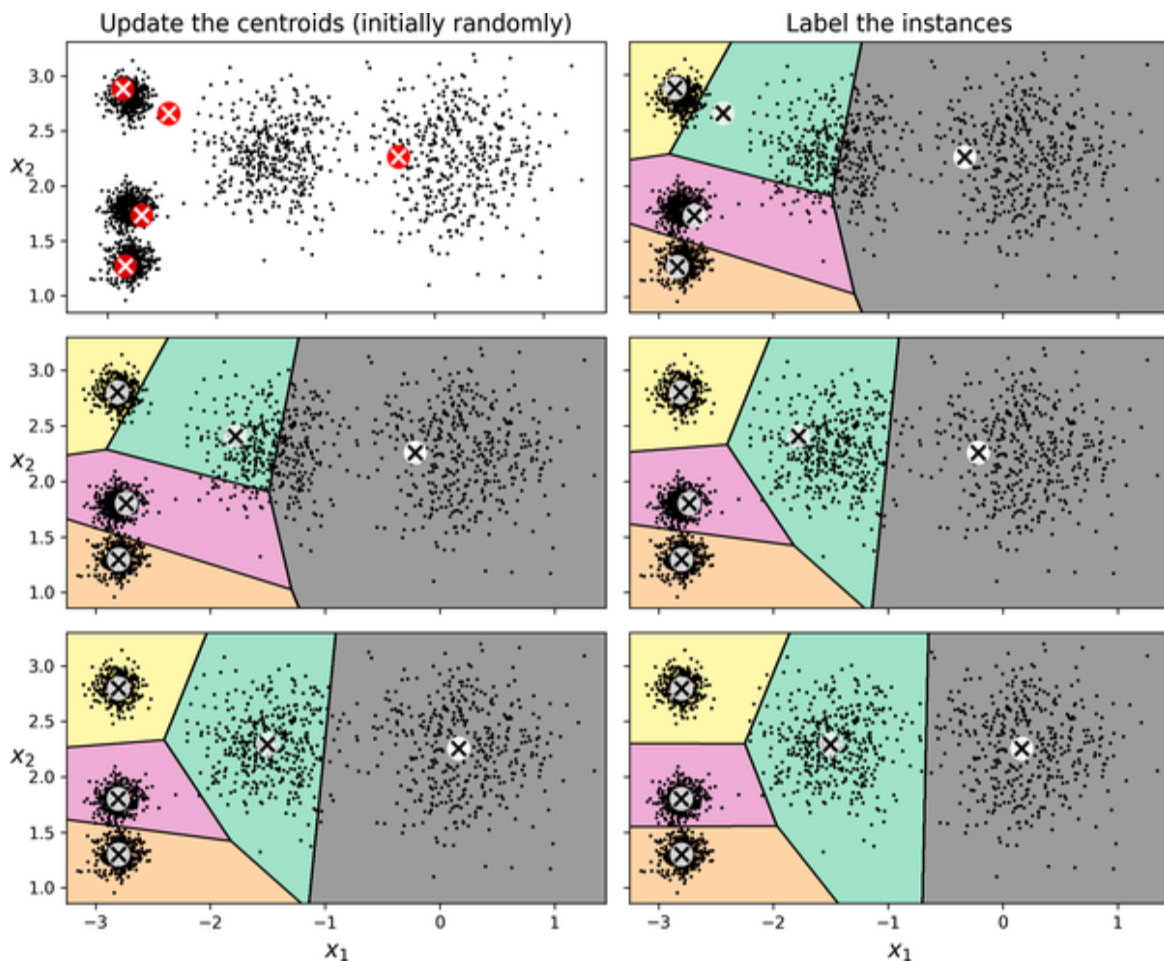


Figure 9-4. The k -means algorithm

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.

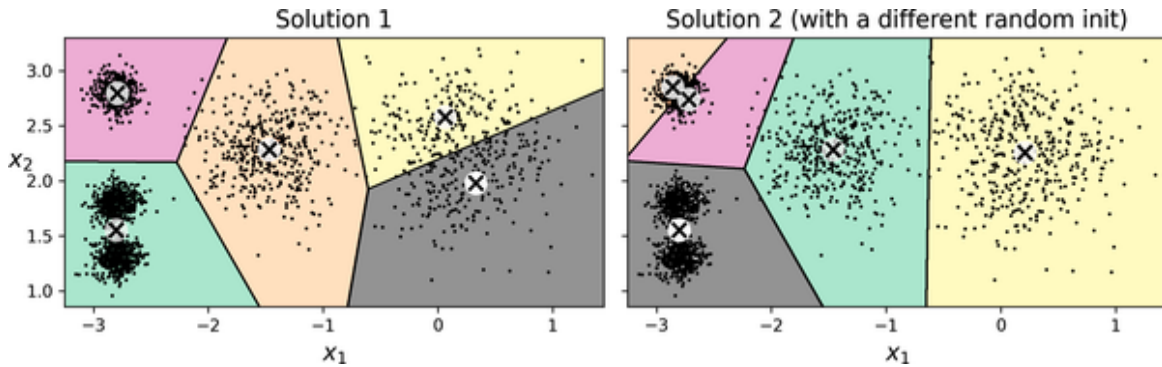


Figure 9-5. Suboptimal solutions due to unlucky centroid initializations

Let's take a 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 centroids. 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 (it's 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. The paper 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 $\mathbf{c}^{(1)}$, chosen uniformly at random from the dataset.
2. 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.
3. 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, and sometimes it can even slow down training significantly; it depends on the dataset. 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, which you can use just 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.

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.

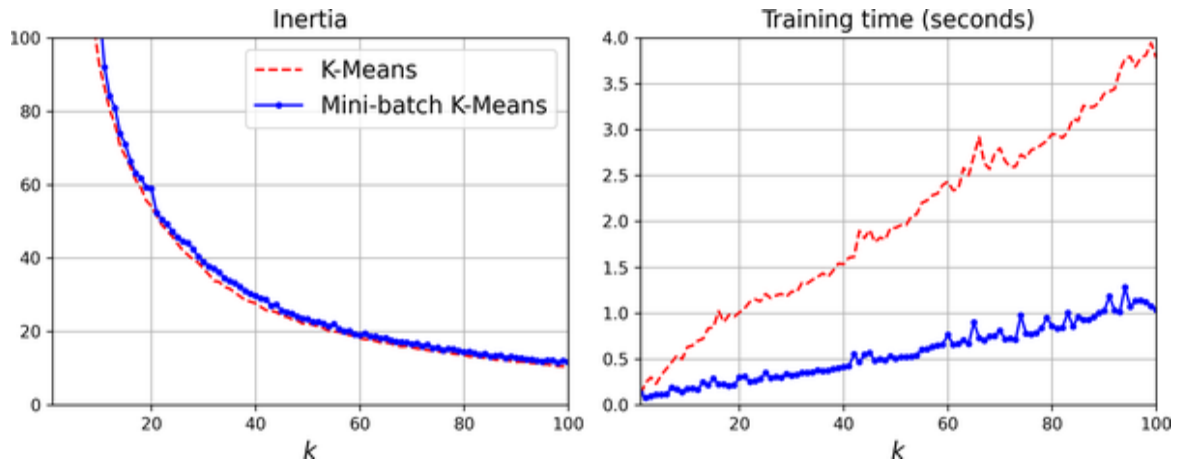


Figure 9-6. Mini-batch k -means has a higher inertia than k -means (left) but it is much faster (right), especially as k increases

Finding the optimal number of clusters

So far, we've 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 won't 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](#), for this dataset setting k to 3 or 8 results in fairly bad models.

You might be thinking that you could just pick the model with the lowest inertia. Unfortunately, it is not that simple. The inertia for $k=3$ is about 653.2, which is much higher than for $k=5$ (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 . When we do this, the curve often contains an inflexion point called the *elbow* (see [Figure 9-8](#)).

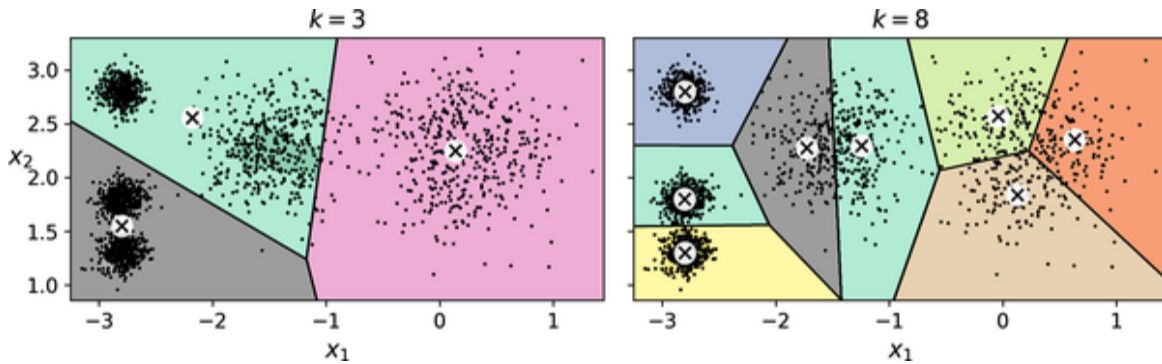


Figure 9-7. Bad choices for the number of clusters: when k is too small, separate clusters get merged (left), and when k is too large, some clusters get chopped into multiple pieces (right)

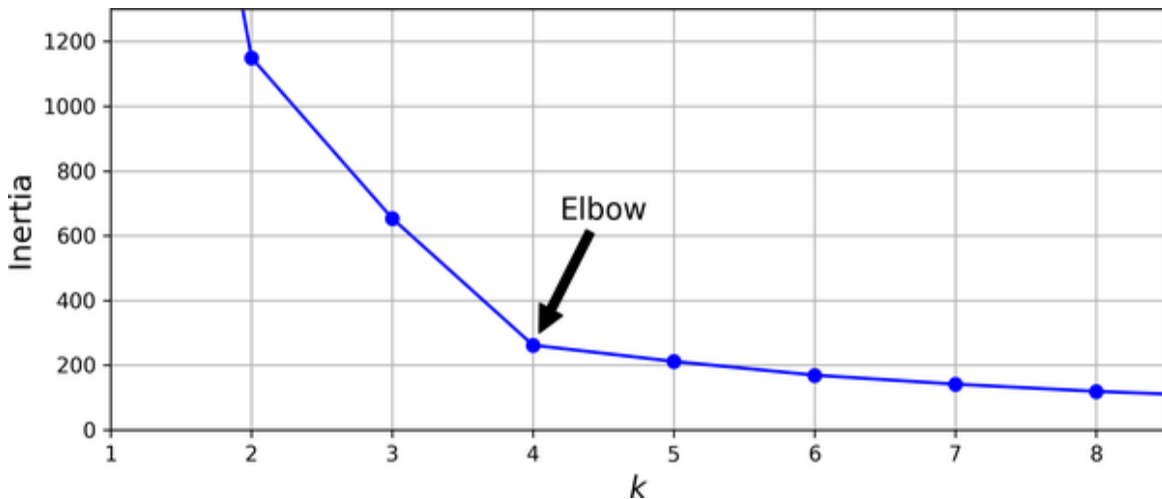


Figure 9-8. Plotting the inertia as a function of the number of clusters k

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, we might think 4 was 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 (but also more computationally expensive) approach 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; 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](#)).

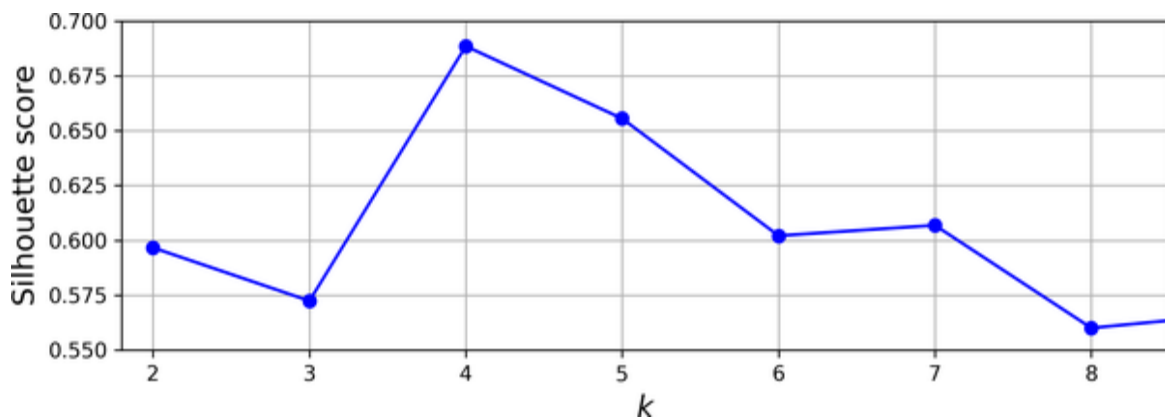


Figure 9-9. Selecting the number of clusters k using the silhouette score

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 highlights 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 we plot every instance's silhouette coefficient, sorted by the clusters 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).

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. Here 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.

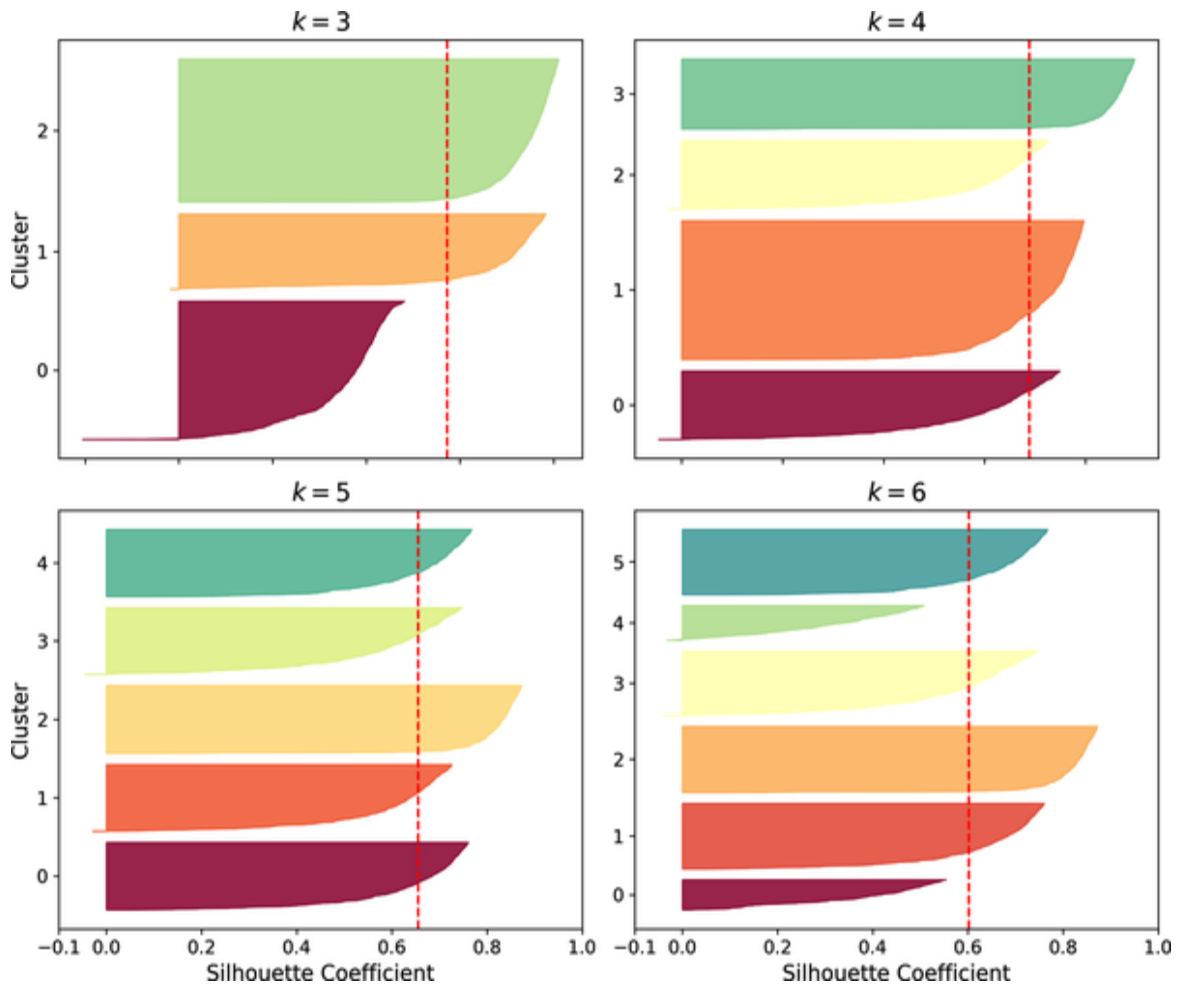


Figure 9-10. Analyzing the silhouette diagrams for various values of k

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.

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.

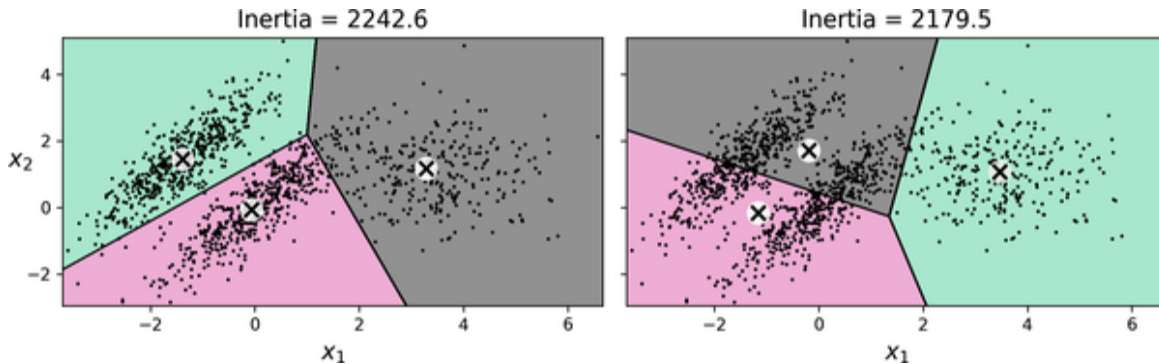


Figure 9-11. *k*-means fails to cluster these ellipsoidal blobs properly

TIP

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.

We'll start by importing the Pillow package (successor to the Python Imaging Library, PIL), which we'll then use 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, which only have one), 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 additional light frequencies (like infrared)).

The following code reshapes the array to get a long list of RGB colors, then it clusters these colors using *k*-means with eight clusters. 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])
```

We can then 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, we'll find the image closest to the centroid. We'll 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 the 50 representative images.

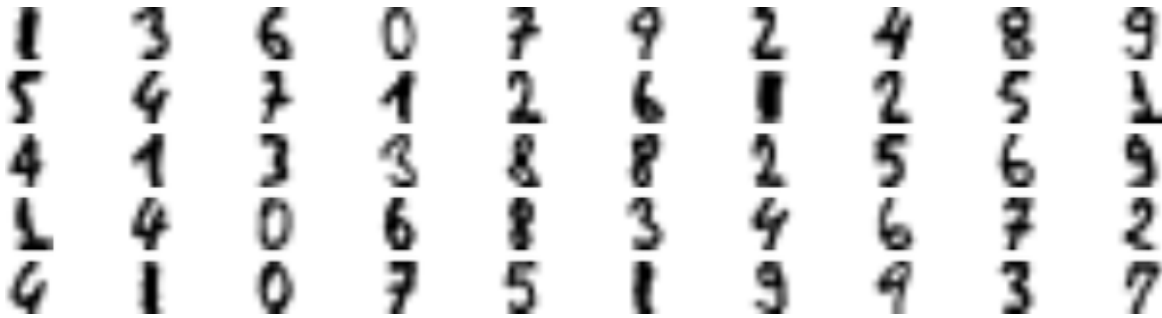


Figure 9-13. Fifty representative digit images (one per cluster)

Let's look at each image and manually label them:

```
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% of 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
```

TIP

Scikit-Learn also offers two classes that can propagate labels automatically: `LabelSpreading` and `LabelPropagation` in the `sklearn.semi_supervised` package. Both classes 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, and 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., where 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

The *density-based spatial clustering of applications with noise* (DBSCAN) 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 5](#):

```
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.

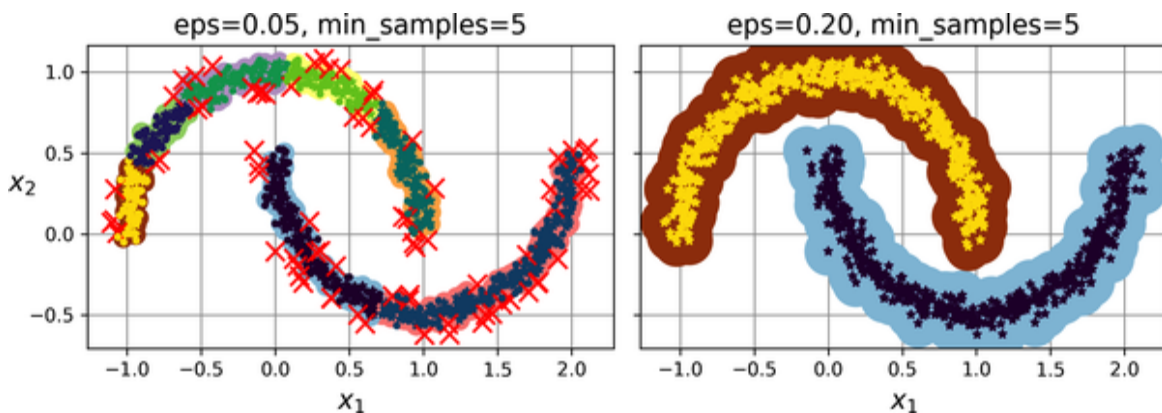


Figure 9-14. DBSCAN clustering using two different neighborhood radiuses

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 clusters 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])
```

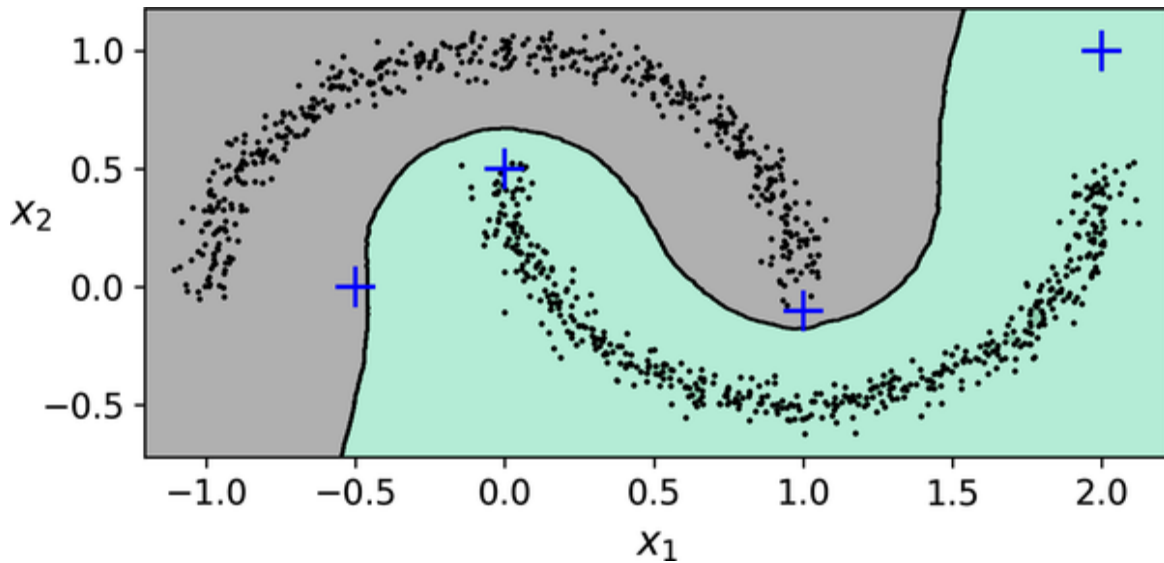


Figure 9-15. Decision boundary between two clusters

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, however, 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.

TIP

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 balanced iterative reducing and clustering using hierarchies (BIRCH) 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 $\boldsymbol{\Sigma}^{(j)}$. This is noted $\mathbf{x}^{(i)} \sim \mathcal{N}(\boldsymbol{\mu}^{(j)}, \boldsymbol{\Sigma}^{(j)})$.

So what can you do with such a model? Well, given the dataset \mathbf{X} , you typically want to start by estimating the weights $\boldsymbol{\phi}$ and all the distribution parameters $\boldsymbol{\mu}^{(1)}$ to $\boldsymbol{\mu}^{(k)}$ and $\boldsymbol{\Sigma}^{(1)}$ to $\boldsymbol{\Sigma}^{(k)}$. 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 ($\boldsymbol{\mu}^{(1)}$ to $\boldsymbol{\mu}^{(k)}$), but also their size, shape, and orientation ($\boldsymbol{\Sigma}^{(1)}$ to $\boldsymbol{\Sigma}^{(k)}$), 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.

WARNING

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.

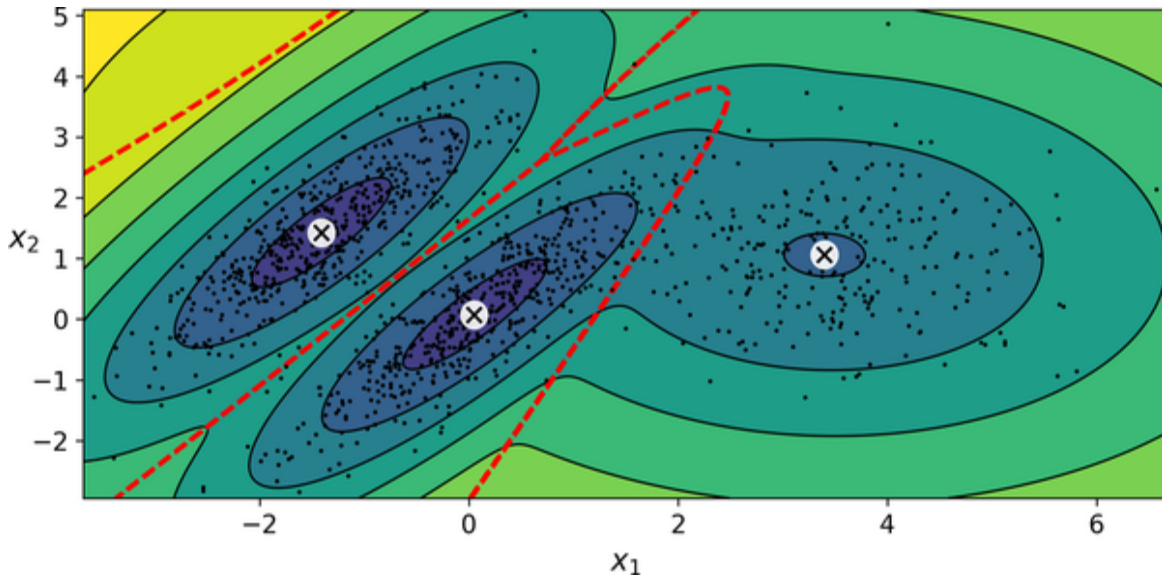


Figure 9-16. Cluster means, decision boundaries, and density contours of a trained Gaussian mixture model

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".

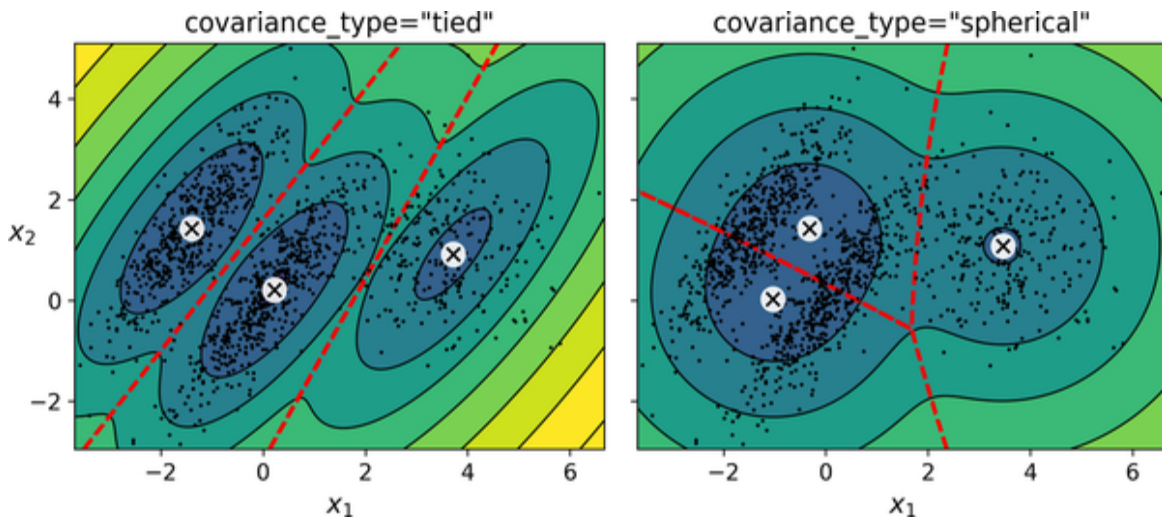


Figure 9-17. Gaussian mixtures for tied clusters (left) and spherical clusters (right)

NOTE

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. We'll see how in the next section.

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.

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.

TIP

Gaussian mixture models try to fit all the data, including the outliers; 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).

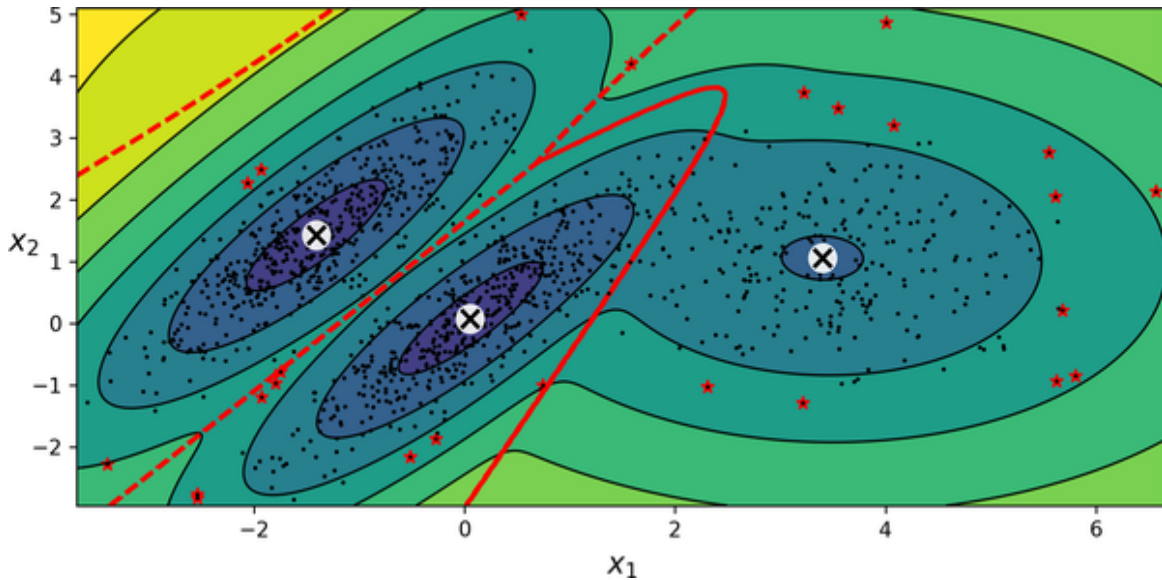


Figure 9-18. Anomaly detection using a Gaussian mixture model

Just like *k*-means, the `GaussianMixture` algorithm requires you to specify the number of clusters. So how can you find that number?

Selecting the Number of Clusters

With *k*-means, you can 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)

$$BIC = \log(m)p - 2 \log(\hat{\mathcal{L}})$$

$$AIC = 2p - 2 \log(\hat{\mathcal{L}})$$

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.

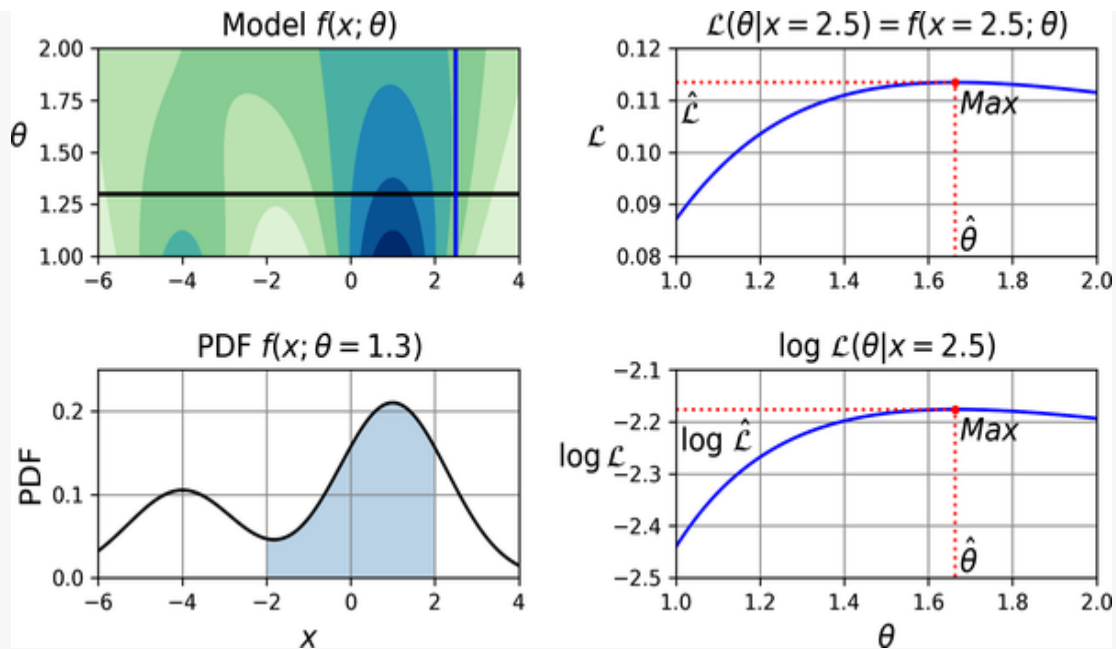


Figure 9-19. A model's parametric function (top left), and some derived functions: a PDF (lower left), a likelihood function (top right), and a log likelihood function (lower right)

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-right 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.

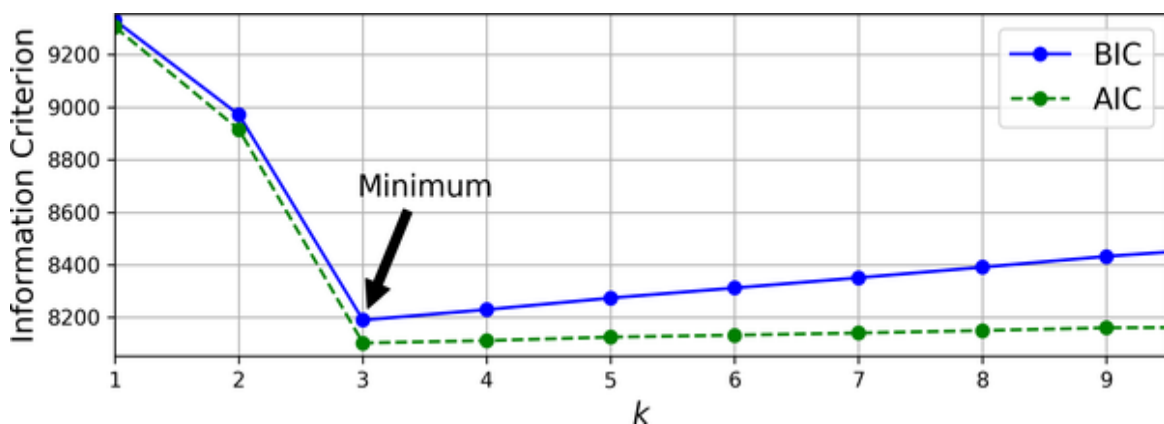


Figure 9-20. AIC and BIC for different numbers of clusters k

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](#)).

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.

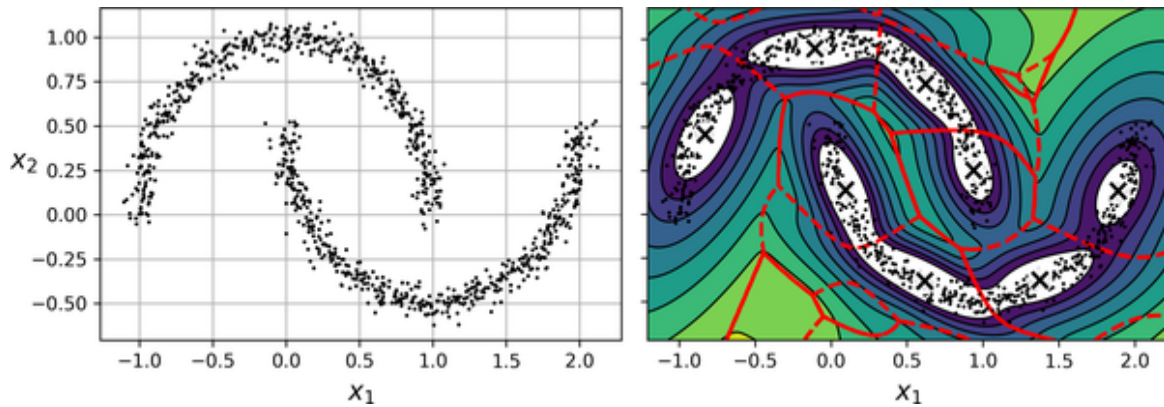


Figure 9-21. Fitting a Gaussian mixture to nonellipsoidal clusters

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. Forty 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 will 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

Chapter 8. Dimensionality Reduction

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 you 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.

WARNING

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. 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, data visualization 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 first 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 locally linear embedding (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.

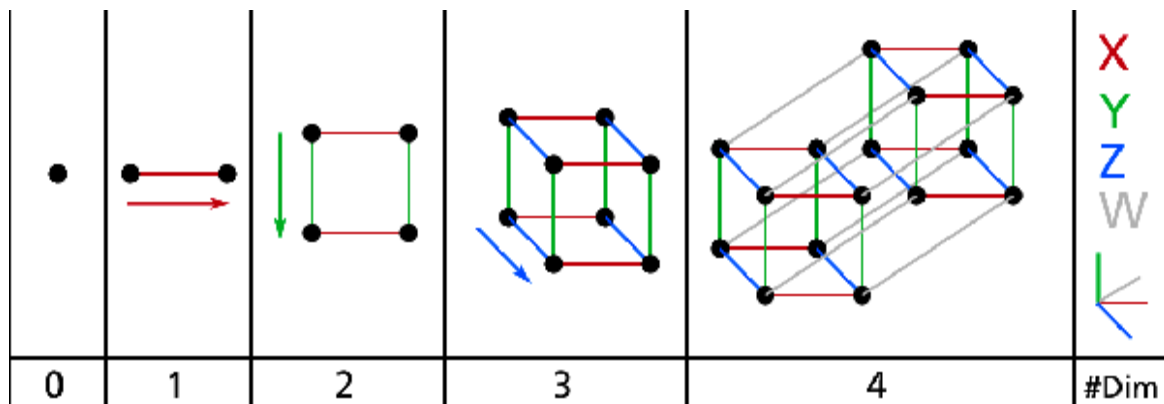


Figure 8-1. Point, segment, square, cube, and tesseract (0D to 4D hypercubes)²

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 3D unit cube, the average distance will be roughly 0.66. But what about two points picked randomly in a 1,000,000-dimensional unit 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 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.

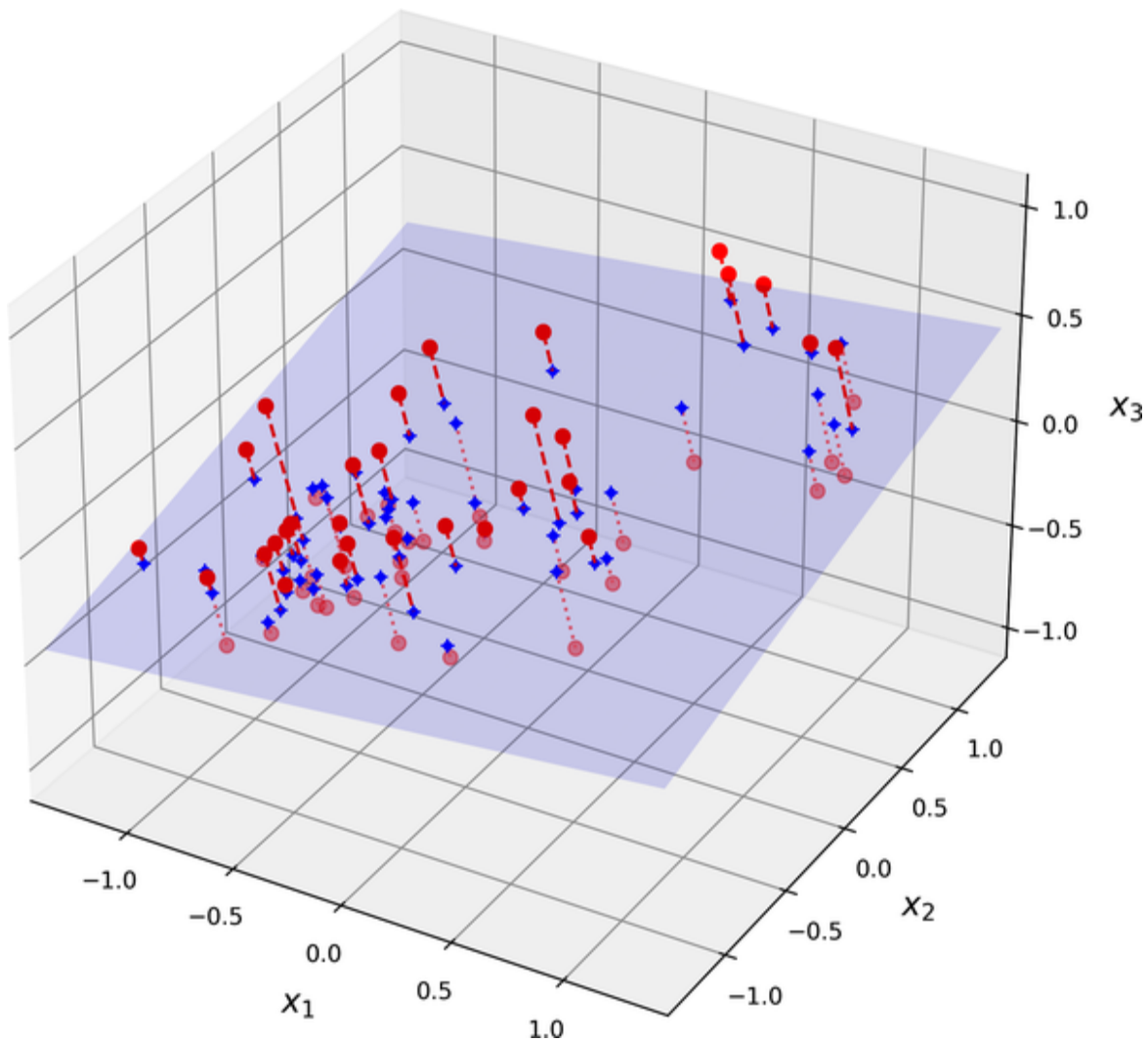


Figure 8-2. A 3D dataset lying close to a 2D subspace

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.

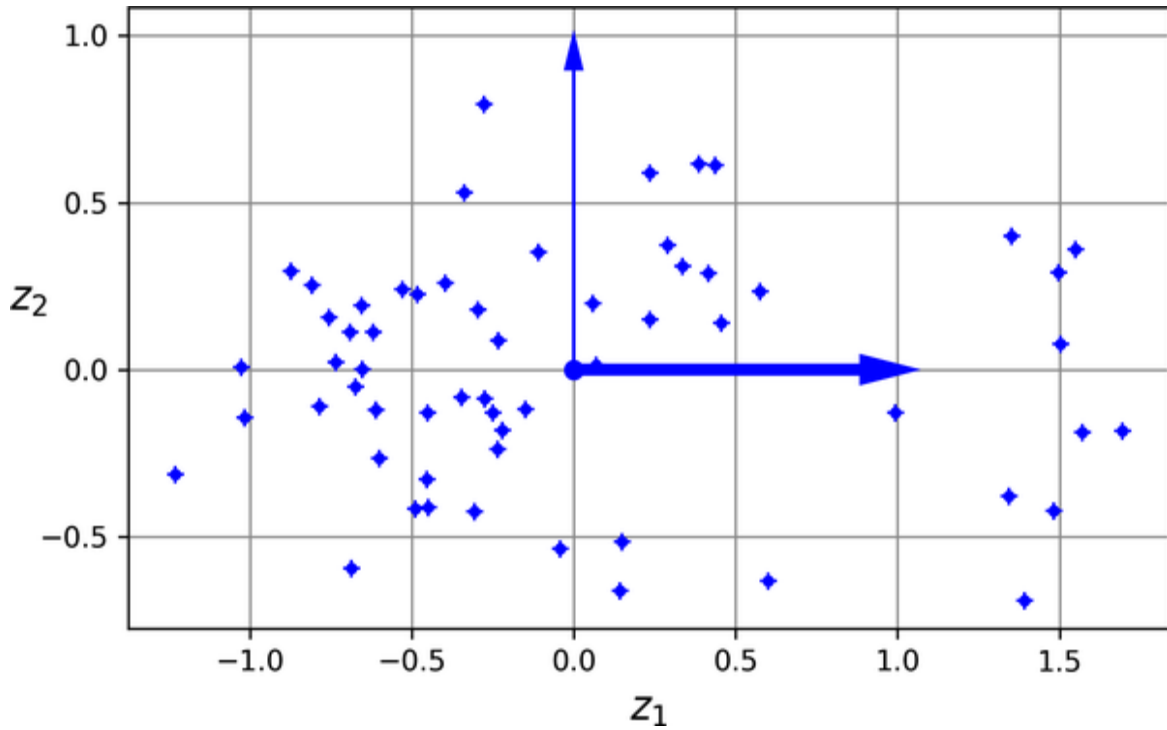


Figure 8-3. The new 2D dataset after projection

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](#).

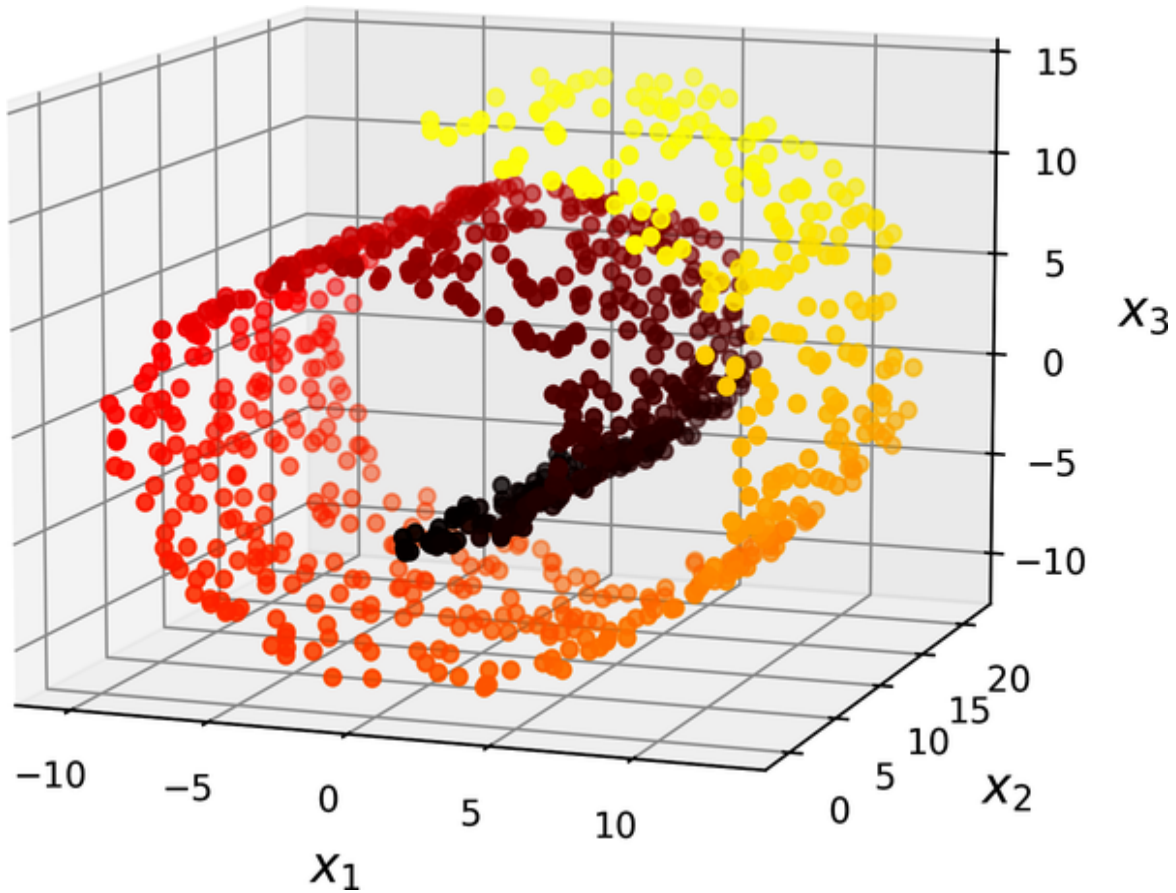


Figure 8-4. Swiss roll dataset

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.

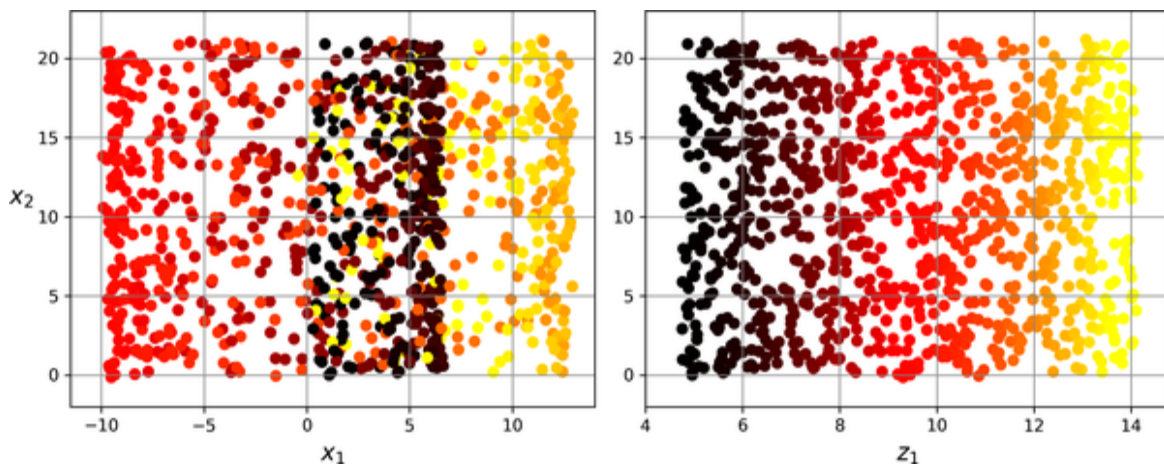


Figure 8-5. Squashing by projecting onto a plane (left) versus unrolling the Swiss roll (right)

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.

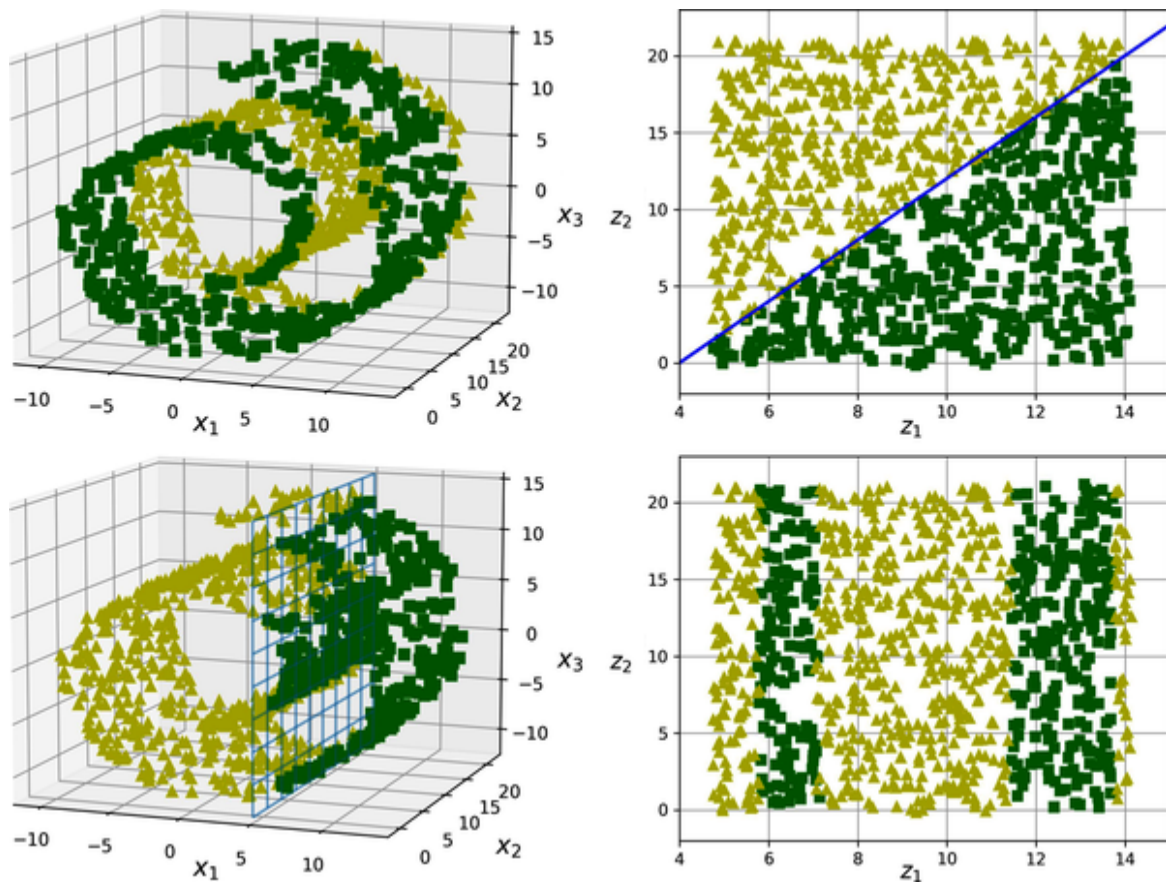


Figure 8-6. The decision boundary may not always be simpler with lower dimensions

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).

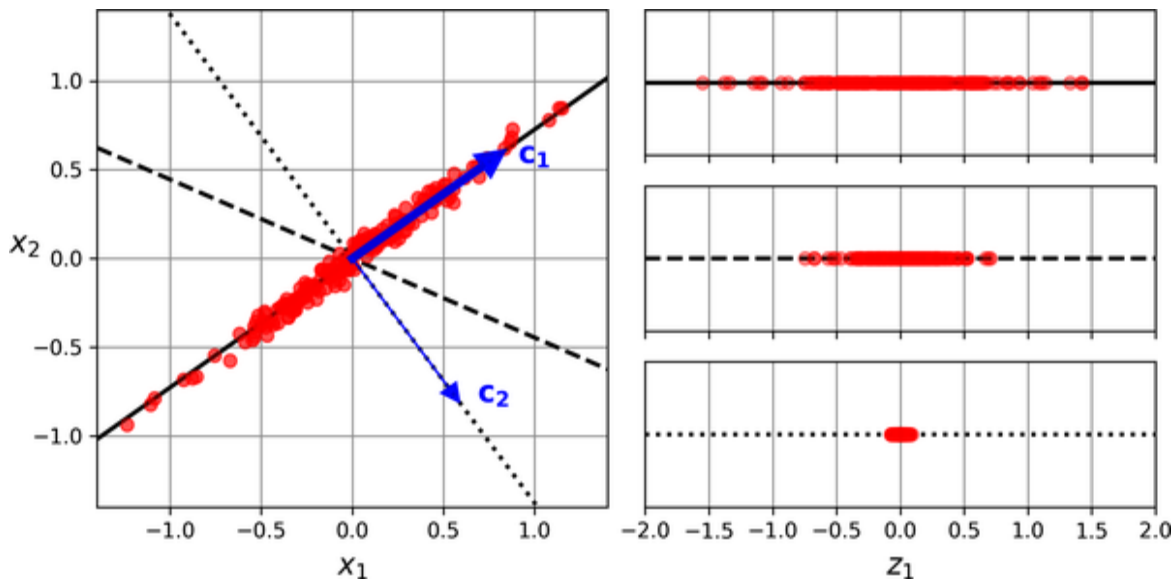


Figure 8-7. Selecting the subspace on which to project

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

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 the 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.

NOTE

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 $\mathbf{U} \mathbf{\Sigma} \mathbf{V}^T$, where \mathbf{V} contains the unit vectors that define all the principal components that you are looking for, as shown in [Equation 8-1](#).

Equation 8-1. Principal components matrix

$$\mathbf{V} = \begin{pmatrix} | & | & \cdots & | \\ \mathbf{c}_1 & \mathbf{c}_2 & \cdots & \mathbf{c}_n \\ | & | & & | \end{pmatrix}$$

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]
```

WARNING

PCA assumes that the dataset is centered around the origin. As you 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 us 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—say, 95% (An exception to this rule, of course, is if 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 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’s 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.

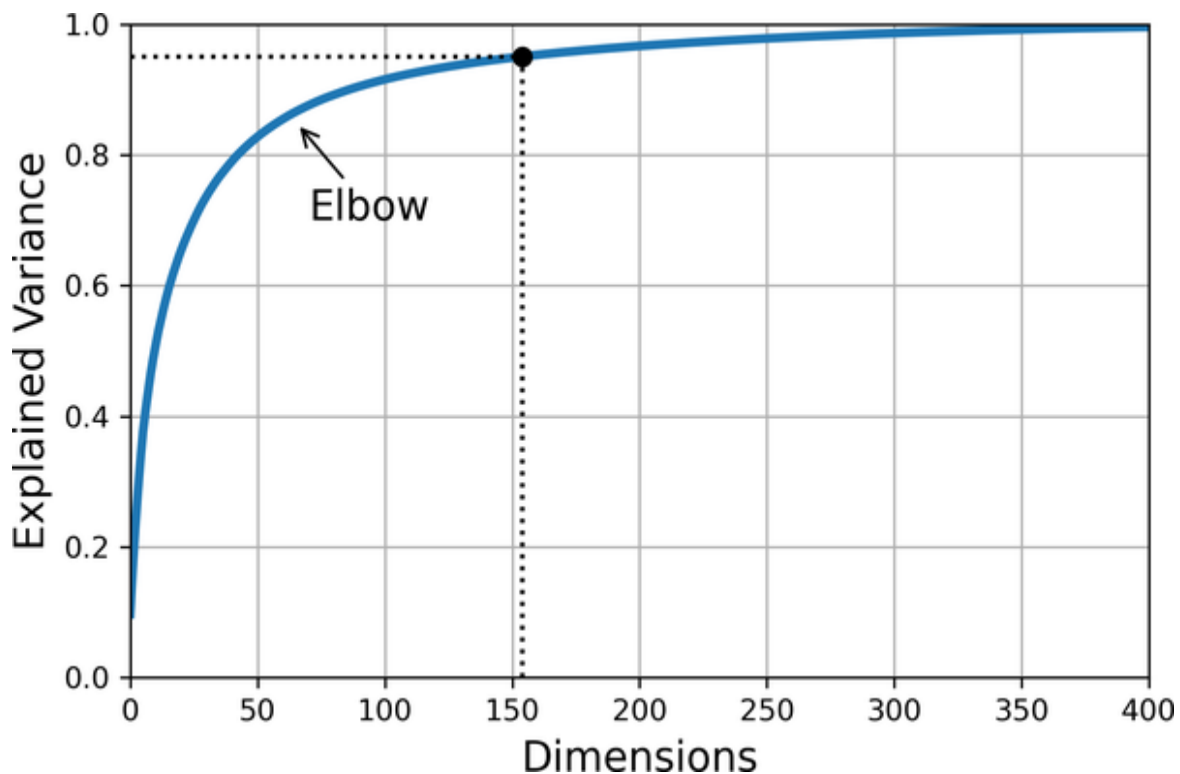


Figure 8-8. Explained variance as a function of the number of dimensions

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 2 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.

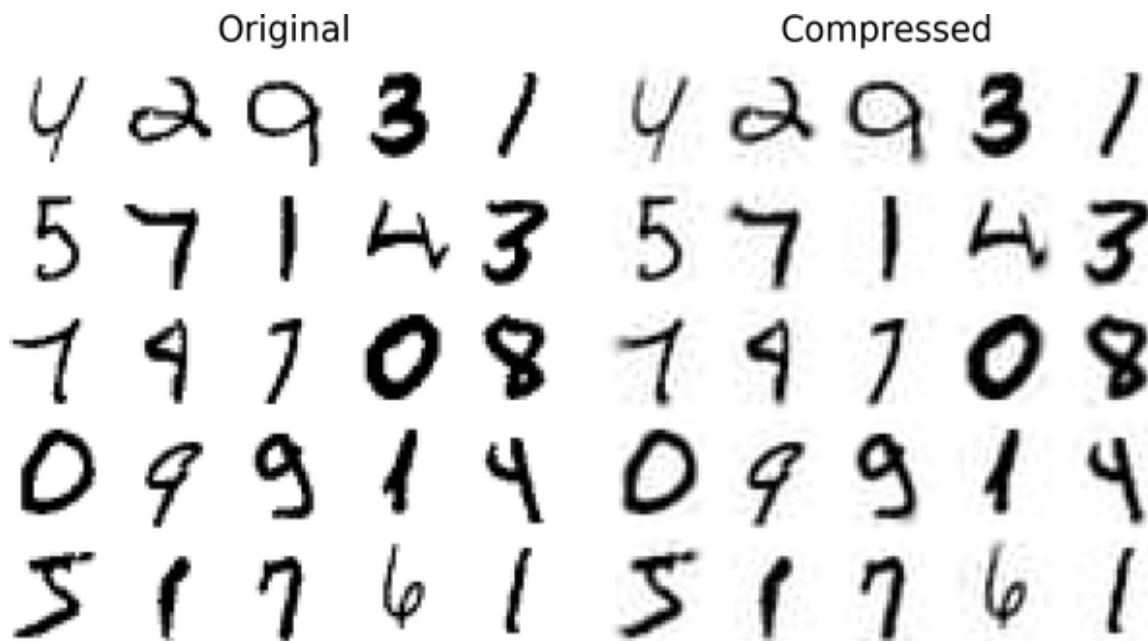


Figure 8-9. MNIST compression that preserves 95% of the variance

The equation for the inverse transformation is shown in **Equation 8-3**.

Equation 8-3. PCA inverse transformation, back to the original number of dimensions

$$\mathbf{X}_{\text{recovered}} = \mathbf{X}_{d\text{-proj}} \mathbf{W}_d^T$$

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)
```

TIP

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 the `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 that allow you to split the training set into mini-batches and feed these in 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 `mmap` 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 memory-mapped (`mmap`) file and copy the MNIST training set to it, then call `flush()` to ensure that any data still in the 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 `mmap` array:

```

filename = "my_mnist.mmap"
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 `mmap` 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
inc_pca = IncrementalPCA(n_components=154, batch_size=batch_size)
inc_pca.fit(X_mmap)

```

WARNING

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 you 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,  $\epsilon$  = 5_000, 0.1
>>> d = johnson_lindenstrauss_min_dim(m, eps= $\epsilon$ )
>>> d
7300
```

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 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. 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 the preceding code (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 also provides a second random projection transformer, known as `SparseRandomProjection`. It determines the target dimensionality in the same way, generates a random matrix of the same shape, and 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 1.2 GB in the preceding example! And it's also much faster, both to generate the random matrix and to reduce dimensionality: about 50% faster in this case. 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 1 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 $v = 1/\sqrt{dr}$.

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
```

WARNING

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.

NOTE

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 a fruit fly 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.

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:

```
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 the Swiss roll. We don't use it in this example, but it

can be used as a target for a nonlinear 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 of modeling the manifold.

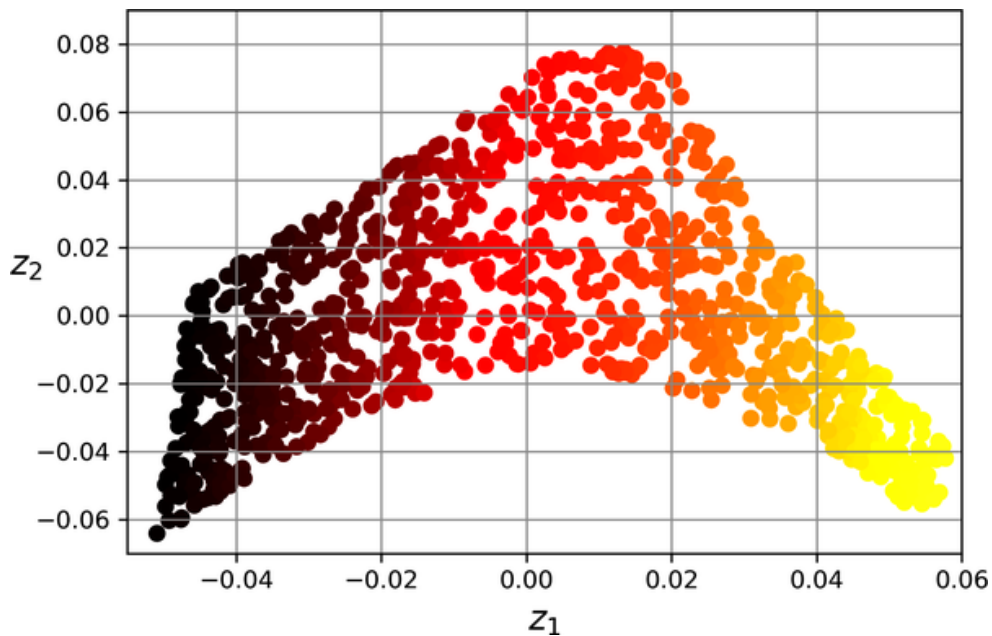


Figure 8-10. Unrolled Swiss roll using LLE

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 1: linearly modeling local relationships

$$\widehat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmin}} \sum_{i=1}^m \left(\mathbf{x}^{(i)} - \sum_{j=1}^m w_{i,j} \mathbf{x}^{(j)} \right)^2$$

subject to $\begin{cases} w_{i,j} = 0 & \text{if } \mathbf{x}^{(j)} \text{ is not one of the } k \text{ n.n. of } \mathbf{x}^{(i)} \\ \sum_{j=1}^m w_{i,j} = 1 & \text{for } i = 1, 2, \dots, m \end{cases}$

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 2: 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 nonlinear.

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 that, during training, 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. t-SNE does a reasonable job of flattening the Swiss roll, preserving a bit of curvature, and it also amplifies clusters, tearing the roll apart. Again, this might be good or bad, depending on the downstream task.

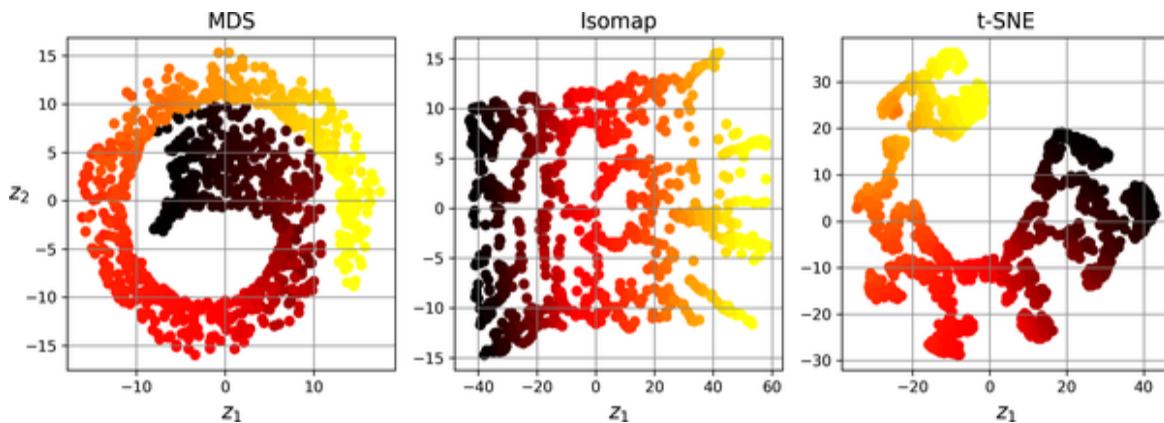


Figure 8-11. Using various techniques to reduce the Swiss roll to 2D

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?

Chapter 18. Reinforcement Learning

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. However, 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 humans**³ 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 with the victory of their system AlphaGo against Lee Sedol, a legendary professional player of the game of Go, in March 2016 and against Ke Jie, the world champion, in May 2017. 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.

So how did DeepMind (bought by Google for over \$500 million in 2014) 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, including a discussion of Markov decision processes. 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](#)):

1. 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.
2. 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.
3. Similarly, the agent can be the program playing a board game such as Go. It only gets a reward if it wins.
4. 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.
5. 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.

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 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.

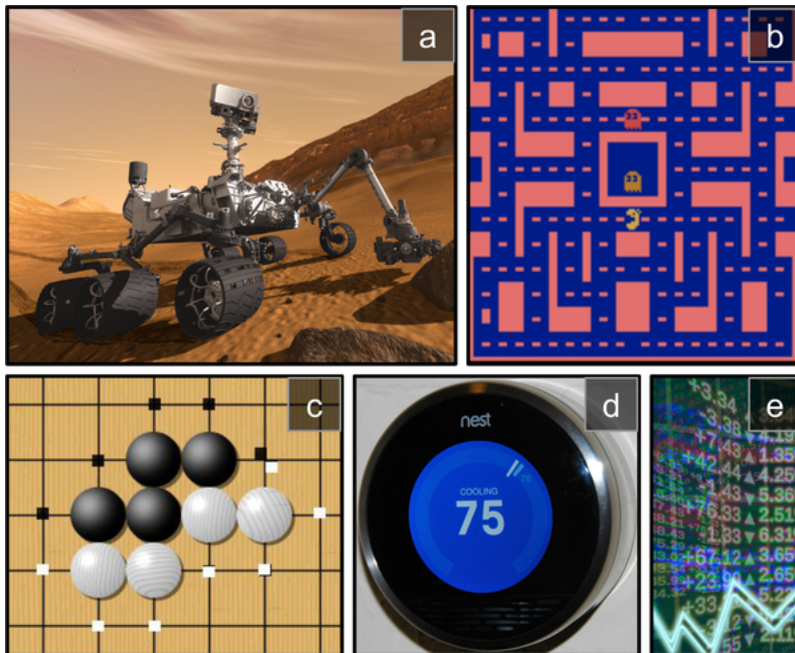


Figure 18-1. Reinforcement learning examples: (a) robotics, (b) Ms. Pac-Man, (c) Go player, (d) thermostat, (e) automatic trader⁵

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).

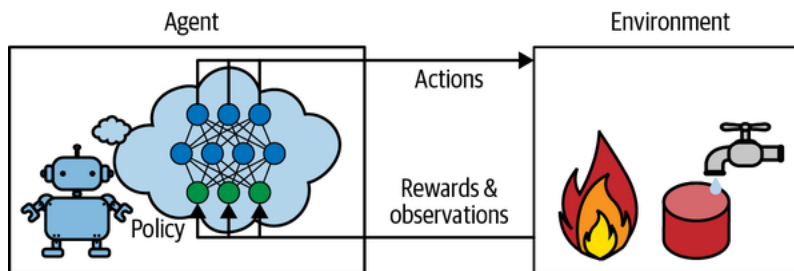


Figure 18-2. Reinforcement learning using a neural network policy

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

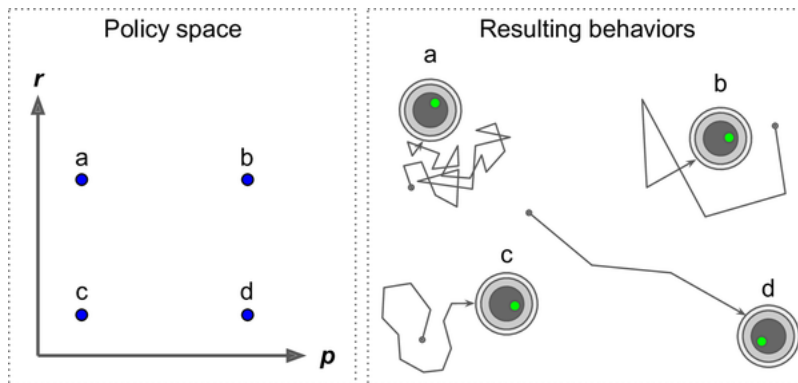


Figure 18-3. Four points in the policy space (left) and the agent's corresponding behavior (right)

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.

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. However, 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 might 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), that you can use to train agents, compare them, or develop new RL algorithms.

OpenAI Gym is preinstalled on Colab, but it's an older version, so you'll need to replace it with the latest one. You also need to 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; otherwise, enter these commands:

```
# Only run these commands on Colab or Kaggle!
%pip install -q -U gym
%pip install -q -U gym[classic_control,box2d,atari,accept-rom-license]
```

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 various kinds of environments. This includes classic environments from *control theory*—the science of controlling dynamical systems—such as balancing a pole on a cart. It also includes environments based on the Box2D library—

a 2D physics engine for games. Lastly, it includes environments based on the Arcade Learning Environment (ALE), which is an emulator for Atari 2600 games. Several Atari game ROMs are downloaded automatically, and by running this code you agree with Atari's ROM licenses.

With that, you're ready to use OpenAI Gym. Let's import it and make an environment:

```
import gym
env = gym.make("CartPole-v1", render_mode="rgb_array")
```

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](#)). This is a classic control task.

TIP

The `gym.envs.registry` dictionary contains the names and specifications of all the available environments.

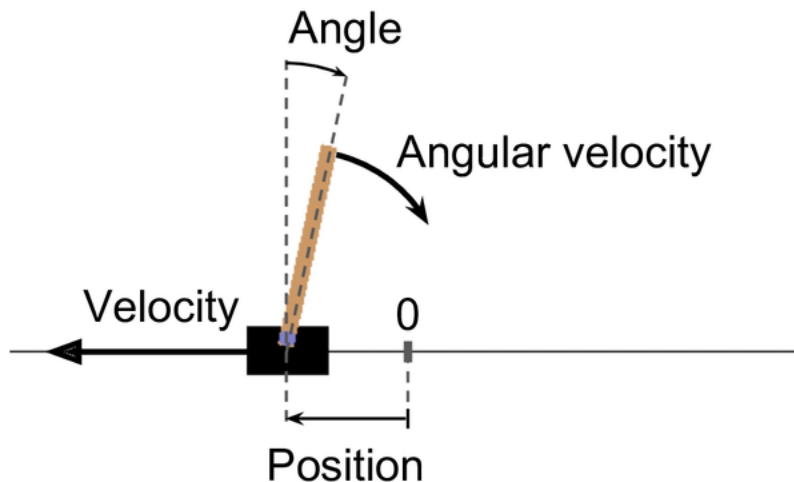


Figure 18-4. The CartPole environment

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 representing the cart's horizontal position ($0.0 = \text{center}$), its velocity (positive means right), the angle of the pole ($0.0 = \text{vertical}$), and its angular velocity (positive means clockwise). The `reset()` method also returns a dictionary that may contain extra environment-specific information. This can be useful for debugging or for training. For example, in many Atari environments, it contains the number of lives left. However, in the CartPole environment, this dictionary is empty.

```
>>> obs, info = env.reset(seed=42)
>>> obs
array([ 0.0273956 , -0.00611216,  0.03585979,  0.0197368 ], dtype=float32)
>>> info
{}
```

Let's call the `render()` method to render this environment as an image. Since we set `render_mode="rgb_array"` when creating the environment, the image will be returned as a NumPy array:

```
>>> img = env.render()
>>> 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 or right. 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, truncated, info = env.step(action)
>>> obs
array([ 0.02727336,  0.18847767,  0.03625453, -0.26141977], dtype=float32)
>>> reward
1.0
>>> done
False
>>> truncated
False
>>> info
{}
```

The `step()` method executes the desired action and returns five 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.

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.

truncated

This value will be `True` when an episode is interrupted early, for example by an environment wrapper that imposes a maximum number of steps per episode (see Gym's documentation for more details on environment wrappers). Some RL algorithms treat truncated episodes differently from episodes finished normally (i.e., when `done` is `True`), but in this chapter we will treat them identically.

info

This environment-specific dictionary may provide extra information, just like the one returned by the `reset()` method.

TIP

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, info = env.reset(seed=episode)
    for step in range(200):
        action = basic_policy(obs)
        obs, reward, done, truncated, info = env.step(action)
        episode_rewards += reward
        if done or truncated:
            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 this chapter's 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.

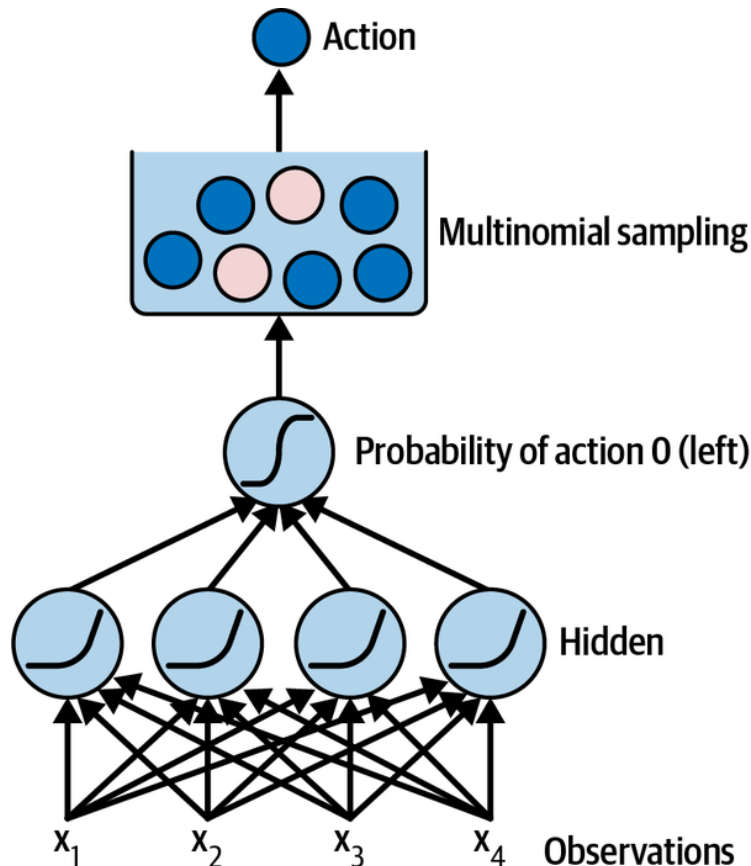


Figure 18-5. Neural network policy

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.

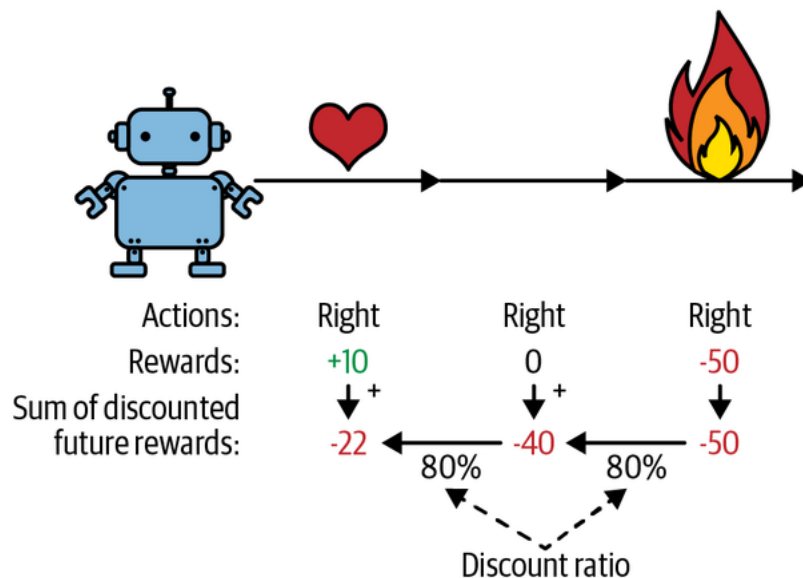


Figure 18-6. Computing an action's return: the sum of discounted future rewards

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. 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 1992**⁴¹ 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, truncated, info = env.step(int(action))
    return obs, reward, done, truncated, grads
```

Let's walk through 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.
- We now 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, whether it is truncated 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, info = env.reset()
        for step in range(n_max_steps):
            obs, reward, done, truncated, grads = play_one_step(
                env, obs, model, loss_fn)
            current_rewards.append(reward)
            current_grads.append(grads)
            if done or truncated:
                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, 1.0727777 ])]
```

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!

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).

TIP

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.

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 explore 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 consider *Markov decision processes* (MDPs).

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.

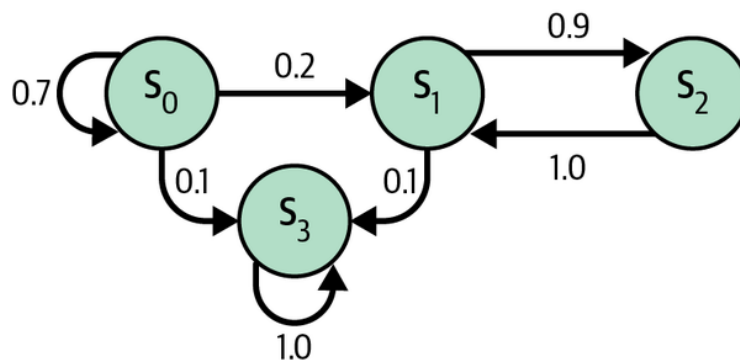


Figure 18-7. Example of a Markov chain

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**.¹² 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).

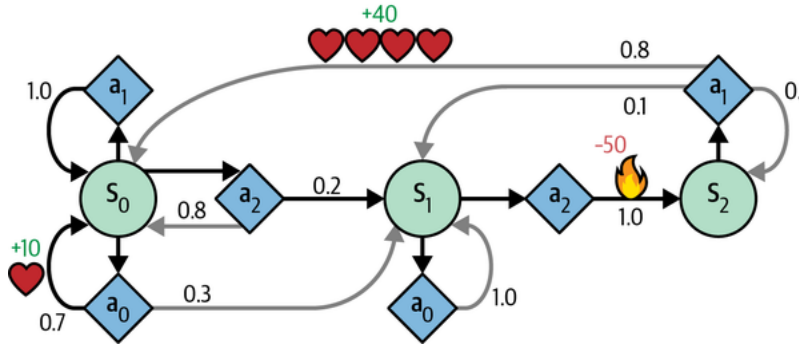


Figure 18-8. Example of a Markov decision process

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 the state, 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: first initialize all the state value estimates to zero, and then 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.

NOTE

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.

Let's look at 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: $\pi^*(s) = \underset{a}{\operatorname{argmax}} Q^*(s, a)$.

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 zero (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, we can find 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 (TD) learning* algorithm is very similar to the Q-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: $V_{k+1}(s) \leftarrow V_k(s) + \alpha \cdot \delta_k(s, r, s')$ with $\delta_k(s, r, s')$

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: $V(s) \xleftarrow{\alpha} r + \gamma \cdot V(s')$.

TIP

TD learning has many similarities with stochastic gradient descent, including the fact that it handles one sample at a time. Moreover, just like SGD, 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 will 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!

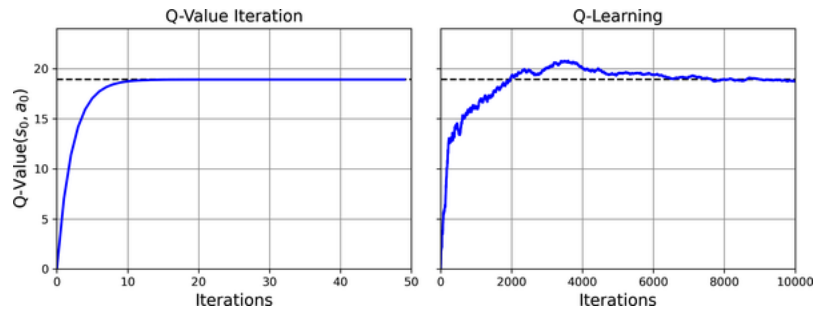


Figure 18-9. Learning curve of the Q-value iteration algorithm versus the Q-learning algorithm

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

$$Q(s, a) \leftarrow r + \gamma \cdot \max_{a'} f(Q(st, at), N(st, at))$$

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., the distances 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, we 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], verbose=0)[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)
```

TIP

A *deque* is a queue elements can be efficiently added to or removed from 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 six elements: a state s , the action a that the agent took, the resulting reward r , the next state s' it reached, a Boolean indicating whether the episode ended at that point (*done*), and finally another Boolean indicating whether the episode was truncated at that point. We will need a small function to sample a random batch of experiences from the replay buffer. It will return six NumPy arrays corresponding to the six 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]
    return [
        np.array([experience[field_index] for experience in batch])
        for field_index in range(6)
    ] # [states, actions, rewards, next_states, dones, truncateds]
```

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, truncated, info = env.step(action)
    replay_buffer.append((state, action, reward, next_state, done, truncated))
    return next_state, reward, done, truncated, 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, truncateds = experiences
    next_Q_values = model.predict(next_states, verbose=0)
    max_next_Q_values = next_Q_values.max(axis=1)
    runs = 1.0 - (dones | truncateds) # episode is not done or truncated
    target_Q_values = rewards + runs * 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))
```

Here's what's happening in 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.
- We want to use the DQN to compute the Q-value for each experienced state-action pair, but 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.
- Next, 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, info = env.reset()
    for step in range(200):
        epsilon = max(1 - episode / 500, 0.01)
        obs, reward, done, truncated, info = play_one_step(env, obs, epsilon)
        if done or truncated:
            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 or truncated, 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.

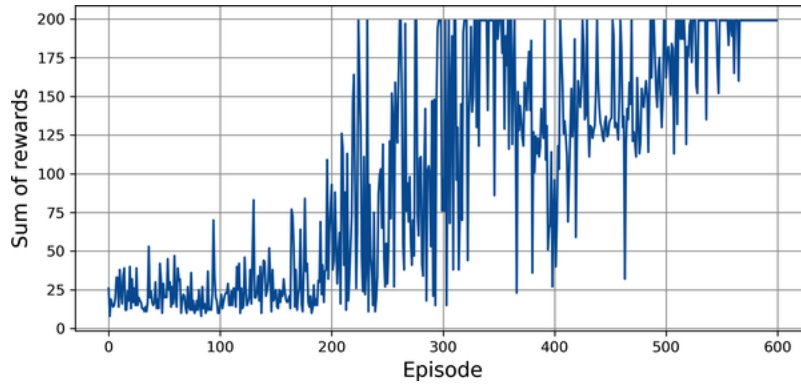


Figure 18-10. Learning curve of the deep Q-learning algorithm

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 soon after it looked like it had finally stabilized near the max reward, at around episode 320, its score again 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.

NOTE

Reinforcement learning is notoriously difficult, largely because of the training instabilities and the huge sensitivity to the choice of hyperparameter values and random seeds.¹³ 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!

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, verbose=0)
```

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 the 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,¹⁴ 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, the researchers 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, done, truncateds = experiences
    next_Q_values = model.predict(next_states, verbose=0) # 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, verbose=0) * 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; we'll look at that next.

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](#)¹⁵ 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 $\delta = r + \gamma V(s') - V(s)$. A large TD error indicates that a transition (s, a, s') is very surprising, and thus probably worth learning from.¹⁶ 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 priority is set to $p = |\delta|$ (plus a small constant to ensure that every experience has a nonzero 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 paper](#)¹⁷ 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 DDQN 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)
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](#):¹⁸ 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*¹⁹

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 in 1949 by Nicholas Metropolis and Stanislaw Ulam. It selects the best move after running many simulations, repeatedly exploring the search tree starting from the current position, and spending 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 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 involved three more neural networks, and was more complicated, but it was simplified in the *AlphaGo Zero paper*,²⁰ which uses a single neural network to both select moves and evaluate game states. The *AlphaZero paper*²¹ 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*²² continued to improve upon this algorithm, 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-critic (A3C)*²³

This is 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)

A2C is 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-critic (SAC)*²⁴

SAC is 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 algorithm has demonstrated an amazing sample efficiency (contrary to all the previous algorithms, which learn very slowly).

*Proximal policy optimization (PPO)*²⁵

This algorithm by John Schulman and other OpenAI researchers is based on A2C, but it 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 optimization*²⁶ (TRPO) algorithm, also by OpenAI. OpenAI made the news in April 2019 with its AI called OpenAI Five, based on the PPO algorithm, which defeated the world champions at the multiplayer game *Dota 2*.

*Curiosity-based exploration*²⁷

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 (OEL)

The objective of 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 paper](#)²⁸ by a team of researchers from Uber AI introduced the *POET algorithm*, which generates multiple simulated 2D environments with bumps and holes and trains one agent per environment: the agent's goal is to walk as fast as possible while avoiding the obstacles. The algorithm 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](#),²⁹ as well as DeepMind's [2021 paper](#)³⁰ on this topic.

TIP

If you'd like to learn more about reinforcement learning, check out the book *Reinforcement Learning* by Phil Winder (O'Reilly).

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?
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) then 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>.

-
- 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.
 - 4 Check out the videos of DeepMind's system learning to play *Space Invaders*, *Breakout*, and other video games at <https://homl.info/dqn3>.
 - 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](#).
 - 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.
 - 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).
 - 11 Ronald J. Williams, "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning", *Machine Learning* 8 (1992): 229–256.
 - 12 Richard Bellman, "A Markovian Decision Process", *Journal of Mathematics and Mechanics* 6, no. 5 (1957): 679–684.
 - 13 A great [2018 post](#) by Alex Irpan nicely lays out RL's biggest difficulties and limitations.
 - 14 Hado van Hasselt et al., "Deep Reinforcement Learning with Double Q-Learning", *Proceedings of the 30th AAAI Conference on Artificial Intelligence* (2015): 2094–2100.
 - 15 Tom Schaul et al., "Prioritized Experience Replay", arXiv preprint arXiv:1511.05952 (2015).
 - 16 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).
 - 17 Ziyu Wang et al., "Dueling Network Architectures for Deep Reinforcement Learning", arXiv preprint arXiv:1511.06581 (2015).
 - 18 Matteo Hessel et al., "Rainbow: Combining Improvements in Deep Reinforcement Learning", arXiv preprint arXiv:1710.02298 (2017): 3215–3222.
 - 19 David Silver et al., "Mastering the Game of Go with Deep Neural Networks and Tree Search", *Nature* 529 (2016): 484–489.

Part II. Neural Networks and Deep Learning

OceanofPDF.com

Chapter 10. Introduction to Artificial Neural Networks with Keras

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), machine learning models 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 to fly. 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

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, 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 you 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 you 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 had been invented, such as support vector machines (see [Chapter 5](#)). These techniques seemed to offer better results 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).

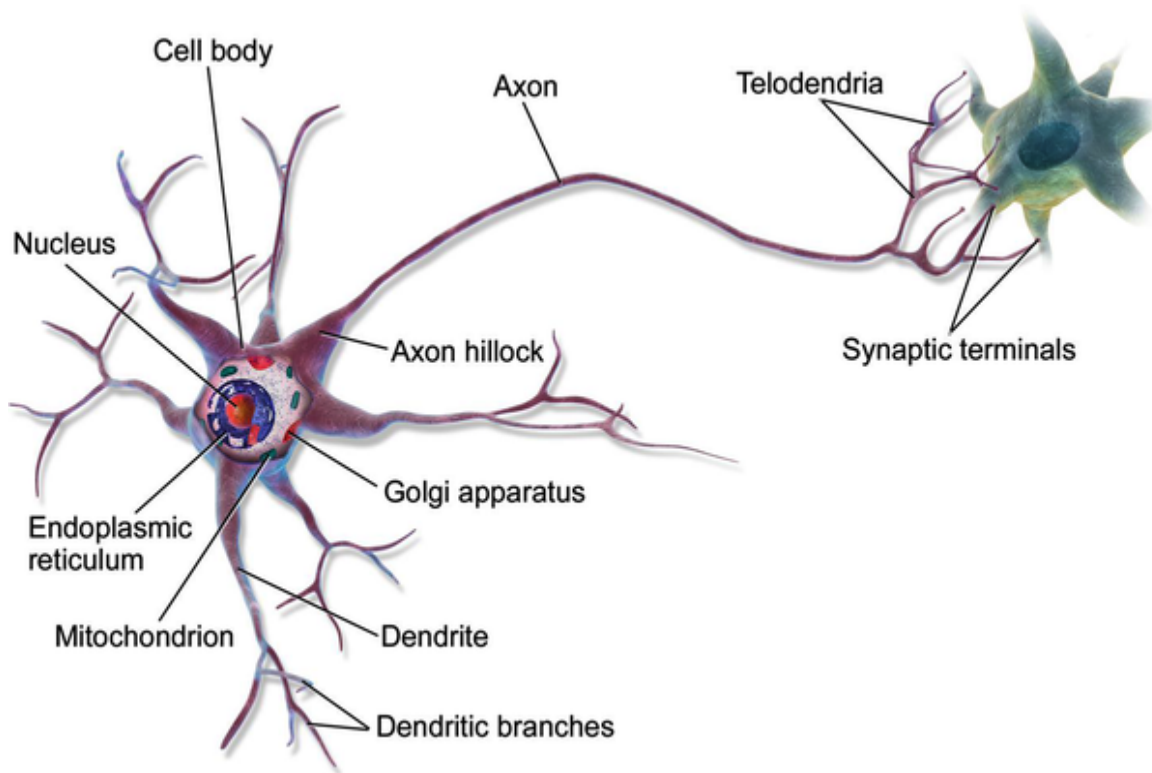


Figure 10-1. A biological neuron⁴

Thus, individual biological neurons seem to behave in a simple way, but they're 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 the subject of active research, but some parts of the brain have been mapped. These efforts show that neurons are often organized in consecutive layers, especially in the cerebral cortex (the outer layer of the brain), as shown in **Figure 10-2**.

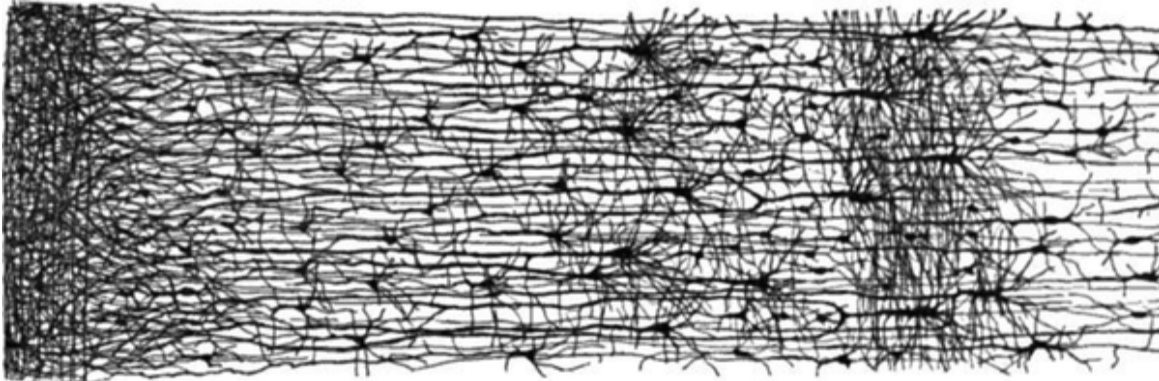


Figure 10-2. Multiple layers in a biological neural network (human cortex)⁶

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, McCulloch and Pitts showed that even with such a simplified model it is possible to build a network of artificial neurons that can compute 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.

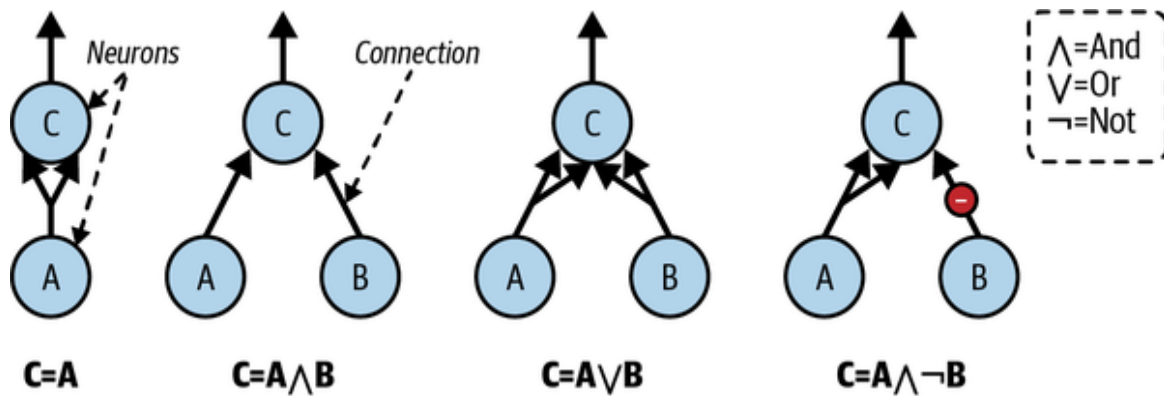


Figure 10-3. ANNs performing simple logical computations

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.
- 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 .

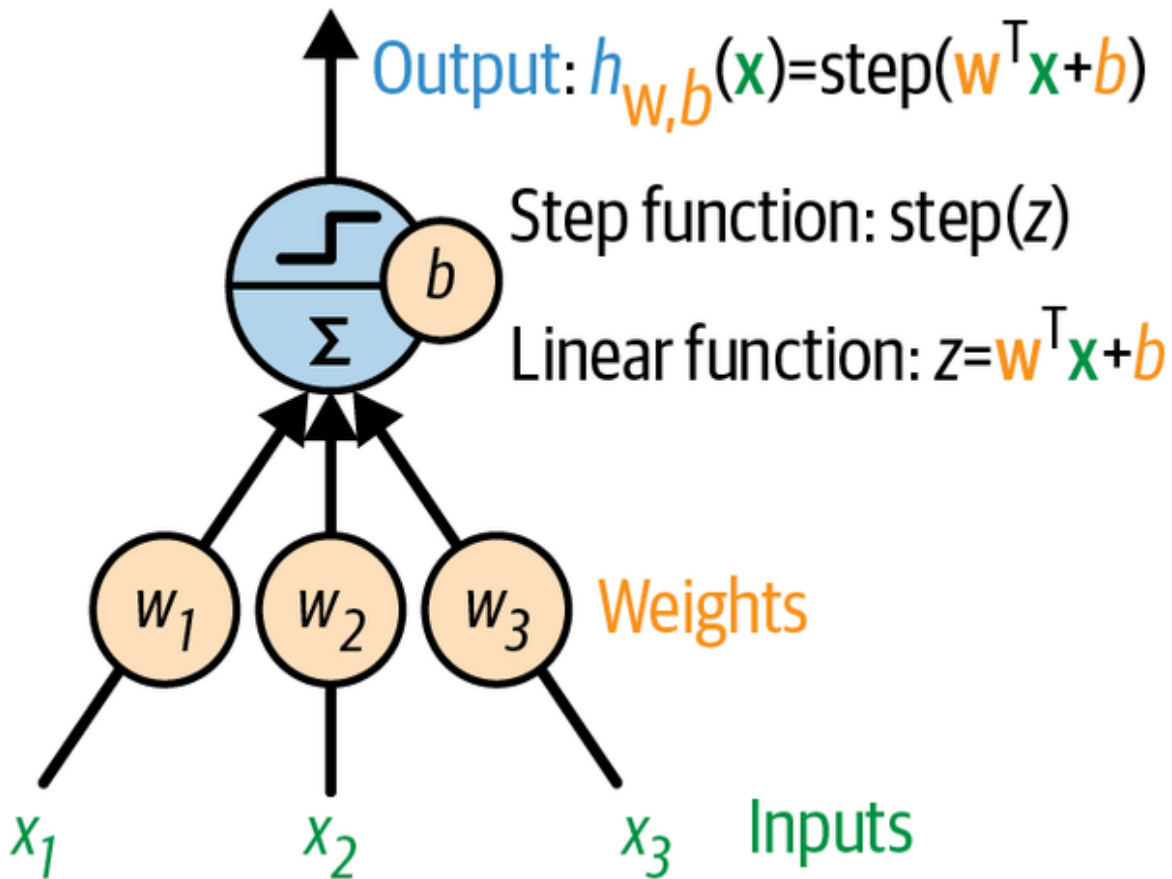


Figure 10-4. TLU: an artificial neuron that computes a weighted sum of its inputs $\mathbf{w}^T \mathbf{x}$, plus a bias term b , then applies a step function

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](#).

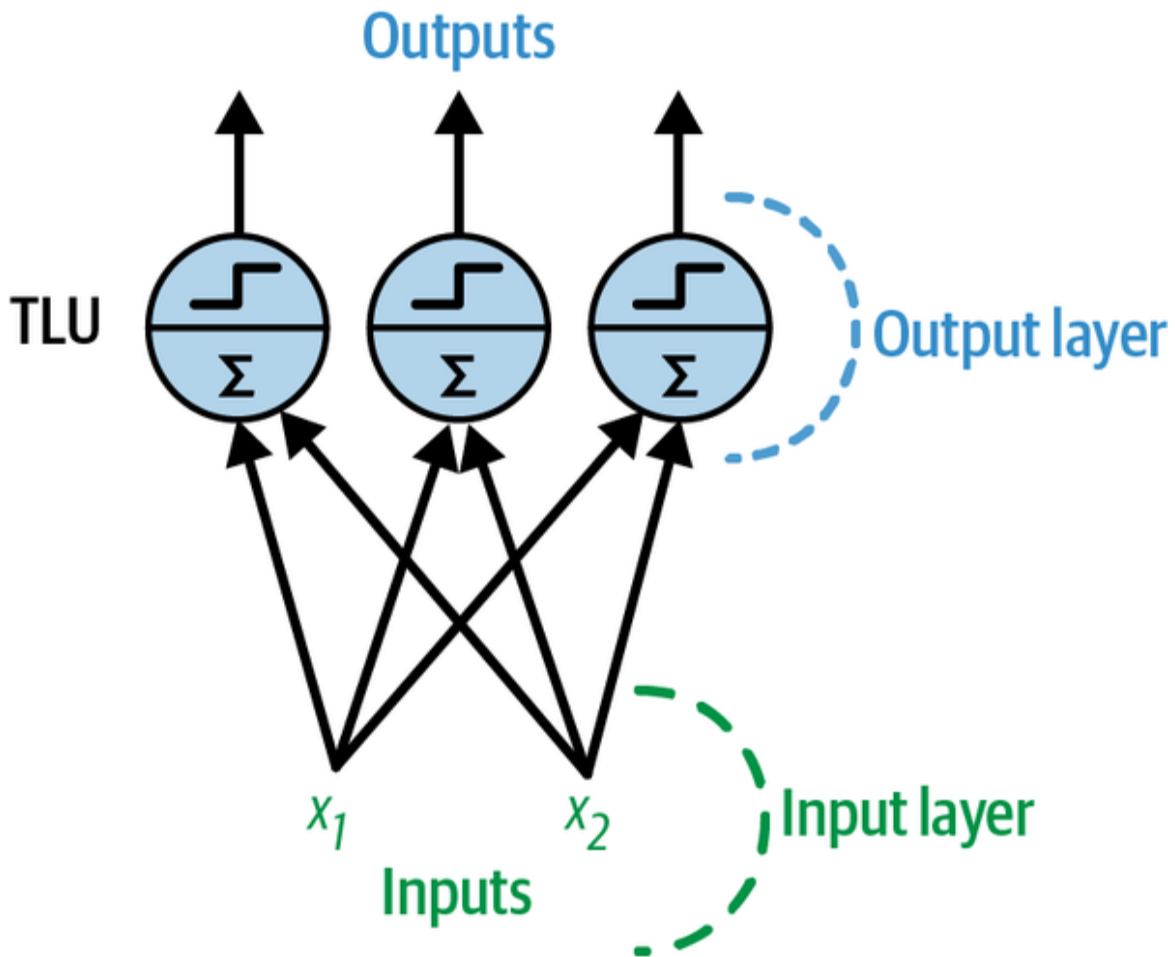


Figure 10-5. Architecture of a perceptron with two inputs and three output neurons

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 (we will discuss other activation functions shortly).

NOTE

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, $\mathbf{XW} + \mathbf{b}$ 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 then 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 that 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).

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

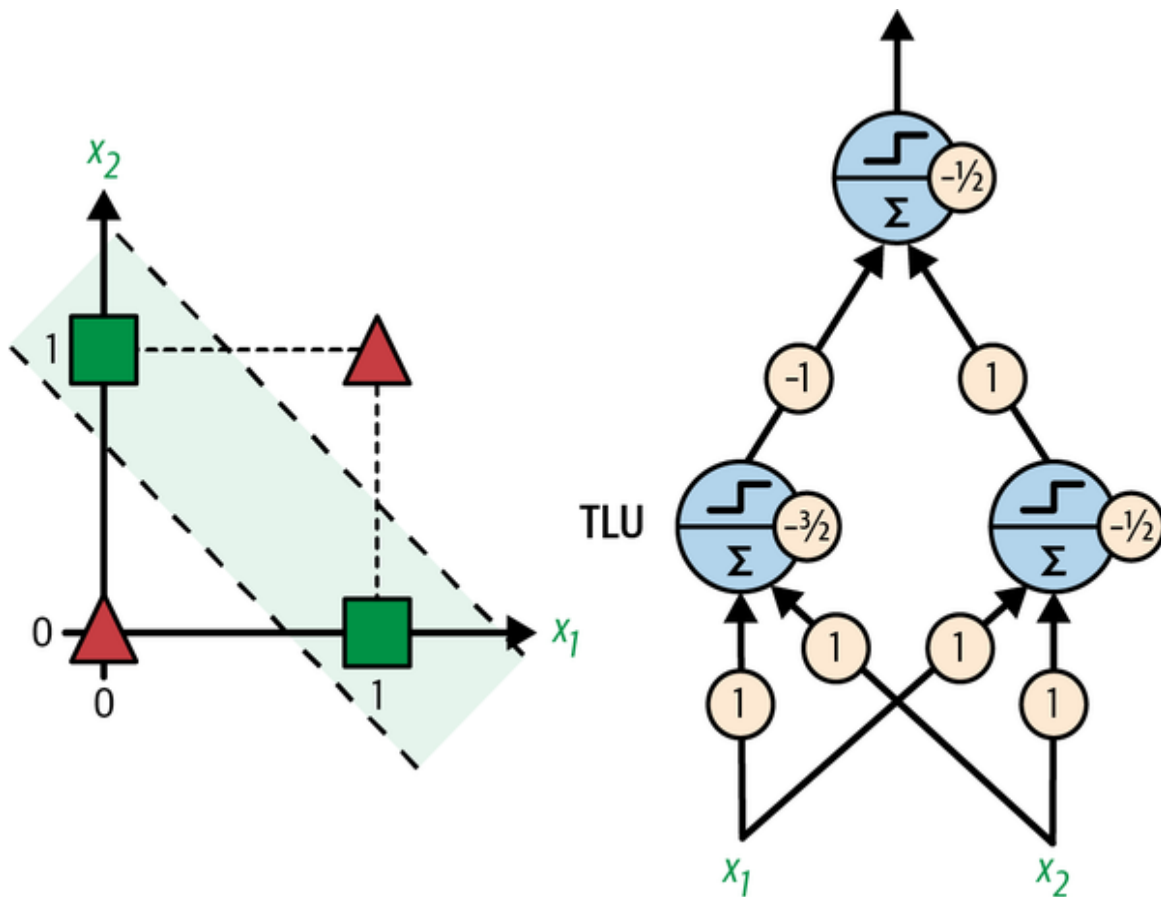


Figure 10-6. XOR classification problem and an MLP that solves it

NOTE

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.

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*.

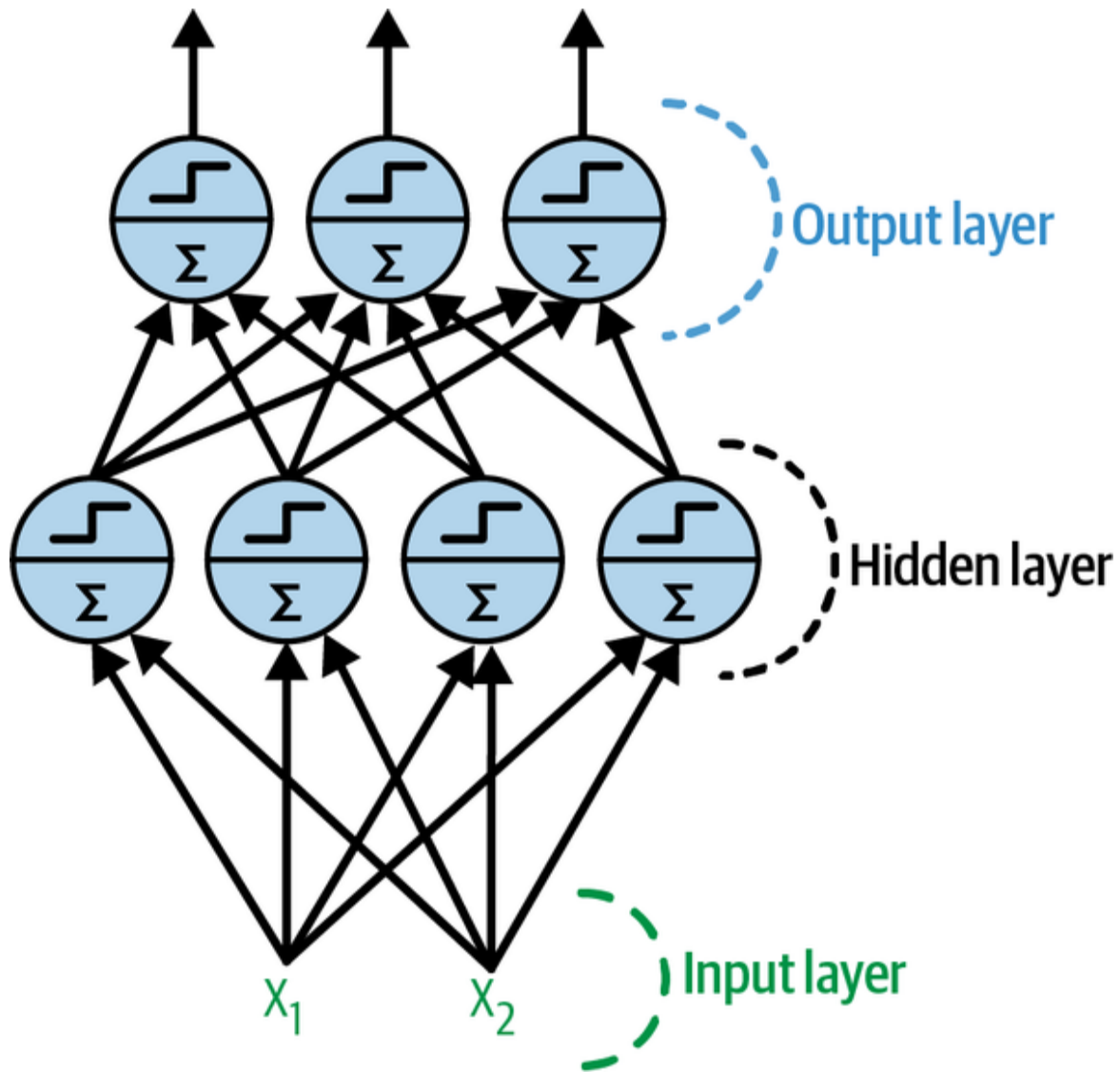


Figure 10-7. Architecture of a multilayer perceptron with two inputs, one hidden layer of four neurons, and three output neurons

NOTE

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 of using gradient descent to train neural networks, but as we saw in [Chapter 4](#), this requires computing the gradients of the model's error with regard 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.

Then, in 1970, a researcher named Seppo Linnainmaa introduced in his master's 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).

NOTE

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's thesis was not about neural nets, it was more general. It was several more years before

backprop started to be used to train neural networks, but it still wasn't mainstream. Then, in 1985, David Rumelhart, Geoffrey Hinton, and Ronald Williams published a **groundbreaking paper**¹⁰ 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.

Let's run through how backpropagation works 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 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 it 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.

WARNING

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 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).

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.

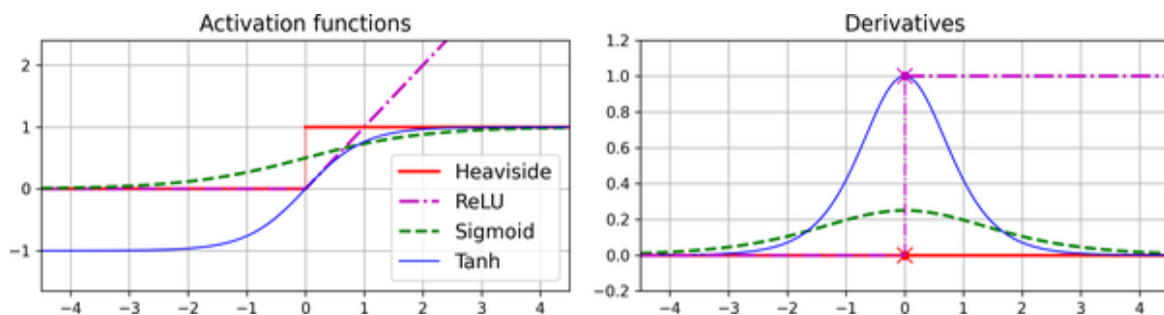


Figure 10-8. Activation functions (left) and their derivatives (right)

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.

WARNING

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. As you saw in [Chapter 3](#), this is called multiclass 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.

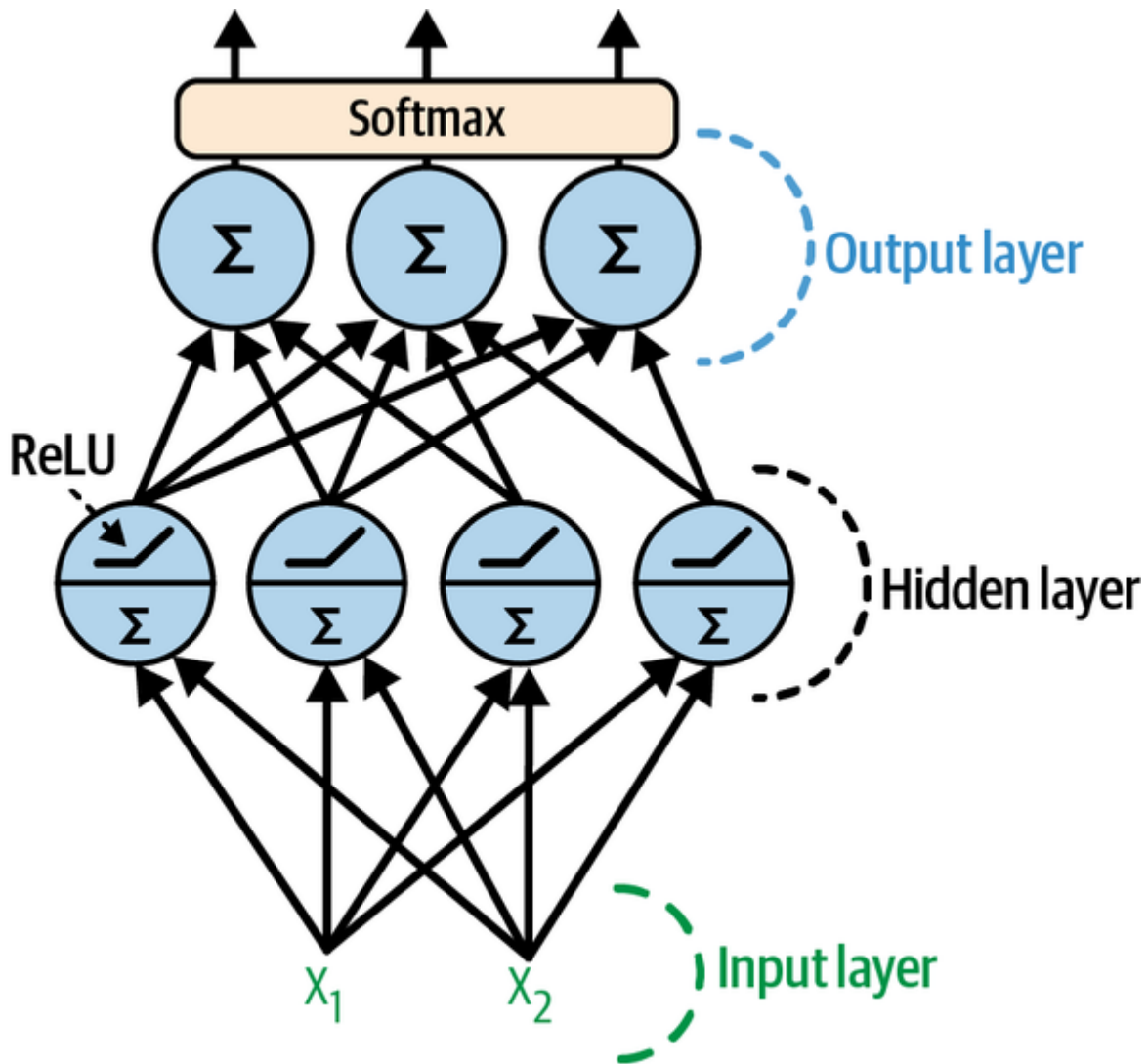


Figure 10-9. A modern MLP (including ReLU and softmax) for classification

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 (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	Typically 1 to 5 layers, depending on the task		
# output neurons	1	1 per binary label	1 per class
Output layer activation	Sigmoid	Sigmoid	Softmax
Loss function	X-entropy	X-entropy	X-entropy

TIP

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.

NOTE

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. Similarly, 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. Other popular deep learning libraries include [PyTorch by Facebook](#) and [JAX by Google](#).¹³

Now let's use Keras! We will start by building an MLP for image classification.

NOTE

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://homl.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 shuffled and split into a training set (60,000 images)

and a test set (10,000 images), but we'll 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:]
```

TIP

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:

- 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 (an `Input` layer) and add it to the model. 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. Its 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 doesn't have any parameters; it's 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.

TIP

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 (you will see some non-trainable parameters later in this chapter):

```
>>> model.summary()
Model: "sequential"
```

Layer (type)	Output Shape	Param #
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.

Each layer 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). Keras also ensures that the name is globally unique, even across models, by appending an index if needed, as in "dense_2". But why does it bother making the names unique across models? Well, this makes it possible to merge models easily without getting name conflicts.

TIP

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., 0.], dtype=float32)
>>> biases.shape
(300,)

```

Notice that the Dense layer initialized the connection weights randomly (which is needed to break symmetry, as discussed earlier), and the biases were initialized to zeros, which is fine. If you 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'll discuss initializers further in [Chapter 11](#), and the full list is at <https://keras.io/api/layers/initializers>.

NOTE

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"])
```

NOTE

Using `loss="sparse_categorical_crossentropy"` is the equivalent of using `loss=tf.keras.losses.sparse_categorical_crossentropy`. Similarly, using `optimizer="sgd"` is the equivalent of using `optimizer=tf.keras.optimizers.SGD()`, and using `metrics=["accuracy"]` is the equivalent of using `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 explanation. 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.

TIP

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 autodiff.

NOTE

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.

TIP

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 you can 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 1 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.

TIP

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()

```

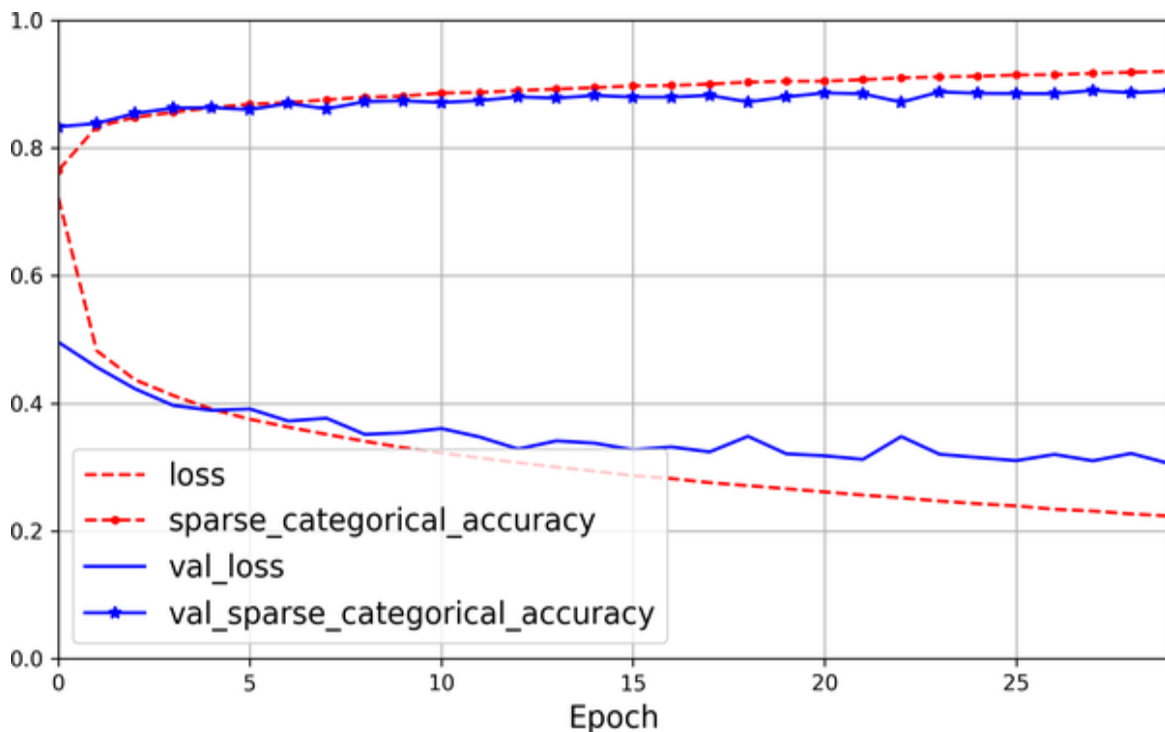


Figure 10-11. Learning curves: the mean training loss and accuracy measured over each epoch, and the mean validation loss and accuracy measured at the end of each epoch

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. This is 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 actually the case. 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. This is 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 you 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

Now let's use the model's `predict()` method to make predictions on new instances. Since we don't have actual new instances, we'll 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)
```

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()` method 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)
```

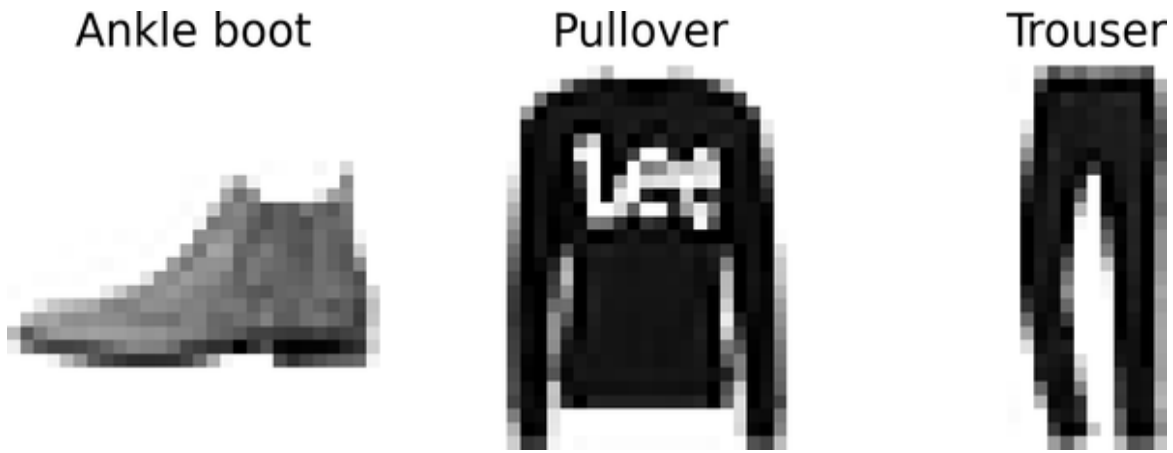


Figure 10-12. Correctly classified Fashion MNIST images

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](#)). Let's take a look:

```

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)

```

NOTE

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.

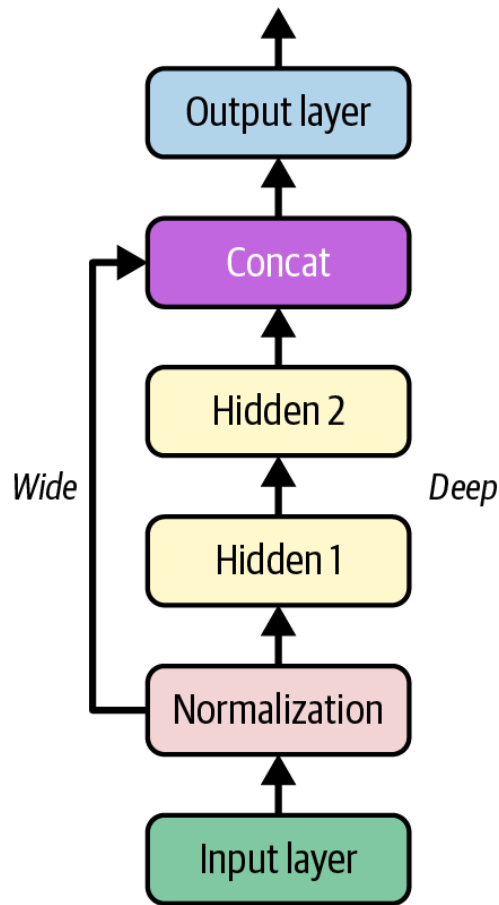


Figure 10-13. Wide & Deep neural network

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([normalized, 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 you 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 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, as illustrated in [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). We can do this as follows:

```
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])
```

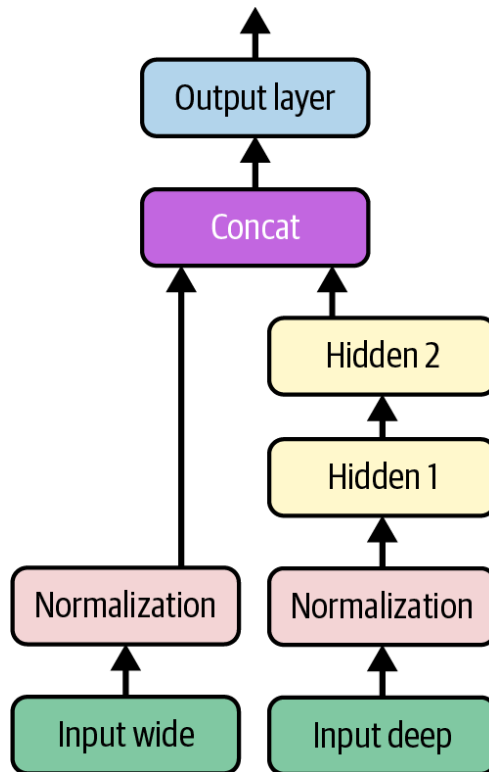


Figure 10-14. Handling multiple inputs

There are a few things to note in this example, compared to the previous one:

- 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))
```

TIP

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}`, if 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.

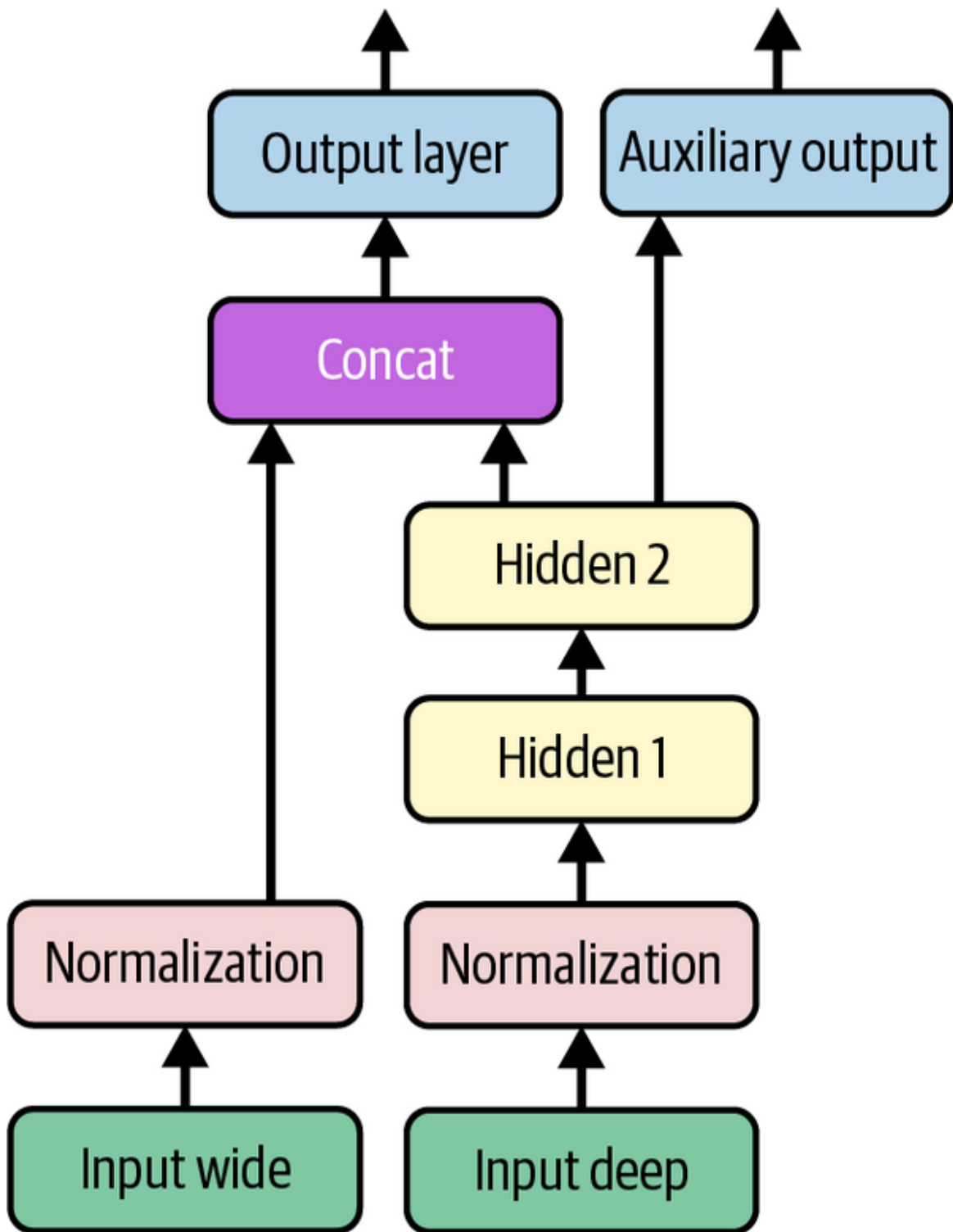


Figure 10-15. Handling multiple outputs, in this example to add an auxiliary output for regularization

Adding an extra output is quite easy: we just connect it to the appropriate layer and add it to the 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"])
```

TIP

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 we train the model, we 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`, we 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 we 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
```

TIP

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. However, 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. Next, we'll 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.

With this approach, you 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 like the previous one, except we separate the creation of the layers¹⁷ in the constructor from their usage in the `call()` method. And we don't need to create the Input objects: we can use the `input` argument to the `call()` method.

Now that we have a model instance, we 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 we did with the functional API.

The big difference with this API is that you can include 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.models.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.

TIP

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 (you 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, such as 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.

NOTE

If you set `save_format="h5"` or use a filename that ends 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.

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.

TIP

Many other callbacks are 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 plug-in to visualize profiling data. If you followed the installation instructions at <https://homl.info/install> to run everything locally, then you already have the plug-in 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 logfiles 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 that 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])
```

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 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, connects to this server and displays the user interface directly inside of Jupyter. The server, listens on the first available TCP port greater than or equal to 6006 (or you can set the port you want using the `--port` option).

```

%load_ext tensorboard
%tensorboard --logdir=./my_logs

```

TIP

If you're running everything on your own machine, it's 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 in the second run it went down a bit faster thanks to the higher learning rate.

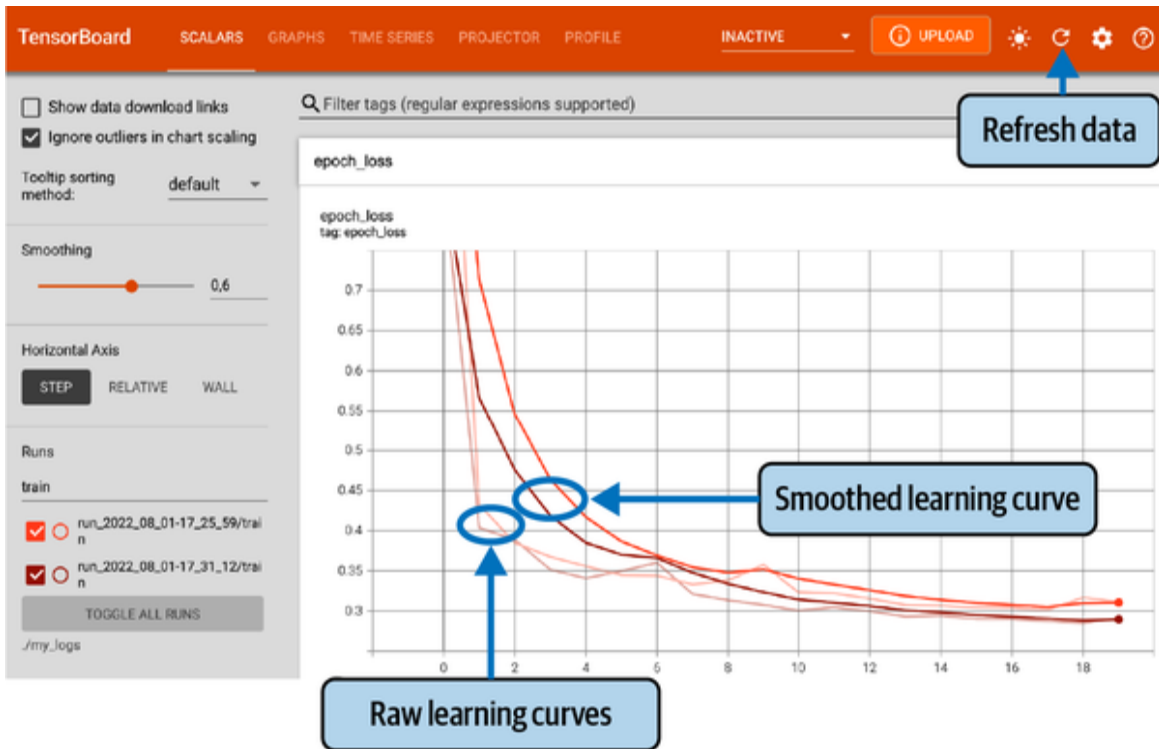


Figure 10-16. Visualizing learning curves with TensorBoard

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 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 audio at different time steps. As you can see, TensorBoard is a useful tool even beyond TensorFlow or deep learning.

TIP

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, you'll need to 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 an 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) 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 hyperparameter specifications. Then, in this example, it runs 5 trials; 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. The `fit()` method takes a `HyperParameters` object and a compiled model as an argument, as well as all the `model.fit()` arguments, and fits the model and returns 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 $1 / \text{factor}$ 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 scatterplot 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 in the `validation_epoch_accuracy` column: this will filter only the

hyperparameter combinations that reached a good performance. Click one of the hyperparameter combinations, and the corresponding learning curves will appear at the bottom of the page. Take some time to go through each tab; this 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 the 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.

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 you 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 slightly 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.

TIP

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 optimization 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 examine 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 paper](#)²¹

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 research points in the opposite direction, however. For example, 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) along with various techniques such as warming up the learning rate (i.e., starting training with a small learning rate, then ramping it up, as discussed in [Chapter 11](#)) and to obtain very short training times, without any generalization gap. So, one strategy is to try to using a large batch size, with 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, but 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.

TIP

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 paper](#)²⁴ 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 `tf.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 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

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" in the pipeline and finds the preprocessing ColumnTransformer. Next, it looks for a transformer named "geo" inside this ColumnTransformer and finds the ClusterSimilarity transformer we used on the latitude and longitude attributes. Then 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](#)).

TIP

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 3-fold cross validation. This means 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 setting `max_features` to 8.

TIP

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.

You can access the best estimator 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.
- If 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 `HalvingRandomSearchCV` and `HalvingGridSearchCV` hyperparameter search classes. 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 that are 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 that model's prediction. We will cover this topic in more detail in [Chapter 7](#).

Analyzing 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'),
 (0.05446998753775219, 'rooms_per_house__ratio'),
 (0.05262301809680712, 'people_per_house__ratio'),
 (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).

TIP

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), looks at its `feature_importances_` attribute, and selects the most useful features. Then when you call `transform()`, it 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, northern or southern, minority or not, etc. Creating subsets of your validation set for each category takes a bit of work, 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 currently 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()`. You get a fairly large interval from 39,275 to 43,467, and your 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. That's 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")
```

TIP

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): [...]
def ratio_name(function_transformer, feature_names_in): [...]
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.



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 Vertex AI 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 you 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 do 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 assess the model's performance. For example, suppose you trained an image classification model (we'll look at these in [Chapter 3](#)) to detect various 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 the 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.

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* (MLOps), 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 [Kaggle](#): 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 support vector machines 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`.