3 Introduction to Keras and TensorFlow

This chapter covers

- A closer look at TensorFlow, Keras, and their relationship
- Setting up a deep learning workspace
- An overview of how core deep learning concepts translate to Keras and TensorFlow

This chapter is meant to give you everything you need to start doing deep learning in practice. I'll give you a quick presentation of Keras (https://keras.io) and TensorFlow (https://tensorflow.org), the Python-based deep learning tools that we'll use throughout the book. You'll find out how to set up a deep learning workspace, with TensorFlow, Keras, and GPU support. Finally, building on top of the first contact you had with Keras and TensorFlow in chapter 2, we'll review the core components of neural networks and how they translate to the Keras and TensorFlow APIs.

By the end of this chapter, you'll be ready to move on to practical, real-world applications, which will start with chapter 4.

3.1 What's TensorFlow?

TensorFlow is a Python-based, free, open source machine learning platform, developed primarily by Google. Much like NumPy, the primary purpose of TensorFlow is to enable engineers and researchers to manipulate mathematical expressions over numerical tensors. But TensorFlow goes far beyond the scope of NumPy in the following ways:

- It can automatically compute the gradient of any differentiable expression (as you saw in chapter 2), making it highly suitable for machine learning.
- It can run not only on CPUs, but also on GPUs and TPUs, highly parallel hardware accelerators.
- Computation defined in TensorFlow can be easily distributed across many machines.
- TensorFlow programs can be exported to other runtimes, such as C++, JavaScript (for browser-based applications), or TensorFlow Lite (for applications running on mobile devices or embedded devices), etc. This makes TensorFlow applications easy to deploy in practical settings.

It's important to keep in mind that TensorFlow is much more than a single library. It's really a platform, home to a vast ecosystem of components, some developed by Google and some developed by third parties. For instance, there's TF-Agents for reinforcement-learning research, TFX for industry-strength machine learning workflow management, TensorFlow Serving for production deployment, and there's the TensorFlow Hub repository of pretrained models. Together, these components cover a very wide range of use cases, from cutting-edge research to large-scale production applications.

TensorFlow scales fairly well: for instance, scientists from Oak Ridge National Lab have used it to train a 1.1 exaFLOPS extreme weather forecasting model on the 27,000 GPUs of the IBM Summit supercomputer. Likewise, Google has used TensorFlow to develop very compute-intensive deep learning applications, such as the chess-playing and Go-playing agent AlphaZero. For your own models, if you have the budget, you can realistically hope to scale to around 10 petaFLOPS on a small TPU pod or a large cluster of GPUs rented on Google Cloud or AWS. That would still be around 1% of the peak compute power of the top supercomputer in 2019!

3.2 What's Keras?

Keras is a deep learning API for Python, built on top of TensorFlow, that provides a convenient way to define and train any kind of deep learning model. Keras was initially developed for research, with the aim of enabling fast deep learning experimentation.

Through TensorFlow, Keras can run on top of different types of hardware (see figure 3.1)—GPU, TPU, or plain CPU—and can be seamlessly scaled to thousands of machines.

Keras is known for prioritizing the developer experience. It's an API for human beings, not machines. It follows best practices for reducing cognitive load: it offers

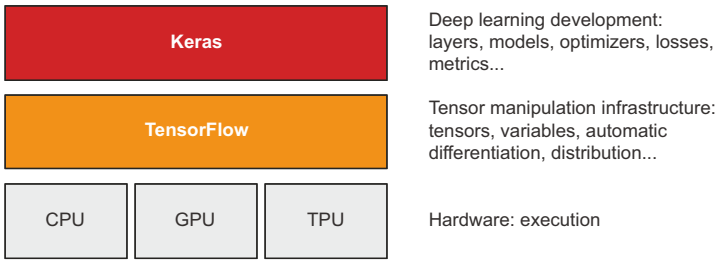


Figure 3.1 Keras and TensorFlow: TensorFlow is a low-level tensor computing platform, and Keras is a high-level deep learning API

consistent and simple workflows, it minimizes the number of actions required for common use cases, and it provides clear and actionable feedback upon user error. This makes Keras easy to learn for a beginner, and highly productive to use for an expert.

Keras has well over a million users as of late 2021, ranging from academic researchers, engineers, and data scientists at both startups and large companies to graduate students and hobbyists. Keras is used at Google, Netflix, Uber, CERN, NASA, Yelp, Instacart, Square, and hundreds of startups working on a wide range of problems across every industry. Your YouTube recommendations originate from Keras models. The Waymo self-driving cars are developed with Keras models. Keras is also a popular framework on Kaggle, the machine learning competition website, where most deep learning competitions have been won using Keras.

Because Keras has a large and diverse user base, it doesn't force you to follow a single "true" way of building and training models. Rather, it enables a wide range of different workflows, from the very high level to the very low level, corresponding to different user profiles. For instance, you have an array of ways to build models and an array of ways to train them, each representing a certain trade-off between usability and flexibility. In chapter 5, we'll review in detail a good fraction of this spectrum of workflows. You could be using Keras like you would use Scikit-learn—just calling `fit()` and letting the framework do its thing—or you could be using it like NumPy—taking full control of every little detail.

This means that everything you're learning now as you're getting started will still be relevant once you've become an expert. You can get started easily and then gradually dive into workflows where you're writing more and more logic from scratch. You won't have to switch to an entirely different framework as you go from student to researcher, or from data scientist to deep learning engineer.

This philosophy is not unlike that of Python itself! Some languages only offer one way to write programs—for instance, object-oriented programming or functional programming. Meanwhile, Python is a multiparadigm language: it offers an array of possible usage patterns that all work nicely together. This makes Python suitable to a wide range of very different use cases: system administration, data science, machine learning

engineering, web development . . . or just learning how to program. Likewise, you can think of Keras as the Python of deep learning: a user-friendly deep learning language that offers a variety of workflows to different user profiles.

3.3 Keras and TensorFlow: A brief history

Keras predates TensorFlow by eight months. It was released in March 2015, and TensorFlow was released in November 2015. You may ask, if Keras is built on top of TensorFlow, how it could exist before TensorFlow was released? Keras was originally built on top of Theano, another tensor-manipulation library that provided automatic differentiation and GPU support—the earliest of its kind. Theano, developed at the Montréal Institute for Learning Algorithms (MILA) at the Université de Montréal, was in many ways a precursor of TensorFlow. It pioneered the idea of using static computation graphs for automatic differentiation and for compiling code to both CPU and GPU.

In late 2015, after the release of TensorFlow, Keras was refactored to a multibackend architecture: it became possible to use Keras with either Theano or TensorFlow, and switching between the two was as easy as changing an environment variable. By September 2016, TensorFlow had reached a level of technical maturity where it became possible to make it the default backend option for Keras. In 2017, two new additional backend options were added to Keras: CNTK (developed by Microsoft) and MXNet (developed by Amazon). Nowadays, both Theano and CNTK are out of development, and MXNet is not widely used outside of Amazon. Keras is back to being a single-backend API—on top of TensorFlow.

Keras and TensorFlow have had a symbiotic relationship for many years. Throughout 2016 and 2017, Keras became well known as the user-friendly way to develop TensorFlow applications, funneling new users into the TensorFlow ecosystem. By late 2017, a majority of TensorFlow users were using it through Keras or in combination with Keras. In 2018, the TensorFlow leadership picked Keras as TensorFlow's official high-level API. As a result, the Keras API is front and center in TensorFlow 2.0, released in September 2019—an extensive redesign of TensorFlow and Keras that takes into account over four years of user feedback and technical progress.

By this point, you must be eager to start running Keras and TensorFlow code in practice. Let's get you started.

3.4 Setting up a deep learning workspace

Before you can get started developing deep learning applications, you need to set up your development environment. It's highly recommended, although not strictly necessary, that you run deep learning code on a modern NVIDIA GPU rather than your computer's CPU. Some applications—in particular, image processing with convolutional networks—will be excruciatingly slow on CPU, even a fast multicore CPU. And even for applications that can realistically be run on CPU, you'll generally see the speed increase by a factor of 5 or 10 by using a recent GPU.

To do deep learning on a GPU, you have three options:

- Buy and install a physical NVIDIA GPU on your workstation.
- Use GPU instances on Google Cloud or AWS EC2.
- Use the free GPU runtime from Colaboratory, a hosted notebook service offered by Google (for details about what a "notebook" is, see the next section).

Colaboratory is the easiest way to get started, as it requires no hardware purchase and no software installation—just open a tab in your browser and start coding. It's the option we recommend for running the code examples in this book. However, the free version of Colaboratory is only suitable for small workloads. If you want to scale up, you'll have to use the first or second option.

If you don't already have a GPU that you can use for deep learning (a recent, high-end NVIDIA GPU), then running deep learning experiments in the cloud is a simple, low-cost way for you to move to larger workloads without having to buy any additional hardware. If you're developing using Jupyter notebooks, the experience of running in the cloud is no different from running locally.

But if you're a heavy user of deep learning, this setup isn't sustainable in the long term—or even for more than a few months. Cloud instances aren't cheap: you'd pay $2.48 per hour for a V100 GPU on Google Cloud in mid-2021. Meanwhile, a solid consumer-class GPU will cost you somewhere between $1,500 and $2,500—a price that has been fairly stable over time, even as the specs of these GPUs keep improving. If you're a heavy user of deep learning, consider setting up a local workstation with one or more GPUs.

Additionally, whether you're running locally or in the cloud, it's better to be using a Unix workstation. Although it's technically possible to run Keras on Windows directly, we don't recommend it. If you're a Windows user and you want to do deep learning on your own workstation, the simplest solution to get everything running is to set up an Ubuntu dual boot on your machine, or to leverage Windows Subsystem for Linux (WSL), a compatibility layer that enables you to run Linux applications from Windows. It may seem like a hassle, but it will save you a lot of time and trouble in the long run.

3.4.1 Jupyter notebooks: The preferred way to run deep learning experiments

Jupyter notebooks are a great way to run deep learning experiments—in particular, the many code examples in this book. They're widely used in the data science and machine learning communities. A *notebook* is a file generated by the Jupyter Notebook app (https://jupyter.org) that you can edit in your browser. It mixes the ability to execute Python code with rich text-editing capabilities for annotating what you're doing. A notebook also allows you to break up long experiments into smaller pieces that can be executed independently, which makes development interactive and means you don't have to rerun all of your previous code if something goes wrong late in an experiment.

I recommend using Jupyter notebooks to get started with Keras, although that isn't a requirement: you can also run standalone Python scripts or run code from within an IDE such as PyCharm. All the code examples in this book are available as open source notebooks; you can download them from GitHub at github.com/fchollet/deep-learning-with-python-notebooks.

3.4.2 Using Colaboratory

Colaboratory (or Colab for short) is a free Jupyter notebook service that requires no installation and runs entirely in the cloud. Effectively, it's a web page that lets you write and execute Keras scripts right away. It gives you access to a free (but limited) GPU runtime and even a TPU runtime, so you don't have to buy your own GPU. Colaboratory is what we recommend for running the code examples in this book.

First steps with Colaboratory

To get started with Colab, go to https://colab.research.google.com and click the New Notebook button. You'll see the standard Notebook interface shown in figure 3.2.

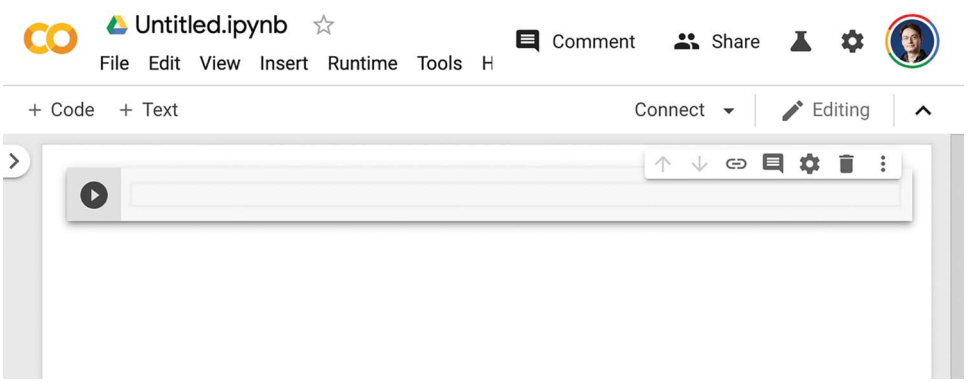


Figure 3.2 A Colab notebook

You'll notice two buttons in the toolbar: + Code and + Text. They're for creating executable Python code cells and annotation text cells, respectively. After entering code in a code cell, Pressing Shift-Enter will execute it (see figure 3.3).

In a text cell, you can use Markdown syntax (see figure 3.4). Pressing Shift-Enter on a text cell will render it.

Text cells are useful for giving a readable structure to your notebooks: use them to annotate your code with section titles and long explanation paragraphs or to embed figures. Notebooks are meant to be a multimedia experience!

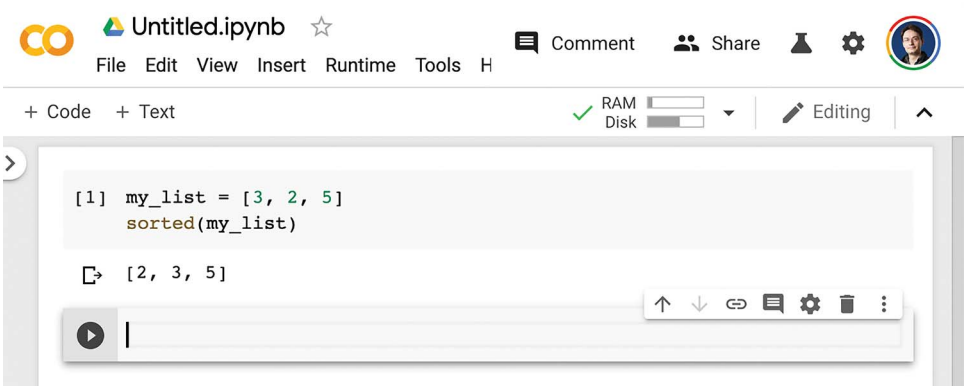


Figure 3.3 Creating a code cell

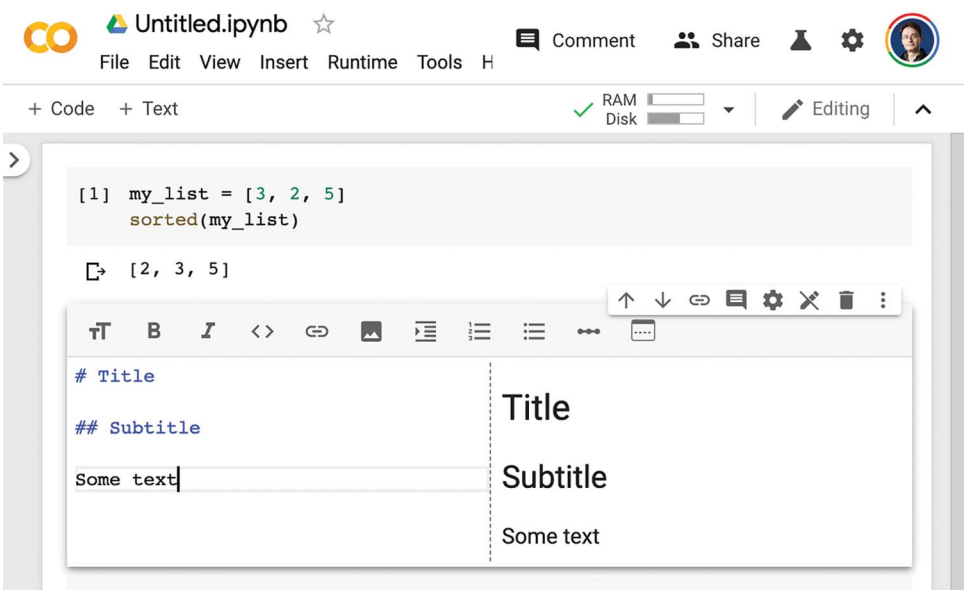


Figure 3.4 Creating a text cell

INSTALLING PACKAGES WITH PIP

The default Colab environment already comes with TensorFlow and Keras installed, so you can start using it right away without any installation steps required. But if you ever need to install something with `pip`, you can do so by using the following syntax in a code cell (note that the line starts with `!` to indicate that it is a shell command rather than Python code):

```
!pip install package_name
```

USING THE GPU RUNTIME

To use the GPU runtime with Colab, select Runtime > Change Runtime Type in the menu and select GPU for the Hardware Accelerator (see figure 3.5).

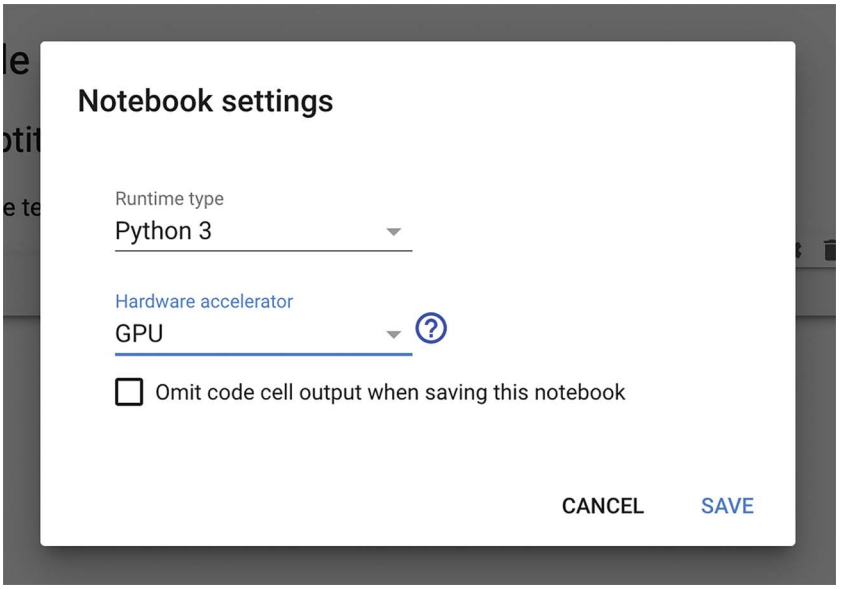


Figure 3.5 Using the GPU runtime with Colab

TensorFlow and Keras will automatically execute on GPU if a GPU is available, so there's nothing more you need to do after you've selected the GPU runtime.

You'll notice that there's also a TPU runtime option in that Hardware Accelerator dropdown menu. Unlike the GPU runtime, using the TPU runtime with TensorFlow and Keras does require a bit of manual setup in your code. We'll cover this in chapter 13. For the time being, we recommend that you stick to the GPU runtime to follow along with the code examples in the book.

You now have a way to start running Keras code in practice. Next, let's see how the key ideas you learned about in chapter 2 translate to Keras and TensorFlow code.

3.5 First steps with TensorFlow

As you saw in the previous chapters, training a neural network revolves around the following concepts:

- First, low-level tensor manipulation—the infrastructure that underlies all modern machine learning. This translates to TensorFlow APIs:
  - *Tensors*, including special tensors that store the network's state (*variables*)
  - *Tensor operations* such as addition, `relu`, `matmul`

  - *Backpropagation*, a way to compute the gradient of mathematical expressions (handled in TensorFlow via the `GradientTape` object)
- Second, high-level deep learning concepts. This translates to Keras APIs:
  - *Layers*, which are combined into a *model*
  - A *loss function*, which defines the feedback signal used for learning
  - An *optimizer*, which determines how learning proceeds
  - *Metrics* to evaluate model performance, such as accuracy
  - A *training loop* that performs mini-batch stochastic gradient descent

In the previous chapter, you already had a first light contact with some of the corresponding TensorFlow and Keras APIs: you've briefly used TensorFlow's `Variable` class, the `matmul` operation, and the `GradientTape`. You've instantiated Keras `Dense` layers, packed them into a `Sequential` model, and trained that model with the `fit()` method.

Now let's take a deeper dive into how all of these different concepts can be approached in practice using TensorFlow and Keras.

3.5.1 Constant tensors and variables

To do anything in TensorFlow, we're going to need some tensors. Tensors need to be created with some initial value. For instance, you could create all-ones or all-zeros tensors (see listing 3.1), or tensors of values drawn from a random distribution (see listing 3.2).

Listing 3.1 All-ones or all-zeros tensors

```
>>> import tensorflow as tf
>>> x = tf.ones(shape=(2, 1))
>>> print(x)
tf.Tensor(
[[1.]
 [1.]], shape=(2, 1), dtype=float32)
>>> x = tf.zeros(shape=(2, 1))
>>> print(x)
tf.Tensor(
[[0.]
 [0.]], shape=(2, 1), dtype=float32)
```

Equivalent to np.ones(shape=(2, 1))

Equivalent to np.zeros(shape=(2, 1))

Listing 3.2 Random tensors

```
>>> x = tf.random.normal(shape=(3, 1), mean=0., stddev=1.)
>>> print(x)
tf.Tensor(
[[-0.14208166]
 [-0.95319825]
 [ 1.1096532 ]], shape=(3, 1), dtype=float32)
>>> x = tf.random.uniform(shape=(3, 1), minval=0., maxval=1.)
>>> print(x)
tf.Tensor(
```

Tensor of random values drawn from a normal distribution with mean 0 and standard deviation 1. Equivalent to np.random.normal(size=(3, 1), loc=0., scale=1.).

Tensor of random values drawn from a uniform distribution between 0 and 1. Equivalent to np.random.uniform(size=(3, 1), low=0., high=1.).

```
[[0.33779848]
 [0.06692922]
 [0.7749394 ]], shape=(3, 1), dtype=float32)
```

A significant difference between NumPy arrays and TensorFlow tensors is that TensorFlow tensors aren't assignable: they're constant. For instance, in NumPy, you can do the following.

Listing 3.3 NumPy arrays are assignable

```
import numpy as np
x = np.ones(shape=(2, 2))
x[0, 0] = 0.
```

Try to do the same thing in TensorFlow, and you will get an error: "EagerTensor object does not support item assignment."

Listing 3.4 TensorFlow tensors are not assignable

```
x = tf.ones(shape=(2, 2))
x[0, 0] = 0.
```

This will fail, as a tensor isn't assignable.

To train a model, we'll need to update its state, which is a set of tensors. If tensors aren't assignable, how do we do it? That's where *variables* come in. `tf.Variable` is the class meant to manage modifiable state in TensorFlow. You've already briefly seen it in action in the training loop implementation at the end of chapter 2.

To create a variable, you need to provide some initial value, such as a random tensor.

Listing 3.5 Creating a TensorFlow variable

```
>>> v = tf.Variable(initial_value=tf.random.normal(shape=(3, 1)))
>>> print(v)
array([[-0.75133973],
       [-0.4872893 ],
       [ 1.6626885 ]], dtype=float32)>
```

The state of a variable can be modified via its `assign` method, as follows.

Listing 3.6 Assigning a value to a TensorFlow variable

```
>>> v.assign(tf.ones((3, 1)))
array([[1.],
       [1.],
       [1.]], dtype=float32)>
```

It also works for a subset of the coefficients.

Listing 3.7 Assigning a value to a subset of a TensorFlow variable

```
>>> v[0, 0].assign(3.)
array([[3.],
       [1.],
       [1.]], dtype=float32)>
```

Similarly, `assign_add()` and `assign_sub()` are efficient equivalents of `+=` and `-=`, as shown next.

Listing 3.8 Using `assign_add()`

```
>>> v.assign_add(tf.ones((3, 1)))
array([[2.],
       [2.],
       [2.]], dtype=float32)>
```

3.5.2 Tensor operations: Doing math in TensorFlow

Just like NumPy, TensorFlow offers a large collection of tensor operations to express mathematical formulas. Here are a few examples.

Listing 3.9 A few basic math operations

```
a = tf.ones((2, 2))
b = tf.square(a)
c = tf.sqrt(a)
d = b + c
e = tf.matmul(a, b)
e *= d
```

Take the square.

Take the square root.

Add two tensors (element-wise).

Take the product of two tensors (as discussed in chapter 2).

Multiply two tensors (element-wise).

Importantly, each of the preceding operations gets executed on the fly: at any point, you can print what the current result is, just like in NumPy. We call this *eager execution*.

3.5.3 A second look at the GradientTape API

So far, TensorFlow seems to look a lot like NumPy. But here's something NumPy can't do: retrieve the gradient of any differentiable expression with respect to any of its inputs. Just open a `GradientTape` scope, apply some computation to one or several input tensors, and retrieve the gradient of the result with respect to the inputs.

Listing 3.10 Using the `GradientTape`

```
input_var = tf.Variable(initial_value=3.)
with tf.GradientTape() as tape:
    result = tf.square(input_var)
gradient = tape.gradient(result, input_var)
```

This is most commonly used to retrieve the gradients of the loss of a model with respect to its weights: `gradients = tape.gradient(loss, weights)`. You saw this in action in chapter 2.

So far, you've only seen the case where the input tensors in `tape.gradient()` were TensorFlow variables. It's actually possible for these inputs to be any arbitrary tensor. However, only *trainable variables* are tracked by default. With a constant tensor, you'd have to manually mark it as being tracked by calling `tape.watch()` on it.

Listing 3.11 Using `GradientTape` with constant tensor inputs

```
input_const = tf.constant(3.)
with tf.GradientTape() as tape:
   tape.watch(input_const)
   result = tf.square(input_const)
gradient = tape.gradient(result, input_const)
```

Why is this necessary? Because it would be too expensive to preemptively store the information required to compute the gradient of anything with respect to anything. To avoid wasting resources, the tape needs to know what to watch. Trainable variables are watched by default because computing the gradient of a loss with regard to a list of trainable variables is the most common use of the gradient tape.

The gradient tape is a powerful utility, even capable of computing *second-order gradients*, that is to say, the gradient of a gradient. For instance, the gradient of the position of an object with regard to time is the speed of that object, and the second-order gradient is its acceleration.

If you measure the position of a falling apple along a vertical axis over time and find that it verifies `position(time) = 4.9 * time ** 2`, what is its acceleration? Let's use two nested gradient tapes to find out.

Listing 3.12 Using nested gradient tapes to compute second-order gradients

```
time = tf.Variable(0.)
with tf.GradientTape() as outer_tape:
    with tf.GradientTape() as inner_tape:
        position =  4.9 * time ** 2
    speed = inner_tape.gradient(position, time)
acceleration = outer_tape.gradient(speed, time)
```

We use the outer tape to compute the gradient of the gradient from the inner tape. Naturally, the answer is 4.9 * 2 = 9.8.

3.5.4 An end-to-end example: A linear classifier in pure TensorFlow

You know about tensors, variables, and tensor operations, and you know how to compute gradients. That's enough to build any machine learning model based on gradient descent. And you're only at chapter 3!

In a machine learning job interview, you may be asked to implement a linear classifier from scratch in TensorFlow: a very simple task that serves as a filter between candidates who have some minimal machine learning background and those who don't.

Let's get you past that filter and use your newfound knowledge of TensorFlow to implement such a linear classifier.

First, let's come up with some nicely linearly separable synthetic data to work with: two classes of points in a 2D plane. We'll generate each class of points by drawing their coordinates from a random distribution with a specific covariance matrix and a specific mean. Intuitively, the covariance matrix describes the shape of the point cloud, and the mean describes its position in the plane (see figure 3.6). We'll reuse the same covariance matrix for both point clouds, but we'll use two different mean values—the point clouds will have the same shape, but different positions.

Listing 3.13 Generating two classes of random points in a 2D plane

```
num_samples_per_class = 1000
negative_samples = np.random.multivariate_normal(
    mean=[0, 3],
    cov=[[1, 0.5],[0.5, 1]],
    size=num_samples_per_class)
positive_samples = np.random.multivariate_normal(
    mean=[3, 0],
    cov=[[1, 0.5],[0.5, 1]],
    size=num_samples_per_class)
```

Generate the first class of points: 1000 random 2D points. cov=[[1, 0.5],[0.5, 1]] corresponds to an oval-like point cloud oriented from bottom left to top right.

Generate the other class of points with a different mean and the same covariance matrix.

In the preceding code, `negative_samples` and `positive_samples` are both arrays with shape `(1000, 2)`. Let's stack them into a single array with shape `(2000, 2)`.

Listing 3.14 Stacking the two classes into an array with shape (2000, 2)

```
inputs = np.vstack((negative_samples, positive_samples)).astype(np.float32)
```

Let's generate the corresponding target labels, an array of zeros and ones of shape `(2000, 1)`, where `targets[i, 0]` is 0 if `inputs[i]` belongs to class 0 (and inversely).

Listing 3.15 Generating the corresponding targets (0 and 1)

```
targets = np.vstack((np.zeros((num_samples_per_class, 1), dtype="float32"),
                     np.ones((num_samples_per_class, 1), dtype="float32")))
```

Next, let's plot our data with Matplotlib.

Listing 3.16 Plotting the two point classes (see figure 3.6)

```
import matplotlib.pyplot as plt
plt.scatter(inputs[:, 0], inputs[:, 1], c=targets[:, 0])
plt.show()
```

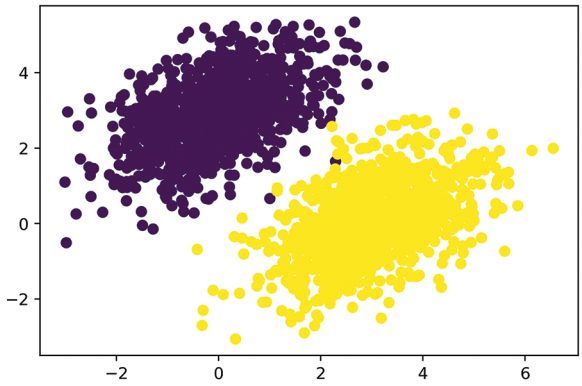


Figure 3.6 Our synthetic data: two classes of random points in the 2D plane

Now let's create a linear classifier that can learn to separate these two blobs. A linear classifier is an affine transformation (`prediction = W • input + b`) trained to minimize the square of the difference between predictions and the targets.

As you'll see, it's actually a much simpler example than the end-to-end example of a toy two-layer neural network you saw at the end of chapter 2. However, this time you should be able to understand everything about the code, line by line.

Let's create our variables, `W` and `b`, initialized with random values and with zeros, respectively.

Listing 3.17 Creating the linear classifier variables

The inputs will be 2D points.

The output predictions will be a single score per sample (close to 0 if the sample is predicted to be in class 0, and close to 1 if the sample is predicted to be in class 1).

```
input_dim = 2
output_dim = 1
W = tf.Variable(initial_value=tf.random.uniform(shape=(input_dim, output_dim)))
b = tf.Variable(initial_value=tf.zeros(shape=(output_dim,)))
```

Here's our forward pass function.

Listing 3.18 The forward pass function

```
def model(inputs):
    return tf.matmul(inputs, W) + b
```

Because our linear classifier operates on 2D inputs, `W` is really just two scalar coefficients, `w1` and `w2`: `W = [[w1], [w2]]`. Meanwhile, `b` is a single scalar coefficient. As such, for a given input point `[x, y]`, its prediction value is `prediction = [[w1], [w2]] • [x, y] + b = w1 * x + w2 * y + b`.

The following listing shows our loss function.

Listing 3.19 The mean squared error loss function

```
def square_loss(targets, predictions):
    per_sample_losses = tf.square(targets - predictions)
    return tf.reduce_mean(per_sample_losses)
```

per_sample_losses will be a tensor with the same shape as targets and predictions, containing per-sample loss scores.

We need to average these per-sample loss scores into a single scalar loss value: this is what reduce_mean does.

Next is the training step, which receives some training data and updates the weights `W` and `b` so as to minimize the loss on the data.

Listing 3.20 The training step function

```
learning_rate = 0.1

def training_step(inputs, targets):
    with tf.GradientTape() as tape:
        predictions = model(inputs)
        loss = square_loss(predictions, targets)
    grad_loss_wrt_W, grad_loss_wrt_b = tape.gradient(loss, [W, b])
    W.assign_sub(grad_loss_wrt_W * learning_rate)
    b.assign_sub(grad_loss_wrt_b * learning_rate)
    return loss
```

Retrieve the gradient of the loss with regard to weights.

Forward pass, inside a gradient tape scope

Update the weights.

For simplicity, we'll do *batch training* instead of *mini-batch training*: we'll run each training step (gradient computation and weight update) for all the data, rather than iterate over the data in small batches. On one hand, this means that each training step will take much longer to run, since we'll compute the forward pass and the gradients for 2,000 samples at once. On the other hand, each gradient update will be much more effective at reducing the loss on the training data, since it will encompass information from all training samples instead of, say, only 128 random samples. As a result, we will need many fewer steps of training, and we should use a larger learning rate than we would typically use for mini-batch training (we'll use `learning_rate = 0.1`, defined in listing 3.20).

Listing 3.21 The batch training loop

```
for step in range(40):
    loss = training_step(inputs, targets)
    print(f"Loss at step {step}: {loss:.4f}")
```

After 40 steps, the training loss seems to have stabilized around 0.025. Let's plot how our linear model classifies the training data points. Because our targets are zeros and ones, a given input point will be classified as "0" if its prediction value is below 0.5, and as "1" if it is above 0.5 (see figure 3.7):

```
predictions = model(inputs)
plt.scatter(inputs[:, 0], inputs[:, 1], c=predictions[:, 0] > 0.5)
plt.show()
```

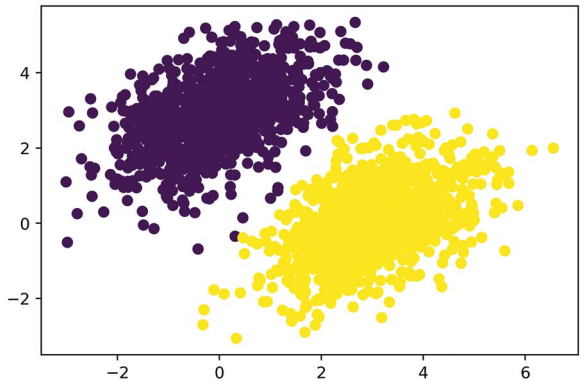


Figure 3.7 Our model's predictions on the training inputs: pretty similar to the training targets

Recall that the prediction value for a given point `[x, y]` is simply `prediction == [[w1], [w2]] • [x, y] + b == w1 * x + w2 * y + b`. Thus, class 0 is defined as `w1 * x + w2 * y + b < 0.5`, and class 1 is defined as `w1 * x + w2 * y + b > 0.5`. You'll notice that what you're looking at is really the equation of a line in the 2D plane: `w1 * x + w2 * y + b = 0.5`. Above the line is class 1, and below the line is class 0. You may be used to seeing line equations in the format `y = a * x + b`; in the same format, our line becomes `y = - w1 / w2 * x + (0.5 - b) / w2`.

Let's plot this line (shown in figure 3.8):

Generate 100 regularly spaced numbers between –1 and 4, which we will use to plot our line.

This is our line's equation.

Plot our line ("-r" means "plot it as a red line").

```
x = np.linspace(-1, 4, 100)
y = - W[0] /  W[1] * x + (0.5 - b) / W[1]
plt.plot(x, y, "-r")
plt.scatter(inputs[:, 0], inputs[:, 1], c=predictions[:, 0] > 0.5)
```

Plot our model's predictions on the same plot.

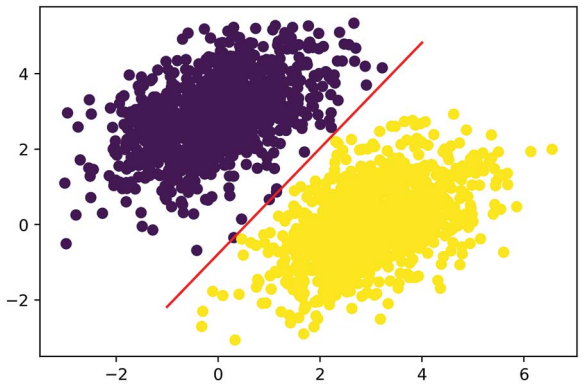


Figure 3.8 Our model, visualized as a line

This is really what a linear classifier is all about: finding the parameters of a line (or, in higher-dimensional spaces, a hyperplane) neatly separating two classes of data.

3.6 Anatomy of a neural network: Understanding core Keras APIs

At this point, you know the basics of TensorFlow, and you can use it to implement a toy model from scratch, such as the batch linear classifier in the previous section, or the toy neural network at the end of chapter 2. That's a solid foundation to build upon. It's now time to move on to a more productive, more robust path to deep learning: the Keras API.

3.6.1 Layers: The building blocks of deep learning

The fundamental data structure in neural networks is the *layer*, to which you were introduced in chapter 2. A layer is a data processing module that takes as input one or more tensors and that outputs one or more tensors. Some layers are stateless, but more frequently layers have a state: the layer's *weights*, one or several tensors learned with stochastic gradient descent, which together contain the network's *knowledge*.

Different types of layers are appropriate for different tensor formats and different types of data processing. For instance, simple vector data, stored in rank-2 tensors of shape `(samples, features)`, is often processed by *densely connected* layers, also called *fully connected* or *dense* layers (the `Dense` class in Keras). Sequence data, stored in rank-3 tensors of shape `(samples, timesteps, features)`, is typically processed by *recurrent* layers, such as an `LSTM` layer, or 1D convolution layers (`Conv1D`). Image data, stored in rank-4 tensors, is usually processed by 2D convolution layers (`Conv2D`).

You can think of layers as the LEGO bricks of deep learning, a metaphor that is made explicit by Keras. Building deep learning models in Keras is done by clipping together compatible layers to form useful data-transformation pipelines.

THE BASE LAYER CLASS IN KERAS

A simple API should have a single abstraction around which everything is centered. In Keras, that's the `Layer` class. Everything in Keras is either a `Layer` or something that closely interacts with a `Layer`.

A `Layer` is an object that encapsulates some state (weights) and some computation (a forward pass). The weights are typically defined in a `build()` (although they could also be created in the constructor, `__init__()`), and the computation is defined in the `call()` method.

In the previous chapter, we implemented a `NaiveDense` class that contained two weights `W` and `b` and applied the computation `output = activation(dot(input, W) + b)`. This is what the same layer would look like in Keras.

Listing 3.22 A `Dense` layer implemented as a `Layer` subclass

```
from tensorflow import keras

class SimpleDense(keras.layers.Layer):
```

All Keras layers inherit from the base Layer class.

```
def __init__(self, units, activation=None):
    super().__init__()
    self.units = units
    self.activation = activation

def build(self, input_shape):
    input_dim = input_shape[-1]
    self.W = self.add_weight(shape=(input_dim, self.units),
                             initializer="random_normal")
    self.b = self.add_weight(shape=(self.units,),
                             initializer="zeros")

def call(self, inputs):
    y = tf.matmul(inputs, self.W) + self.b
    if self.activation is not None:
        y = self.activation(y)
    return y
```

Weight creation takes place in the build() method.

We define the forward pass computation in the call() method.

add_weight() is a shortcut method for creating weights. It is also possible to create standalone variables and assign them as layer attributes, like self.W = tf.Variable(tf.random.uniform(w_shape)).

In the next section, we'll cover in detail the purpose of these `build()` and `call()` methods. Don't worry if you don't understand everything just yet!

Once instantiated, a layer like this can be used just like a function, taking as input a TensorFlow tensor:

```
>>> my_dense = SimpleDense(units=32, activation=tf.nn.relu)
>>> input_tensor = tf.ones(shape=(2, 784))
>>> output_tensor = my_dense(input_tensor)
>>> print(output_tensor.shape)
(2, 32))
```

Instantiate our layer, defined previously.

Create some test inputs.

Call the layer on the inputs, just like a function.

You're probably wondering, why did we have to implement `call()` and `build()`, since we ended up using our layer by plainly calling it, that is to say, by using its `__call__()` method? It's because we want to be able to create the state just in time. Let's see how that works.

Automatic shape inference: Building layers on the fly

Just like with LEGO bricks, you can only "clip" together layers that are compatible. The notion of *layer compatibility* here refers specifically to the fact that every layer will only accept input tensors of a certain shape and will return output tensors of a certain shape. Consider the following example:

```
from tensorflow.keras import layers
layer = layers.Dense(32, activation="relu")
```

A dense layer with 32 output units

This layer will return a tensor where the first dimension has been transformed to be 32. It can only be connected to a downstream layer that expects 32-dimensional vectors as its input.

When using Keras, you don't have to worry about size compatibility most of the time, because the layers you add to your models are dynamically built to match the shape of the incoming layer. For instance, suppose you write the following:

```
from tensorflow.keras import models
from tensorflow.keras import layers
model = models.Sequential([
    layers.Dense(32, activation="relu"),
    layers.Dense(32)
])
```

The layers didn't receive any information about the shape of their inputs—instead, they automatically inferred their input shape as being the shape of the first inputs they see.

In the toy version of the `Dense` layer we implemented in chapter 2 (which we named `NaiveDense`), we had to pass the layer's input size explicitly to the constructor in order to be able to create its weights. That's not ideal, because it would lead to models that look like this, where each new layer needs to be made aware of the shape of the layer before it:

```
model = NaiveSequential([
    NaiveDense(input_size=784, output_size=32, activation="relu"),
    NaiveDense(input_size=32, output_size=64, activation="relu"),
    NaiveDense(input_size=64, output_size=32, activation="relu"),
    NaiveDense(input_size=32, output_size=10, activation="softmax")
])
```

It would be even worse if the rules used by a layer to produce its output shape are complex. For instance, what if our layer returned outputs of shape `(batch, input_size * 2 if input_size % 2 == 0 else input_size * 3)`?

If we were to reimplement our `NaiveDense` layer as a Keras layer capable of automatic shape inference, it would look like the previous `SimpleDense` layer (see listing 3.22), with its `build()` and `call()` methods.

In `SimpleDense`, we no longer create weights in the constructor like in the `NaiveDense` example; instead, we create them in a dedicated state-creation method, `build()`, which receives as an argument the first input shape seen by the layer. The `build()` method is called automatically the first time the layer is called (via its `__call__()` method). In fact, that's why we defined the computation in a separate `call()` method rather than in the `__call__()` method directly. The `__call__()` method of the base layer schematically looks like this:

```
def __call__(self, inputs):
    if not self.built:
        self.build(inputs.shape)
        self.built = True
    return self.call(inputs)
```

With automatic shape inference, our previous example becomes simple and neat:

```
model = keras.Sequential([
    SimpleDense(32, activation="relu"),
    SimpleDense(64, activation="relu"),
```

```
    SimpleDense(32, activation="relu"),
    SimpleDense(10, activation="softmax")
])
```

Note that automatic shape inference is not the only thing that the `Layer` class's `__call__()` method handles. It takes care of many more things, in particular routing between *eager* and *graph* execution (a concept you'll learn about in chapter 7), and input masking (which we'll cover in chapter 11). For now, just remember: when implementing your own layers, put the forward pass in the `call()` method.

3.6.2 From layers to models

A deep learning model is a graph of layers. In Keras, that's the `Model` class. Until now, you've only seen `Sequential` models (a subclass of `Model`), which are simple stacks of layers, mapping a single input to a single output. But as you move forward, you'll be exposed to a much broader variety of network topologies. These are some common ones:

- Two-branch networks
- Multihead networks
- Residual connections

Network topology can get quite involved. For instance, figure 3.9 shows the topology of the graph of layers of a Transformer, a common architecture designed to process text data.

There are generally two ways of building such models in Keras: you could directly subclass the `Model` class, or you could use the Functional API, which lets you do more with less code. We'll cover both approaches in chapter 7.

The topology of a model defines a *hypothesis space*. You may remember that in chapter 1 we described machine learning as searching for useful representations of some input data, within a predefined *space of possibilities*, using guidance from a feedback signal. By choosing a network topology, you constrain your space of possibilities (hypothesis space) to a specific series of tensor operations, mapping input data to output data. What you'll then be searching for is a good set of values for the weight tensors involved in these tensor operations.

To learn from data, you have to make assumptions about it. These assumptions define what can be learned. As such, the structure of your hypothesis space—the architecture of your model—is extremely important. It encodes the assumptions you make about your problem, the prior knowledge that the model starts with. For instance, if you're working on a two-class classification problem with a model made of a single `Dense` layer with no activation (a pure affine transformation), you are assuming that your two classes are linearly separable.

Picking the right network architecture is more an art than a science, and although there are some best practices and principles you can rely on, only practice can help you become a proper neural-network architect. The next few chapters will both teach

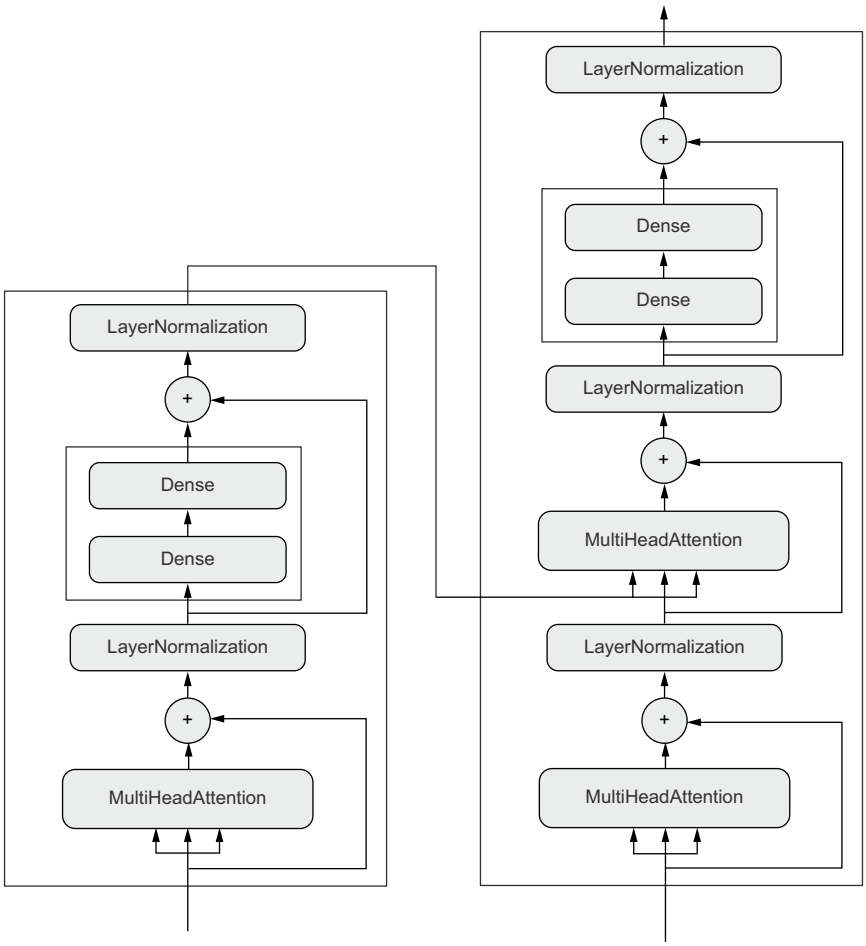


Figure 3.9 The Transformer architecture (covered in chapter 11). There's a lot going on here. Throughout the next few chapters, you'll climb your way up to understanding it.

you explicit principles for building neural networks and help you develop intuition as to what works or doesn't work for specific problems. You'll build a solid intuition about what type of model architectures work for different kinds of problems, how to build these networks in practice, how to pick the right learning configuration, and how to tweak a model until it yields the results you want to see.

3.6.3 The "compile" step: Configuring the learning process

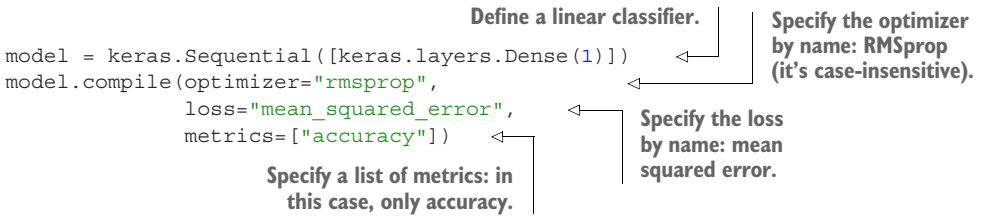
Once the model architecture is defined, you still have to choose three more things:

- *Loss function (objective function)*—The quantity that will be minimized during training. It represents a measure of success for the task at hand.

- *Optimizer*—Determines how the network will be updated based on the loss function. It implements a specific variant of stochastic gradient descent (SGD).
- *Metrics*—The measures of success you want to monitor during training and validation, such as classification accuracy. Unlike the loss, training will not optimize directly for these metrics. As such, metrics don't need to be differentiable.

Once you've picked your loss, optimizer, and metrics, you can use the built-in `compile()` and `fit()` methods to start training your model. Alternatively, you could also write your own custom training loops—we'll cover how to do this in chapter 7. It's a lot more work! For now, let's take a look at `compile()` and `fit()`.

The `compile()` method configures the training process—you've already been introduced to it in your very first neural network example in chapter 2. It takes the arguments `optimizer`, `loss`, and `metrics` (a list):




```
model = keras.Sequential([keras.layers.Dense(1)])
model.compile(optimizer="rmsprop",
              loss="mean_squared_error",
              metrics=["accuracy"])
```

In the preceding call to `compile()`, we passed the optimizer, loss, and metrics as strings (such as `"rmsprop"`). These strings are actually shortcuts that get converted to Python objects. For instance, `"rmsprop"` becomes `keras.optimizers.RMSprop()`. Importantly, it's also possible to specify these arguments as object instances, like this:

```
model.compile(optimizer=keras.optimizers.RMSprop(),
              loss=keras.losses.MeanSquaredError(),
              metrics=[keras.metrics.BinaryAccuracy()])
```

This is useful if you want to pass your own custom losses or metrics, or if you want to further configure the objects you're using—for instance, by passing a `learning_rate` argument to the optimizer:

```
model.compile(optimizer=keras.optimizers.RMSprop(learning_rate=1e-4),
              loss=my_custom_loss,
              metrics=[my_custom_metric_1, my_custom_metric_2])
```

In chapter 7, we'll cover how to create custom losses and metrics. In general, you won't have to create your own losses, metrics, or optimizers from scratch, because Keras offers a wide range of built-in options that is likely to include what you need:

Optimizers:

- `SGD` (with or without momentum)
- `RMSprop`
- `Adam`

- `Adagrad`
- Etc.

Losses:

- `CategoricalCrossentropy`
- `SparseCategoricalCrossentropy`
- `BinaryCrossentropy`
- `MeanSquaredError`
- `KLDivergence`
- `CosineSimilarity`
- Etc.

Metrics:

- `CategoricalAccuracy`
- `SparseCategoricalAccuracy`
- `BinaryAccuracy`
- `AUC`
- `Precision`
- `Recall`
- Etc.

Throughout this book, you'll see concrete applications of many of these options.

3.6.4 Picking a loss function

Choosing the right loss function for the right problem is extremely important: your network will take any shortcut it can to minimize the loss, so if the objective doesn't fully correlate with success for the task at hand, your network will end up doing things you may not have wanted. Imagine a stupid, omnipotent AI trained via SGD with this poorly chosen objective function: "maximizing the average well-being of all humans alive." To make its job easier, this AI might choose to kill all humans except a few and focus on the well-being of the remaining ones—because average well-being isn't affected by how many humans are left. That might not be what you intended! Just remember that all neural networks you build will be just as ruthless in lowering their loss function—so choose the objective wisely, or you'll have to face unintended side effects.

Fortunately, when it comes to common problems such as classification, regression, and sequence prediction, there are simple guidelines you can follow to choose the correct loss. For instance, you'll use binary crossentropy for a two-class classification problem, categorical crossentropy for a many-class classification problem, and so on. Only when you're working on truly new research problems will you have to develop your own loss functions. In the next few chapters, we'll detail explicitly which loss functions to choose for a wide range of common tasks.

3.6.5 Understanding the fit() method

After `compile()` comes `fit()`. The `fit()` method implements the training loop itself. These are its key arguments:

- The *data* (inputs and targets) to train on. It will typically be passed either in the form of NumPy arrays or a TensorFlow `Dataset` object. You'll learn more about the `Dataset` API in the next chapters.
- The number of *epochs* to train for: how many times the training loop should iterate over the data passed.
- The batch size to use within each epoch of mini-batch gradient descent: the number of training examples considered to compute the gradients for one weight update step.

Listing 3.23 Calling `fit()` with NumPy data

```
history = model.fit(
    inputs,
    targets,
    epochs=5,
    batch_size=128
)
```

The input examples, as a NumPy array

The corresponding training targets, as a NumPy array

The training loop will iterate over the data 5 times.

The training loop will iterate over the data in batches of 128 examples.

The call to `fit()` returns a `History` object. This object contains a `history` field, which is a dict mapping keys such as `"loss"` or specific metric names to the list of their per-epoch values.

```
>>> history.history
{"binary_accuracy": [0.855, 0.9565, 0.9555, 0.95, 0.951],
 "loss": [0.6573270302042366,
          0.07434618508815766,
          0.07687718723714351,
          0.07412414988875389,
          0.07617757616937161]}
```

3.6.6 Monitoring loss and metrics on validation data

The goal of machine learning is not to obtain models that perform well on the training data, which is easy—all you have to do is follow the gradient. The goal is to obtain models that perform well in general, and particularly on data points that the model has never encountered before. Just because a model performs well on its training data doesn't mean it will perform well on data it has never seen! For instance, it's possible that your model could end up merely *memorizing* a mapping between your training samples and their targets, which would be useless for the task of predicting targets for data the model has never seen before. We'll go over this point in much more detail in chapter 5.

To keep an eye on how the model does on new data, it's standard practice to reserve a subset of the training data as *validation data*: you won't be training the model on this data, but you will use it to compute a loss value and metrics value. You do this by using the `validation_data` argument in `fit()`. Like the training data, the validation data could be passed as NumPy arrays or as a TensorFlow `Dataset` object.

Listing 3.24 Using the `validation_data` argument

```
model = keras.Sequential([keras.layers.Dense(1)])
model.compile(optimizer=keras.optimizers.RMSprop(learning_rate=0.1),
              loss=keras.losses.MeanSquaredError(),
              metrics=[keras.metrics.BinaryAccuracy()])

indices_permutation = np.random.permutation(len(inputs))
shuffled_inputs = inputs[indices_permutation]
shuffled_targets = targets[indices_permutation]

num_validation_samples = int(0.3 * len(inputs))
val_inputs = shuffled_inputs[:num_validation_samples]
val_targets = shuffled_targets[:num_validation_samples]
training_inputs = shuffled_inputs[num_validation_samples:]
training_targets = shuffled_targets[num_validation_samples:]
model.fit(
    training_inputs,
    training_targets,
    epochs=5,
    batch_size=16,
    validation_data=(val_inputs, val_targets)
)
```

To avoid having samples from only one class in the validation data, shuffle the inputs and targets using a random indices permutation.

Reserve 30% of the training inputs and targets for validation (we'll exclude these samples from training and reserve them to compute the validation loss and metrics).

Training data, used to update the weights of the model

Validation data, used only to monitor the validation loss and metrics

The value of the loss on the validation data is called the "validation loss," to distinguish it from the "training loss." Note that it's essential to keep the training data and validation data strictly separate: the purpose of validation is to monitor whether what the model is learning is actually useful on new data. If any of the validation data has been seen by the model during training, your validation loss and metrics will be flawed.

Note that if you want to compute the validation loss and metrics after the training is complete, you can call the `evaluate()` method:

```
loss_and_metrics = model.evaluate(val_inputs, val_targets, batch_size=128)
```

`evaluate()` will iterate in batches (of size `batch_size`) over the data passed and return a list of scalars, where the first entry is the validation loss and the following entries are the validation metrics. If the model has no metrics, only the validation loss is returned (rather than a list).

3.6.7 Inference: Using a model after training

Once you've trained your model, you're going to want to use it to make predictions on new data. This is called *inference*. To do this, a naive approach would simply be to `__call__()` the model:

```
predictions = model(new_inputs)
```

Takes a NumPy array or TensorFlow tensor and returns a TensorFlow tensor

However, this will process all inputs in `new_inputs` at once, which may not be feasible if you're looking at a lot of data (in particular, it may require more memory than your GPU has).

A better way to do inference is to use the `predict()` method. It will iterate over the data in small batches and return a NumPy array of predictions. And unlike `__call__()`, it can also process TensorFlow `Dataset` objects.

```
predictions = model.predict(new_inputs, batch_size=128)
```

Takes a NumPy array or a Dataset and returns a NumPy array

For instance, if we use `predict()` on some of our validation data with the linear model we trained earlier, we get scalar scores that correspond to the model's prediction for each input sample:

```
>>> predictions = model.predict(val_inputs, batch_size=128)
>>> print(predictions[:10])
[[0.3590725 ]
 [0.82706255]
 [0.74428225]
 [0.682058  ]
 [0.7312616 ]
 [0.6059811 ]
 [0.78046083]
 [0.025846  ]
 [0.16594526]
 [0.72068727]]
```

For now, this is all you need to know about Keras models. You are ready to move on to solving real-world machine learning problems with Keras in the next chapter.

Summary

- TensorFlow is an industry-strength numerical computing framework that can run on CPU, GPU, or TPU. It can automatically compute the gradient of any differentiable expression, it can be distributed to many devices, and it can export programs to various external runtimes—even JavaScript.
- Keras is the standard API for doing deep learning with TensorFlow. It's what we'll use throughout this book.
- Key TensorFlow objects include tensors, variables, tensor operations, and the gradient tape.

- The central class of Keras is the `Layer`. A *layer* encapsulates some weights and some computation. Layers are assembled into *models*.
- Before you start training a model, you need to pick an *optimizer*, a *loss*, and some *metrics*, which you specify via the `model.compile()` method.
- To train a model, you can use the `fit()` method, which runs mini-batch gradient descent for you. You can also use it to monitor your loss and metrics on *validation data*, a set of inputs that the model doesn't see during training.
- Once your model is trained, you use the `model.predict()` method to generate predictions on new inputs.

4 Getting started with neural networks: Classification and regression

This chapter covers

- Your first examples of real-world machine learning workflows
- Handling classification problems over vector data
- Handling continuous regression problems over vector data

This chapter is designed to get you started using neural networks to solve real problems. You'll consolidate the knowledge you gained from chapters 2 and 3, and you'll apply what you've learned to three new tasks covering the three most common use cases of neural networks—binary classification, multiclass classification, and scalar regression:

- Classifying movie reviews as positive or negative (binary classification)
- Classifying news wires by topic (multiclass classification)
- Estimating the price of a house, given real-estate data (scalar regression)

These examples will be your first contact with end-to-end machine learning workflows: you'll get introduced to data preprocessing, basic model architecture principles, and model evaluation.

Classification and regression glossary

Classification and regression involve many specialized terms. You've come across some of them in earlier examples, and you'll see more of them in future chapters. They have precise, machine learning–specific definitions, and you should be familiar with them:

- *Sample* or *input*—One data point that goes into your model.
- *Prediction* or *output*—What comes out of your model.
- *Target*—The truth. What your model should ideally have predicted, according to an external source of data.
- *Prediction error* or *loss value*—A measure of the distance between your model's prediction and the target.
- *Classes*—A set of possible labels to choose from in a classification problem. For example, when classifying cat and dog pictures, "dog" and "cat" are the two classes.
- *Label*—A specific instance of a class annotation in a classification problem. For instance, if picture #1234 is annotated as containing the class "dog," then "dog" is a label of picture #1234.
- *Ground-truth* or *annotations*—All targets for a dataset, typically collected by humans.
- *Binary classification*—A classification task where each input sample should be categorized into two exclusive categories.
- *Multiclass classification*—A classification task where each input sample should be categorized into more than two categories: for instance, classifying handwritten digits.
- *Multilabel classification*—A classification task where each input sample can be assigned multiple labels. For instance, a given image may contain both a cat and a dog and should be annotated both with the "cat" label and the "dog" label. The number of labels per image is usually variable.
- *Scalar regression*—A task where the target is a continuous scalar value. Predicting house prices is a good example: the different target prices form a continuous space.
- *Vector regression*—A task where the target is a set of continuous values: for example, a continuous vector. If you're doing regression against multiple values (such as the coordinates of a bounding box in an image), then you're doing vector regression.
- *Mini-batch* or *batch*—A small set of samples (typically between 8 and 128) that are processed simultaneously by the model. The number of samples is often a power of 2, to facilitate memory allocation on GPU. When training, a mini-batch is used to compute a single gradient-descent update applied to the weights of the model.

By the end of this chapter, you'll be able to use neural networks to handle simple classification and regression tasks over vector data. You'll then be ready to start building a more principled, theory-driven understanding of machine learning in chapter 5.

4.1 Classifying movie reviews: A binary classification example

Two-class classification, or binary classification, is one of the most common kinds of machine learning problems. In this example, you'll learn to classify movie reviews as positive or negative, based on the text content of the reviews.

4.1.1 The IMDB dataset

You'll work with the IMDB dataset: a set of 50,000 highly polarized reviews from the Internet Movie Database. They're split into 25,000 reviews for training and 25,000 reviews for testing, each set consisting of 50% negative and 50% positive reviews.

Just like the MNIST dataset, the IMDB dataset comes packaged with Keras. It has already been preprocessed: the reviews (sequences of words) have been turned into sequences of integers, where each integer stands for a specific word in a dictionary. This enables us to focus on model building, training, and evaluation. In chapter 11, you'll learn how to process raw text input from scratch.

The following code will load the dataset (when you run it the first time, about 80 MB of data will be downloaded to your machine).

Listing 4.1 Loading the IMDB dataset

```
from tensorflow.keras.datasets import imdb
(train_data, train_labels), (test_data, test_labels) = imdb.load_data(
    num_words=10000)
```

The argument `num_words=10000` means you'll only keep the top 10,000 most frequently occurring words in the training data. Rare words will be discarded. This allows us to work with vector data of manageable size. If we didn't set this limit, we'd be working with 88,585 unique words in the training data, which is unnecessarily large. Many of these words only occur in a single sample, and thus can't be meaningfully used for classification.

The variables `train_data` and `test_data` are lists of reviews; each review is a list of word indices (encoding a sequence of words). `train_labels` and `test_labels` are lists of 0s and 1s, where 0 stands for *negative* and 1 stands for *positive*:

```
>>> train_data[0]
[1, 14, 22, 16, ... 178, 32]
>>> train_labels[0]
1
```

Because we're restricting ourselves to the top 10,000 most frequent words, no word index will exceed 10,000:

```
>>> max([max(sequence) for sequence in train_data])
9999
```

For kicks, here's how you can quickly decode one of these reviews back to English words.

Listing 4.2 Decoding reviews back to text

```
word_index = imdb.get_word_index()
reverse_word_index = dict(
    [(value, key) for (key, value) in word_index.items()])
decoded_review = " ".join(
    [reverse_word_index.get(i - 3, "?") for i in train_data[0]])
```

word_index is a dictionary mapping words to an integer index.

Reverses it, mapping integer indices to words

Decodes the review. Note that the indices are offset by 3 because 0, 1, and 2 are reserved indices for "padding," "start of sequence," and "unknown."

4.1.2 Preparing the data

You can't directly feed lists of integers into a neural network. They all have different lengths, but a neural network expects to process contiguous batches of data. You have to turn your lists into tensors. There are two ways to do that:

- Pad your lists so that they all have the same length, turn them into an integer tensor of shape `(samples, max_length)`, and start your model with a layer capable of handling such integer tensors (the `Embedding` layer, which we'll cover in detail later in the book).
- *Multi-hot encode* your lists to turn them into vectors of 0s and 1s. This would mean, for instance, turning the sequence `[8, 5]` into a 10,000-dimensional vector that would be all 0s except for indices 8 and 5, which would be 1s. Then you could use a `Dense` layer, capable of handling floating-point vector data, as the first layer in your model.

Let's go with the latter solution to vectorize the data, which you'll do manually for maximum clarity.

Listing 4.3 Encoding the integer sequences via multi-hot encoding

```
import numpy as np
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        for j in sequence:
            results[i, j] = 1.
    return results
x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
```

Creates an all-zero matrix of shape (len(sequences), dimension)

Sets specific indices of results[i] to 1s

Vectorized training data

Vectorized test data

Here's what the samples look like now:

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

You should also vectorize your labels, which is straightforward:

```
y_train = np.asarray(train_labels).astype("float32")
y_test = np.asarray(test_labels).astype("float32")
```

Now the data is ready to be fed into a neural network.

4.1.3 Building your model

The input data is vectors, and the labels are scalars (1s and 0s): this is one of the simplest problem setups you'll ever encounter. A type of model that performs well on such a problem is a plain stack of densely connected (`Dense`) layers with `relu` activations.

There are two key architecture decisions to be made about such a stack of `Dense` layers:

- How many layers to use
- How many units to choose for each layer

In chapter 5, you'll learn formal principles to guide you in making these choices. For the time being, you'll have to trust me with the following architecture choices:

- Two intermediate layers with 16 units each
- A third layer that will output the scalar prediction regarding the sentiment of the current review

Figure 4.1 shows what the model looks like. And the following listing shows the Keras implementation, similar to the MNIST example you saw previously.

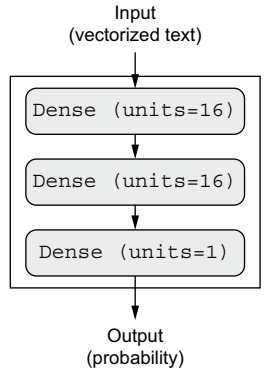


Figure 4.1 The three-layer model

Listing 4.4 Model definition

```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    layers.Dense(16, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
```

The first argument being passed to each `Dense` layer is the number of *units* in the layer: the dimensionality of representation space of the layer. You remember from chapters 2 and 3 that each such `Dense` layer with a `relu` activation implements the following chain of tensor operations:

```
output = relu(dot(input, W) + b)
```

Having 16 units means the weight matrix `W` will have shape `(input_dimension, 16)`: the dot product with `W` will project the input data onto a 16-dimensional representation space (and then you'll add the bias vector `b` and apply the `relu` operation). You can intuitively understand the dimensionality of your representation space as "how much freedom you're allowing the model to have when learning internal representations." Having more units (a higher-dimensional representation space) allows your model to learn more-complex representations, but it makes the model more computationally expensive and may lead to learning unwanted patterns (patterns that will improve performance on the training data but not on the test data).

The intermediate layers use `relu` as their activation function, and the final layer uses a sigmoid activation so as to output a probability (a score between 0 and 1 indicating how likely the sample is to have the target "1": how likely the review is to be positive). A `relu` (rectified linear unit) is a function meant to zero out negative values (see figure 4.2), whereas a sigmoid "squashes" arbitrary values into the `[0, 1]` interval (see figure 4.3), outputting something that can be interpreted as a probability.

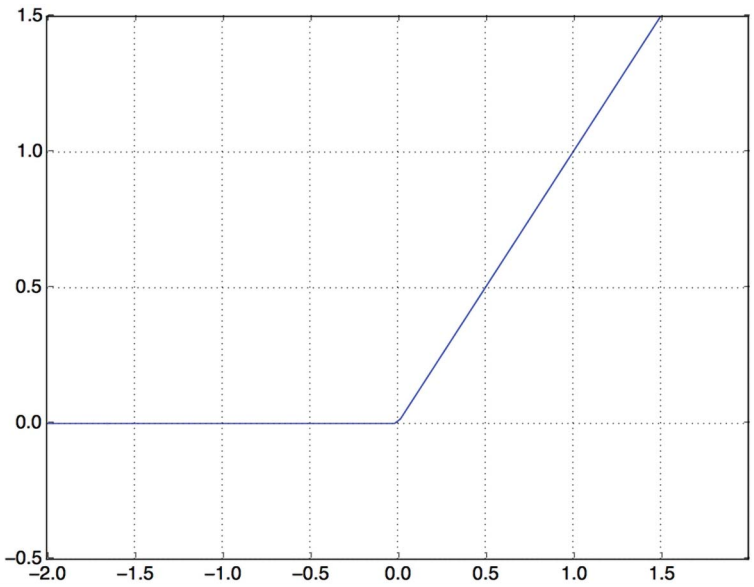


Figure 4.2 The rectified linear unit function

Finally, you need to choose a loss function and an optimizer. Because you're facing a binary classification problem and the output of your model is a probability (you end your model with a single-unit layer with a sigmoid activation), it's best to use the `binary_crossentropy` loss. It isn't the only viable choice: for instance, you could use `mean_squared_error`. But crossentropy is usually the best choice when you're dealing

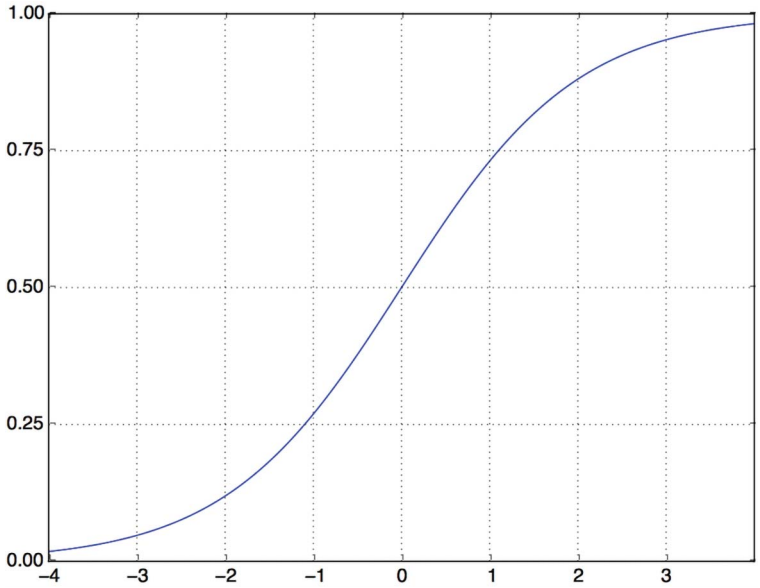


Figure 4.3 The sigmoid function

What are activation functions, and why are they necessary?

Without an activation function like `relu` (also called a *non-linearity*), the `Dense` layer would consist of two linear operations—a dot product and an addition:

```
output = dot(input, W) + b
```

The layer could only learn *linear transformations* (affine transformations) of the input data: the *hypothesis space* of the layer would be the set of all possible linear transformations of the input data into a 16-dimensional space. Such a hypothesis space is too restricted and wouldn't benefit from multiple layers of representations, because a deep stack of linear layers would still implement a linear operation: adding more layers wouldn't extend the hypothesis space (as you saw in chapter 2).

In order to get access to a much richer hypothesis space that will benefit from deep representations, you need a non-linearity, or activation function. `relu` is the most popular activation function in deep learning, but there are many other candidates, which all come with similarly strange names: `prelu`, `elu`, and so on.

with models that output probabilities. *Crossentropy* is a quantity from the field of information theory that measures the distance between probability distributions or, in this case, between the ground-truth distribution and your predictions.

As for the choice of the optimizer, we'll go with `rmsprop`, which is a usually a good default choice for virtually any problem.

Here's the step where we configure the model with the `rmsprop` optimizer and the `binary_crossentropy` loss function. Note that we'll also monitor accuracy during training.

Listing 4.5 Compiling the model

```
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
```

4.1.4 Validating your approach

As you learned in chapter 3, a deep learning model should never be evaluated on its training data—it's standard practice to use a validation set to monitor the accuracy of the model during training. Here, we'll create a validation set by setting apart 10,000 samples from the original training data.

Listing 4.6 Setting aside a validation set

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]
y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

We will now train the model for 20 epochs (20 iterations over all samples in the training data) in mini-batches of 512 samples. At the same time, we will monitor loss and accuracy on the 10,000 samples that we set apart. We do so by passing the validation data as the `validation_data` argument.

Listing 4.7 Training your model

```
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

On CPU, this will take less than 2 seconds per epoch—training is over in 20 seconds. At the end of every epoch, there is a slight pause as the model computes its loss and accuracy on the 10,000 samples of the validation data.

Note that the call to `model.fit()` returns a `History` object, as you saw in chapter 3. This object has a member `history`, which is a dictionary containing data about everything that happened during training. Let's look at it:

```
>>> history_dict = history.history
>>> history_dict.keys()
[u"accuracy", u"loss", u"val_accuracy", u"val_loss"]
```

The dictionary contains four entries: one per metric that was being monitored during training and during validation. In the following two listings, let's use Matplotlib to plot the training and validation loss side by side (see figure 4.4), as well as the training and validation accuracy (see figure 4.5). Note that your own results may vary slightly due to a different random initialization of your model.

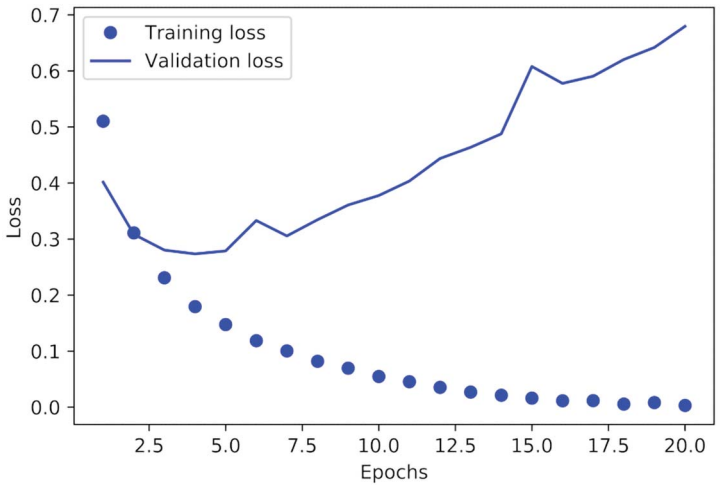


Figure 4.4 Training and validation loss

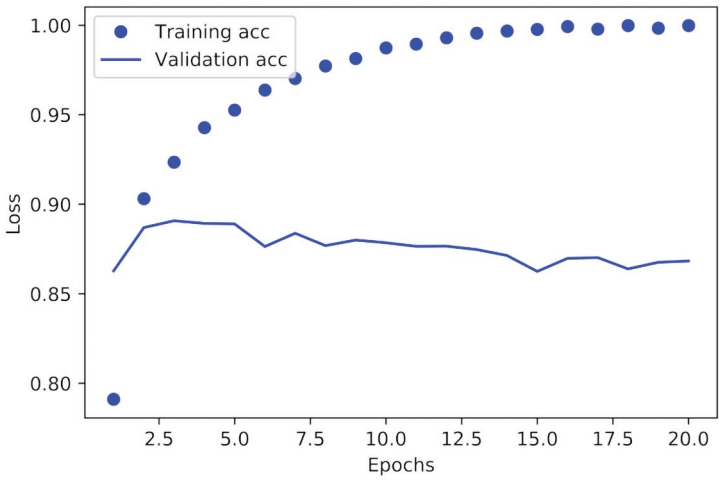


Figure 4.5 Training and validation accuracy

Listing 4.8 Plotting the training and validation loss

```
import matplotlib.pyplot as plt
history_dict = history.history
loss_values = history_dict["loss"]
val_loss_values = history_dict["val_loss"]
epochs = range(1, len(loss_values) + 1)
plt.plot(epochs, loss_values, "bo", label="Training loss")
plt.plot(epochs, val_loss_values, "b", label="Validation loss")
plt.title("Training and validation loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()
```

"bo" is for "blue dot."

"b" is for "solid blue line."

Listing 4.9 Plotting the training and validation accuracy

```
plt.clf()
acc = history_dict["accuracy"]
val_acc = history_dict["val_accuracy"]
plt.plot(epochs, acc, "bo", label="Training acc")
plt.plot(epochs, val_acc, "b", label="Validation acc")
plt.title("Training and validation accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```

Clears the figure

As you can see, the training loss decreases with every epoch, and the training accuracy increases with every epoch. That's what you would expect when running gradient-descent optimization—the quantity you're trying to minimize should be less with every iteration. But that isn't the case for the validation loss and accuracy: they seem to peak at the fourth epoch. This is an example of what we warned against earlier: a model that performs better on the training data isn't necessarily a model that will do better on data it has never seen before. In precise terms, what you're seeing is *overfitting*: after the fourth epoch, you're overoptimizing on the training data, and you end up learning representations that are specific to the training data and don't generalize to data outside of the training set.

In this case, to prevent overfitting, you could stop training after four epochs. In general, you can use a range of techniques to mitigate overfitting, which we'll cover in chapter 5.

Let's train a new model from scratch for four epochs and then evaluate it on the test data.

Listing 4.10 Retraining a model from scratch

```
model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    layers.Dense(16, activation="relu"),
```

```
    layers.Dense(1, activation="sigmoid")
])
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.fit(x_train, y_train, epochs=4, batch_size=512)
results = model.evaluate(x_test, y_test)
```

The final results are as follows:

```
>>> results
[0.2929924130630493, 0.88327999999999995]
```

The first number, 0.29, is the test loss, and the second number, 0.88, is the test accuracy.

This fairly naive approach achieves an accuracy of 88%. With state-of-the-art approaches, you should be able to get close to 95%.

4.1.5 Using a trained model to generate predictions on new data

After having trained a model, you'll want to use it in a practical setting. You can generate the likelihood of reviews being positive by using the `predict` method, as you've learned in chapter 3:

```
>>> model.predict(x_test)
array([[ 0.98006207]
       [ 0.99758697]
       [ 0.99975556]
       ...,
       [ 0.82167041]
       [ 0.02885115]
       [ 0.65371346]], dtype=float32)
```

As you can see, the model is confident for some samples (0.99 or more, or 0.01 or less) but less confident for others (0.6, 0.4).

4.1.6 Further experiments

The following experiments will help convince you that the architecture choices you've made are all fairly reasonable, although there's still room for improvement:

- You used two representation layers before the final classification layer. Try using one or three representation layers, and see how doing so affects validation and test accuracy.
- Try using layers with more units or fewer units: 32 units, 64 units, and so on.
- Try using the `mse` loss function instead of `binary_crossentropy`.
- Try using the `tanh` activation (an activation that was popular in the early days of neural networks) instead of `relu`.

4.1.7 Wrapping up

Here's what you should take away from this example:

- You usually need to do quite a bit of preprocessing on your raw data in order to be able to feed it—as tensors—into a neural network. Sequences of words can be encoded as binary vectors, but there are other encoding options too.
- Stacks of `Dense` layers with `relu` activations can solve a wide range of problems (including sentiment classification), and you'll likely use them frequently.
- In a binary classification problem (two output classes), your model should end with a `Dense` layer with one unit and a `sigmoid` activation: the output of your model should be a scalar between 0 and 1, encoding a probability.
- With such a scalar sigmoid output on a binary classification problem, the loss function you should use is `binary_crossentropy`.
- The `rmsprop` optimizer is generally a good enough choice, whatever your problem. That's one less thing for you to worry about.
- As they get better on their training data, neural networks eventually start overfitting and end up obtaining increasingly worse results on data they've never seen before. Be sure to always monitor performance on data that is outside of the training set.

4.2 Classifying newswires: A multiclass classification example

In the previous section, you saw how to classify vector inputs into two mutually exclusive classes using a densely connected neural network. But what happens when you have more than two classes?

In this section, we'll build a model to classify Reuters newswires into 46 mutually exclusive topics. Because we have many classes, this problem is an instance of *multiclass classification*, and because each data point should be classified into only one category, the problem is more specifically an instance of *single-label multiclass classification*. If each data point could belong to multiple categories (in this case, topics), we'd be facing a *multilabel multiclass classification* problem.

4.2.1 The Reuters dataset

You'll work with the *Reuters dataset*, a set of short newswires and their topics, published by Reuters in 1986. It's a simple, widely used toy dataset for text classification. There are 46 different topics; some topics are more represented than others, but each topic has at least 10 examples in the training set.

Like IMDB and MNIST, the Reuters dataset comes packaged as part of Keras. Let's take a look.

Listing 4.11 Loading the Reuters dataset

```
from tensorflow.keras.datasets import reuters
(train_data, train_labels), (test_data, test_labels) = reuters.load_data(
    num_words=10000)
```

As with the IMDB dataset, the argument `num_words=10000` restricts the data to the 10,000 most frequently occurring words found in the data.

You have 8,982 training examples and 2,246 test examples:

```
>>> len(train_data)
8982
>>> len(test_data)
2246
```

As with the IMDB reviews, each example is a list of integers (word indices):

```
>>> train_data[10]
[1, 245, 273, 207, 156, 53, 74, 160, 26, 14, 46, 296, 26, 39, 74, 2979,
3554, 14, 46, 4689, 4329, 86, 61, 3499, 4795, 14, 61, 451, 4329, 17, 12]
```

Here's how you can decode it back to words, in case you're curious.

Listing 4.12 Decoding newswires back to text

```
word_index = reuters.get_word_index()
reverse_word_index = dict(
    [(value, key) for (key, value) in word_index.items()])
decoded_newswire = " ".join(
    [reverse_word_index.get(i - 3, "?") for i in     train_data[0]])
```

Note that the indices are offset by 3 because 0, 1, and 2 are reserved indices for "padding," "start of sequence," and "unknown."

The label associated with an example is an integer between 0 and 45—a topic index:

```
>>> train_labels[10]
3
```

4.2.2 Preparing the data

You can vectorize the data with the exact same code as in the previous example.

Listing 4.13 Encoding the input data

```
x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
```

Vectorized training data

Vectorized test data

To vectorize the labels, there are two possibilities: you can cast the label list as an integer tensor, or you can use *one-hot encoding*. One-hot encoding is a widely used format for categorical data, also called *categorical encoding*. In this case, one-hot encoding of the labels consists of embedding each label as an all-zero vector with a 1 in the place of the label index. The following listing shows an example.

Listing 4.14 Encoding the labels

```
def to_one_hot(labels, dimension=46):
    results = np.zeros((len(labels), dimension))
```

```
    for i, label in enumerate(labels):
        results[i, label] = 1.
    return results
y_train = to_one_hot(train_labels)      <-- Vectorized training labels
y_test = to_one_hot(test_labels)        <-- Vectorized test labels
```

Note that there is a built-in way to do this in Keras:

```
from tensorflow.keras.utils import to_categorical
y_train = to_categorical(train_labels)
y_test = to_categorical(test_labels)
```

4.2.3 Building your model

This topic-classification problem looks similar to the previous movie-review classification problem: in both cases, we're trying to classify short snippets of text. But there is a new constraint here: the number of output classes has gone from 2 to 46. The dimensionality of the output space is much larger.

In a stack of `Dense` layers like those we've been using, each layer can only access information present in the output of the previous layer. If one layer drops some information relevant to the classification problem, this information can never be recovered by later layers: each layer can potentially become an information bottleneck. In the previous example, we used 16-dimensional intermediate layers, but a 16-dimensional space may be too limited to learn to separate 46 different classes: such small layers may act as information bottlenecks, permanently dropping relevant information.

For this reason we'll use larger layers. Let's go with 64 units.

Listing 4.15 Model definition

```
model = keras.Sequential([
    layers.Dense(64, activation="relu"),
    layers.Dense(64, activation="relu"),
    layers.Dense(46, activation="softmax")
])
```

There are two other things you should note about this architecture.

First, we end the model with a `Dense` layer of size 46. This means for each input sample, the network will output a 46-dimensional vector. Each entry in this vector (each dimension) will encode a different output class.

Second, the last layer uses a `softmax` activation. You saw this pattern in the MNIST example. It means the model will output a *probability distribution* over the 46 different output classes—for every input sample, the model will produce a 46-dimensional output vector, where `output[i]` is the probability that the sample belongs to class `i`. The 46 scores will sum to 1.

The best loss function to use in this case is `categorical_crossentropy`. It measures the distance between two probability distributions: here, between the probability

distribution output by the model and the true distribution of the labels. By minimizing the distance between these two distributions, you train the model to output something as close as possible to the true labels.

Listing 4.16 Compiling the model

```
model.compile(optimizer="rmsprop",
              loss="categorical_crossentropy",
              metrics=["accuracy"])
```

4.2.4 Validating your approach

Let's set apart 1,000 samples in the training data to use as a validation set.

Listing 4.17 Setting aside a validation set

```
x_val = x_train[:1000]
partial_x_train = x_train[1000:]
y_val = y_train[:1000]
partial_y_train = y_train[1000:]
```

Now, let's train the model for 20 epochs.

Listing 4.18 Training the model

```
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

And finally, let's display its loss and accuracy curves (see figures 4.6 and 4.7).

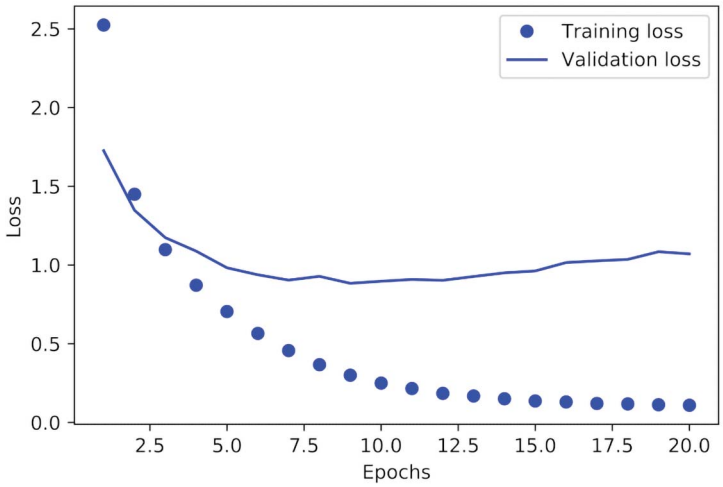


Figure 4.6 Training and validation loss

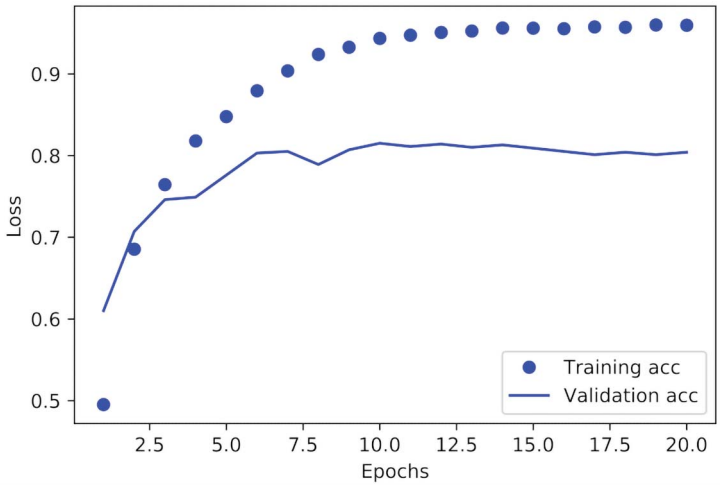


Figure 4.7 Training and validation accuracy

Listing 4.19 Plotting the training and validation loss

```
loss = history.history["loss"]
val_loss = history.history["val_loss"]
epochs = range(1, len(loss) + 1)
plt.plot(epochs, loss, "bo", label="Training loss")
plt.plot(epochs, val_loss, "b", label="Validation loss")
plt.title("Training and validation loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()
```

Listing 4.20 Plotting the training and validation accuracy

```
plt.clf()      <-- Clears the figure
acc = history.history["accuracy"]
val_acc = history.history["val_accuracy"]
plt.plot(epochs, acc, "bo", label="Training accuracy")
plt.plot(epochs, val_acc, "b", label="Validation accuracy")
plt.title("Training and validation accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```

The model begins to overfit after nine epochs. Let's train a new model from scratch for nine epochs and then evaluate it on the test set.

Listing 4.21 Retraining a model from scratch

```
model = keras.Sequential([
    layers.Dense(64, activation="relu"),
```

```
    layers.Dense(64, activation="relu"),
    layers.Dense(46, activation="softmax")
])
model.compile(optimizer="rmsprop",
              loss="categorical_crossentropy",
              metrics=["accuracy"])
model.fit(x_train,
          y_train,
          epochs=9,
          batch_size=512)
results = model.evaluate(x_test, y_test)
```

Here are the final results:

```
>>> results
[0.9565213431445807, 0.79697239536954589]
```

This approach reaches an accuracy of ~80%. With a balanced binary classification problem, the accuracy reached by a purely random classifier would be 50%. But in this case, we have 46 classes, and they may not be equally represented. What would be the accuracy of a random baseline? We could try quickly implementing one to check this empirically:

```
>>> import copy
>>> test_labels_copy = copy.copy(test_labels)
>>> np.random.shuffle(test_labels_copy)
>>> hits_array = np.array(test_labels) == np.array(test_labels_copy)
>>> hits_array.mean()
0.18655387355298308
```

As you can see, a random classifier would score around 19% classification accuracy, so the results of our model seem pretty good in that light.

4.2.5 Generating predictions on new data

Calling the model's `predict` method on new samples returns a class probability distribution over all 46 topics for each sample. Let's generate topic predictions for all of the test data:

```
predictions = model.predict(x_test)
```

Each entry in "predictions" is a vector of length 46:

```
>>> predictions[0].shape
(46,)
```

The coefficients in this vector sum to 1, as they form a probability distribution:

```
>>> np.sum(predictions[0])
1.0
```

The largest entry is the predicted class—the class with the highest probability:

```
>>> np.argmax(predictions[0])
4
```

4.2.6 A different way to handle the labels and the loss

We mentioned earlier that another way to encode the labels would be to cast them as an integer tensor, like this:

```
y_train = np.array(train_labels)
y_test = np.array(test_labels)
```

The only thing this approach would change is the choice of the loss function. The loss function used in listing 4.21, `categorical_crossentropy`, expects the labels to follow a categorical encoding. With integer labels, you should use `sparse_categorical_crossentropy`:

```
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
```

This new loss function is still mathematically the same as `categorical_crossentropy`; it just has a different interface.

4.2.7 The importance of having sufficiently large intermediate layers

We mentioned earlier that because the final outputs are 46-dimensional, you should avoid intermediate layers with many fewer than 46 units. Now let's see what happens when we introduce an information bottleneck by having intermediate layers that are significantly less than 46-dimensional: for example, 4-dimensional.

Listing 4.22 A model with an information bottleneck

```
model = keras.Sequential([
    layers.Dense(64, activation="relu"),
    layers.Dense(4, activation="relu"),
    layers.Dense(46, activation="softmax")
])
model.compile(optimizer="rmsprop",
              loss="categorical_crossentropy",
              metrics=["accuracy"])
model.fit(partial_x_train,
          partial_y_train,
          epochs=20,
          batch_size=128,
          validation_data=(x_val, y_val))
```

The model now peaks at ~71% validation accuracy, an 8% absolute drop. This drop is mostly due to the fact that we're trying to compress a lot of information (enough

information to recover the separation hyperplanes of 46 classes) into an intermediate space that is too low-dimensional. The model is able to cram *most* of the necessary information into these four-dimensional representations, but not all of it.

4.2.8 Further experiments

Like in the previous example, I encourage you to try out the following experiments to train your intuition about the kind of configuration decisions you have to make with such models:

- Try using larger or smaller layers: 32 units, 128 units, and so on.
- You used two intermediate layers before the final softmax classification layer. Now try using a single intermediate layer, or three intermediate layers.

4.2.9 Wrapping up

Here's what you should take away from this example:

- If you're trying to classify data points among *N* classes, your model should end with a `Dense` layer of size *N*.
- In a single-label, multiclass classification problem, your model should end with a `softmax` activation so that it will output a probability distribution over the *N* output classes.
- Categorical crossentropy is almost always the loss function you should use for such problems. It minimizes the distance between the probability distributions output by the model and the true distribution of the targets.
- There are two ways to handle labels in multiclass classification:
  - Encoding the labels via categorical encoding (also known as one-hot encoding) and using `categorical_crossentropy` as a loss function
  - Encoding the labels as integers and using the `sparse_categorical_crossentropy` loss function
- If you need to classify data into a large number of categories, you should avoid creating information bottlenecks in your model due to intermediate layers that are too small.

4.3 Predicting house prices: A regression example

The two previous examples were considered classification problems, where the goal was to predict a single discrete label of an input data point. Another common type of machine learning problem is *regression*, which consists of predicting a continuous value instead of a discrete label: for instance, predicting the temperature tomorrow, given meteorological data or predicting the time that a software project will take to complete, given its specifications.

NOTE Don't confuse *regression* and the *logistic regression* algorithm. Confusingly, logistic regression isn't a regression algorithm—it's a classification algorithm.

4.3.1 The Boston housing price dataset

In this section, we'll attempt to predict the median price of homes in a given Boston suburb in the mid-1970s, given data points about the suburb at the time, such as the crime rate, the local property tax rate, and so on. The dataset we'll use has an interesting difference from the two previous examples. It has relatively few data points: only 506, split between 404 training samples and 102 test samples. And each *feature* in the input data (for example, the crime rate) has a different scale. For instance, some values are proportions, which take values between 0 and 1, others take values between 1 and 12, others between 0 and 100, and so on.

Listing 4.23 Loading the Boston housing dataset

```
from tensorflow.keras.datasets import boston_housing
(train_data, train_targets), (test_data, test_targets) = (
    boston_housing.load_data())
```

Let's look at the data:

```
>>> train_data.shape
(404, 13)
>>> test_data.shape
(102, 13)
```

As you can see, we have 404 training samples and 102 test samples, each with 13 numerical features, such as per capita crime rate, average number of rooms per dwelling, accessibility to highways, and so on.

The targets are the median values of owner-occupied homes, in thousands of dollars:

```
>>> train_targets
[ 15.2,  42.3,  50. ...  19.4,  19.4,  29.1]
```

The prices are typically between $10,000 and $50,000. If that sounds cheap, remember that this was the mid-1970s, and these prices aren't adjusted for inflation.

4.3.2 Preparing the data

It would be problematic to feed into a neural network values that all take wildly different ranges. The model might be able to automatically adapt to such heterogeneous data, but it would definitely make learning more difficult. A widespread best practice for dealing with such data is to do feature-wise normalization: for each feature in the input data (a column in the input data matrix), we subtract the mean of the feature and divide by the standard deviation, so that the feature is centered around 0 and has a unit standard deviation. This is easily done in NumPy.

Listing 4.24 Normalizing the data

```
mean = train_data.mean(axis=0)
train_data -= mean
```

```
std = train_data.std(axis=0)
train_data /= std
test_data -= mean
test_data /= std
```

Note that the quantities used for normalizing the test data are computed using the training data. You should never use any quantity computed on the test data in your workflow, even for something as simple as data normalization.

4.3.3 Building your model

Because so few samples are available, we'll use a very small model with two intermediate layers, each with 64 units. In general, the less training data you have, the worse overfitting will be, and using a small model is one way to mitigate overfitting.

Listing 4.25 Model definition

```
def build_model():
    model = keras.Sequential([
        layers.Dense(64, activation="relu"),
        layers.Dense(64, activation="relu"),
        layers.Dense(1)
    ])
    model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
    return model
```

Because we need to instantiate the same model multiple times, we use a function to construct it.

The model ends with a single unit and no activation (it will be a linear layer). This is a typical setup for scalar regression (a regression where you're trying to predict a single continuous value). Applying an activation function would constrain the range the output can take; for instance, if you applied a `sigmoid` activation function to the last layer, the model could only learn to predict values between 0 and 1. Here, because the last layer is purely linear, the model is free to learn to predict values in any range.

Note that we compile the model with the `mse` loss function—*mean squared error*, the square of the difference between the predictions and the targets. This is a widely used loss function for regression problems.

We're also monitoring a new metric during training: *mean absolute error* (MAE). It's the absolute value of the difference between the predictions and the targets. For instance, an MAE of 0.5 on this problem would mean your predictions are off by $500 on average.

4.3.4 Validating your approach using K-fold validation

To evaluate our model while we keep adjusting its parameters (such as the number of epochs used for training), we could split the data into a training set and a validation set, as we did in the previous examples. But because we have so few data points, the validation set would end up being very small (for instance, about 100 examples). As a consequence, the validation scores might change a lot depending on which data points we chose for validation and which we chose for training: the validation scores

might have a high *variance* with regard to the validation split. This would prevent us from reliably evaluating our model.

The best practice in such situations is to use *K-fold* cross-validation (see figure 4.8).

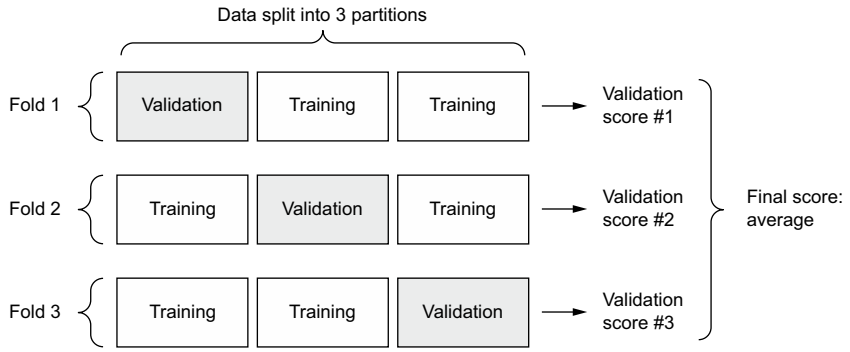


Figure 4.8 K-fold cross-validation with K=3

It consists of splitting the available data into K partitions (typically $K = 4$ or 5), instantiating K identical models, and training each one on $K - 1$ partitions while evaluating on the remaining partition. The validation score for the model used is then the average of the K validation scores obtained. In terms of code, this is straightforward.

Listing 4.26 K-fold validation

```
k = 4
num_val_samples = len(train_data) // k
num_epochs = 100
all_scores = []
for i in range(k):
    print(f"Processing fold #{i}")
    val_data = train_data[i * num_val_samples: (i + 1) * num_val_samples]
    val_targets = train_targets[i * num_val_samples: (i + 1) * num_val_samples]
    partial_train_data = np.concatenate(
        [train_data[:i * num_val_samples],
         train_data[(i + 1) * num_val_samples:]],
        axis=0)
    partial_train_targets = np.concatenate(
        [train_targets[:i * num_val_samples],
         train_targets[(i + 1) * num_val_samples:]],
        axis=0)
    model = build_model()
    model.fit(partial_train_data, partial_train_targets,
              epochs=num_epochs, batch_size=16, verbose=0)
    val_mse, val_mae = model.evaluate(val_data, val_targets, verbose=0)
    all_scores.append(val_mae)
```

Prepares the validation data: data from partition #k

Prepares the training data: data from all other partitions

Builds the Keras model (already compiled)

Trains the model (in silent mode, verbose = 0)

Evaluates the model on the validation data

Running this with `num_epochs = 100` yields the following results:

```
>>> all_scores
[2.112449, 3.0801501, 2.6483836, 2.4275346]
>>> np.mean(all_scores)
2.5671294
```

The different runs do indeed show rather different validation scores, from 2.1 to 3.1. The average (2.6) is a much more reliable metric than any single score—that's the entire point of K-fold cross-validation. In this case, we're off by $2,600 on average, which is significant considering that the prices range from $10,000 to $50,000.

Let's try training the model a bit longer: 500 epochs. To keep a record of how well the model does at each epoch, we'll modify the training loop to save the per-epoch validation score log for each fold.

Listing 4.27 Saving the validation logs at each fold

```
num_epochs = 500
all_mae_histories = []
for i in range(k):
    print(f"Processing fold #{i}")
    val_data = train_data[i * num_val_samples: (i + 1) * num_val_samples]
    val_targets = train_targets[i * num_val_samples: (i + 1) * num_val_samples]
    partial_train_data = np.concatenate(
        [train_data[:i * num_val_samples],
         train_data[(i + 1) * num_val_samples:]],
        axis=0)
    partial_train_targets = np.concatenate(
        [train_targets[:i * num_val_samples],
         train_targets[(i + 1) * num_val_samples:]],
        axis=0)
    model = build_model()
    history = model.fit(partial_train_data, partial_train_targets,
                        validation_data=(val_data, val_targets),
                        epochs=num_epochs, batch_size=16, verbose=0)
    mae_history = history.history["val_mae"]
    all_mae_histories.append(mae_history)
```

Prepares the validation data: data from partition #k

Prepares the training data: data from all other partitions

Builds the Keras model (already compiled)

Trains the model (in silent mode, verbose=0)

We can then compute the average of the per-epoch MAE scores for all folds.

Listing 4.28 Building the history of successive mean K-fold validation scores

```
average_mae_history = [
    np.mean([x[i] for x in all_mae_histories]) for i in range(num_epochs)]
```

Let's plot this; see figure 4.9.

Listing 4.29 Plotting validation scores

```
plt.plot(range(1, len(average_mae_history) + 1), average_mae_history)
plt.xlabel("Epochs")
plt.ylabel("Validation MAE")
plt.show()
```

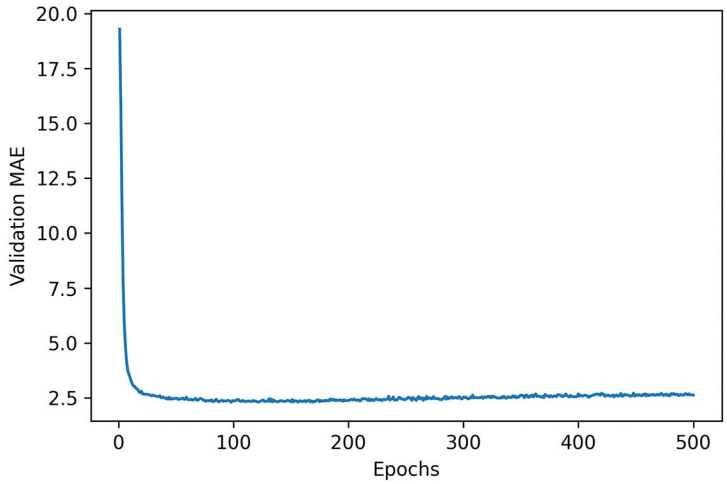


Figure 4.9 Validation MAE by epoch

It may be a little difficult to read the plot, due to a scaling issue: the validation MAE for the first few epochs is dramatically higher than the values that follow. Let's omit the first 10 data points, which are on a different scale than the rest of the curve.

Listing 4.30 Plotting validation scores, excluding the first 10 data points

```
truncated_mae_history = average_mae_history[10:]
plt.plot(range(1, len(truncated_mae_history) + 1), truncated_mae_history)
plt.xlabel("Epochs")
plt.ylabel("Validation MAE")
plt.show()
```

As you can see in figure 4.10, validation MAE stops improving significantly after 120–140 epochs (this number includes the 10 epochs we omitted). Past that point, we start overfitting.

Once you're finished tuning other parameters of the model (in addition to the number of epochs, you could also adjust the size of the intermediate layers), you can train a final production model on all of the training data, with the best parameters, and then look at its performance on the test data.

Listing 4.31 Training the final model

```
model = build_model()
model.fit(train_data, train_targets,
          epochs=130, batch_size=16, verbose=0)
test_mse_score, test_mae_score = model.evaluate(test_data, test_targets)
```

Gets a fresh, compiled model

Trains it on the entirety of the data

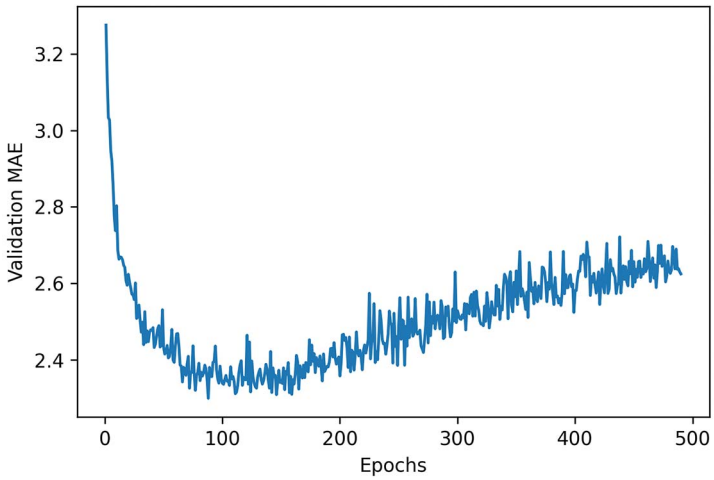


Figure 4.10 Validation MAE by epoch, excluding the first 10 data points

Here's the final result:

```
>>> test_mae_score
2.4642276763916016
```

We're still off by a bit under $2,500. It's an improvement! Just like with the two previous tasks, you can try varying the number of layers in the model, or the number of units per layer, to see if you can squeeze out a lower test error.

4.3.5 Generating predictions on new data

When calling `predict()` on our binary classification model, we retrieved a scalar score between 0 and 1 for each input sample. With our multiclass classification model, we retrieved a probability distribution over all classes for each sample. Now, with this scalar regression model, `predict()` returns the model's guess for the sample's price in thousands of dollars:

```
>>> predictions = model.predict(test_data)
>>> predictions[0]
array([9.990133], dtype=float32)
```

The first house in the test set is predicted to have a price of about $10,000.

4.3.6 Wrapping up

Here's what you should take away from this scalar regression example:

- Regression is done using different loss functions than we used for classification. Mean squared error (MSE) is a loss function commonly used for regression.

- Similarly, evaluation metrics to be used for regression differ from those used for classification; naturally, the concept of accuracy doesn't apply for regression. A common regression metric is mean absolute error (MAE).
- When features in the input data have values in different ranges, each feature should be scaled independently as a preprocessing step.
- When there is little data available, using K-fold validation is a great way to reliably evaluate a model.
- When little training data is available, it's preferable to use a small model with few intermediate layers (typically only one or two), in order to avoid severe overfitting.

Summary

- The three most common kinds of machine learning tasks on vector data are binary classification, multiclass classification, and scalar regression.
  - The "Wrapping up" sections earlier in the chapter summarize the important points you've learned regarding each task.
  - Regression uses different loss functions and different evaluation metrics than classification.
- You'll usually need to preprocess raw data before feeding it into a neural network.
- When your data has features with different ranges, scale each feature independently as part of preprocessing.
- As training progresses, neural networks eventually begin to overfit and obtain worse results on never-before-seen data.
- If you don't have much training data, use a small model with only one or two intermediate layers, to avoid severe overfitting.
- If your data is divided into many categories, you may cause information bottlenecks if you make the intermediate layers too small.
- When you're working with little data, K-fold validation can help reliably evaluate your model.

5 Fundamentals of machine learning

This chapter covers

- Understanding the tension between generalization and optimization, the fundamental issue in machine learning
- Evaluation methods for machine learning models
- Best practices to improve model fitting
- Best practices to achieve better generalization

After the three practical examples in chapter 4, you should be starting to feel familiar with how to approach classification and regression problems using neural networks, and you've witnessed the central problem of machine learning: overfitting. This chapter will formalize some of your new intuition about machine learning into a solid conceptual framework, highlighting the importance of accurate model evaluation and the balance between training and generalization.

5.1 Generalization: The goal of machine learning

In the three examples presented in chapter 4—predicting movie reviews, topic classification, and house-price regression—we split the data into a training set, a validation set, and a test set. The reason not to evaluate the models on the same data they

were trained on quickly became evident: after just a few epochs, performance on never-before-seen data started diverging from performance on the training data, which always improves as training progresses. The models started to *overfit*. Overfitting happens in every machine learning problem.

The fundamental issue in machine learning is the tension between optimization and generalization. *Optimization* refers to the process of adjusting a model to get the best performance possible on the training data (the *learning* in *machine learning*), whereas *generalization* refers to how well the trained model performs on data it has never seen before. The goal of the game is to get good generalization, of course, but you don't control generalization; you can only fit the model to its training data. If you do that *too well*, overfitting kicks in and generalization suffers.

But what causes overfitting? How can we achieve good generalization?

5.1.1 Underfitting and overfitting

For the models you saw in the previous chapter, performance on the held-out validation data started improving as training went on and then inevitably peaked after a while. This pattern (illustrated in figure 5.1) is universal. You'll see it with any model type and any dataset.

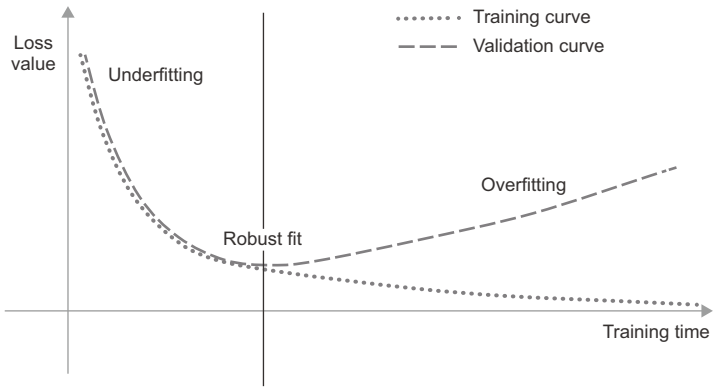


Figure 5.1 Canonical overfitting behavior

At the beginning of training, optimization and generalization are correlated: the lower the loss on training data, the lower the loss on test data. While this is happening, your model is said to be *underfit*: there is still progress to be made; the network hasn't yet modeled all relevant patterns in the training data. But after a certain number of iterations on the training data, generalization stops improving, validation metrics stall and then begin to degrade: the model is starting to overfit. That is, it's beginning to learn patterns that are specific to the training data but that are misleading or irrelevant when it comes to new data.

Overfitting is particularly likely to occur when your data is noisy, if it involves uncertainty, or if it includes rare features. Let's look at concrete examples.

Noisy training data

In real-world datasets, it's fairly common for some inputs to be invalid. Perhaps a MNIST digit could be an all-black image, for instance, or something like figure 5.2.

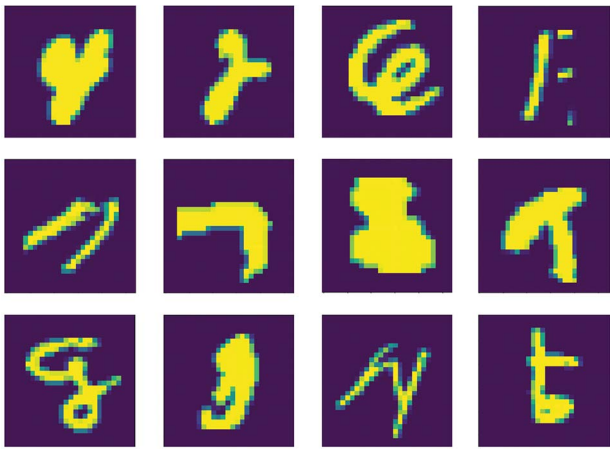


Figure 5.2 Some pretty weird MNIST training samples

What are these? I don't know either. But they're all part of the MNIST training set. What's even worse, however, is having perfectly valid inputs that end up mislabeled, like those in figure 5.3.

Figure 5.3 Mislabeled MNIST training samples

If a model goes out of its way to incorporate such outliers, its generalization performance will degrade, as shown in figure 5.4. For instance, a 4 that looks very close to the mislabeled 4 in figure 5.3 may end up getting classified as a 9.

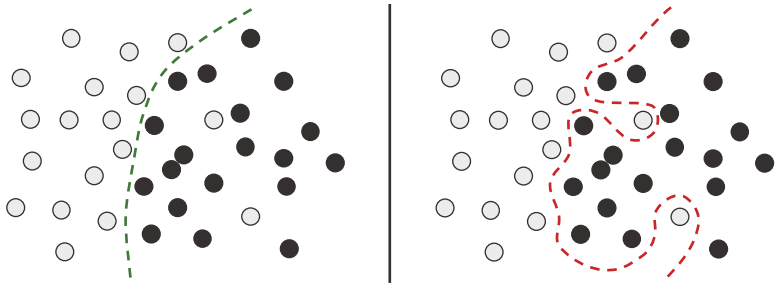


Figure 5.4 Dealing with outliers: robust fit vs. overfitting

AMBIGUOUS FEATURES

Not all data noise comes from inaccuracies—even perfectly clean and neatly labeled data can be noisy when the problem involves uncertainty and ambiguity. In classification tasks, it is often the case that some regions of the input feature space are associated with multiple classes at the same time. Let's say you're developing a model that takes an image of a banana and predicts whether the banana is unripe, ripe, or rotten. These categories have no objective boundaries, so the same picture might be classified as either unripe or ripe by different human labelers. Similarly, many problems involve randomness. You could use atmospheric pressure data to predict whether it will rain tomorrow, but the exact same measurements may be followed sometimes by rain and sometimes by a clear sky, with some probability.

A model could overfit to such probabilistic data by being too confident about ambiguous regions of the feature space, like in figure 5.5. A more robust fit would ignore individual data points and look at the bigger picture.

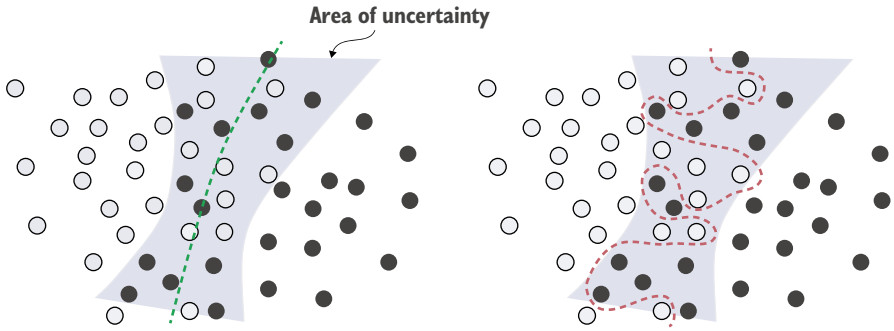


Figure 5.5 Robust fit vs. overfitting giving an ambiguous area of the feature space

Rare features and spurious correlations

If you've only ever seen two orange tabby cats in your life, and they both happened to be terribly antisocial, you might infer that orange tabby cats are generally likely to be antisocial. That's overfitting: if you had been exposed to a wider variety of cats, including more orange ones, you'd have learned that cat color is not well correlated with character.

Likewise, machine learning models trained on datasets that include rare feature values are highly susceptible to overfitting. In a sentiment classification task, if the word "cherimoya" (a fruit native to the Andes) only appears in one text in the training data, and this text happens to be negative in sentiment, a poorly regularized model might put a very high weight on this word and always classify new texts that mention cherimoyas as negative, whereas, objectively, there's nothing negative about the cherimoya.[1]

Importantly, a feature value doesn't need to occur only a couple of times to lead to spurious correlations. Consider a word that occurs in 100 samples in your training data and that's associated with a positive sentiment 54% of the time and with a negative sentiment 46% of the time. That difference may well be a complete statistical fluke, yet your model is likely to learn to leverage that feature for its classification task. This is one of the most common sources of overfitting.

Here's a striking example. Take MNIST. Create a new training set by concatenating 784 white noise dimensions to the existing 784 dimensions of the data, so half of the data is now noise. For comparison, also create an equivalent dataset by concatenating 784 all-zeros dimensions. Our concatenation of meaningless features does not at all affect the information content of the data: we're only adding something. Human classification accuracy wouldn't be affected by these transformations at all.

Listing 5.1 Adding white noise channels or all-zeros channels to MNIST

```
from tensorflow.keras.datasets import mnist
import numpy as np

(train_images, train_labels), _ = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255

train_images_with_noise_channels = np.concatenate(
    [train_images, np.random.random((len(train_images), 784))], axis=1)

train_images_with_zeros_channels = np.concatenate(
    [train_images, np.zeros((len(train_images), 784))], axis=1)
```

Now, let's train the model from chapter 2 on both of these training sets.

[1] Mark Twain even called it "the most delicious fruit known to men."

Listing 5.2 Training the same model on MNIST data with noise channels or all-zero channels

```
from tensorflow import keras
from tensorflow.keras import layers

def get_model():
    model = keras.Sequential([
        layers.Dense(512, activation="relu"),
        layers.Dense(10, activation="softmax")
    ])
    model.compile(optimizer="rmsprop",
                  loss="sparse_categorical_crossentropy",
                  metrics=["accuracy"])
    return model

model = get_model()
history_noise = model.fit(
    train_images_with_noise_channels, train_labels,
    epochs=10,
    batch_size=128,
    validation_split=0.2)

model = get_model()
history_zeros = model.fit(
    train_images_with_zeros_channels, train_labels,
    epochs=10,
    batch_size=128,
    validation_split=0.2)
```

Let's compare how the validation accuracy of each model evolves over time.

Listing 5.3 Plotting a validation accuracy comparison

```
import matplotlib.pyplot as plt
val_acc_noise = history_noise.history["val_accuracy"]
val_acc_zeros = history_zeros.history["val_accuracy"]
epochs = range(1, 11)
plt.plot(epochs, val_acc_noise, "b-",
         label="Validation accuracy with noise channels")
plt.plot(epochs, val_acc_zeros, "b--",
         label="Validation accuracy with zeros channels")
plt.title("Effect of noise channels on validation accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
```

Despite the data holding the same information in both cases, the validation accuracy of the model trained with noise channels ends up about one percentage point lower (see figure 5.6)—purely through the influence of spurious correlations. The more noise channels you add, the further accuracy will degrade.

Noisy features inevitably lead to overfitting. As such, in cases where you aren't sure whether the features you have are informative or distracting, it's common to do *feature*

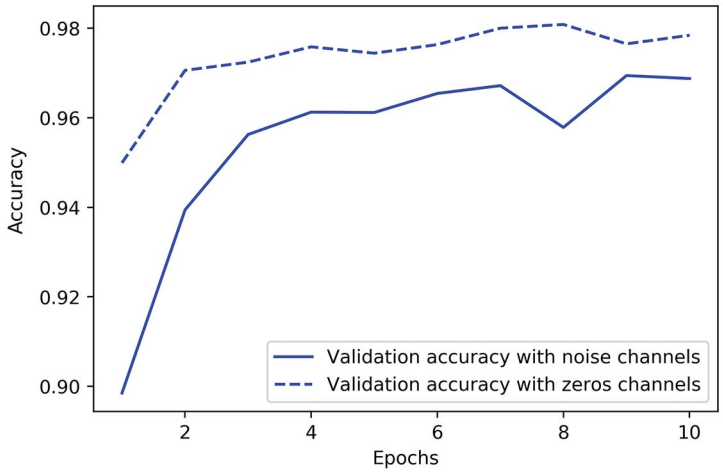


Figure 5.6 Effect of noise channels on validation accuracy

selection before training. Restricting the IMDB data to the top 10,000 most common words was a crude form of feature selection, for instance. The typical way to do feature selection is to compute some usefulness score for each feature available—a measure of how informative the feature is with respect to the task, such as the mutual information between the feature and the labels—and only keep features that are above some threshold. Doing this would filter out the white noise channels in the preceding example.

5.1.2 The nature of generalization in deep learning

A remarkable fact about deep learning models is that they can be trained to fit anything, as long as they have enough representational power.

Don't believe me? Try shuffling the MNIST labels and train a model on that. Even though there is no relationship whatsoever between the inputs and the shuffled labels, the training loss goes down just fine, even with a relatively small model. Naturally, the validation loss does not improve at all over time, since there is no possibility of generalization in this setting.

Listing 5.4 Fitting an MNIST model with randomly shuffled labels

```
(train_images, train_labels), _ = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255

random_train_labels = train_labels[:]
np.random.shuffle(random_train_labels)

model = keras.Sequential([
    layers.Dense(512, activation="relu"),
```

```
    layers.Dense(10, activation="softmax")
])
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
model.fit(train_images, random_train_labels,
          epochs=100,
          batch_size=128,
          validation_split=0.2)
```

In fact, you don't even need to do this with MNIST data—you could just generate white noise inputs and random labels. You could fit a model on that, too, as long as it has enough parameters. It would just end up memorizing specific inputs, much like a Python dictionary.

If this is the case, then how come deep learning models generalize at all? Shouldn't they just learn an ad hoc mapping between training inputs and targets, like a fancy `dict`? What expectation can we have that this mapping will work for new inputs?

As it turns out, the nature of generalization in deep learning has rather little to do with deep learning models themselves, and much to do with the structure of information in the real world. Let's take a look at what's really going on here.

THE MANIFOLD HYPOTHESIS

The input to an MNIST classifier (before preprocessing) is a 28 × 28 array of integers between 0 and 255. The total number of possible input values is thus 256 to the power of 784—much greater than the number of atoms in the universe. However, very few of these inputs would look like valid MNIST samples: actual handwritten digits only occupy a tiny *subspace* of the parent space of all possible 28 × 28 `uint8` arrays. What's more, this subspace isn't just a set of points sprinkled at random in the parent space: it is highly structured.

First, the subspace of valid handwritten digits is *continuous*: if you take a sample and modify it a little, it will still be recognizable as the same handwritten digit. Further, all samples in the valid subspace are *connected* by smooth paths that run through the subspace. This means that if you take two random MNIST digits A and B, there exists a sequence of "intermediate" images that morph A into B, such that two consecutive digits are very close to each other (see figure 5.7). Perhaps there will be a few ambiguous shapes close to the boundary between two classes, but even these shapes would still look very digit-like.

In technical terms, you would say that handwritten digits form a *manifold* within the space of possible 28 × 28 `uint8` arrays. That's a big word, but the concept is pretty intuitive. A "manifold" is a lower-dimensional subspace of some parent space that is locally similar to a linear (Euclidian) space. For instance, a smooth curve in the plane is a 1D manifold within a 2D space, because for every point of the curve, you can draw a tangent (the curve can be approximated by a line at every point). A smooth surface within a 3D space is a 2D manifold. And so on.

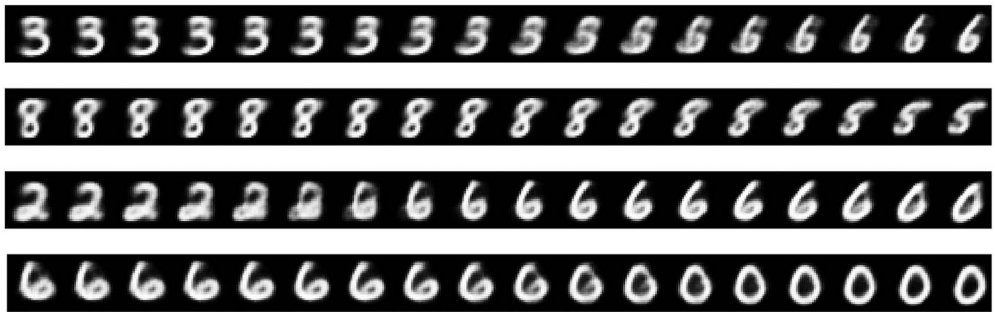


Figure 5.7 Different MNIST digits gradually morphing into one another, showing that the space of handwritten digits forms a "manifold." This image was generated using code from chapter 12.

More generally, the *manifold hypothesis* posits that all natural data lies on a low-dimensional manifold within the high-dimensional space where it is encoded. That's a pretty strong statement about the structure of information in the universe. As far as we know, it's accurate, and it's the reason why deep learning works. It's true for MNIST digits, but also for human faces, tree morphology, the sounds of the human voice, and even natural language.

The manifold hypothesis implies that

- Machine learning models only have to fit relatively simple, low-dimensional, highly structured subspaces within their potential input space (latent manifolds).
- Within one of these manifolds, it's always possible to *interpolate* between two inputs, that is to say, morph one into another via a continuous path along which all points fall on the manifold.

The ability to interpolate between samples is the key to understanding generalization in deep learning.

Interpolation as a source of generalization

If you work with data points that can be interpolated, you can start making sense of points you've never seen before by relating them to other points that lie close on the manifold. In other words, you can make sense of the *totality* of the space using only a *sample* of the space. You can use interpolation to fill in the blanks.

Note that interpolation on the latent manifold is different from linear interpolation in the parent space, as illustrated in figure 5.8. For instance, the average of pixels between two MNIST digits is usually not a valid digit.

Crucially, while deep learning achieves generalization via interpolation on a learned approximation of the data manifold, it would be a mistake to assume that interpolation is *all* there is to generalization. It's the tip of the iceberg. Interpolation can only help you make sense of things that are very close to what you've seen before:

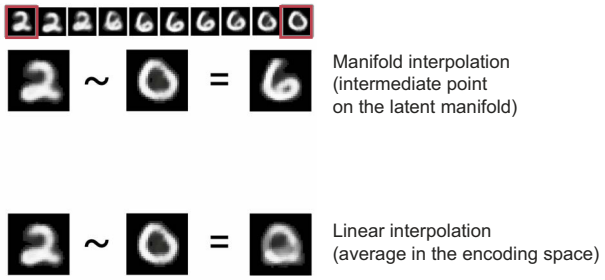


Figure 5.8 Difference between linear interpolation and interpolation on the latent manifold. Every point on the latent manifold of digits is a valid digit, but the average of two digits usually isn't.

it enables *local generalization*. But remarkably, humans deal with extreme novelty all the time, and they do just fine. You don't need to be trained in advance on countless examples of every situation you'll ever have to encounter. Every single one of your days is different from any day you've experienced before, and different from any day experienced by anyone since the dawn of humanity. You can switch between spending a week in NYC, a week in Shanghai, and a week in Bangalore without requiring thousands of lifetimes of learning and rehearsal for each city.

Humans are capable of *extreme generalization*, which is enabled by cognitive mechanisms other than interpolation: abstraction, symbolic models of the world, reasoning, logic, common sense, innate priors about the world—what we generally call *reason*, as opposed to intuition and pattern recognition. The latter are largely interpolative in nature, but the former isn't. Both are essential to intelligence. We'll talk more about this in chapter 14.

WHY DEEP LEARNING WORKS

Remember the crumpled paper ball metaphor from chapter 2? A sheet of paper represents a 2D manifold within 3D space (see figure 5.9). A deep learning model is a tool for uncrumpling paper balls, that is, for disentangling latent manifolds.

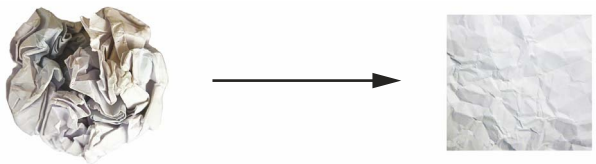


Figure 5.9 Uncrumpling a complicated manifold of data

A deep learning model is basically a very high-dimensional curve—a curve that is smooth and continuous (with additional constraints on its structure, originating from model architecture priors), since it needs to be differentiable. And that curve is fitted to data points via gradient descent, smoothly and incrementally. By its very nature, deep learning is about taking a big, complex curve—a manifold—and incrementally adjusting its parameters until it fits some training data points.

The curve involves enough parameters that it could fit anything—indeed, if you let your model train for long enough, it will effectively end up purely memorizing its training data and won't generalize at all. However, the data you're fitting to isn't made of isolated points sparsely distributed across the underlying space. Your data forms a highly structured, low-dimensional manifold within the input space—that's the manifold hypothesis. And because fitting your model curve to this data happens gradually and smoothly over time as gradient descent progresses, there will be an intermediate point during training at which the model roughly approximates the natural manifold of the data, as you can see in figure 5.10.

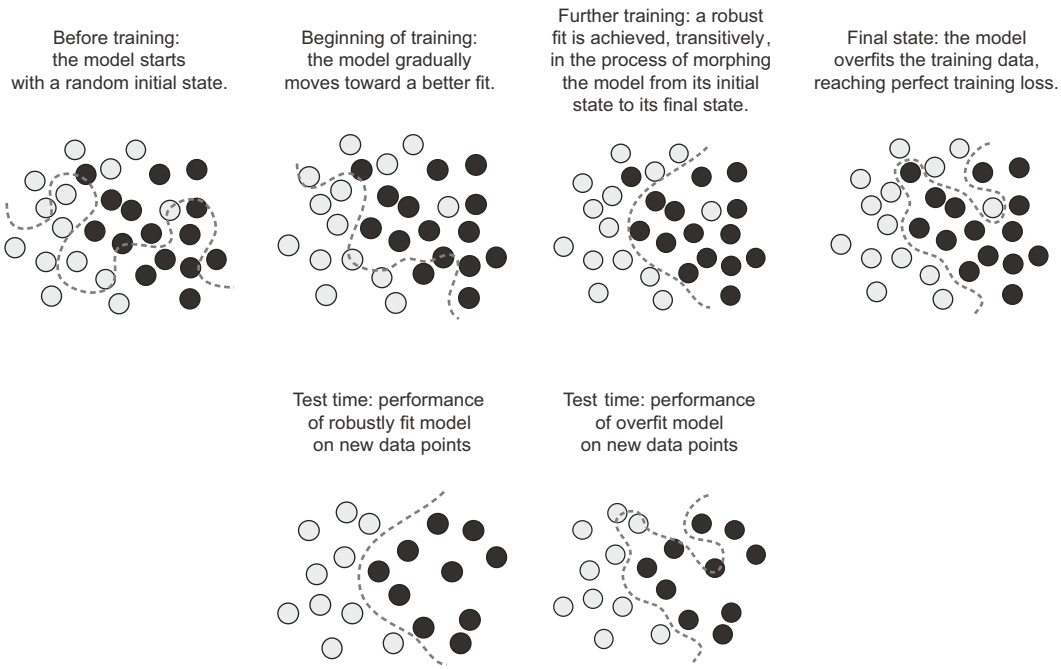


Figure 5.10 Going from a random model to an overfit model, and achieving a robust fit as an intermediate state

Moving along the curve learned by the model at that point will come close to moving along the actual latent manifold of the data—as such, the model will be capable of making sense of never-before-seen inputs via interpolation between training inputs.

Besides the trivial fact that they have sufficient representational power, there are a few properties of deep learning models that make them particularly well-suited to learning latent manifolds:

- Deep learning models implement a smooth, continuous mapping from their inputs to their outputs. It has to be smooth and continuous because it must be differentiable, by necessity (you couldn't do gradient descent otherwise).

This smoothness helps approximate latent manifolds, which follow the same properties.

- Deep learning models tend to be structured in a way that mirrors the "shape" of the information in their training data (via architecture priors). This is particularly the case for image-processing models (discussed in chapters 8 and 9) and sequence-processing models (chapter 10). More generally, deep neural networks structure their learned representations in a hierarchical and modular way, which echoes the way natural data is organized.

TRAINING DATA IS PARAMOUNT

While deep learning is indeed well suited to manifold learning, the power to generalize is more a consequence of the natural structure of your data than a consequence of any property of your model. You'll only be able to generalize if your data forms a manifold where points can be interpolated. The more informative and the less noisy your features are, the better you will be able to generalize, since your input space will be simpler and better structured. Data curation and feature engineering are essential to generalization.

Further, because deep learning is curve fitting, for a model to perform well *it needs to be trained on a dense sampling of its input space*. A "dense sampling" in this context means that the training data should densely cover the entirety of the input data manifold (see figure 5.11). This is especially true near decision boundaries. With a sufficiently dense sampling, it becomes possible to make sense of new inputs by interpolating between past training inputs without having to use common sense, abstract reasoning, or external knowledge about the world—all things that machine learning models have no access to.

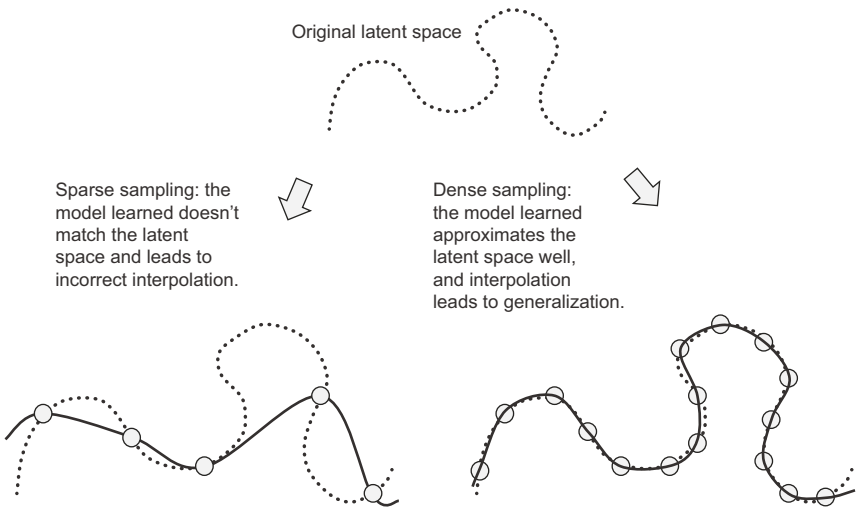


Figure 5.11 A dense sampling of the input space is necessary in order to learn a model capable of accurate generalization.

As such, you should always keep in mind that the best way to improve a deep learning model is to train it on more data or better data (of course, adding overly noisy or inaccurate data will harm generalization). A denser coverage of the input data manifold will yield a model that generalizes better. You should never expect a deep learning model to perform anything more than crude interpolation between its training samples, and thus you should do everything you can to make interpolation as easy as possible. The only thing you will find in a deep learning model is what you put into it: the priors encoded in its architecture and the data it was trained on.

When getting more data isn't possible, the next best solution is to modulate the quantity of information that your model is allowed to store, or to add constraints on the smoothness of the model curve. If a network can only afford to memorize a small number of patterns, or very regular patterns, the optimization process will force it to focus on the most prominent patterns, which have a better chance of generalizing well. The process of fighting overfitting this way is called *regularization*. We'll review regularization techniques in depth in section 5.4.4.

Before you can start tweaking your model to help it generalize better, you'll need a way to assess how your model is currently doing. In the following section, you'll learn how you can monitor generalization during model development: model evaluation.

5.2 Evaluating machine learning models

You can only control what you can observe. Since your goal is to develop models that can successfully generalize to new data, it's essential to be able to reliably measure the generalization power of your model. In this section, I'll formally introduce the different ways you can evaluate machine learning models. You've already seen most of them in action in the previous chapter.

5.2.1 Training, validation, and test sets

Evaluating a model always boils down to splitting the available data into three sets: training, validation, and test. You train on the training data and evaluate your model on the validation data. Once your model is ready for prime time, you test it one final time on the test data, which is meant to be as similar as possible to production data. Then you can deploy the model in production.

You may ask, why not have two sets: a training set and a test set? You'd train on the training data and evaluate on the test data. Much simpler!

The reason is that developing a model always involves tuning its configuration: for example, choosing the number of layers or the size of the layers (called the *hyperparameters* of the model, to distinguish them from the *parameters*, which are the network's weights). You do this tuning by using as a feedback signal the performance of the model on the validation data. In essence, this tuning is a form of *learning*: a search for a good configuration in some parameter space. As a result, tuning the configuration of the model based on its performance on the validation set can quickly result in *overfitting to the validation set*, even though your model is never directly trained on it.

Central to this phenomenon is the notion of *information leaks*. Every time you tune a hyperparameter of your model based on the model's performance on the validation set, some information about the validation data leaks into the model. If you do this only once, for one parameter, then very few bits of information will leak, and your validation set will remain reliable for evaluating the model. But if you repeat this many times—running one experiment, evaluating on the validation set, and modifying your model as a result—then you'll leak an increasingly significant amount of information about the validation set into the model.

At the end of the day, you'll end up with a model that performs artificially well on the validation data, because that's what you optimized it for. You care about performance on completely new data, not on the validation data, so you need to use a completely different, never-before-seen dataset to evaluate the model: the test dataset. Your model shouldn't have had access to *any* information about the test set, even indirectly. If anything about the model has been tuned based on test set performance, then your measure of generalization will be flawed.

Splitting your data into training, validation, and test sets may seem straightforward, but there are a few advanced ways to do it that can come in handy when little data is available. Let's review three classic evaluation recipes: simple holdout validation, K-fold validation, and iterated K-fold validation with shuffling. We'll also talk about the use of common-sense baselines to check that your training is going somewhere.

SIMPLE HOLDOUT VALIDATION

Set apart some fraction of your data as your test set. Train on the remaining data, and evaluate on the test set. As you saw in the previous sections, in order to prevent information leaks, you shouldn't tune your model based on the test set, and therefore you should *also* reserve a validation set.

Schematically, holdout validation looks like figure 5.12. Listing 5.5 shows a simple implementation.

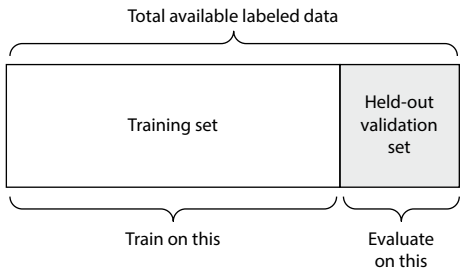


Figure 5.12 Simple holdout validation split

Listing 5.5 Holdout validation (note that labels are omitted for simplicity)

```
num_validation_samples = 10000
np.random.shuffle(data)
```

Shuffling the data is usually appropriate.

Defines the validation set

```
validation_data = data[:num_validation_samples]
training_data = data[num_validation_samples:]
model = get_model()
model.fit(training_data, ...)
validation_score = model.evaluate(validation_data, ...)
...
model = get_model()
model.fit(np.concatenate([training_data,
                          validation_data]), ...)
test_score = model.evaluate(test_data, ...)
```

Defines the training set

Trains a model on the training data, and evaluates it on the validation data

At this point you can tune your model, retrain it, evaluate it, tune it again.

Once you've tuned your hyperparameters, it's common to train your final model from scratch on all non-test data available.

This is the simplest evaluation protocol, and it suffers from one flaw: if little data is available, then your validation and test sets may contain too few samples to be statistically representative of the data at hand. This is easy to recognize: if different random shuffling rounds of the data before splitting end up yielding very different measures of model performance, then you're having this issue. K-fold validation and iterated K-fold validation are two ways to address this, as discussed next.

K-FOLD VALIDATION

With this approach, you split your data into `K` partitions of equal size. For each partition `i`, train a model on the remaining `K - 1` partitions, and evaluate it on partition `i`. Your final score is then the averages of the K scores obtained. This method is helpful when the performance of your model shows significant variance based on your train-test split. Like holdout validation, this method doesn't exempt you from using a distinct validation set for model calibration.

Schematically, K-fold cross-validation looks like figure 5.13. Listing 5.6 shows a simple implementation.

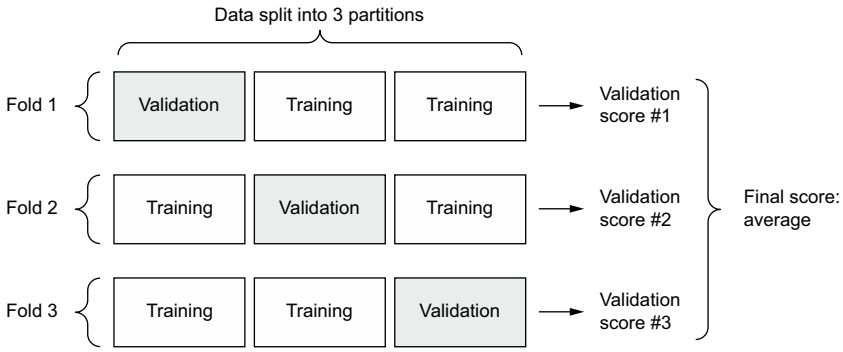


Figure 5.13 K-fold cross-validation with K=3

Listing 5.6 K-fold cross-validation (note that labels are omitted for simplicity)

```
k = 3
num_validation_samples = len(data) // k
np.random.shuffle(data)
validation_scores = []
for fold in range(k):
    validation_data = data[num_validation_samples * fold:
                           num_validation_samples * (fold + 1)]
    training_data = np.concatenate(
        data[:num_validation_samples * fold],
        data[num_validation_samples * (fold + 1):])
    model = get_model()
    model.fit(training_data, ...)
    validation_score = model.evaluate(validation_data, ...)
    validation_scores.append(validation_score)
validation_score = np.average(validation_scores)
model = get_model()
model.fit(data, ...)
test_score = model.evaluate(test_data, ...)
```

Selects the validation-data partition

Creates a brand-new instance of the model (untrained)

Validation score: average of the validation scores of the k folds

Trains the final model on all non-test data available

Uses the remainder of the data as training data. Note that the + operator represents list concatenation, not summation.

ITERATED K-FOLD VALIDATION WITH SHUFFLING

This one is for situations in which you have relatively little data available and you need to evaluate your model as precisely as possible. I've found it to be extremely helpful in Kaggle competitions. It consists of applying K-fold validation multiple times, shuffling the data every time before splitting it `K` ways. The final score is the average of the scores obtained at each run of K-fold validation. Note that you end up training and evaluating `P * K` models (where `P` is the number of iterations you use), which can be very expensive.

5.2.2 Beating a common-sense baseline

Besides the different evaluation protocols you have available, one last thing you should know about is the use of common-sense baselines.

Training a deep learning model is a bit like pressing a button that launches a rocket in a parallel world. You can't hear it or see it. You can't observe the manifold learning process—it's happening in a space with thousands of dimensions, and even if you projected it to 3D, you couldn't interpret it. The only feedback you have is your validation metrics—like an altitude meter on your invisible rocket.

It's particularly important to be able to tell whether you're getting off the ground at all. What was the altitude you started at? Your model seems to have an accuracy of 15%—is that any good? Before you start working with a dataset, you should always pick a trivial baseline that you'll try to beat. If you cross that threshold, you'll know you're doing something right: your model is actually using the information in the input data to make predictions that generalize, and you can keep going. This baseline could be

the performance of a random classifier, or the performance of the simplest non-machine learning technique you can imagine.

For instance, in the MNIST digit-classification example, a simple baseline would be a validation accuracy greater than 0.1 (random classifier); in the IMDB example, it would be a validation accuracy greater than 0.5. In the Reuters example, it would be around 0.18-0.19, due to class imbalance. If you have a binary classification problem where 90% of samples belong to class A and 10% belong to class B, then a classifier that always predicts A already achieves 0.9 in validation accuracy, and you'll need to do better than that.

Having a common-sense baseline you can refer to is essential when you're getting started on a problem no one has solved before. If you can't beat a trivial solution, your model is worthless—perhaps you're using the wrong model, or perhaps the problem you're tackling can't even be approached with machine learning in the first place. Time to go back to the drawing board.

5.2.3 Things to keep in mind about model evaluation

Keep an eye out for the following when you're choosing an evaluation protocol:

- *Data representativeness*—You want both your training set and test set to be representative of the data at hand. For instance, if you're trying to classify images of digits, and you're starting from an array of samples where the samples are ordered by their class, taking the first 80% of the array as your training set and the remaining 20% as your test set will result in your training set containing only classes 0–7, whereas your test set will contain only classes 8–9. This seems like a ridiculous mistake, but it's surprisingly common. For this reason, you usually should *randomly shuffle* your data before splitting it into training and test sets.
- *The arrow of time*—If you're trying to predict the future given the past (for example, tomorrow's weather, stock movements, and so on), you should not randomly shuffle your data before splitting it, because doing so will create a *temporal leak*: your model will effectively be trained on data from the future. In such situations, you should always make sure all data in your test set is *posterior* to the data in the training set.
- *Redundancy in your data*—If some data points in your data appear twice (fairly common with real-world data), then shuffling the data and splitting it into a training set and a validation set will result in redundancy between the training and validation sets. In effect, you'll be testing on part of your training data, which is the worst thing you can do! Make sure your training set and validation set are disjoint.

Having a reliable way to evaluate the performance of your model is how you'll be able to monitor the tension at the heart of machine learning—between optimization and generalization, underfitting and overfitting.

5.3 Improving model fit

To achieve the perfect fit, you must first overfit. Since you don't know in advance where the boundary lies, you must cross it to find it. Thus, your initial goal as you start working on a problem is to achieve a model that shows some generalization power and that is able to overfit. Once you have such a model, you'll focus on refining generalization by fighting overfitting.

There are three common problems you'll encounter at this stage:

- Training doesn't get started: your training loss doesn't go down over time.
- Training gets started just fine, but your model doesn't meaningfully generalize: you can't beat the common-sense baseline you set.
- Training and validation loss both go down over time, and you can beat your baseline, but you don't seem to be able to overfit, which indicates you're still underfitting.

Let's see how you can address these issues to achieve the first big milestone of a machine learning project: getting a model that has some generalization power (it can beat a trivial baseline) and that is able to overfit.

5.3.1 Tuning key gradient descent parameters

Sometimes training doesn't get started, or it stalls too early. Your loss is stuck. This is *always* something you can overcome: remember that you can fit a model to random data. Even if nothing about your problem makes sense, you should *still* be able to train something—if only by memorizing the training data.

When this happens, it's always a problem with the configuration of the gradient descent process: your choice of optimizer, the distribution of initial values in the weights of your model, your learning rate, or your batch size. All these parameters are interdependent, and as such it is usually sufficient to tune the learning rate and the batch size while keeping the rest of the parameters constant.

Let's look at a concrete example: let's train the MNIST model from chapter 2 with an inappropriately large learning rate of value 1.

Listing 5.7 Training an MNIST model with an incorrectly high learning rate

```
(train_images, train_labels), _ = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255

model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(10, activation="softmax")
])
model.compile(optimizer=keras.optimizers.RMSprop(1.),
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
model.fit(train_images, train_labels,
          epochs=10,
```

```
          batch_size=128,
          validation_split=0.2)
```

The model quickly reaches a training and validation accuracy in the 30%–40% range, but cannot get past that. Let's try to lower the learning rate to a more reasonable value of `1e-2`.

Listing 5.8 The same model with a more appropriate learning rate

```
model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(10, activation="softmax")
])
model.compile(optimizer=keras.optimizers.RMSprop(1e-2),
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
model.fit(train_images, train_labels,
          epochs=10,
          batch_size=128,
          validation_split=0.2)
```

The model is now able to train.

If you find yourself in a similar situation, try

- Lowering or increasing the learning rate. A learning rate that is too high may lead to updates that vastly overshoot a proper fit, like in the preceding example, and a learning rate that is too low may make training so slow that it appears to stall.
- Increasing the batch size. A batch with more samples will lead to gradients that are more informative and less noisy (lower variance).

You will, eventually, find a configuration that gets training started.

5.3.2 Leveraging better architecture priors

You have a model that fits, but for some reason your validation metrics aren't improving at all. They remain no better than what a random classifier would achieve: your model trains but doesn't generalize. What's going on?

This is perhaps the worst machine learning situation you can find yourself in. It indicates that *something is fundamentally wrong with your approach*, and it may not be easy to tell what. Here are some tips.

First, it may be that the input data you're using simply doesn't contain sufficient information to predict your targets: the problem as formulated is not solvable. This is what happened earlier when we tried to fit an MNIST model where the labels were shuffled: the model would train just fine, but validation accuracy would stay stuck at 10%, because it was plainly impossible to generalize with such a dataset.

It may also be that the kind of model you're using is not suited for the problem at hand. For instance, in chapter 10, you'll see an example of a timeseries prediction

problem where a densely connected architecture isn't able to beat a trivial baseline, whereas a more appropriate recurrent architecture does manage to generalize well. Using a model that makes the right assumptions about the problem is essential to achieve generalization: you should leverage the right architecture priors.

In the following chapters, you'll learn about the best architectures to use for a variety of data modalities—images, text, timeseries, and so on. In general, you should always make sure to read up on architecture best practices for the kind of task you're attacking—chances are you're not the first person to attempt it.

5.3.3 Increasing model capacity

If you manage to get to a model that fits, where validation metrics are going down, and that seems to achieve at least some level of generalization power, congratulations: you're almost there. Next, you need to get your model to start overfitting.

Consider the following small model—a simple logistic regression—trained on MNIST pixels.

Listing 5.9 A simple logistic regression on MNIST

```
model = keras.Sequential([layers.Dense(10, activation="softmax")])
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
history_small_model = model.fit(
    train_images, train_labels,
    epochs=20,
    batch_size=128,
    validation_split=0.2)
```

You get loss curves that look like figure 5.14:

```
import matplotlib.pyplot as plt
val_loss = history_small_model.history["val_loss"]
epochs = range(1, 21)
plt.plot(epochs, val_loss, "b--",
         label="Validation loss")
plt.title("Effect of insufficient model capacity on validation loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
```

Validation metrics seem to stall, or to improve very slowly, instead of peaking and reversing course. The validation loss goes to 0.26 and just stays there. You can fit, but you can't clearly overfit, even after many iterations over the training data. You're likely to encounter similar curves often in your career.

Remember that it should always be possible to overfit. Much like the problem where the training loss doesn't go down, this is an issue that can always be solved. If

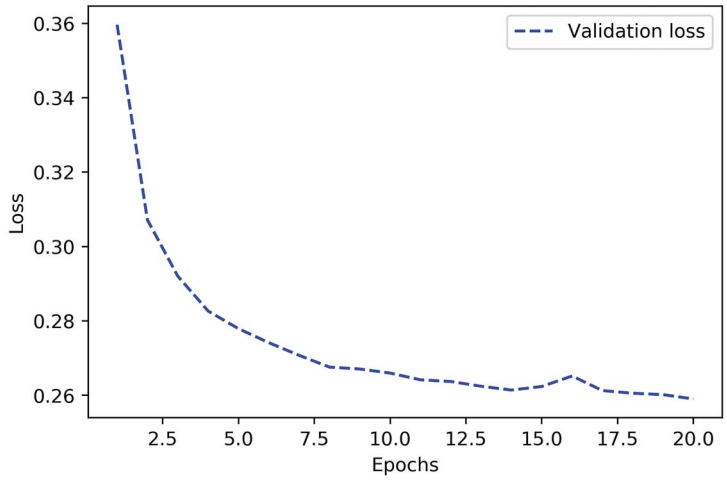


Figure 5.14 Effect of insufficient model capacity on loss curves

you can't seem to be able to overfit, it's likely a problem with the *representational power* of your model: you're going to need a bigger model, one with more *capacity*, that is to say, one able to store more information. You can increase representational power by adding more layers, using bigger layers (layers with more parameters), or using kinds of layers that are more appropriate for the problem at hand (better architecture priors).

Let's try training a bigger model, one with two intermediate layers with 96 units each:

```
model = keras.Sequential([
    layers.Dense(96, activation="relu"),
    layers.Dense(96, activation="relu"),
    layers.Dense(10, activation="softmax"),
])
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
history_large_model = model.fit(
    train_images, train_labels,
    epochs=20,
    batch_size=128,
    validation_split=0.2)
```

The validation curve now looks exactly like it should: the model fits fast and starts overfitting after 8 epochs (see figure 5.15).

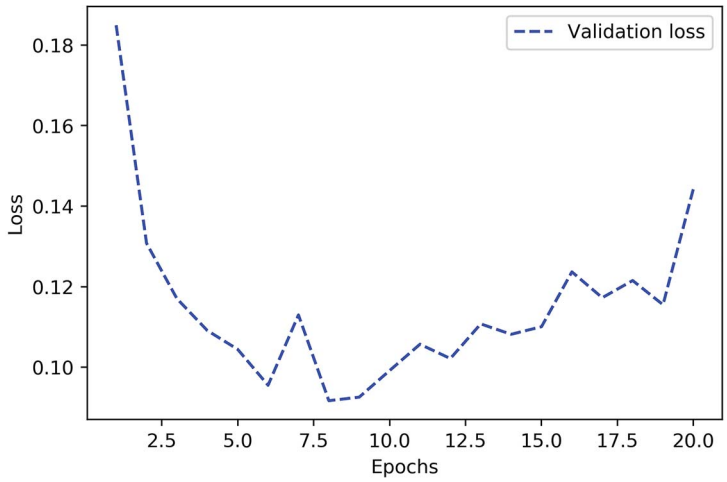


Figure 5.15 Validation loss for a model with appropriate capacity

5.4 Improving generalization

Once your model has shown itself to have some generalization power and to be able to overfit, it's time to switch your focus to maximizing generalization.

5.4.1 Dataset curation

You've already learned that generalization in deep learning originates from the latent structure of your data. If your data makes it possible to smoothly interpolate between samples, you will be able to train a deep learning model that generalizes. If your problem is overly noisy or fundamentally discrete, like, say, list sorting, deep learning will not help you. Deep learning is curve fitting, not magic.

As such, it is essential that you make sure that you're working with an appropriate dataset. Spending more effort and money on data collection almost always yields a much greater return on investment than spending the same on developing a better model.

- Make sure you have enough data. Remember that you need a *dense sampling* of the input-cross-output space. More data will yield a better model. Sometimes, problems that seem impossible at first become solvable with a larger dataset.
- Minimize labeling errors—visualize your inputs to check for anomalies, and proofread your labels.
- Clean your data and deal with missing values (we'll cover this in the next chapter).
- If you have many features and you aren't sure which ones are actually useful, do feature selection.

A particularly important way to improve the generalization potential of your data is *feature engineering*. For most machine learning problems, feature engineering is a key ingredient for success. Let's take a look.

5.4.2 Feature engineering

Feature engineering is the process of using your own knowledge about the data and about the machine learning algorithm at hand (in this case, a neural network) to make the algorithm work better by applying hardcoded (non-learned) transformations to the data before it goes into the model. In many cases, it isn't reasonable to expect a machine learning model to be able to learn from completely arbitrary data. The data needs to be presented to the model in a way that will make the model's job easier.

Let's look at an intuitive example. Suppose you're trying to develop a model that can take as input an image of a clock and can output the time of day (see figure 5.16).

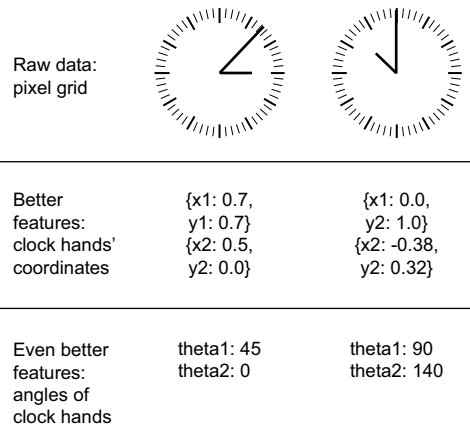


Figure 5.16 Feature engineering for reading the time on a clock

If you choose to use the raw pixels of the image as input data, you have a difficult machine learning problem on your hands. You'll need a convolutional neural network to solve it, and you'll have to expend quite a bit of computational resources to train the network.

But if you already understand the problem at a high level (you understand how humans read time on a clock face), you can come up with much better input features for a machine learning algorithm: for instance, it's easy to write a five-line Python script to follow the black pixels of the clock hands and output the `(x, y)` coordinates of the tip of each hand. Then a simple machine learning algorithm can learn to associate these coordinates with the appropriate time of day.

You can go even further: do a coordinate change, and express the `(x, y)` coordinates as polar coordinates with regard to the center of the image. Your input will become the angle `theta` of each clock hand. At this point, your features are making the problem so easy that no machine learning is required; a simple rounding operation and dictionary lookup are enough to recover the approximate time of day.

That's the essence of feature engineering: making a problem easier by expressing it in a simpler way. Make the latent manifold smoother, simpler, better organized. Doing so usually requires understanding the problem in depth.

Before deep learning, feature engineering used to be the most important part of the machine learning workflow, because classical shallow algorithms didn't have hypothesis spaces rich enough to learn useful features by themselves. The way you presented the data to the algorithm was absolutely critical to its success. For instance, before convolutional neural networks became successful on the MNIST digit-classification problem, solutions were typically based on hardcoded features such as the number of loops in a digit image, the height of each digit in an image, a histogram of pixel values, and so on.

Fortunately, modern deep learning removes the need for most feature engineering, because neural networks are capable of automatically extracting useful features from raw data. Does this mean you don't have to worry about feature engineering as long as you're using deep neural networks? No, for two reasons:

- Good features still allow you to solve problems more elegantly while using fewer resources. For instance, it would be ridiculous to solve the problem of reading a clock face using a convolutional neural network.
- Good features let you solve a problem with far less data. The ability of deep learning models to learn features on their own relies on having lots of training data available; if you have only a few samples, the information value in their features becomes critical.

5.4.3 Using early stopping

In deep learning, we always use models that are vastly overparameterized: they have way more degrees of freedom than the minimum necessary to fit to the latent manifold of the data. This overparameterization is not an issue, because *you never fully fit a deep learning model*. Such a fit wouldn't generalize at all. You will always interrupt training long before you've reached the minimum possible training loss.

Finding the exact point during training where you've reached the most generalizable fit—the exact boundary between an underfit curve and an overfit curve—is one of the most effective things you can do to improve generalization.

In the examples in the previous chapter, we would start by training our models for longer than needed to figure out the number of epochs that yielded the best validation metrics, and then we would retrain a new model for exactly that number of epochs. This is pretty standard, but it requires you to do redundant work, which can sometimes be expensive. Naturally, you could just save your model at the end of each epoch, and once you've found the best epoch, reuse the closest saved model you have. In Keras, it's typical to do this with an `EarlyStopping` callback, which will interrupt training as soon as validation metrics have stopped improving, while remembering the best known model state. You'll learn to use callbacks in chapter 7.

5.4.4 Regularizing your model

Regularization techniques are a set of best practices that actively impede the model's ability to fit perfectly to the training data, with the goal of making the model perform better during validation. This is called "regularizing" the model, because it tends to make the model simpler, more "regular," its curve smoother, more "generic"; thus it is less specific to the training set and better able to generalize by more closely approximating the latent manifold of the data.

Keep in mind that regularizing a model is a process that should always be guided by an accurate evaluation procedure. You will only achieve generalization if you can measure it.

Let's review some of the most common regularization techniques and apply them in practice to improve the movie-classification model from chapter 4.

Reducing the network's size

You've already learned that a model that is too small will not overfit. The simplest way to mitigate overfitting is to reduce the size of the model (the number of learnable parameters in the model, determined by the number of layers and the number of units per layer). If the model has limited memorization resources, it won't be able to simply memorize its training data; thus, in order to minimize its loss, it will have to resort to learning compressed representations that have predictive power regarding the targets—precisely the type of representations we're interested in. At the same time, keep in mind that you should use models that have enough parameters that they don't underfit: your model shouldn't be starved for memorization resources. There is a compromise to be found between *too much capacity* and *not enough capacity*.

Unfortunately, there is no magical formula to determine the right number of layers or the right size for each layer. You must evaluate an array of different architectures (on your validation set, not on your test set, of course) in order to find the correct model size for your data. The general workflow for finding an appropriate model size is to start with relatively few layers and parameters, and increase the size of the layers or add new layers until you see diminishing returns with regard to validation loss.

Let's try this on the movie-review classification model. The following listing shows our original model.

Listing 5.10 Original model

```
from tensorflow.keras.datasets import imdb
(train_data, train_labels), _ = imdb.load_data(num_words=10000)

def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results
train_data = vectorize_sequences(train_data)
```

```
model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    layers.Dense(16, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
history_original = model.fit(train_data, train_labels,
                             epochs=20, batch_size=512, validation_split=0.4)
```

Now let's try to replace it with this smaller model.

Listing 5.11 Version of the model with lower capacity

```
model = keras.Sequential([
    layers.Dense(4, activation="relu"),
    layers.Dense(4, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
history_smaller_model = model.fit(
    train_data, train_labels,
    epochs=20, batch_size=512, validation_split=0.4)
```

Figure 5.17 shows a comparison of the validation losses of the original model and the smaller model.

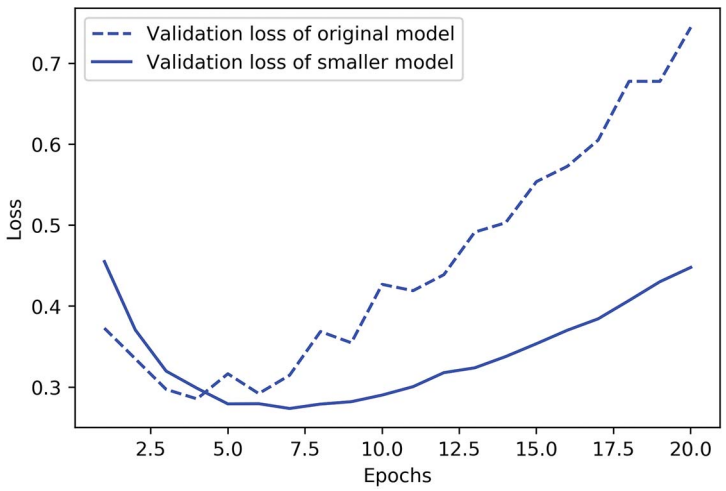


Figure 5.17 Original model vs. smaller model on IMDB review classification

As you can see, the smaller model starts overfitting later than the reference model (after six epochs rather than four), and its performance degrades more slowly once it starts overfitting.

Now, let's add to our benchmark a model that has much more capacity—far more than the problem warrants. While it is standard to work with models that are significantly overparameterized for what they're trying to learn, there can definitely be such a thing as *too much* memorization capacity. You'll know your model is too large if it starts overfitting right away and if its validation loss curve looks choppy with high-variance (although choppy validation metrics could also be a symptom of using an unreliable validation process, such as a validation split that's too small).

Listing 5.12 Version of the model with higher capacity

```
model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(512, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
history_larger_model = model.fit(
    train_data, train_labels,
    epochs=20, batch_size=512, validation_split=0.4)
```

Figure 5.18 shows how the bigger model fares compared with the reference model.

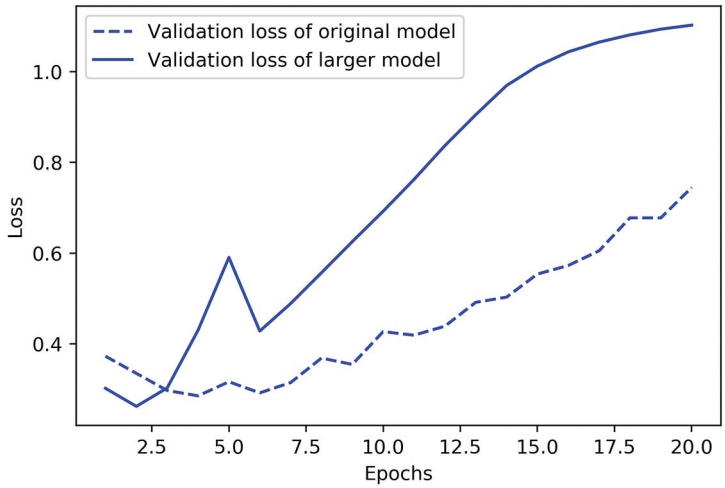


Figure 5.18 Original model vs. much larger model on IMDB review classification

The bigger model starts overfitting almost immediately, after just one epoch, and it overfits much more severely. Its validation loss is also noisier. It gets training loss near zero very quickly. The more capacity the model has, the more quickly it can model the training data (resulting in a low training loss), but the more susceptible it is to overfitting (resulting in a large difference between the training and validation loss).

ADDING WEIGHT REGULARIZATION

You may be familiar with the principle of *Occam's razor*: given two explanations for something, the explanation most likely to be correct is the simplest one—the one that makes fewer assumptions. This idea also applies to the models learned by neural networks: given some training data and a network architecture, multiple sets of weight values (multiple *models*) could explain the data. Simpler models are less likely to overfit than complex ones.

A *simple model* in this context is a model where the distribution of parameter values has less entropy (or a model with fewer parameters, as you saw in the previous section). Thus, a common way to mitigate overfitting is to put constraints on the complexity of a model by forcing its weights to take only small values, which makes the distribution of weight values more *regular*. This is called *weight regularization*, and it's done by adding to the loss function of the model a cost associated with having large weights. This cost comes in two flavors:

- *L1 regularization*—The cost added is proportional to the *absolute value of the weight coefficients* (the *L1 norm* of the weights).
- *L2 regularization*—The cost added is proportional to the *square of the value of the weight coefficients* (the *L2 norm* of the weights). L2 regularization is also called *weight decay* in the context of neural networks. Don't let the different name confuse you: weight decay is mathematically the same as L2 regularization.

In Keras, weight regularization is added by passing *weight regularizer instances* to layers as keyword arguments. Let's add L2 weight regularization to our initial movie-review classification model.

Listing 5.13 Adding L2 weight regularization to the model

```
from tensorflow.keras import regularizers
model = keras.Sequential([
    layers.Dense(16,
                 kernel_regularizer=regularizers.l2(0.002),
                 activation="relu"),
    layers.Dense(16,
                 kernel_regularizer=regularizers.l2(0.002),
                 activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
```

```
history_l2_reg = model.fit(
    train_data, train_labels,
    epochs=20, batch_size=512, validation_split=0.4)
```

In the preceding listing, `l2(0.002)` means every coefficient in the weight matrix of the layer will add `0.002 * weight_coefficient_value ** 2` to the total loss of the model. Note that because this penalty is *only added at training time*, the loss for this model will be much higher at training than at test time.

Figure 5.19 shows the impact of the L2 regularization penalty. As you can see, the model with L2 regularization has become much more resistant to overfitting than the reference model, even though both models have the same number of parameters.

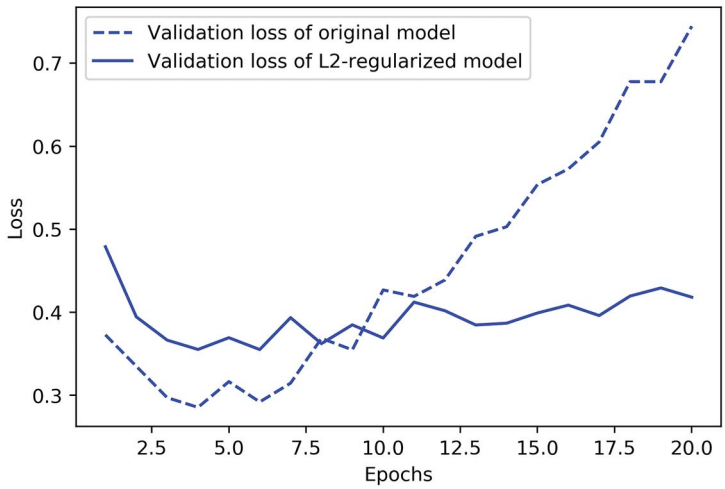


Figure 5.19 Effect of L2 weight regularization on validation loss

As an alternative to L2 regularization, you can use one of the following Keras weight regularizers.

Listing 5.14 Different weight regularizers available in Keras

```
from tensorflow.keras import regularizers
regularizers.l1(0.001)
regularizers.l1_l2(l1=0.001, l2=0.001)
```

L1 regularization

Simultaneous L1 and L2 regularization

Note that weight regularization is more typically used for smaller deep learning models. Large deep learning models tend to be so overparameterized that imposing constraints on weight values hasn't much impact on model capacity and generalization. In these cases, a different regularization technique is preferred: dropout.

ADDING DROPOUT

Dropout is one of the most effective and most commonly used regularization techniques for neural networks; it was developed by Geoff Hinton and his students at the University of Toronto. Dropout, applied to a layer, consists of randomly *dropping out* (setting to zero) a number of output features of the layer during training. Let's say a given layer would normally return a vector `[0.2, 0.5, 1.3, 0.8, 1.1]` for a given input sample during training. After applying dropout, this vector will have a few zero entries distributed at random: for example, `[0, 0.5, 1.3, 0, 1.1]`. The *dropout rate* is the fraction of the features that are zeroed out; it's usually set between 0.2 and 0.5. At test time, no units are dropped out; instead, the layer's output values are scaled down by a factor equal to the dropout rate, to balance for the fact that more units are active than at training time.

Consider a NumPy matrix containing the output of a layer, `layer_output`, of shape `(batch_size, features)`. At training time, we zero out at random a fraction of the values in the matrix:

```
layer_output *= np.random.randint(0, high=2, size=layer_output.shape)
```

At training time, drops out 50% of the units in the output

At test time, we scale down the output by the dropout rate. Here, we scale by 0.5 (because we previously dropped half the units):

```
layer_output *= 0.5
```

At test time

Note that this process can be implemented by doing both operations at training time and leaving the output unchanged at test time, which is often the way it's implemented in practice (see figure 5.20):

At training time

```
layer_output *= np.random.randint(0, high=2, size=layer_output.shape)
layer_output /= 0.5
```

Note that we're scaling up rather than scaling down in this case.

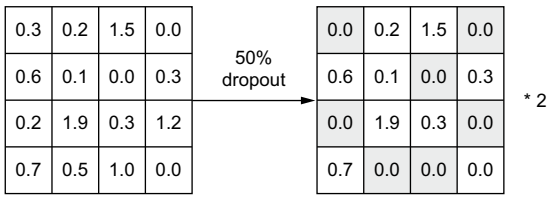


Figure 5.20 Dropout applied to an activation matrix at training time, with rescaling happening during training. At test time the activation matrix is unchanged.

This technique may seem strange and arbitrary. Why would this help reduce overfitting? Hinton says he was inspired by, among other things, a fraud-prevention mechanism used by banks. In his own words, "I went to my bank. The tellers kept changing and I asked one of them why. He said he didn't know but they got moved around a lot.

I figured it must be because it would require cooperation between employees to successfully defraud the bank. This made me realize that randomly removing a different subset of neurons on each example would prevent conspiracies and thus reduce overfitting." The core idea is that introducing noise in the output values of a layer can break up happenstance patterns that aren't significant (what Hinton refers to as *conspiracies*), which the model will start memorizing if no noise is present.

In Keras, you can introduce dropout in a model via the `Dropout` layer, which is applied to the output of the layer right before it. Let's add two `Dropout` layers in the IMDB model to see how well they do at reducing overfitting.

Listing 5.15 Adding dropout to the IMDB model

```
model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    layers.Dropout(0.5),
    layers.Dense(16, activation="relu"),
    layers.Dropout(0.5),
    layers.Dense(1, activation="sigmoid")
])
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
history_dropout = model.fit(
    train_data, train_labels,
    epochs=20, batch_size=512, validation_split=0.4)
```

Figure 5.21 shows a plot of the results. This is a clear improvement over the reference model—it also seems to be working much better than L2 regularization, since the lowest validation loss reached has improved.

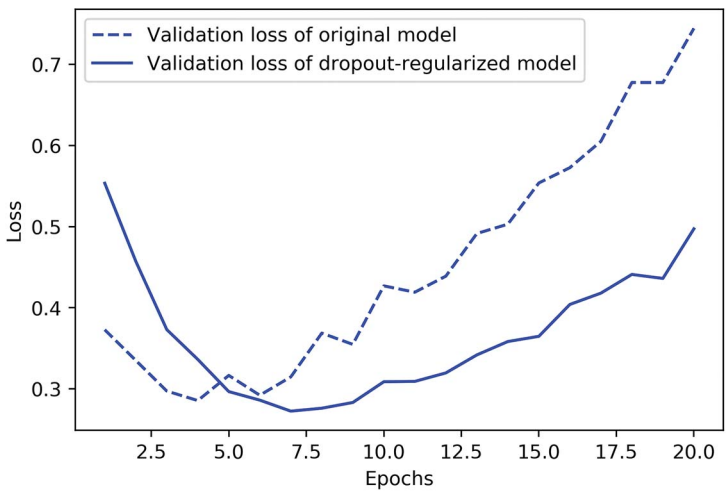


Figure 5.21 Effect of dropout on validation loss

To recap, these are the most common ways to maximize generalization and prevent overfitting in neural networks:

- Get more training data, or better training data.
- Develop better features.
- Reduce the capacity of the model.
- Add weight regularization (for smaller models).
- Add dropout.

Summary

- The purpose of a machine learning model is to *generalize*: to perform accurately on never-before-seen inputs. It's harder than it seems.
- A deep neural network achieves generalization by learning a parametric model that can successfully *interpolate* between training samples—such a model can be said to have learned the "latent manifold" of the training data. This is why deep learning models can only make sense of inputs that are very close to what they've seen during training.
- The fundamental problem in machine learning is *the tension between optimization and generalization*: to attain generalization, you must first achieve a good fit to the training data, but improving your model's fit to the training data will inevitably start hurting generalization after a while. Every single deep learning best practice deals with managing this tension.
- The ability of deep learning models to generalize comes from the fact that they manage to learn to approximate the *latent manifold* of their data, and can thus make sense of new inputs via interpolation.
- It's essential to be able to accurately evaluate the generalization power of your model while you're developing it. You have at your disposal an array of evaluation methods, from simple holdout validation to K-fold cross-validation and iterated K-fold cross-validation with shuffling. Remember to always keep a completely separate test set for final model evaluation, since information leaks from your validation data to your model may have occurred.
- When you start working on a model, your goal is first to achieve a model that has some generalization power and that can overfit. Best practices for doing this include tuning your learning rate and batch size, leveraging better architecture priors, increasing model capacity, or simply training longer.
- As your model starts overfitting, your goal switches to improving generalization through *model regularization*. You can reduce your model's capacity, add dropout or weight regularization, and use early stopping. And naturally, a larger or better dataset is always the number one way to help a model generalize.