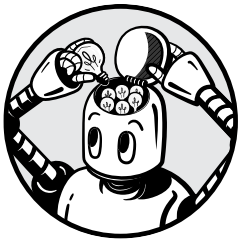
B1

SCIKIT-LEARN



This chapter is a bonus chapter for my book *Deep Learning: A Visual Approach*. You can order the book from No Starch Press at *https://nostarch.com/deep-learning-visual-approach/*. The official version of this chapter can be found for free on my GitHub at *https://github.com/blueberrymusic/* (look for the repository "Deep-Learning-A-Visual-Approach"). All of the figures in this chapter, and all of the notebooks with complete, running implementations of the code discussed here, can also be found for free on the book's GitHub repository.

Machine learning is a practical field. Although this book is focused on concepts, there's nothing quite like putting something into practice to transform ideas into understanding.

Implementing an idea makes us face decisions that we might have otherwise brushed off as easy or unimportant, and can reveal misunderstandings and holes in our knowledge. In a field like machine learning, where so much is still an art, and we make decisions based on intuition and

what we've learned from previous mistakes and successes, experience is invaluable.

In short, writing our own code is a challenging, instructive, and vastly illuminating activity. Unfortunately, it's also time-consuming, and there's a lot of work on infrastructure that isn't relevant to machine learning.

If you're of a mind to write your own code, I strongly encourage you to do it! It's well worth the effort. There are many guides online and offline to help you along that path [Müller-Guido16][Raschka15][VanderPlas16].

On the other hand, if you'd like to use existing library code, that's fine, too. Life is short, and we need to put our time and energies where we each feel they're best directed.

In this chapter, we'll take the approach of learning use an existing library. Once you're familiar with what different techniques do, you can always write your own implementations if you feel the itch.

Our library of choice is a popular Python-based toolkit for machine learning called *scikit-learn* (pronounced sy′-kit-lern). We're using scikit-learn because it's widely used, well supported and documented, stable, open-source, and fast.

We'll be using scikit-learn version 0.24, released in December 2020 [scikit-learn20]. Versions released after that are likely to be compatible with the code we'll be using. Downloading and installing the library is usually easy on most systems. See the download instructions on the library's home page for detailed instructions.

A pleasant way to use Python is to work interactively with a Python interpreter. The free Jupyter system provides a simple and flexible interface to interpreted Python that runs inside of any major browser [Jupyter17]. Jupyter notebooks containing running versions of all the programs in this chapter are available for free download on GitHub at *https://github.com/blueberrymusic*. Open the repo named *Deep-Learning-A-Visual-Approach*, then *Notebooks*, then *Bonus01-Scikit-Learn*.

Introduction

The Python library *scikit-learn* is a free library of common machine-learning algorithms.

To use it well requires some familiarity with the Python language and the NumPy library. NumPy (pronounced num′-pie) is a Python library focused on manipulating and calculating with numerical data. This nesting of dependencies is shown in Figure B1-1.

There are several other scientific "kits" in Python, each called "scikit-something." For example, *scikit-image* is a popular system for image processing. These kits are free and open-source projects produced and maintained by serious developers who largely pursue these efforts on their free time as a public service. Another widely-used library is named *SciPy* (pronounced sie′-pie), and it provides a wealth of routines for scientific and engineering work.

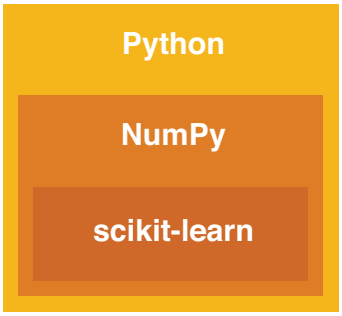


Figure B1-1: The scikit-learn library uses routines from the numerical calculation library NumPy, which in turn depends upon Python.

Getting deep into Python, NumPy, or scikit-learn, is not an afternoon project. Each of these entities is big, and has a lot to offer. On the other hand, a weekend of work on any of them can provide a solid start.

If you're new to any of these topics, the References section offers some resources for getting started. If you ever see a term or routine that seems unfamiliar, fountains of documentation on all of these topics are just an internet search away. In this chapter, our focus is on scikit-learn, so we'll assume that you are familiar enough with Python and NumPy to at least follow along.

Even though this chapter is focused on scikit-learn, we won't go deep into the library. Like any substantial piece of software, scikit-learn has its own conventions, defaults, assumptions, limitations, shortcuts, best practices, common pitfalls, easily-committed mistakes, and so on. And each routine within the library has its own specific details that are important to know in order to get the best results. To give these topics proper attention would require a book of its own, and indeed such books exist [Müller-Guido16].

Our goal in this chapter is to give an overview of the library and how it's used. For details on any routine or technique, consult the online documentation, reference books, or even some of the helpful web pages and blog posts online.

Python Conventions and Libraries

The code examples in this chapter (and the accompanying notebooks) are written for Python 3.7.6, but any version of Python 3 should work.

Scikit-learn is *big*. The library offers around 400 routines, which range from small utilities to large machine-learning algorithms bundled up into a single library call. These routines are extensively documented on the library's website. As of this writing, the current release (version 0.24) groups the library's routines into a whopping 35 categories.

Since we're taking a very general overview here, we'll invent our own smaller set of categories to organize our discussion: *Estimators*, *Clustering*, *Transformations*, *Data Refinement*, *Ensembles*, *Automation*, *Datasets*, and *Utilities*.

Many elements of scikit-learn are arranged like interlocking pieces, and some are designed to be used in particular combinations. We'll see some examples of those combinations later when we demonstrate specific algorithms.

Scikit-learn is not directed to neural networks, which are the hallmark of deep learning. Rather, these tools are intended for both standalone data science, and as utilities for neural networks, helping us explore, understand, and modify our data. Many deep learning projects begin with a thorough exploration of the data carried out with scikit-learn.

This chapter inevitably has code listings, but we have tried to keep these small and focused. Full code listings are great resources for learning the details of a system, and how to manage issues like structuring data and calling the right operations in the proper sequence. On the other hand, long blocks of code are boring to read. And real projects usually spend considerable time and effort on little details that are particular to that specific program, and don't aid our understanding of the general principles.

So while we will build real systems to do real work, we'll keep these project-specific steps to a minimum. We'll typically build up our projects one step at a time. So we'll show and discuss each step, but then consider it part of the program and not repeat that code every time as the program grows. In a few instances, we will gather up a bunch of code into one big listing, but usually we leave the complete summaries to the Jupyter notebooks that accompany the chapter, along with details like reading and writing data, and creating plots and other graphics.

To use scikit-learn, we must `import` it first, since Python doesn't load it automatically. When we import scikit-learn, it goes by the shorthand name `sklearn`. This name is just the top-level name for the library. The routines themselves are organized into a set of *modules*, which we also need to import.

There are two popular ways to import a module. The first is to import the whole module. Then we can name anything in that module in our code by prefixing it with the module's name, followed by a period. Listing B1-1 shows this approach, where we create an object named `Ridge`, which comes from the scikit-learn module named `linear_model`.

```
from sklearn import linear_model

ridge_estimator = linear_model.Ridge()
```

Listing B1-1: We can import the whole module `linear_model` and then use it as a prefix to name the `Ridge` object. Here we make a `Ridge` object with no arguments and save it in the variable `ridge_estimator`.

The other approach is to import only what we need from the module, and then we can use that object without referring to its module in the code, as in Listing B1-2.

```
from sklearn.linear_model import Ridge

ridge_estimator = Ridge()
```

Listing B1-2: We can import just the `Ridge` object from `linear_model`, and then we don't need to name the module when we use the object. The result of this code is identical to the result of Listing B1-1.

Generally speaking, it's usually easier during development to import the whole module, since then we can try out different objects from that module without constantly changing the `import` statement. When we're all done, we often go back and clean things up to import only what we're actually using, since that's considered a bit cleaner. In practice, both importing the whole module and importing just specific pieces are common, and often even mixed in the same file.

We will be using NumPy a lot, so we'll frequently import it as well. Conventionally, when we import NumPy we use the abbreviation `np`. This is just a widely used convention, and not a rule.

We also frequently include the `math` module. The convention is not to rename this library, so we refer to it in our code as `math`.

Another common library is the `matplotlib` graphics library which is the standard way to produce graphs and other plots. This too has a conventional name. The module we usually use from this library is called `pyplot`, and the tradition is to call this `plt`, or "plot" without the o. It's weird and arbitrary, but `plt` is used almost universally.

Finally, the Seaborn library offers some additional graphics routines, and also modifies the visuals produced by `matplotlib` to look more attractive [Waskom17]. When we import Seaborn, it is conventional to call it `sns` (this is a rather elaborate inside joke. According to the author of Seaborn, sns refers to the initials of the fictional Communications Director Samuel Norman Seaborn on the TV show *The West Wing* [Github14]). In this chapter our only use of Seaborn will be to import it, which will cause it to replace `matplotlib`'s default aesthetics with its own, causing our graphics to look much nicer. We do this by calling `sns.set()`, and we usually place that call on the same line as the `import` statement, separated by a semicolon.

Listing B1-3 shows typical `import` statements for these other libraries.

```
import sklearn
import numpy as np
import math
import matplotlib.pyplot as plt
import seaborn as sns ; sns.set()
```

Listing B1-3: This set of `import` statements will be our starting point in almost every project.

Seaborn lets us scale up the text in all of our graphs and plots at once. For instance, to scale up the font to about 14 point we could replace the `set()`call with `sns.set(font_scale=1.3)`. We'll leave the text size as-is in this chapter.

To find the right module to import for each scikit-learn routine we want to use, we can refer to the online API Reference. That reference also contains a complete breakdown of everything in scikit-learn, though often in very terse language. The API Reference is great when we treat it as just that, a reference when we want to remember what something's called or how to use it. But because of its brevity, it's less useful for explanations. That information is best found from one of the books or sites listed in the References section.

The people who manage scikit-learn have very high standards for including new routines, so additions are rare. The library is updated when there are important bug fixes and other improvements, and it's very occasionally re-organized. As a result, some routines become marked as "deprecated," which means that they are planned for removal, but are left in for a while so people can update their code. If you call a deprecated routine, you get a warning that often suggests how to update your code to be compliant with the new way of doing things. Deprecated routines are often later subsumed into more general library features, or simply re-organized into a different module.

Many of the routines in scikit-learn are made available to us through *objects*, in the object-oriented sense. For example, if we want to analyze some data in a particular way, we usually create an instance of an object designed for that kind of analysis. Then we get things done by calling that object's methods, instructing the object to do things like analyze a set of data, calculate statistics on the data, process it in some way, or predict values for new data.

Thanks to Python's built-in garbage collection, we don't have to worry about recycling or disposing of our objects when we're through with them. Python will automatically reclaim memory for us when it's safe to do so.

Using objects in this way works very well in practice, since it lets us keep our focus on what we want to do, and less on the mechanics.

A good way to learn a huge library like scikit-learn is to casually look up each routine or object the first time we see it used, often while typing in code from a reference, or studying someone else's program. A quick look at the options and defaults can help flesh out our sense of what the routine is able to do. Later, while working on another project, something may ring a bell, or otherwise draw us back to this routine or object.

Then we can look at the documentation again and read it more closely. In this way we can gradually learn a bit more about the breadth and depth of each feature as we find ourselves using it, which is much easier than trying to memorize everything about every object and routine the first time we encounter it.

Almost all objects and routines in `sklearn` accept multiple arguments. Many of these are optional and take on carefully-chosen default values if we don't refer to them. For simplicity, in this chapter we'll usually pass only the mandatory arguments, and leave all the optional arguments at their defaults.

Estimators

See the Jupyter notebook Bonus01-Scikit-Learn-1-Estimators.ipynb.

We use scikit-learn's *estimators* to carry out supervised learning. We create an estimator object, and then give it our labeled training data to learn from. We can later give our object a new sample it hasn't seen before, and it will do its best to predict the correct value.

All estimators have a core set of common routines and usage patterns.

As we mentioned earlier, each estimator is an individual *object* in the object-oriented programming sense. The object knows how to accept data, learn from it, and then describe new data it hasn't seen before, maintaining everything it needs to remember in its own instance variables. So the general process is to first *create* the object, and then send it *messages* (that is, call its built-in routines, or methods) causing it to take actions. Some of those actions only modify the internal state of the estimator object, while others compute results that are returned to us.

Figure B1-2 shows what we'll be aiming for in this section. We'll start with a bunch of points, and use a scikit-learn routine to find the best straight line through them.

Figure B1-2: straight line fit by a scikit-learn estimator

Creation

The first step is to *create*, or *instantiate*, the *estimator object*, often just called the *estimator*. As an example, let's instantiate an estimator called `Ridge` from the `linear_model` module. The `Ridge` object is a versatile estimator for doing this sort of regression task, and it includes built-in regularization. We just name the object (prefixing it with its module name), along with any arguments we want or need to provide. Since we're sticking with all the defaults, we won't pass any arguments. Listing B1-4 shows the code.

```
from sklearn import linear_model

ridge_estimator = linear_model.Ridge()
```

Listing B1-4: Creating a `Ridge` estimator object

After this assignment, the variable `ridge_estimator` refers to an object that implements the Ridge regression algorithm. When we use the simplest version of the estimator with 2D data, as we're doing here, this finds the best straight line that goes through the data.

All estimators have two routines named `fit()` and `predict()`. We use these respectively to teach our estimator, and to have it evaluate new data for us.

Learning with fit()

The first routine we'll look at is called `fit()`, and it is provided by every estimator. It takes two mandatory arguments, containing the samples that the estimator will learn from and the values (or targets) associated with them. The samples are usually arranged as a big numerical NumPy grid (called an *array*, even when there are many dimensions), where each row contains one sample.

Let's look at one example of generating data in the right format, so we can get a feeling for the process. We'll create the data shown on the left of Figure B1-2.

In Listing B1-5, we start by seeding NumPy's pseudo-random number generator, so we'll get back the same values every time. This lets us re-run the code and still generate the same data. Then we set the number of points we want to make in `num_pts`. We'll make our data based on a piece of a sine wave (a nice curve that's built into the `math` library), but we'll add a little noise to each value. Then we run a loop, appending point X and Y values to the arrays `x_vals` and `y_vals`. The array `x_vals` contains our samples, and `y_vals` contains the target value for each corresponding sample.

```
np.random.seed(42)
num_pts = 100
noise_range = 0.2
x_vals = []
y_vals = []
(x_left, x_right) = (-2, 2)
for i in range(num_pts):
    x = np.random.uniform(x_left, x_right)
    y = np.random.uniform(-noise_range, noise_range) + (2*math.sin(x))
    x_vals.append(x)
    y_vals.append(y)
```

Listing B1-5: Making data for training

The `x_vals` variable holds a list. But the `Ridge` estimator (like many others) wants to see its input data in the form of a 2D grid, where each entry is a list of features on its own row. Since we have only one feature, we can use NumPy's `reshape()` routine to turn `x_vals` into a column. Listing B1-6 shows this step.

```
x_column = np.reshape(x_vals, [len(x_vals), 1])
```

Listing B1-6: Reshaping our `x_vals` data into a column

Now that we have our data in the form expected by our `Ridge` object, we hand the samples over and ask it to find a straight line through the points. The first argument gives the samples and the second argument gives the labels associated with those samples. In this case, the first argument contains the X coordinate of each point, and the second contains the Y coordinate. We could turn the second argument into a column as well if we wanted, but `fit()` is happy to accept this argument as a 1D list, as shown in Listing B1-7.

```
ridge_estimator.fit(x_column, y_vals)
```

Listing B1-7: We train an estimator by calling its `fit()` method with the training data as an argument.

The `fit()` routine analyzes the input data and saves its results internally in the `Ridge` object. We can think of `fit()` as meaning, "analyze this data, and save the results so that you can answer future questions on this and related data."

Conceptually, we can think of `fit()` like a tailor who starts with a bolt of fabric, and then upon meeting and measuring a customer, proceeds to cut, sew, and *fit* a custom set of clothes for that person. In the same way, `fit()` calculates data that is customized to this particular input data, and saves that inside the estimator. If we call this object's `fit()` again with different set of training samples, it will start over and build a new set of internal data to fit that new input. This all happens inside of the object, so the `fit()` routine doesn't return anything new (for convenience, it does return a reference to the object it was called on. So in our case, it just returns the same `ridge_estimator` we just used to call `fit()` itself).

This means that we can make multiple estimator objects and keep them all around at the same time. We might train them all on the same data and then compare their results, or we might make many instances of the same estimator and train it with different data sets. The key thing is that each object is independent of the others, containing the parameters needed to answer questions about the data it was given.

Our program so far puts together everything from Listings B1-4 to B1-7.

Predicting with predict()

Once we've trained our estimator, we can ask it to evaluate new samples with `predict()`. This takes at least one argument, containing the new samples that we want the estimator to assign values to. The routine returns the information that describes the new data. Usually, this is a NumPy array that contains either one number or one class per sample.

Since our estimator has created a straight line to the data, we can get back that line by asking for the Y value for any two different values of X. Let's ask it for the Y values at the X values corresponding to the leftmost and rightmost values in our input data, which we can find by asking NumPy for the minimum and maximum values in the column array of X values. We'll give the estimator these minimum and maximum X values, and ask it

for the corresponding Y values. We have to turn the X values into 2D arrays because that's what `predict()` expects, so we just nest them both in two pairs of brackets, making a list that holds a list that holds our one item. The values that come back are the endpoints of our line. Listing B1-8 shows the calls.

```
x_left = np.min(x_column)
x_right = np.max(x_column)
y_left = ridge_estimator.predict([[x_left]])
y_right = ridge_estimator.predict([[x_right]]])
```

Listing B1-8: Getting the Y values of the straight line at two values of X

Let's put this all together. We'll import the stuff we need, make the data, make the estimator, fit the data, and get the left and right Y values of the line through the data. We'll just combine Listings B1-4 through B1-8. To keep things short, we'll replace the data-generating step in Listing B1-5 with a comment.

Listing B1-9 shows the code.

```
import numpy as np
import math
from sklearn.linear_model import Ridge

# Make the data. Save the samples in column form
# as x_column, and the target values as y_vals

# Make our estimator
ridge_estimator = Ridge()

# Call fit() to find the best straight-line fit to our data.
ridge_estimator.fit(x_column, y_vals)

# Find the Y values at the left and right ends of the data
x_left = np.min(x_column)
x_right = np.max(x_column)
y_left = ridge_estimator.predict([[x_left]])
y_right = ridge_estimator.predict([[x_right]])
```

Listing B1-9: Using the `Ridge` object to fit some 2D data

Figure B1-3 repeats Figure B1-2, showing the result of Listing B1-9, after we add a few lines of code to create the plots. We just draw all the data points, and then draw a red line from the point `(x_left, y_left)` to the point `(x_right, y_right)`.

We leave the plotting details to the code in the notebooks. Information about using `matplotlib` and `seaborn` can be found in each library's own documentation online, or in online or print references [VanderPlas16].

Now that we have the endpoints for this line, we know the equation of the line. We can find the estimated Y value for any X by just plugging that X value into the line's equation (or we could be lazy and ask our estimator to do that for us).

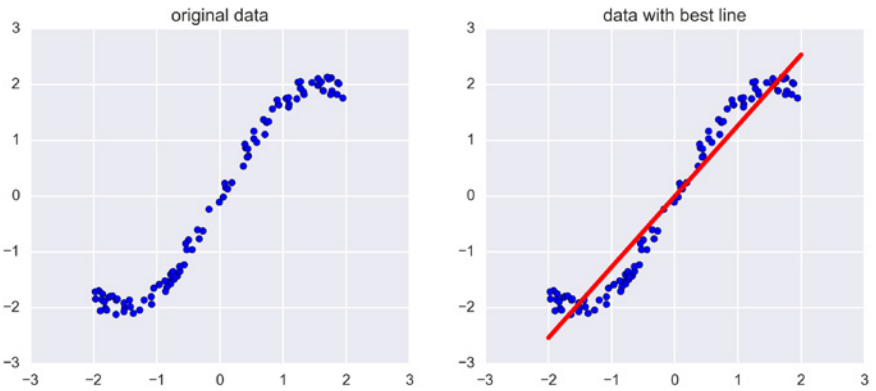


Figure B1-3: A straight line fit by `Ridge` to 2D data, using the code of Listing B1-9

Using decision_function() and predict_proba()

We just saw how to use an estimator to solve a regression problem.

The procedure is very similar for classification problems, which use classification estimators.

For classifiers, the `predict()` function gives us a single category as a result. But sometimes we want to know how "close" a sample came to being categorized into each of the other classes. For example, a classifier might decide that a given photo has a 49% chance of being a tiger, but a 48% chance of being a leopard (with the other 3% being other types of cats). The `predict()` function would tell us the photo is a tiger, but we might want to know that the leopard was a very close second.

For these situations where we care about the probabilities of our input with respect to all of the possible classes, many categorizers offer some additional options.

The routine `decision_function()` takes a set of samples as input, and returns a "confidence" score for every class and every input, where larger values represent more confidence. Note that it's possible for many categories to have large scores.

By contrast, the routine `predict_proba()` returns a *probability* for each class. Unlike the output of `decision_function()`, all of the probabilities for each sample always add up to 1. This often makes the output of `predict_proba()` easier to understand at a glance than that from `decision_function()`. It also means we can use the output of `predict_proba()` as a probability distribution function, which is sometimes convenient if we're doing additional analysis of the results.

Though all classifiers provide `fit()`, not all of them also offer either or both of these other routines. As always, the API documentation for each classifier tells us what it supports.

Clustering

See the Jupyter notebook Bonus01-Scikit-Learn-2-Clusters.ipynb.

Clustering is an unsupervised learning technique, where we provide the algorithm with a bunch of samples, and it does its best to group similar samples together.

Our input data will be a big collection of 2D points with no other information.

There are lots of clustering options in scikit-learn. Let's use the *k-means* algorithm. The value of *k* tells the algorithm how many clusters to build (though the routine calls that argument `n_clusters`, for "number of clusters," rather than the shorter and more cryptic, though traditional, single letter *k*).

We start by creating a `KMeans` object. To get access to that, we need to import it from its module, `sklearn.cluster`. When we make the object, we can tell it how many clusters we're going to want it to build once we give it data. This argument, named `n_clusters`, defaults to 8 if we don't give it a value of our own. Listing B1-10 shows these steps, including passing a value for the number of clusters.

```
from sklearn.cluster import KMeans

num_clusters = 5
kMeans = KMeans(n_clusters=num_clusters)
```

Listing B1-10: Importing `KMeans` and then creating an instance

Clustering algorithms are often used with data that has many dimensions (or features), perhaps dozens or hundreds. But when we create our clustering object we don't have to tell it how many features we'll be using, because it will extract that information from the data itself when we provide that later via a call to `fit()`.

As usual, let's use 2 dimensions (that is, 2 features per sample) so we can draw pictures of our data and the results. To create our dataset, we'll make 7 random Gaussian blobs, and pick points at random from each one. Figure B1-4 shows the result. We also show the data color-coded by the blob that generated each sample, but that's strictly for our human eyes. The computer only sees a list of X and Y values.

Remember, we're just giving it the black dots from Figure B1-4, so the algorithm knows nothing about these points except for their location. As before, we'll shape our data as a NumPy array that is a big column, where each row has two dimensions, or features: the X and Y values of that row's point. We'll call that data `XY_points`. To build the clusters from this data, we only have to hand it to our object's `fit()` routine. Listing B1-11 shows the code.

```
kMeans.fit(XY_points)
```

Listing B1-11: We give our data to `kMeans` and it will work out the number of clusters we requested when we made the object.

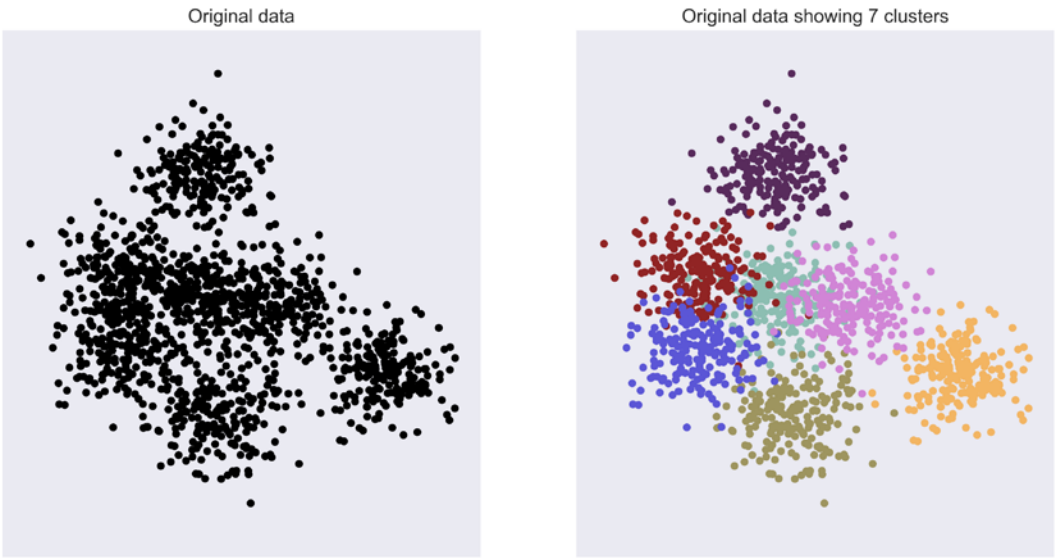


Figure B1-4: The data we'll use for clustering. Left: The data the algorithm will get. Right: A cheat sheet that shows us which Gaussian blob was used to generate each sample.

As soon as `fit()` returns, the clustering is done. To visualize how it broke up our starting data, we'll run through the same data again, and ask for the predicted cluster for each sample. Conveniently, we get this information using the object's `predict()` method, as shown in Listing B1-12.

```
predictions = kMeans.predict(XY_points)
```

Listing B1-12: We can find the predicted cluster for our original data just by handing it to `predict()`.

The variable `predictions` is a NumPy array that's shaped as a column. That is, it has one row for every point in the input, and each row has one value: an integer telling us which cluster the routine has assigned to the corresponding point in our `XY_points` input. For example, the fifth row of `XY_points` holds the X and Y values of a point, and the fifth row of `predictions` holds an integer telling us which cluster that point was assigned to.

Let's run this process for a bunch of different numbers of clusters. Leaving out the plotting code, it looks like Listing B1-13.

```
for num_clusters in range(2, 10):
    kMeans = KMeans(n_clusters=num_clusters)
    kMeans.fit(XY_points)
    predictions = kMeans.predict(XY_points)
    # plot the XY_points and predictions
```

Listing B1-13: Trying out a range of different numbers of clusters

The results are shown in Figure B1-5.

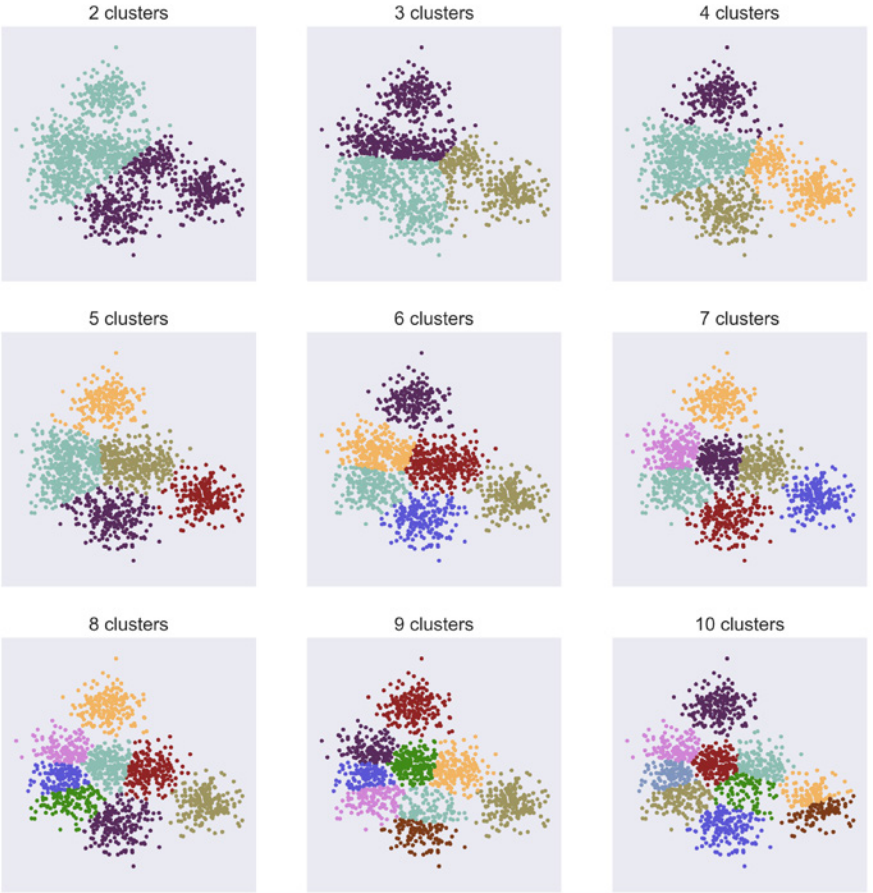


Figure B1-5: The KMeans algorithm clustering our starting data from Figure B1-4 into different numbers of clusters. Each cluster is shown in a different color.

Visually, things start to look reasonable starting at about 5 clusters. Because we used overlapping Gaussians to create our dataset, the groups are not all cleanly distinguished. But starting around 5 clusters we can see that the topmost and rightmost chunks get identified well, with the main mass getting subdivided into smaller regions. Starting at around 9 clusters those outer chunks seem be getting split up themselves. So somewhere between 5 and 8 seems like a good choice, which is a satisfying result knowing that we started with 7 overlapping Gaussians.

Scikit-learn offers multiple clustering routines because they approach the problem in different ways, and produce different kinds of clusters.

Clustering algorithms are a great way to get a feeling for the spatial distribution of our data. A downside of clustering is that we have to pick the number of clusters. There are algorithms that try to pick the best number of clusters for us [Kassambara17], but often we still have to look at the results and decide if they make sense.

Transformations

See the Jupyter notebook Bonus01-Scikit-Learn-3-Transformations.ipynb.

Much of the time we'll want to *transform*, or modify, our data before we use it. Often this is because we want to make sure that the data fits the expectations of the algorithms we'll give it to. For instance, many techniques work best if they receive data in which every feature is centered at 0, and scaled to the range [–1, 1].

Scikit-learn offers a wide range of objects, called *transformers*, that carry out many different kinds of data transformations. Each routine accepts a NumPy array holding sample data, and returns a transformed array. Note that this name unfortunately conflicts with the name of the popular *transformer* architecture, which we discussed in Chapter 20. The name as used here is actually a bit more appropriate, as these routines carry out operations that are traditionally referred to as transformations. The transformer models in Chapter 20 are an entirely different kind of thing, and we won't be referring to them again in this chapter.

We usually choose which transformer to apply based on the estimator we will ultimately give our data to. Each estimator that wants its data in a specific form gives that information in its documentation, but it's up to us to explicitly carry out any transformations they suggest or require.

We learn our transformation from the training data, but it's essential that we then apply the same transformation to all further data. To help us carry this out, each transformer offers distinct routines for creating the transformation object, analyzing data to find the parameters for its transformation, and applying that transformation.

We create the object in the same way that we created other objects above, by calling its creation routine with any arguments we want to pass.

To find the parameters for a specific transformation, we call the `fit()` method, just as we do for an estimator. We give `fit()` our data, and the object analyzes it to determine the parameters for the transformation performed by that object. Then we actually transform data with the `transform()` method, which takes a set of data, and returns the transformed version. Then any time we have data for the trained model, whether it's the original training data, or validation data, or even new data arriving after deployment, we'll first run it through this object's `transform()` method.

Notice how nicely this encapsulates the necessity of retaining the transformation. Our object learns what it needs to do just one time, up front, when we call `fit()`, and then it will apply that same transformation to any data we hand it from then on.

Figure B1-6 illustrates the idea.

After creating a scaling transformer, we give it the training data by calling `fit()`. The scaler analyzes the data and determines the scaling transformation. Then we transform the training data with `transform()` and train our estimator with it. From then on, all new data we want to use also goes through the scaling object's `transform()` routine.

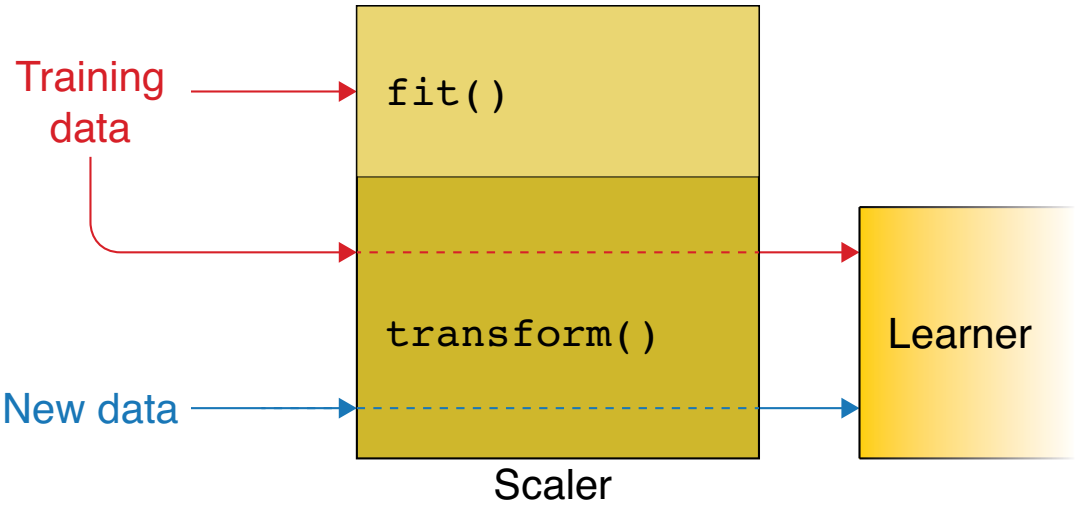


Figure B1-6: Using a scaling transformer to modify our data before training

Let's see this in action. We'll use a transformer with a very obvious visual effect: each feature is scaled to the range [0, 1]. As usual, we'll use 2D data, so there are two features to be scaled: X and y.

We'll use a scikit-learn transformer called the `MinMaxScaler`, which was designed for just this kind of task. We can give it data with any number of features, and by default each feature will be independently transformed to the range [0,1].

Let's start with the data in Figure B1-7. The X values are in the range of about [–1.5, 2.5] and the Y values are in the range of about [–1, 1.5].

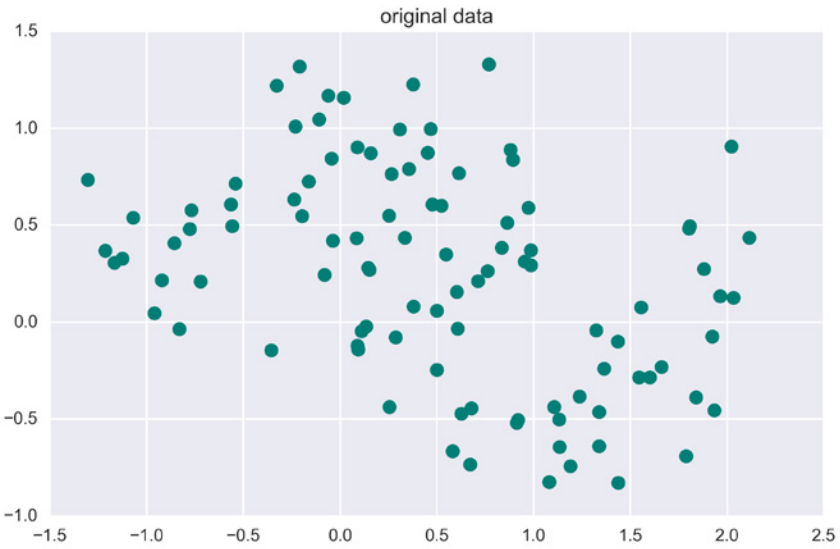


Figure B1-7: The starting two-dimensional (or two-feature) data for scaling.

To use the scaler, as usual we have to first import the proper module from `sklearn`. In this case, the documentation tells us that it's `sklearn.preprocessing`. Our first step is to create the scaler, and save it in a variable, in Listing B1-14.

```
from sklearn.preprocessing import MinMaxScaler

mm_scaler = MinMaxScaler()
```

Listing B1-14: Creating a `MinMaxScaler` object to scale all of our features at once.

Now that we have our object, let's have it analyze our training data and work out the transformation by calling `fit()` with our training data. As before, we'll arrange our data in tabular form, where each row contains all the features for one sample. So our data will have two entries per row (one each for the point's X and Y values). Listing B1-15 shows the call.

```
mm_scaler.fit(training_samples)
```

Listing B1-15: When we call `fit()`, our `MinMaxScaler` will work out the transformation to scale each feature to [0,1].

Now we're ready to transform our data. We just call `transform()` on our set of samples, and save the transformed values. We can then hand these to our estimator for training in Listing B1-16.

```
transformed_training_samples = mm_scaler.transform(training_samples)
```

Listing B1-16: We call `transform()` on any set of data to scale it using the transformation determined when we called `fit()`.

The result of transforming our training data is shown in Figure B1-8.

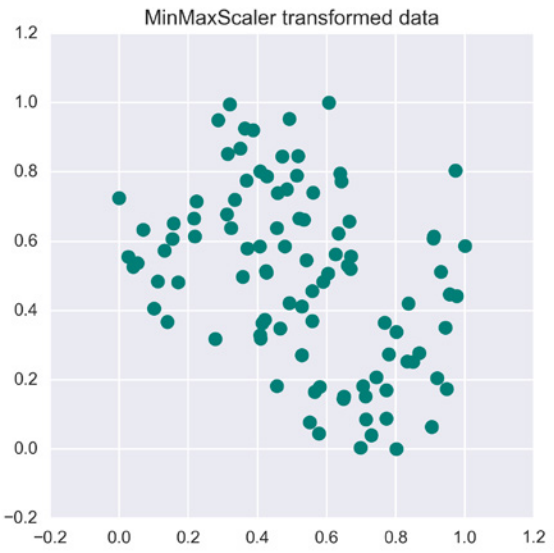


Figure B1-8: Our data from Figure B1-7 transformed by our `MinMaxScaler` so that each feature is independently scaled to [0,1].

We can see that our data now spans the range [0,1] in both X and Y.

Let's suppose that we now we want to evaluate the quality of our estimator using some test, or validation, data. We know that before we give this to our estimator, we have to transform it first in the same way that we transformed the training data. We'll do that in Listing B1-17.

```
transformed_test_samples = mm_scaler.transform(test_samples)
```

Listing B1-17: We call `transform()` on our new data to transform it before using it.

Figure B1-9 shows some test data, and its transformed result.

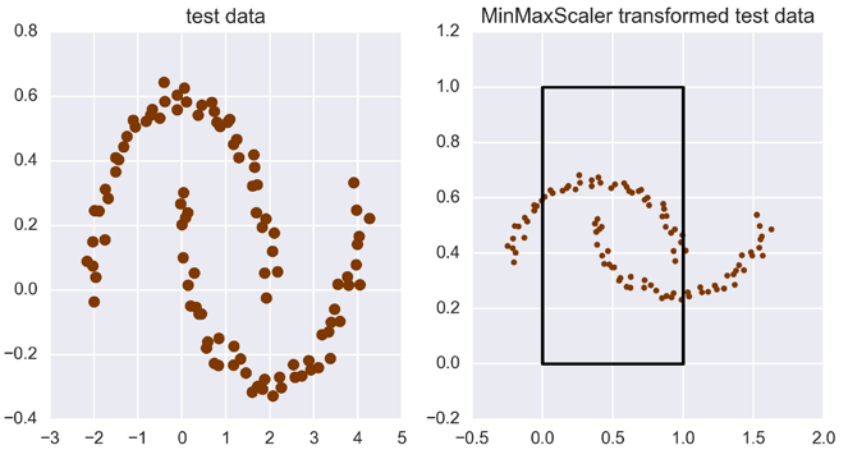


Figure B1-9: Transforming new data. Left: The test data. Right: The test data after transformation. The black rectangle shows the box from (0,0) to (1,1).

Notice that the transformed data is *not* compressed and scaled to the range [0,1] in X or in Y, as we can see from the box from (0,0) to (1,1) in the figure. This is exactly what we'd expect from new data that has larger values than were found in the training data. In this case, the new test data x values are in the range of about [–3, 4], which is much larger than the training data's range of [–1.5, 2.5]. So the transformed X values will fall outside the [0,1] range. The Y range of the test data is about [–0.4, 0.6], which is much less than the training data's range of about [–1, 1.5]. Thus the y values only use up a small piece of the range [0,1] on the Y axis.

So although the transformation shifted and compressed the X values, there's still data falling outside of the X range [0,1]. In the same way, the system shifted and compressed the Y range, but it left a lot of empty space in the Y range [0,1]. Although our estimator works best for data in the range [–1, 1], it will still work well for data that's "close" to that range. The exact meaning of "close" will vary from one estimator to another, so it pays to make sure that the data we give the system is at least vaguely the same as the data it trained on, or the inputs can be much larger or smaller than expected after they're transformed.

Inverse Transformations

We can use `predict()` to get an estimator's output for some data. Remember that these results are based on the data which we fed into the estimator, which we'd transformed first. Whether we're looking at training data, test data, or deployment data, it all went through the transformation.

In the book we discussed a regression problem that asked us to predict the number of cars on a street given the temperature the previous midnight. We transformed the input data, so the value predicted by the estimator was also transformed. We saw that we had to *undo*, or *invert*, the transform so that it represented a number of cars, rather than a value from 0 to 1.

That process is typical. If we want to compare the results that come out of an estimator with our original, un-transformed data, we have to somehow *undo* the transformation. In our scenario above with the `MinMaxScaler`, we want to un-stretch the samples in the exact opposite ways that we originally stretched them.

Every scikit-learn transformer provides a method called `inverse_transform()` for precisely this purpose, where "inverse" means "opposite." So this routine applies the opposite of whatever `transform()` did. Figure B1-10 shows the idea, where we're showing a generic "learner" rather than a specific estimator.

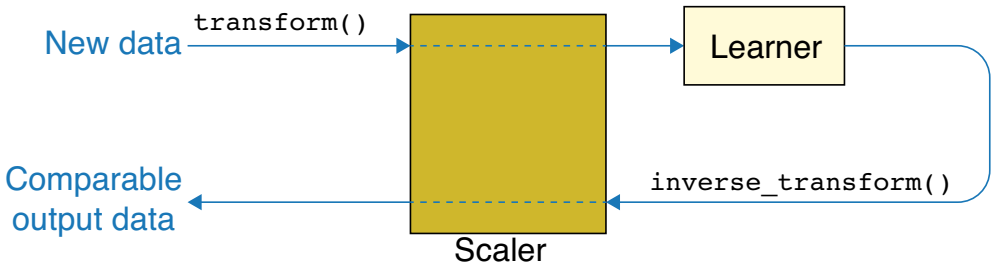


Figure B1-10: To convert data back to its original form, we call the transformer's `inverse_transform()` method on that output data.

If we took the learner out of this loop, the values in the lower-left would be identical to those in the upper-left.

For example, we can feed the samples on the right side of Figure B1-9 to `inverse_transform()`, and we'd get back the samples on the left. Listing B1-18 shows the code.

```
recovered_test_samples = \
    mm_scaler.inverse_transform(transformed_test_samples)
```

Listing B1-18: We call `inverse_transform()` to undo the transformation applied by this object.

Let's see this in action. On the left of Figure B1-11 we see our starting data, which is similar to data we used earlier. The data has an X range of about [0,6] and a Y range of about [–2, 2]. After setting up a new

`MinMaxScaler` and calling `fit()` with this data, and then running it through `transform()`, we get the version on the right, where both ranges are now [0,1].

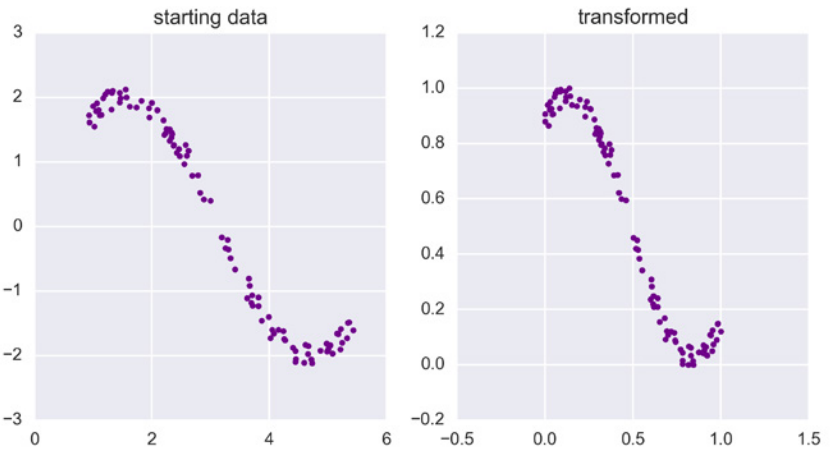


Figure B1-11: Left: Some test data. Right: After setting up a new `MinMaxScaler` object with this data, and then transforming it, both X and Y are in the range [0,1].

Let's now fit a line to our transformed data. We'll use the `Ridge` estimator we saw earlier.

The leftmost image of Figure B1-12 shows the line that our `Ridge` estimator fit to the transformed data, along with the transformed data itself. In the middle image we show that line, along with the *original*, non-transformed input data. It's a terrible match! That's because the line was fit to the transformed data, not the original.

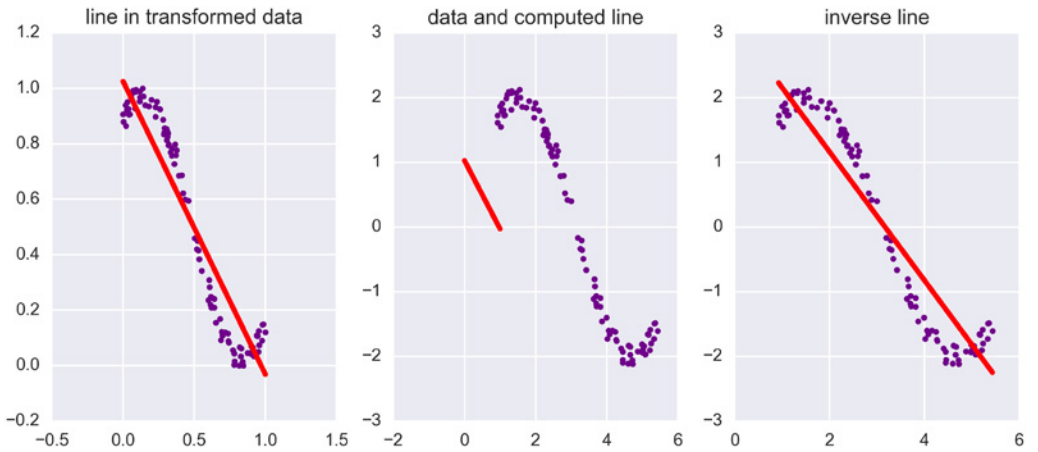


Figure B1-12: Fitting a line. Left: Our transformed data. Middle: Our original data and the line we found from the transformed data (note the different values on the axes). Right: Our inverse-transformed line correctly matches our starting data.

The line in the center figure doesn't match the data at all, because it was fit to the transformed data. To get the line to match up with our input data, we need to run it through the `inverse_transform()` method of the `MinMaxScaler` that we used. To do this, we'll treat the endpoints of the line as two samples, and inverse-transform those two points.

The original data, along with the line after this inverse transformation, are shown on the rightmost image of Figure B1-12.

Data Refinement

See the Jupyter notebook Bonus01-Scikit-Learn-PCA.ipynb.

Sometimes we have too much data.

Maybe some of the features in our data are redundant. For example, if our data records deliveries from a pizzeria, we might have fields for the size of each pizza and the size of the box it should be placed into. It's probably the case that the box size can be predicted from the pizza's size, and vice-versa.

The routines that perform *data refinement* are designed to locate and remove such redundancies from our data either by removing some features altogether, or by making new features out of combinations of others. Another class of routines, such as those related to the Principal Components Analysis (PCA) method, discussed in Chapter 10, are also able to compress data that's related but not entirely redundant. These trade off some loss of information for a simpler database by combining features.

Let's see an example of data compression, from 3D to 2D. Figure B1-13 shows a dataset of points drawn from a 3D blob shaped like a bulging ellipse. The X range is around [-0.8, 0.8], the Y range is around [-0.3, 0.3] and the Z range is about [–0.7, 0.7].

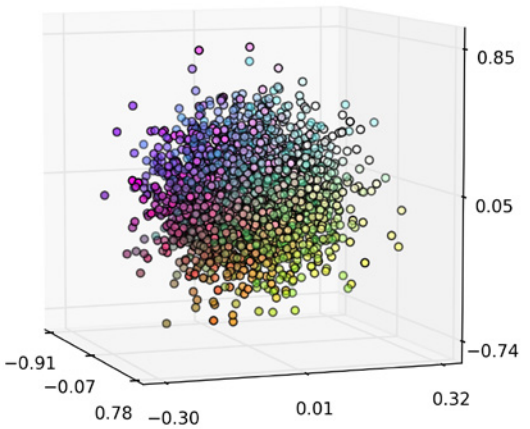


Figure B1-13: Our starting 3D blob. The smallest range is in the Y values. The colors are just to help us keep track of where points are located.

Let's reduce this 3D blob down to just 2 features (that is, we'll compress it into 2D data). We'll use PCA for this. Leaving out the code to build the blob and draw the plots, the core steps are only three: create the `PCA` object (and tell it how many dimensions we want), call `fit()` so it can determine which features to remove, and call `transform()` to apply the transformation. Listing B1-19 shows the code. Here we're also asking PCA to "whiten" the data, which can help PCA produce its best results.

```
from sklearn.decomposition import PCA

# make blob_data, the 3D data forming the "blob"

pca = PCA(n_components=2, whiten=True)
pca.fit(blob_data)
reduced_blob = pca.transform(blob_data)
```

Listing B1-19: To apply PCA, we first make the `PCA` object. Here we tell it to reduce our data down to 2 features, and to whiten each feature along the way. Then we call `fit()` with our 3D data so the PCA algorithm can determine how to reduce it. Finally, we call `transform()` to apply the transformation and get back our reduced, 2D data.

The result is shown in Figure B1-14. Each data point is now described by only two values, rather than three. In other words, the result of PCA is that our data has gone from being three-dimensional to two-dimensional. We requested PCA to save reduce our original 3 features into only 2 while retaining as much information as possible.

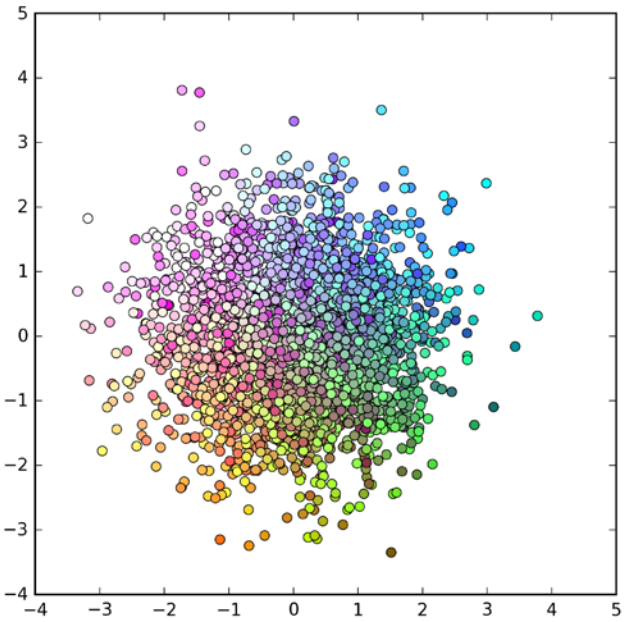


Figure B1-14: Our starting blob of Figure B1-13 after running it through PCA.

Note that it wasn't simply squashed along one dimension. As we discussed in Chapter 10, the algorithm placed a plane through the 3D blob so that it captured as much variance in the data as possible, and then projected each point onto that plane.

Ensembles

See the Jupyter notebook Bonus01-Scikit-Learn-5-Ensembles.ipynb.

Sometimes it's useful to create a bunch of similar but different estimators, and let them all come up with their own predictions. Then we can use some kind of policy (usually voting) to choose the "best" prediction, and return that as the group's best result.

As we saw in Chapter 12, such collections of estimators are called *ensembles*. Let's see how to build and use ensembles in scikit-learn.

The system is set up so that we can treat an ensemble just like any other estimator. That is, we wrap up all the individual estimators into one big estimator, and use that directly. Scikit-learn takes care of all the internal details for us.

So just like any other estimator, our first step in using an ensemble is to create an ensemble object. Then we call its `fit()` method to give it data to analyze. Finally, we can call its `predict()` method to evaluate new data. We don't have to concern ourselves with the fact that there are multiple estimators hiding inside the estimator object we're using.

Some of the ensembles in scikit-learn will make these collections out of any kind of estimator, while others are limited to just classifiers, just regression algorithms, or even just specific instances of algorithms.

Let's build an ensemble to do classification. We'll use a data set of 1000 points, formed into 5 spiral arms, one for each of 5 classes. This starting data is shown in Figure B1-15.

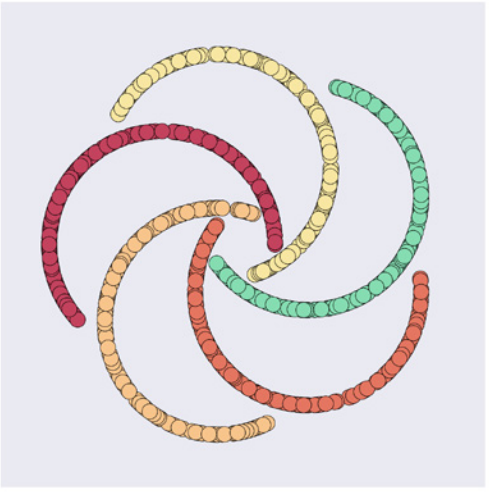


Figure B1-15: Our starting data for ensemble classification. Each of the five arms contains 200 points of a different class.

We'll use 2/3 of these points, randomly selected, as our training data, and the remaining 1/3 as test data.

We'll use a general-purpose ensemble maker which can build a group of classifiers form almost any specific classifier algorithm. The ensemble maker is called `AdaBoostClassifier`. The word `Boost` comes from the fact that internally it uses the technique of *boosting*, which we discussed in Chapter 12.

When we create the ensemble with this object, we tell it which algorithm it should build multiple copies of. We'll use the `RidgeClassifier` classifier, which is the classifier version of the Ridge regression algorithm we used above. Listing B1-20 shows the steps.

```
from sklearn.linear_model import RidgeClassifier
from sklearn.ensemble import AdaBoostClassifier

ridge_ensemble = AdaBoostClassifier(RidgeClassifier(), algorithm='SAMME')
```

Listing B1-20: Creating an ensemble of `RidgeClassifier` objects using the `AdaBoostClassifier` ensemble object. The `algorithm` argument needs to be set to `SAMME` for this classifier (this is not a typo of the word "same"), as explained in the documentation.

The documentation for `AdaBoostClassifier` explains that it has two different modes, depending on what methods are supported by the classifier it's building a collection of. In the case of the `RidgeClassifer`, we need to specify the `SAMME` algorithm (note that this is not the word *same*, but instead an acronym with two `M`s).

We can control how many classifiers go into our ensemble using the optional `n_estimators` argument. For this demonstration we'll leave it at the default value of 50 estimators. We'll also leave all the other arguments (which include the learning rate) at their defaults.

Now that our collection is made, using this ensemble of estimators is just like using one of them. We train the ensemble (and all the estimators inside of it) with samples we pass into it using `fit()`. Since we've build this ensemble for supervised learning of categories, the `fit()` routine takes in both the samples and their labels. Listing B1-21 shows the call to our ensemble's `fit()` routine.

```
ridge_ensemble.fit(training_samples, training_labels)
```

Listing B1-21: Fitting our ensemble object.

Finally, we can get new values out by asking the ensemble to predict them using `predict()`, as in Listing B1-22.

```
predicted_classes = ridge_ensemble.predict(new_samples)
```

Listing B1-22: Using our ensemble object to make new predictions.

Internally, ensembles usually pick the final output by using some kind of voting algorithm, where the most frequently predicted category becomes the final result.

Figure B1-16 shows the classification of our test data from our ensemble of 50 Ridge classifiers. These results aren't terribly encouraging.

Figure B1-16: Using our ensemble of 50 Ridge classifiers. Left: The test data, color-coded by the category each point was assigned to. Middle: The regions created by our ensemble. Right: Overlaying the spiral points on the regions.

The trouble is that we're trying to fit 50 straight lines to this data in a way that will let them correctly categorize our swirling data.

We could try to improve these results by experimenting with the number of estimators, the learning rate, or the arguments to the estimators.

Another approach would be to try another estimator. Let's use decision trees. The only changes we need to make are to import the necessary module, and then tell `AdaBoostClassifier()` to build the ensemble out of these classifiers, as in Listing B1-23.

```
from sklearn.tree import DecisionTreeClassifier

tree_ensemble = AdaBoostClassifier(DecisionTreeClassifier())
```

Listing B1-23: Building an ensemble out of decision trees. For decision trees, we can leave the `algorithm` argument at its default value.

Now we use `fit()` and `predict()` just as before Figure B1-17 shows the results of classifying with 50 decision trees. For this data, and using all the defaults, decision trees turned in a nearly perfect performance!

Figure B1-17: The results of an ensemble made of 50 decision tree classifiers. Left: The test data, color-coded by the category each point was assigned to. Middle: The regions created by our ensemble. Right: Overlaying the spiral points on the regions.

Automation

See the Jupyter notebook Bonus01-Scikit-Learn-6-Automation.ipynb.

See the Jupyter notebook Bonus01-Scikit-Learn-7-Polynomials.ipynb.

Machine learning systems have lots of parameters, and often lots of hyperparameters, too. The distinction is that the system learns its parameters from the data, while we set the hyperparameters. Typical hyperparameters include the learning rate, the number of clusters in a clustering algorithm, the number of estimators in an ensemble, and the amount of regularization to apply.

We often want to try out many values of these hyperparameters to find the combination that gives us the best performance for a given system and data. If we can't find the very best results, we'd at least like to try a few combinations and use the ones that do the best.

If we want to automate this search process, we need two basic pieces. The first automates the selection of hyperparameters, choosing the values that get used to build and train a learner. The second piece evaluates that learner and assigns a score to its performance. We can then choose the combination that resulted in the best score.

Scikit-learn offers tools for both of these steps. We think of them as *automation* tools, since they handle this repetitive process for us.

Cross-Validation

Most learning algorithms have multiple parameters and hyperparameters that control how quickly and how well they learn. Finding the best combination of these values can be difficult, because they depend on the nature of the data we're training with.

As we saw in Chapter 8, we can use *cross-validation* to determine the quality of a model on a given set of data. This technique is particularly attractive when we don't have a big training set, because it doesn't require us to remove a permanent validation set from the training data. Instead, we break up the training into equal-sized pieces called *folds*, and then train and evaluate our model independently many times, using the data in one of the folds as our validation set. The performance of the model is typically reported as the average performance of these multiple versions.

Scikit-learn lets us automate this process. We hand a routine our estimator, our data, and the number of folds we want it to use, and it carries out the whole process for us, reporting the average score when it's done. Many scikit-learn estimators have specialized versions that carry out cross-validation. These are consistently named by appending the routine's usual name with the letters `CV` at the end. They're all listed in the API documentation.

For example, let's suppose we want to use the Ridge classifier we used earlier to learn categories from a set of labeled training data. To evaluate its performance, we'll use cross-validation.

The Ridge classifier is implemented as an object called `RidgeClassifier`. Following the convention we just described, the version of `RidgeClassifier` object with cross-variance built in is called `RidgeClassifierCV`. So our first step is just to make one of these objects.

To carry out cross-validation on this estimator we need only call its `score()` method with our training data and labels. This one call does the whole cross-validation process for us from start to finish. The `score()` routine takes care of chopping up the data into folds, building and evaluating a `RidgeClassifier` for each version of the training data, and then returning the average of the scores. Listing B1-24 shows the steps.

```
from sklearn.linear_model import RidgeClassifierCV

ridge_classifier__cv = RidgeClassifierCV()
mean_accuracy = ridge_classifier_cv.score(training_samples, training_labels)
```

Listing B1-24: To run the `RidgeClassifier` object through cross-validation, we first create an instance of the `RidgeClassifierCV` object. Then we call its `score()` routine with our training data and labels. The result is the average accuracy value from running through the training and validation process for each fold.

We can pass a variety of optional parameters into our `RidgeClassifierCV` object when we create it. Perhaps the most important parameter is the number of folds we want to use (here we left it at the default value of 8).

`RidgeClassifierCV` uses a reasonable fold-making algorithm that just breaks up the data into equal pieces. This often works pretty well, but scikit-learn offers several alternatives, called *cross-validation generators*, that can sometimes do better. For instance, the `StratifiedKFold` cross-validation generator pays attention to the data when it creates the folds, and tries to

distribute the number of instances in each class equally. In other words, it tries to make sure each fold has roughly the same makeup as the entire dataset, which is always a good idea.

To use this approach, we first create a `StratifiedKFold` object, and tell it how many folds to use (that's the value of `K` in the object's name). The name of this argument for this object is unfortunately not named something like "number of folds," but is instead `n_splits`.

The `StratifiedKFold` object doesn't need our data when we create it, because the cross-validator will send the data to it for us when it uses this object to build the folds. We just provide our cross-validation stratifier object as the value of an argument named `cv` to the cross-validation object `RidgeClassiferCV`, and the rest happens automatically. Listing B1-25 shows the code.

```
from sklearn.model_selection import StratifiedKFold

strat_fold = StratifiedKFold(n_splits=10)
ridge_classifier_cv = RidgeClassifierCV(cv=strat_fold)
```

Listing B1-25: To use a stratifier of our own choice, we have to first create it, which means we also have to import it. Here we import the `StratifiedKFold` object and tell it to use 10 folds. We pass that to our `RidgeClassifierCV` object through the argument `cv`.

Let's look at this process visually. The scikit-learn model of cross-validation with a classifier is summarized in Figure B1-18. The input to the system consists of the labeled training data and the number of folds, along with the model constructor (that is, the routine that creates the classifier) and its parameters (which we've been leaving at their defaults in our examples). The routine then loops through each fold, removing it from the data, training on what remains, and evaluating the result on the extracted fold. The resulting scores are produced as output, or are averaged together to present just a single value. In this figure, and those to follow, a wavy box with arrows represents a loop. In this case, it's the loop that runs through the process above once for each fold.

Each time through the loop, one fold is extracted from the training data. The remaining samples are used to train a model, and then we evaluate the result on the validation fold, producing a score. The result is one score for each fold. We either present those scores, or their average, as output.

Figure B1-18 is missing any transformations. That is, we're not scaling or standardizing our data, or running feature compression on it, or any of the other transformations that we've seen, which can help our models learn.

To include these transformations in cross-validation we have to be careful. If we just drop a transformation into the inner loop of Figure B1-18 without thinking things through, we can risk information leakage. We saw in Chapter 10 that this can give us a distorted view of the performance of our model.

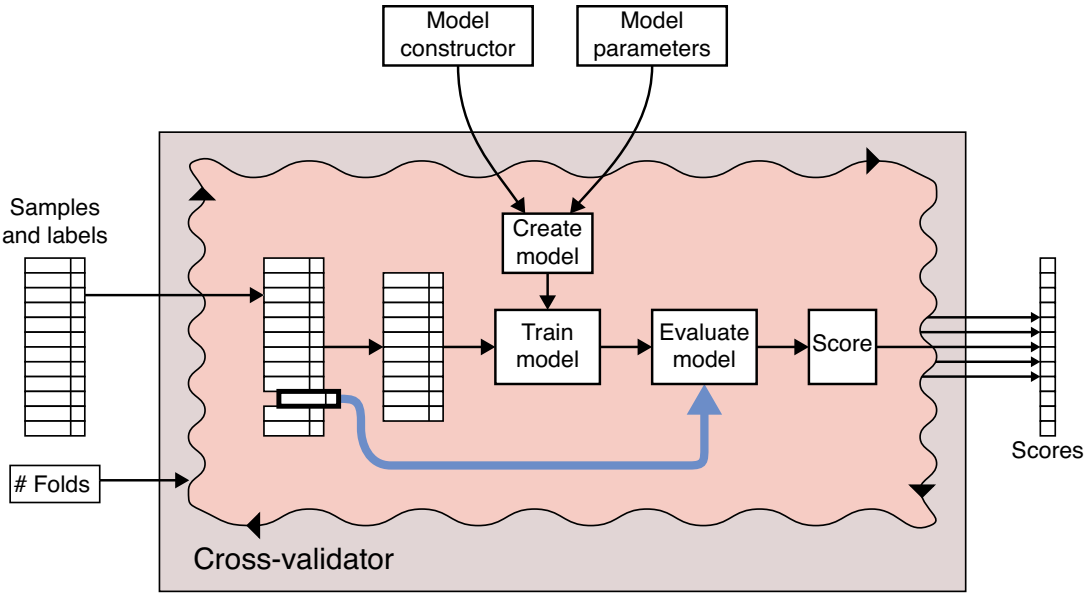


Figure B1-18: A visualization of cross-validation with a classifier. The wavy box with arrows represents the loop of the process.

To include transformations properly we can use another scikit-learn tool called a *pipeline*. A good way to understand pipelines is to see how they're used when we search for hyperparameters, which is an important topic on its own.

So we'll look at searching and pipelines next, and then we'll come back to cross-validation and see how to use a pipeline to include transformations.

Hyperparameter Searching

We often distinguish between *parameters*, which are usually adjusted by the algorithms themselves in response to data, and *hyperparameters*, which we generally set by hand.

For example, suppose we want to run a clustering algorithm on a dataset. The cluster sizes and centers are parameters that are learned from the data, and maintained "inside" the algorithm. In contrast, the number of clusters to use is set by us "outside" of the algorithm, and we call such values hyperparameters.

Finding the best values for all the hyperparameters in a learning system is often a challenging task. There may be many values to tinker with, and they might influence one another. So using an automated approach to searching for them can save us a lot of time and hassle.

We can even generalize the idea of searching for hyperparameter values to searching for a choice of algorithm. For example, we might want to try out several different clustering algorithms, looking for the one that does the best job with our data.

To make this search easier, scikit-learn provides a bunch of routines that automate these types of searches for parameters, hyperparameters, and algorithms.

We identify the types of things we want to search over and the values we want them to try out, and the criteria on which to judge each result. Our search process then runs through every combinations of values, ultimately reporting the set that produced the best results.

Note that although scikit-learn makes it easy to set up and run this search, it's not any faster than if we were to do it by hand (except for the typing time). It just methodically grinds though the values to be searched, then builds, trains, and measures the resulting algorithm, over and over again. The upside is that we can be asleep, or reading a book, or doing anything else we please while the computer works its way through the combinations.

There are two popular ways to approach this kind of search: the *regular grid* and the *random search*.

The regular grid tests every combination of the parameters. The word "grid" is in its name because we can visualize all the combinations it tries as points on a grid. A 2D example is shown in Figure B1-19.

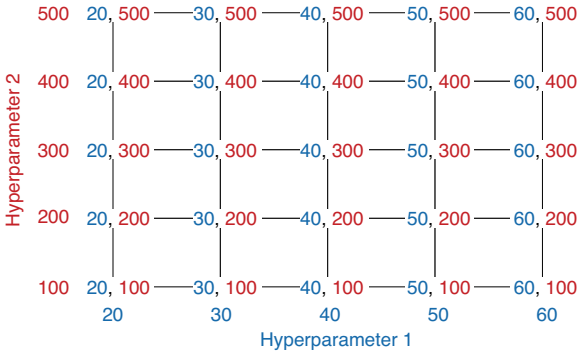


Figure B1-19: A regular grid search using two hyperparameters, each with 5 values. For every combination of the two values, we build, train, and test a new system. There are 25 combinations to be tried.

In this example, we have 5 values for each of two hyperparameters, giving us $5 \times 5 = 25$ combinations to examine.

As we include more parameters to be searched, or more values for each one, the size of this grid, and thus the number of training/measuring steps we have to carry out, will grow correspondingly. This means the whole thing will require more and more time to run. Eventually it could require more time than we have available. For example, if we have 4 hyperparameters to search, and 6 values for each one, that's $6 \times 6 \times 6 \times 6 = 1{,}296$ different combinations. If it takes an hour to train and test each model, that search would take 54 days of non-stop computing!

It would be great to cut away parts of the grid and save time, but that's a risky step because we can't know beforehand what combination of variables might turn up the best results.

A grid search usually proceeds methodically, working its way through all the possible combinations in a predictable, fixed order, such as left to right, and top to bottom.

A faster but less informative alternative is to search these combinations *randomly*, rather than in some fixed order. The algorithm picks a random combination of hyperparameter values that hasn't been tried yet, trains and measures the resulting model, and then picks another untried random combination, and so on. We can think of many different conditions to control when this process ought to stop. For example, the algorithm could stop when it's searched every combination, or it's tried a certain number of combinations that we've specified, or it's run longer than an amount of time that we've specified, or we simply get tired of waiting and stop it manually. Then it returns the best combination it found. Figure B1-20 illustrates the idea. Random sampling can give us an overall feeling for where the big scores are located without exhaustively trying every combination first. Here, we ran through 9 steps of picking a random combination, then built and measured the resulting model.

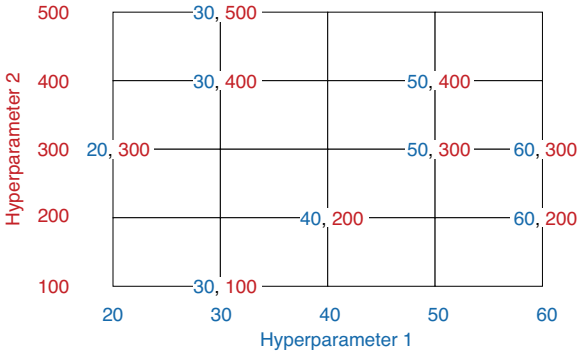


Figure B1-20: A random search doesn't test every combination in sequence, but tests them in random order.

The advantage of this approach over the regular grid is that we can watch the results as they come in, and perhaps get an idea for where we can focus our search. Then we can stop and start again in the neighborhood that looks promising. For example, suppose that in Figure B1-20 the combination (40, 200) performed much better than any of the other combinations. As a result, we might decide to run a new close-up search in that area, perhaps using a regular grid, as in Figure B1-21.

This is called *multiresolution searching*, since we're changing the resolution, or step size, between grid elements as we look more and more closely for the best combination.

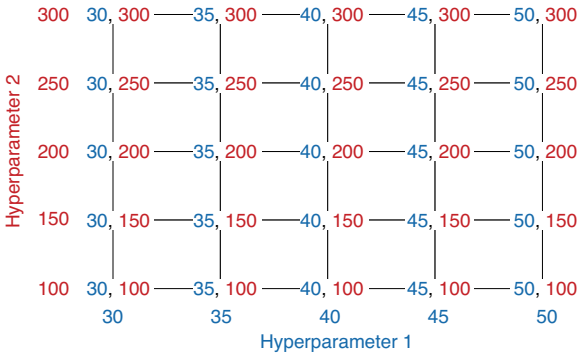


Figure B1-21: When we think we've found the neighborhood of the best values, we can run a new search that zooms in on that region. Here we look around the region where hyperparameter 1 has a value of 40, and hyperparameter 2 is 200.

Let's look at how to use scikit-learn to run a regular, methodical grid search first. We'll see that we set up and call the randomized version in almost an identical way.

Exhaustive Grid Search

The exhaustive, methodical grid-based search is provided by an object called `GridSearchCV` (the `CV` at the end reminds us that the algorithm will use cross-validation to evaluate the performance of each model it tries out).

`GridSearchCV` produces each combination of parameters one by one, builds and trains the model, and then measures how well the corresponding model performs on our data, returning a final score for that model.

Note that the estimator we hand to this routine shouldn't perform cross-validation itself (that is, its name shouldn't end in `CV`), because the grid searcher is handling that step.

Figure B1-22 shows the pieces that go into making this grid-search object.

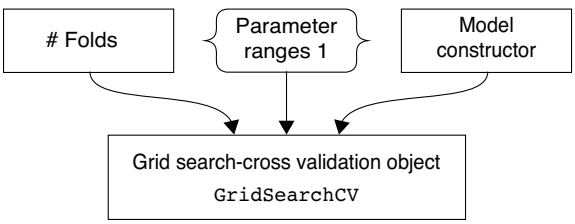


Figure B1-22: The pieces that go into making a basic grid-search object. We supply the number of folds to use during each cross-validation, the parameters we want searched, and the values we want them to take on, and a constructor for the model that we're investigating.

The object takes a number of folds for the cross-validation step (the default is 3), a set of parameters and their values that should be searched, and the routine that builds our learner. There are many more optional parameters that we won't get into.

Conceptually, we can think of the grid search process in two steps. The first step builds a list of every combination of values of the parameters. So if there are just two parameters, we could save this list as a 2D grid. If there are three parameters, we could think of it as a 3D volume, and so on.

The second step runs through those combinations one at a time, building the model, evaluating it with cross-validation, and saving the score. We could save that score in another list (or grid, volume, or bigger structure) of the same shape and size as the list of parameter combinations. Then we can quickly see what score got assigned to each combination.

Figure B1-23 shows a visual summary of the whole process. At the left we see the routine that creates our learning model (perhaps a decision tree algorithm, or a regression algorithm like Ridge). There's also a collection of the parameters we want to vary, and the values we want to try for each one. Let's assume we're interested in exploring two hyperparameters, and we want to try 7 possible values for the first, and 5 possible values for the other.

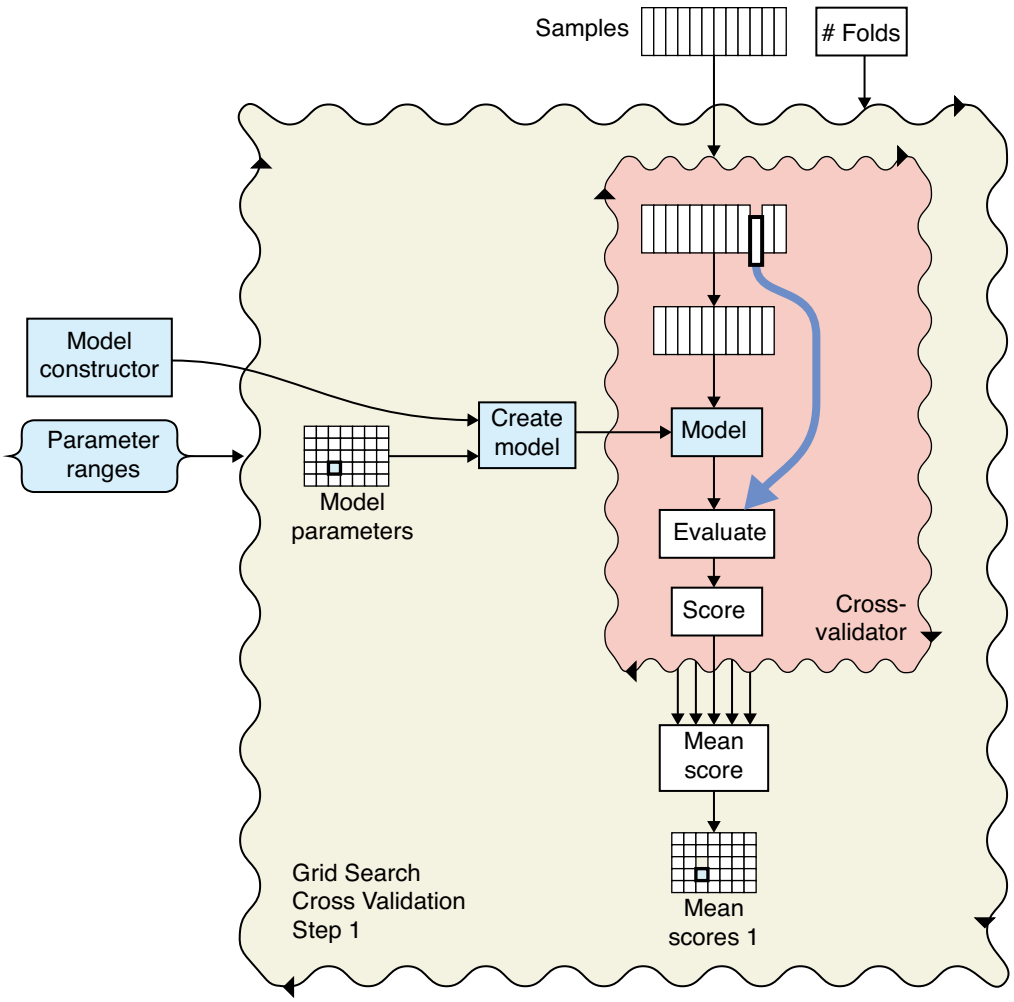


Figure B1-23: Grid searching with cross-validation.

In the first step of processing, we find all combinations of those parameters and values and save them in the grid marked "model parameters." This grid is 7 by 5, with one entry for each combination of values for the two hyperparameters.

Now we start running the loop. The outer wavy box represents the main loop of the grid searcher. Each time through the loop it uses the model constructor, and one set of model parameters from the grid, to create our model. At the top we can see the samples that make up our training set. If we're using a supervised learner, these samples will have labels. We also provide the number of folds to use when cross-validating.

All of this goes into the inner wavy box, which contains the cross-validation step we saw in Figure B1-18 (though turned on its side). The scores from the cross-validation steps are saved and then averaged, with the result saved in a grid the same shape and size as the model parameters, so it's easy to find the score for each combination of parameters.

Once the searching loop is completed, we look through all the scores saved during the search process and find the parameters that produced the best results. We make a new model using those parameters, train it, and return it, as in Figure B1-24.

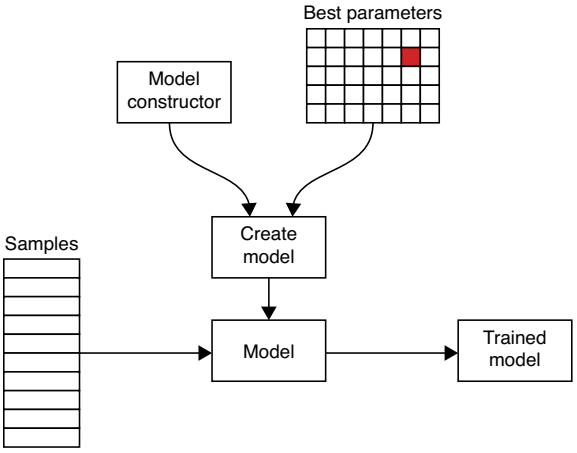


Figure B1-24: We finish the grid-search process by identifying the parameters that gave us the best score. We build a new model from those parameters, train it on all the data, and return that model.

Let's see some code.

We'll use a Ridge classifier from before to classify a bunch of data. The `RidgeClassifier` object takes a lot of arguments, which we've so far been ignoring. Let's pick two of those arguments and search for their best values for a particular set of data.

We'll start with the parameter named `alpha`, which is the regularization strength (this is often called lambda (λ), but it's not unusual to see different Greek letters used for this). Recall from Chapter 9 that regularization helps

us prevent over-fitting. In the `RidgeClassifier`, `alpha` is a floating-point value that's 0 or greater. Larger values mean more, or stronger, regularization. As a first guess, we'll try these six values of alpha: 1, 2, 3, 5, 10, 20.

The second parameter we'll search for is called `solver`. This is the algorithm used internally by `RidgeClassifier` to do its job. Without going into the details of each one, let's say we'd like to try out a few and see if any one performs particularly well. The documentation for `RidgeClassifier` tells us that we refer to these algorithms with a string. Let's arbitrarily pick three of the available options: `'svd'`, `'lsqr'`, and `'sag'`.

Now we need to tell the grid searcher that we want to train and score multiple versions of the `RidgeClassifier`, using these values for the parameters named `alpha` and `solver`.

To communicate which values we want to assign to each parameter, we use a Python *dictionary*. In a nutshell, a dictionary is a list of key/value pairs enclosed in curly braces. Our keys will be the strings that name our parameters, and our values will be lists that the parameters can take on.

Thanks to how Python is structured, we can just name the parameters we want to search on as strings, and use them as the keys in our dictionary. The values we want each argument to take on are named in a list, and assigned to the value for that key. Our dictionary is given in Listing B1-26.

```
parameter_dictionary = {
    'alpha': (1, 2, 3, 5, 10, 20),
    'solver': ('svd', 'lsqr', 'sag')
}
```

Listing B1-26: Building a dictionary to hold the values we want to search over.

When the grid searcher builds a new `RidgeClassifier`, it will assign one value in the `alpha` entry to the `alpha` argument, and one value in the `solver` entry to the `solver` argument. It does this by matching up the names, so the names in our dictionary must exactly match the parameter names used by `RidgeClassifier()`.

The grid searcher will build and score $6 \times 3 = 18$ different models from this dictionary. The models will be made with calls like those in Listing B1-27, though this all happens invisibly to us. Because of how Python's dictionaries are defined, the algorithm might not go through the choices in this order, but that detail will usually also be invisible to us.

```
ridge_model = RidgeClassifier(alpha=1, solver='svd')
ridge_model = RidgeClassifier(alpha=1, solver='lsqr')
ridge_model = RidgeClassifier(alpha=1, solver='sag')

ridge_model = RidgeClassifier(alpha=2, solver='svd')
ridge_model = RidgeClassifier(alpha=2, solver='lsqr')
ridge_model = RidgeClassifier(alpha=2, solver='sag')

ridge_model = RidgeClassifier(alpha=3, solver='svd')
....
```

```
ridge_model = RidgeClassifier(alpha=20, solver='lsqr')
ridge_model = RidgeClassifier(alpha=20, solver='sag')
```

Listing B1-27: The start and end of the 18 models that the grid searcher will automatically build, based on the dictionary of Listing B1-26. Each model is automatically trained and measured using cross-validation. They may not be generated in this order.

To run the search, we make a `GridSearchCV` object with the three items shown in Figure B1-22. These are the number of folds to use, the dictionary of parameters and ranges, and the model constructor. Listing B1-28 shows the code.

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import RidgeClassifier

ridge_model = RidgeClassifier()
parameter_dictionary = {
    'alpha': (1, 2, 3, 5, 10, 20),
    'solver': ('svd', 'lsqr', 'sag')
}
num_folds = 3

grid_searcher = GridSearchCV(estimator=ridge_model,
                             param_grid=parameter_dictionary,
                             cv=num_folds)
```

Listing B1-28: Building a grid searcher. We provide it with the estimator we want it to use (here a `RidgeClassifier`), the parameter dictionary with parameter names and values, and the number of folds to use. We could put the number of folds in the dictionary if we wanted to search different values of that as well. To show that we don't need to, here we're passing it as a constant value.

Hold on a second. Something's fishy here. We saw in Listing B1-27 that the grid searcher is going to make 18 different `RidgeClassifier` objects, each with different parameters. But in Listing B1-28 we made one `RidgeClassifier` and stored it in the variable `ridge_model`, and we gave it no parameters at all. How does the grid searcher take this one object and make 18 new versions of it, each with different parameters?

The answer to this question is also based on how the Python language is set up. Let's look at what's happening at a very high level.

We're providing to the searcher a routine that makes a `RidgeClassifier` object as an argument to its `estimator` parameter. As a result, the grid searcher is able to call that routine with new arguments to make new models. In other words, it can do exactly what Listing B1-27 shows, but using the argument to `estimator`. Since the value we're assigning to this argument is itself a procedure, invoking that procedure is the same as explicitly invoking `RidgeClassifier()`. This mechanism makes it easy for us to later call `GridSearchCV()` with a completely different estimator if we want, just by changing the routine we provide to `estimator`. If the new model takes the same `alpha` and `solver` parameters, then we can leave our dictionary just as it is. If it takes other parameters, we'd want use a different dictionary.

In short, the grid searcher generates all combinations of the parameters, passing each combination to whatever routine we provided as a value of `estimator` in order to create a new instance of the estimator object. In this case, it will make a new `RidgeClassifier` object. It then runs that object through cross-validation using our data and evaluates its performance.

Creating the `GridSearchCV` object prepares it to run the search, but doesn't actually start the process. To run the search, we call the `GridSearchCV` object's `fit()` method. Since in this example we're training a classifier, we'll give it our samples and our labels. Listing B1-29 shows the code.

```
grid_searcher.fit(training_samples, training_labels)
```

Listing B1-29: To run the search, we call the searcher's `fit()` method with our training data and samples.

That's all it takes. We make that call and then everything happens automatically. When this line returns, the system has exhaustively searched all combinations of our parameters.

We should always pause for a moment before starting a grid search, because we can be in for a long wait. A *long* wait. As we saw before, if we're searching a lot of parameters, and each run takes a while, we might wait for hours or weeks (or more!) for `fit()` to finish its work.

When `fit()` is done, we can query our `grid_searcher` object to discover what it found. The object saves its results in internal variables. Each of its variables ends with an underscore to help us avoid confusing those names with our own variables.

The documentation provides a complete list of all the internal variables that contain our results. Let's look at three of the most useful variables. `best_estimator_` (note the trailing underscore) tells us the explicit construction call that the searcher made to make the estimator that resulted in the best score, with all the arguments (including all the defaulted arguments we didn't specify). That best score itself is given by `best_score_` (again with a trailing underscore). If we just want the best set of parameters, then `best_parameters_` (with a trailing underscore again) contains a dictionary that has just the best value for each parameter from our original dictionary.

Let's put this into action. In Figure B1-25 we show some data in a pair of half-moons (generated with the scikit-learn data-making utility `make_moons()`, which we'll see below). On the right we show the results of creating the grid search in Listing B1-28 followed by the fitting step in Listing B1-29. Since we're using a `RidgeClassifier`, we'd expect the straight line that does the best job of separating these two classes, though no straight line is doing to do a perfect job. The figure shows that the classifier found a good straight-line approximation to split the data. A more complicated classifier could have found a more complex and accurate boundary curve.

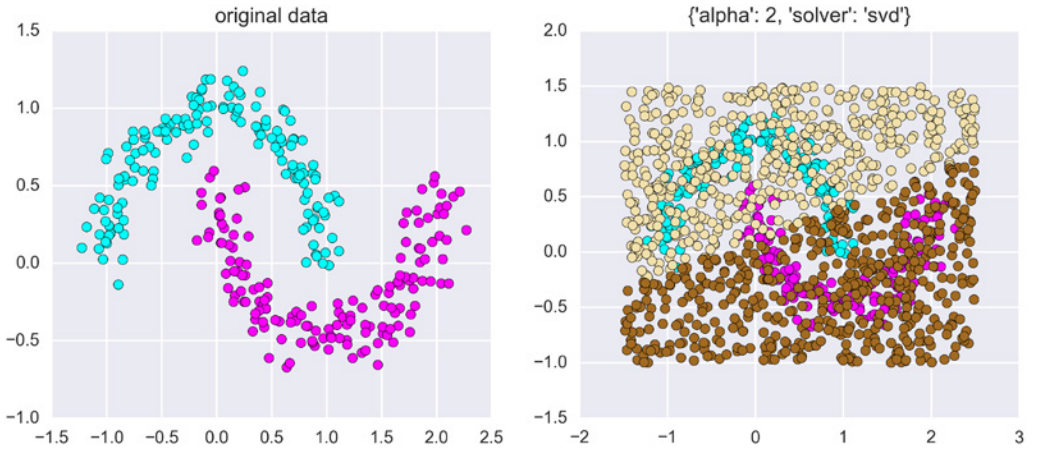


Figure B1-25: On the left, our original training data with 2 categories. On the right, the result of our grid search, with 500 predicted points drawn on top of the original data.

For clarity, Figure B1-26 shows just the test data by itself. Each point is colored by the class it was assigned by the classifier.

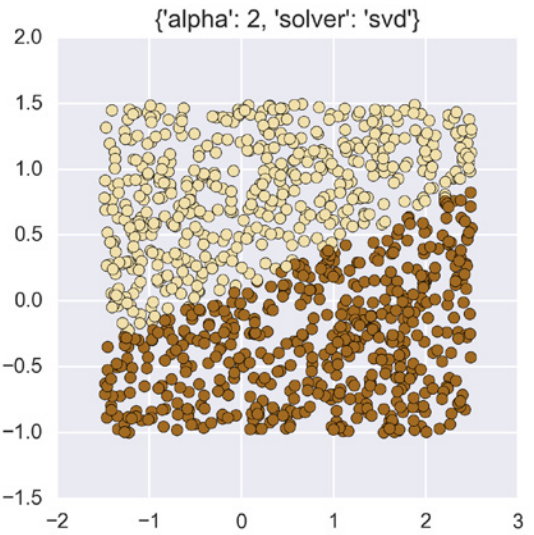


Figure B1-26: Just the test data from Figure B1-25

In Listing B1-30 we show the resulting values of three variables we just discussed, and their values. Outputs from the computer are shown in red.

```
grid_searcher.best_estimator_
    RidgeClassifier(alpha=2, class_weight=None, copy_X=True,
    fit_intercept=True, max_iter=None, normalize=False,
    random_state=None, solver='svd', tol=0.001)
```

```
grid_searcher.best_score_
    0.87
grid_searcher.best_parameters_
    {'solver': 'svd', 'alpha': 2}
```

Listing B1-30: The variables describing the best combination found for our data in Figure B1-25

Note that the best estimator provides us with the full call to `RidgeClassifier` with all of its parameters, including the optional ones. The parameter `best_parameters_` is a dictionary, here with 2 key/value pairs.

The output shows that `best_estimator_` contains everything that's in `best_parameters_`, but the latter restricts itself to just the parameters we were searching on.

If we dive deeper into the variables in our `grid_searcher` object we can look at `cv_results_`, which is a huge dictionary with detailed results from the search. In Figure B1-27, we've plotted data from two entries from that variable, `mean_train_score` and `mean_test_score`, which give us the scores for the training and testing data for each of the 18 parameter combinations (these names don't end with an underscore, because they are members of the `cv_results_` dictionary which does have an underscore. It's a quirk of scikit-learn's naming policy). We can see that in this run that three combinations of the parameters gave us the same, best testing results. That is, the choice of `solver` didn't make any difference in this situation.

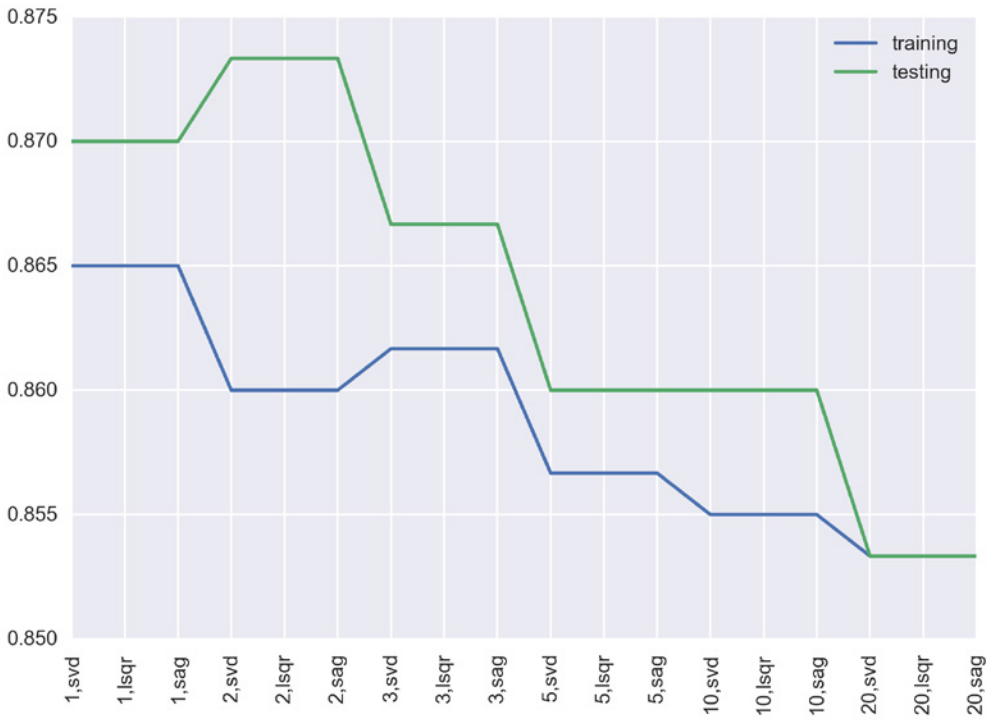


Figure B1-27: Training and testing scores for each of our 18 searches. The best training results came from using `alpha=2` and any of the three `solver` choices.

The results show that using an alpha value of 2 produced the best results on the testing sets, regardless of the algorithm used internally by `RidgeClassifier`. Since the testing data is a proxy for real data, we'd want to use `alpha=2` if we were going to deploy this algorithm. The system picked the `svd` algorithm in the "best" variables from the 3 equally performing combinations because it was the first one encountered.

Random Grid Search

Now that we know how to do the exhaustive version of grid searching, making the switch to the random version is almost no work at all.

Instead of using a `GridSearchCV` object, we use a `RandomizedSearchCV` object (it also comes from the `model_selection` module).

This object takes the same arguments as `GridSearchCV`, but the randomized version also takes an argument called `n_iter` (for "number of iterations"), telling it how many unique, randomly-chosen parameter combinations it should try (it defaults to 10).

Pipelines

Suppose that as part of our grid search, we'd like to include some form of data pre-processing. We might want to normalize or standardize the training set, or do something more complex like running PCA on it.

It would be natural to search for the best parameters for that transformation while we're searching for other parameters, like those we saw above. For example, we might want to try using PCA to reduce an 8-dimensional data set down to 5, 4 and 3 dimensions, and see which (if any) of those choices give us the best performance.

But now we have two objects inside the loop: the pre-processing object and the classifying object. How do we tell the searcher, whether systematic or randomized, to use both of these, and how to we tell it which parameters should be delivered to which object?

The answer is to package up the whole sequence of actions we want performed into a *pipeline*. Then the searcher will proceed as usual, only instead of just calling an estimator, it will call the pipeline, and execute all the steps inside of it.

Let's demonstrate this using the same half-moons data we used in Figure B2-25.

To illustrate the pre-processing step, let's transform our data every time through the loop in such a way that the `RidgeClassifier` object will be able to fit a curve to the data, rather than just a straight line.

To do this, we'll use a pre-processing step to produce *more data* for the classifier. This is a technique we haven't seen before, so let's see what it does before we go into the code to create and use it.

When a `RidgeClassifier` object finds a boundary curve, it builds it from combinations of the features in the samples. If all we give our classifier are the x and y values of the two features in our samples, as we did earlier, then looking at the math we'd see that the most complex shape the classifier can

build from combining them is a straight line. In other words, the reason we got a straight line from `RidgeClassifier` in Figure B1-25 isn't because of the classifier, but because the data we gave it had only the two features of x and y.

If we create some additional features out of the original data, then the classifier will have more to work with. We'll make those features by multiplying together the x and y values in different ways.

For instance, we can multiply the y value with itself a bunch of times. This would make $y \times y$, or $y \times y \times y$, or $y \times y \times y \times y$, and so on. Mathematicians call these *polynomials*. The number of features that are used in these little expressions is called the *degree* of the polynomial. Figure B1-28 shows the shapes of these polynomials made up of repeatedly multiplying x by itself.

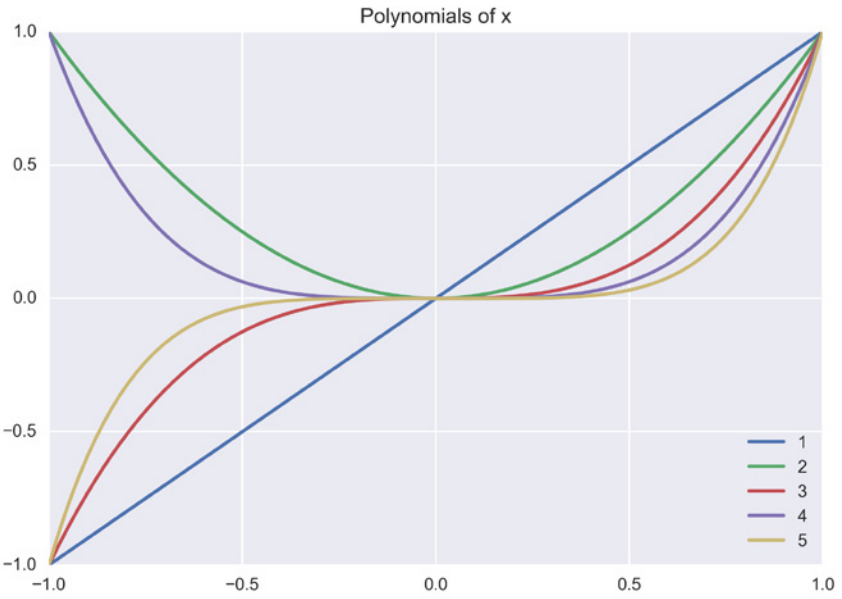


Figure B1-28: The polynomials of x plotted from –1 to 1. The number in the legend is the degree, and tells us how many variables are used to make that curve.

So x by itself is a polynomial of degree 1, $x \times x$ is of degree 2, $x \times x \times x$ is of degree 3, and so on. Note that the odd-numbered polynomials are both negative and positive, while the even-numbered polynomials are strictly 0 or larger.

We've shown just polynomials built from x, but we can also build versions of y multiplied by itself any number of times. And we can also mix x and y together.

For example, the three possible second-degree polynomials are $x \times x$, and $y \times y$, and $x \times y$. When we get up to the third degree we have four types of expressions: $x \times x \times x$, and $x \times x \times y$, and $x \times y \times y$, and $y \times y \times y$. There are only four combinations because the order in which we multiply the values doesn't matter to the end result.

As we mentioned, when the Ridge classifier only gets the polynomial of degree 1 for both x and y (that is, we give it just the values of x and y

themselves), the most complex shape it can make is a straight line. If we also give it the polynomials of degree 2 (as we saw, these are $x \times x$, and $y \times y$, and $x \times y$), then it can combine these to make a more interesting curve. The higher the degree of the polynomials we give to the Ridge algorithm, the more complex a curve it can create.

Figure B1-29 shows just a few combinations of the curves in Figure B1-28 (these curves contain only polynomials of x). This shows that by adding up several of these curves, each with its own strength, we can make some pretty complicated shapes. At each point we found the values of the 5 curves, multiplied each value by a scaling factor for that curve, and added up the results.

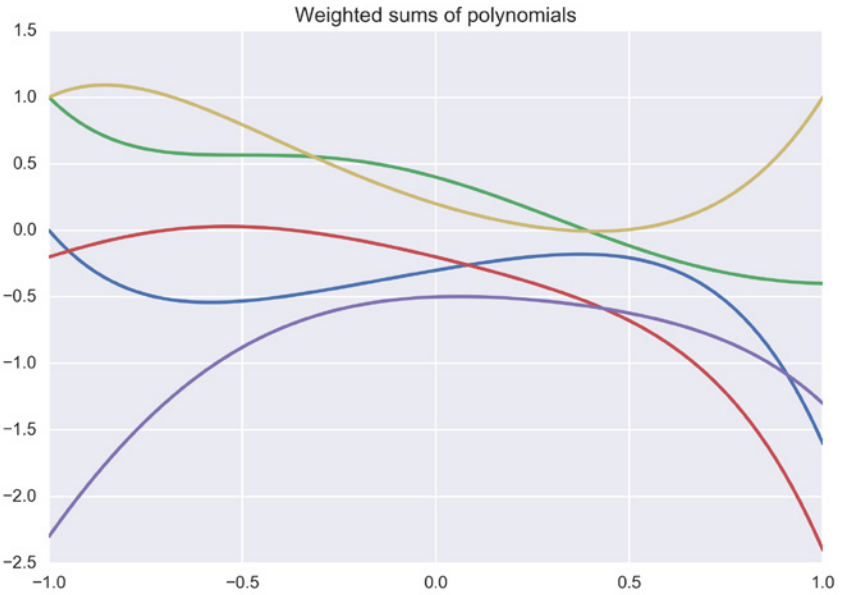


Figure B1-29: Some weighted combinations of the polynomials of x in Figure B1-28

Figure B1-29 shows curves using just the x value. Things get really interesting when we use polynomials in both x and y simultaneously. Without going into the details, we can follow a recipe like that in Figure B1-29 to create a wide variety of 2D curves. Figure B1-30 shows a few examples.

If we give `RidgeClassifier` not just the x and y values, but these polynomials that come from multiplying those values together in different ways, it can create these kinds of curves. This is much better than a straight line!

We say that by creating these additional features, we're *extending* or *augmenting* our original list of features with new *polynomial features* made from the original values of x and y. In our NumPy array of data, it just means that each row gets longer, from having only 2 features to having more of them, computed by multiplying various combinations of x and y.

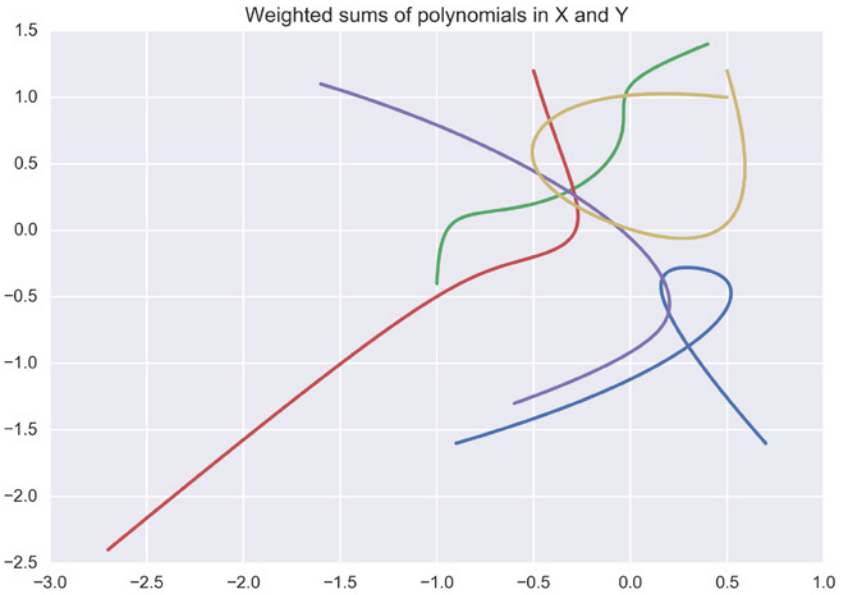


Figure B1-30: Some weighted combinations of polynomials in both X and Y

The first step of our pipeline will be to build these new polynomial features for every sample, so the Ridge classifier will have them to work with. Now we're ready to identify the new object that will do the job, named `PolynomialFeatures`. We tell it the degree of the features we want it to make, and it cranks them out and appends them to each sample. The more polynomials we include with each sample (that is, the higher their degree), the more complicated Ridge's boundary curve can become.

Wait a second. When we create these polynomial features, we're not including any new information to our data set. We're just using what we have and multiplying the values together. How does that give us curves rather than straight lines?

The answer comes down to the details of how these algorithms are implemented. It's true that we're not providing any new information. But `RidgeClassifier` is designed to work with the data it gets, not all possible variations on that data that might help it produce a better answer. So if we give `RidgeClassifier` our simple 2-feature data, it finds a line. By providing it with the polynomials, it uses those as part of its calculations to find a curve.

Conceptually, we could have an argument to `RidgeClassifier` that we could use to tell it to create this extra data all by itself, internally. But scikit-learn is instead set up to make it our responsibility to create that extra data. That way we have complete control over just what we give to `RidgeClassifier`, and thereby control the complexity and shape of the boundaries it can find.

But how much more data is best? As we saw in Chapter 9, at a certain point these increasingly complicated curves can start to overfit the data. So we'll want to stop including new features before that happens. On the other hand, we have the regularization parameter `alpha` in the Ridge classifier

which helps us control overfitting. Maybe by increasing the value of `alpha` we can use more of these combined features, and thus use a more complicated (and perhaps better-fitting) curve.

This is getting very complicated! What's the best balance between curve complexity (given by the parameter `degree` to `PolynomialFeatures`), and regularization strength (given by the parameter `alpha` to `RidgeClassifier`)?

There's no need to try to work this out ourselves. Let's just build a two-step pipeline from these two objects and give the searcher a bunch of values to search. Then it can do the work to find which combination works best. If we use the grid searcher, it will try every combination of every parameter.

Figure B1-31 shows a grid searcher with a two-step pipeline replacing the single model we had in Figure B1-23. The first step creates a `PolynomialFeatures` object, and the second creates a `RidgeClassifier` object that uses the augmented data that comes out of the first step. This is a simplified diagram, because the fold we're using for validation is going through a transformation (marked T) that doesn't seem to be coming from anywhere. We'll return to this diagram in the next section and fill in that gap.

Each step in the pipeline can have its own set of parameters to be searched. As before, each wavy box represents a loop.

Our first step in building our pipeline will be to create our `PolynomialFeatures` object, which produces those extra combinations of *x* and *y*. The only argument we care about now for this object is `degree`, which tells it how many polynomials to build from the original features. Larger values of `degree` will generate and save more of these multiplied-together features, for every sample, which will let the Ridge classifier find more complicated boundary curves.

For the Ridge classifier, let's also search for just one parameter, the regularization strength `alpha`. We'll leave the internal algorithm at its default value (that's the string `auto`, which automatically picks the best algorithm).

Now that we know the two steps that make up our pipeline, let's write the code to create it.

Scikit-learn offers more than one way to build a pipeline. We'll take the approach that's easiest to program and use.

We begin by making the objects that will go into our pipeline. In this case, it's the `PolynomialFeatures` object and the `RidgeClassifier` object. We can make them as in Listing B1-31.

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import RidgeClassifier

pipe_polynomial_features = PolynomialFeatures()
pipe_ridge_classifier = RidgeClassifier()
```

Listing B1-31: We start making our pipeline by creating the objects that will go into it.

As before, neither of these objects has any arguments, because the searcher will fill those in for us as it searches.

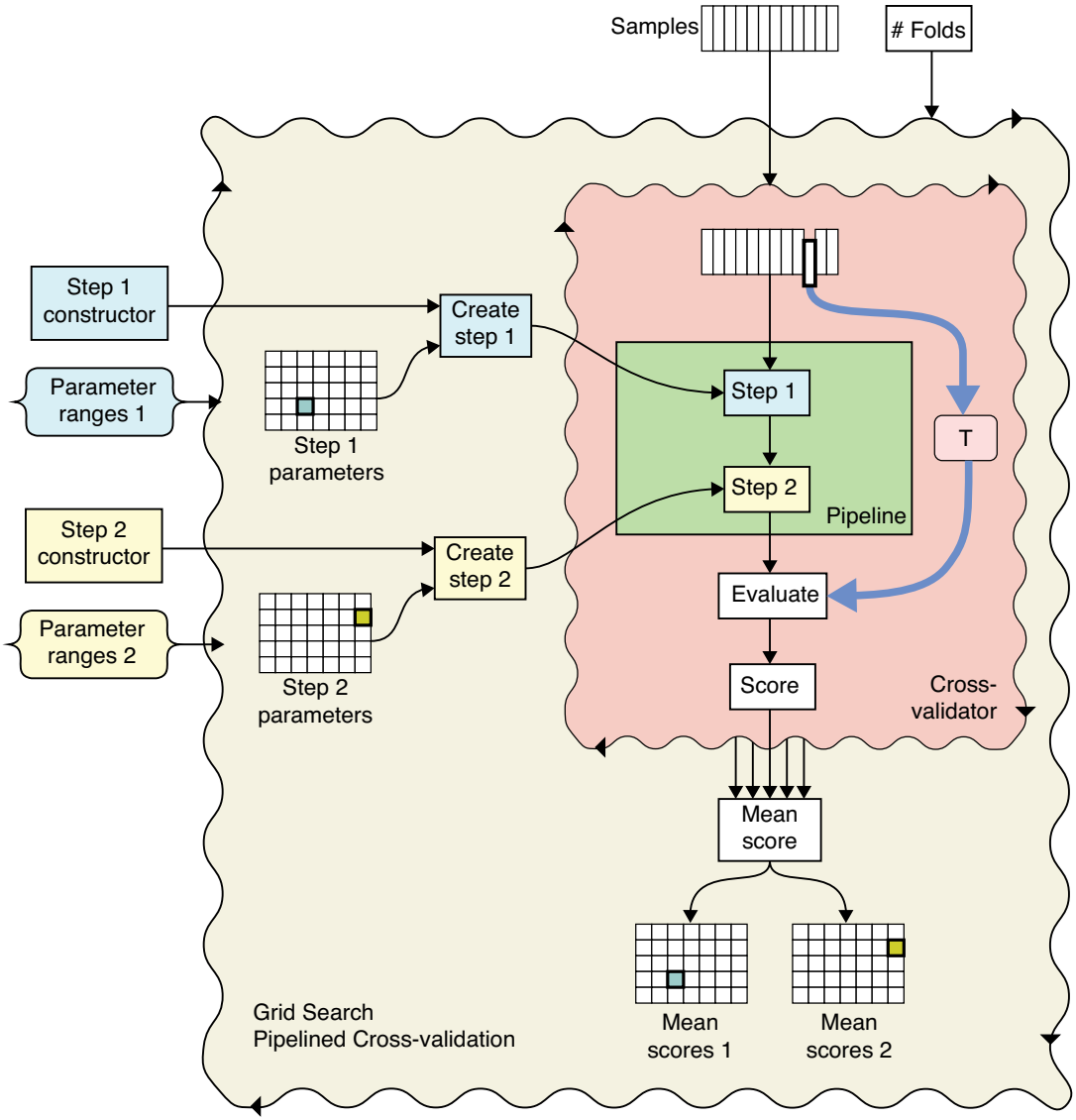


Figure B1-31: A simplified version of our grid search object with a pipeline (the innermost, green box) replacing the single model of Figure B1-23

Now that we have our objects, we can build the pipeline. To do this, we create an object named `Pipeline`, and we hand it a list that contains one entry for each step. Each entry is itself a list with two elements: a name we pick for that step, and the object that implements it.

The pipeline is built from this list of objects, *in the order in which we name them*. The list can be as long as we like, as long as we have a name and object for each step.

Listing B1-32 shows this construction step for our two-step pipeline. Notice the order of the objects. We name the `PolynomialFeatures` object first because it comes first in our intended sequence of steps.

```
from sklearn.pipeline import Pipeline

pipeline = Pipeline([('poly', pipe_polynomial_features), \
                     ('ridge', pipe_ridge_classifier)])
```

Listing B1-32: To create our pipeline object, we give `Pipeline()` a list of steps. Each step is itself a list, containing a name we give to that step, and the object that performs it. Our objects are those we just created in Listing B1-31.

The names we pick when we make the pipeline are like the names we pick for variables: they can be anything we like, but it's best to pick something descriptive. We're using very short names here to conserve space.

We've completed our pipeline, and we will soon hand this to a `GridSearchCV` object as the value of its `estimator` argument.

The last thing to do is make the dictionary of parameters for the searcher to build its combinations from.

If we make our dictionary of parameters for the pipeline in just the same way as we made our dictionary before, sooner or later we'll hit a problem. For example, we know that `RidgeClassifier` takes an argument called `alpha`. What if the `PolynomialFeatures` object also took an argument called `alpha` (it doesn't, but it could)? And what if we wanted to use values (1,2,3) for the first `alpha`, and (`'dog'`, `'cat'`, `'frog'`) for the second `alpha`? We need some way to tell the grid searcher which set of values should go to which parameter in which object.

The way scikit-learn answers this question is both sensible and weird. The sensible part is that we name each parameter using the name of the pipeline object for that parameter (that's why we gave our objects names when we made the pipeline), followed by the name of the parameter. Note that we don't use the name of the object (in our example, `pipe_polynomial_features` or `pipe_ridge_classifier`), but the name of the *pipeline step* (in our example, `poly` or `ridge`).

The weird bit is that we join the pipeline object's name and the parameter's name with *two underscore characters*. That is, two _ in a row, or __. This is easy to confuse with just one underscore, which is unfortunate. But two underscores must be used.

So to refer to the `alpha` parameter for our pipeline step named `ridge`, we'd name it in the dictionary as `ridge__alpha`, with two underscores. Similarly, the `degree` parameter for our `poly` pipeline step is named `poly__degree`, again using two underscores.

We always create our dictionary names by assembling the pipeline step name, two underscores, and the parameter name, even when there's no chance of confusion.

Listing B1-33 shows a dictionary for those two parameters, along with some values we'll try for them.

```
pipe_parameter_dictionary = {
    'poly__degree': (0, 1, 2, 3, 4, 5, 6),
    'ridge__alpha': (0.25, 0.5, 1, 2 )
}
```

Listing B1-33: A dictionary for our pipeline. Note that each parameter is given by the name of the pipeline step and the name of the parameter, joined with two underscore characters.

This will cause the loop to run 7 × 4 = 28 times, but for this small dataset that takes only a few seconds on 2014-era iMac, without even touching the GPU.

Now we're set. We just build our search object as before, and then call its `fit()` routine. Listing B1-34 shows the code.

```
pipe_searcher = GridSearchCV(estimator=pipeline,
                             param_grid=pipe_parameter_dictionary,
                             cv=num_folds)

pipe_searcher.fit(training_samples, training_labels)
```

Listing B1-34: Building our grid search object for our pipeline, and then executing the search.

The results for our half-moon data are shown in Figure B1-32. The best combination the system found was `alpha=0.25` and `degree=5`. Thanks to the additional features we provided to `RidgeClassifier`, it was able to find a curve to split the data.

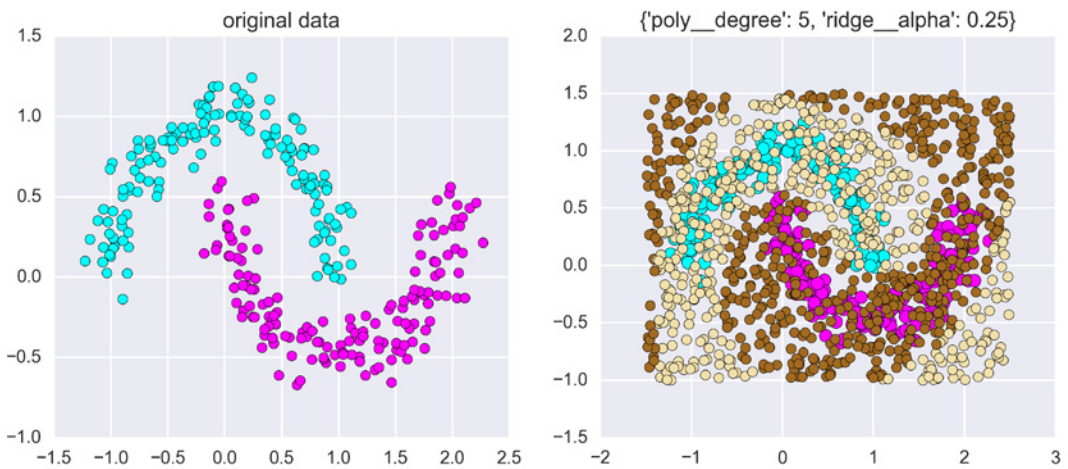


Figure B1-32: Results for searching over our more flexible pipeline object

For clarity, Figure B1-33 shows just the test data by itself.

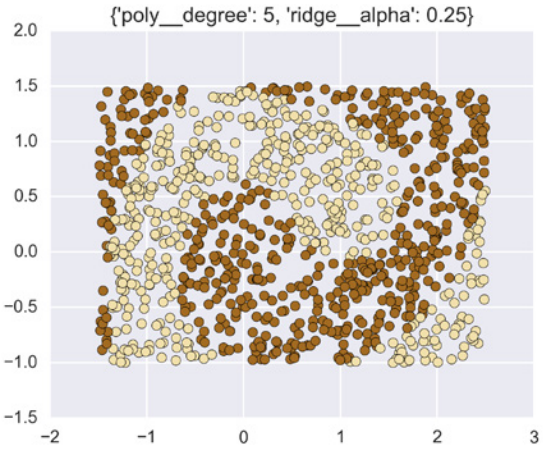


Figure B1-33: Just the test data from Figure B1-25

It's interesting to note that the boundaries are rather symmetrical, which makes sense given the symmetry in the input. We also see them seeming to curve around and re-appear in the corners. This is kind of wild, but entirely permitted by our data. After all, as long as we're classifying all the data correctly (and we are), it doesn't really matter what happens elsewhere (though we probably prefer simplicity when we can get it, just on general principles [Domingos12]).

Since the dots only give an impression of the boundary, let's look at it in high resolution in Figure B1-34. Just to see what things look like away from the data, we'll also zoom out and look at the boundary curve with a larger range of values.

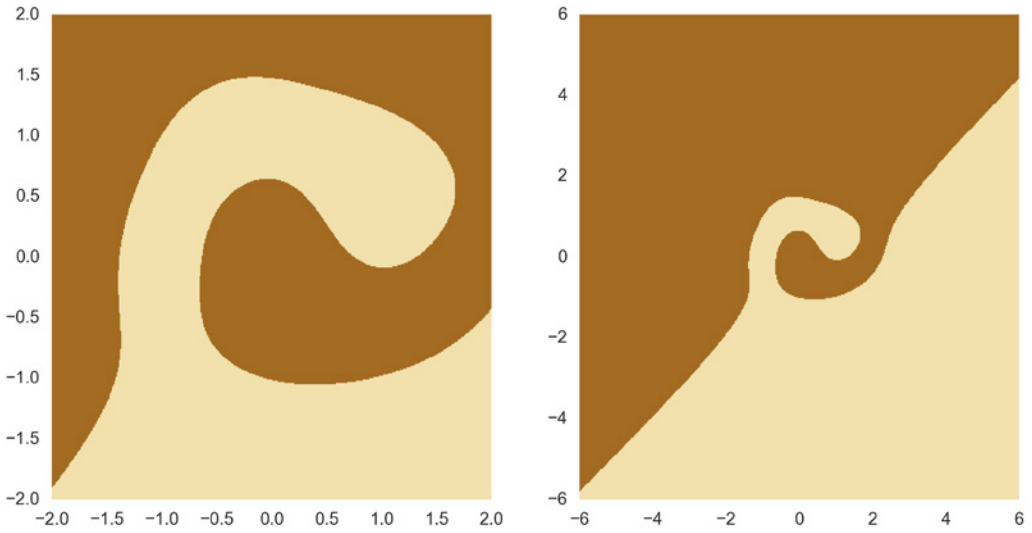


Figure B1-34: Looking at the boundary curves from Figures B1-32 and B1-33. Left: Close up to the data of the two moons, but with a slightly larger range than Figure B1-33. Right: A larger plotting area.

A graph of the scores for the different combinations is shown in Figure B1-35. The best results came from setting `degree` to 5, and `alpha` to 0.25.

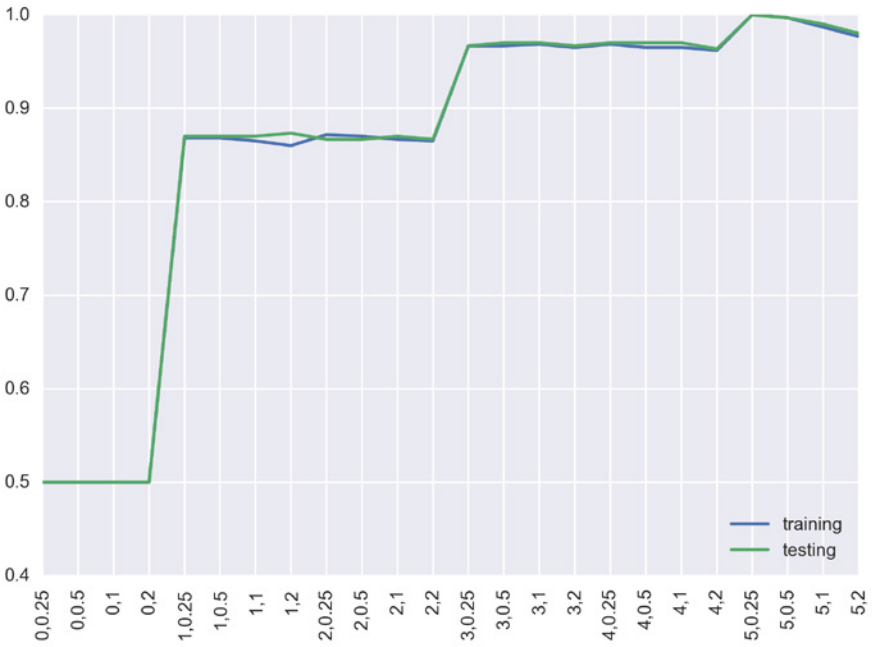


Figure B1-35: Scores from our pipeline search. The straight-line version is shown at the far left, for degree values of 0.

The best combination of `alpha` at 0.25 and `degree` at 5, located near the right end of the graph, produced a perfect training and testing score of 1.0. As we might expect, the regularization parameter didn't have much effect until the curve got complicated enough to start overfitting, and even then on this simple example it didn't make much of a difference.

Looking at the Decision Boundary

Let's take another look at the decision boundaries we've found for our half-moon data.

In Figures B1-25 and B1-26 we found a linear boundary. In those figures, we drew each dot with the color of the class returned by `predict()`. But as we mentioned before, we can get the relative confidences of the classifier from `decision_function()` and `predict_proba()`. So instead of getting just a single integer telling us which class is assigned to each input, we get back a confidence value for each category. Since we have only 2 classes, this means we'll get two floating-point values, one for each category.

Let's plot the values of `decision_function()` for our straight-line fit in 3D. The left diagram in Figure B1-36 shows the result.

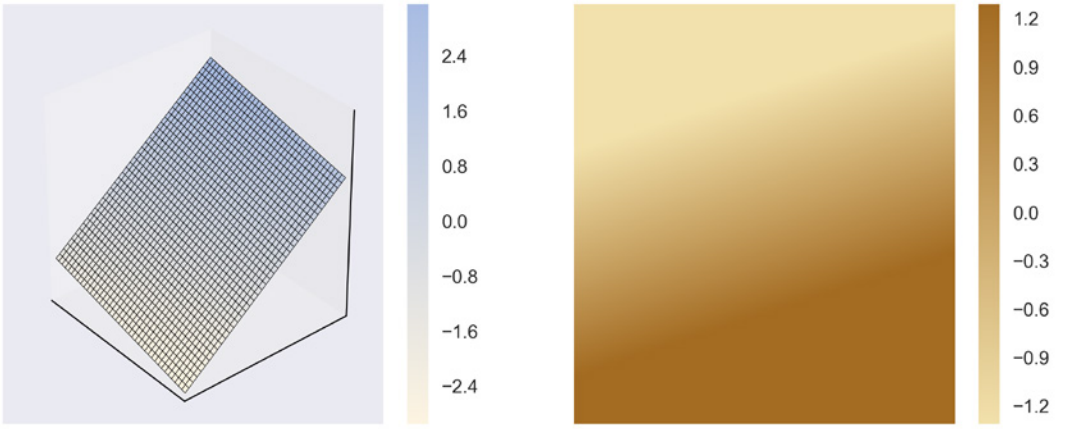


Figure B1-36: Looking at the confidence values for the linear boundary between our two half-moons, as shown in Figures B1-25 and B1-26. Left: The output of `decision_function()`. Right: A top-down view of the 3D plot.

When we look at the confidence of each category, rather than asking for just the most likely one, we can see that there's a transition from one to the other. As we might expect, the boundary from one class to the other isn't instantaneous. This surface is really a single flat plane with no creases.

Looking down on this plot, as in the right of Figure B1-36, we can see a crossover zone where the probabilities smoothly change from 1 to 0 or vice-versa.

Let's look at the values from `decision_function()` for our curved boundary in Figure B1-33 in the same way. We'll plot these in 3D and then look down on the plot, as in Figure B1-37. The soft regions of changing color show where the confidence values are changing.

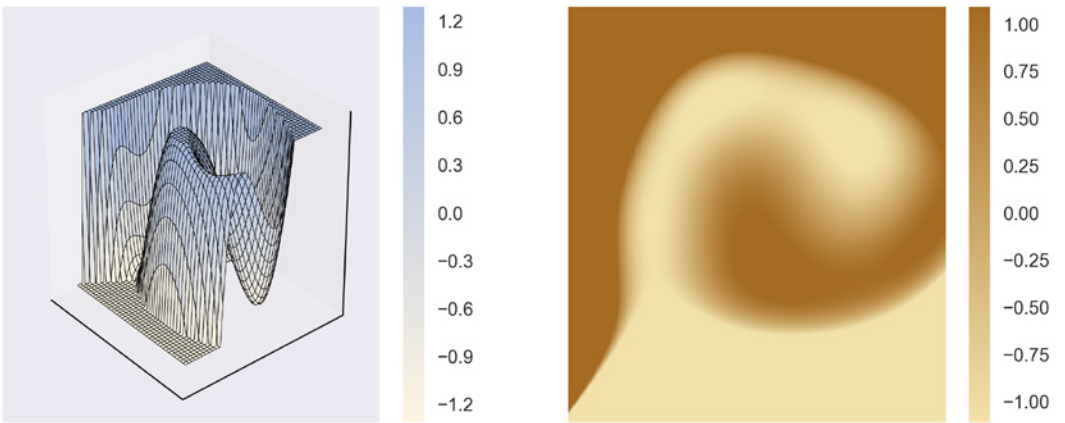


Figure B1-37: The curved boundary in 3D, and viewed from the top. The flat regions are because we've clamped the values to [–1, 1] to better see the structure near the middle. Left: A 3D view. Notice how smooth the surfaces are. Right: A top-down view of the 3D plot.

We can see that there's a smooth transition zone between the two categories (we manually clamped the output of `decision_function()` to [–1, 1] so we could focus on what's happening in that range). In the cross-over regions, the probability of a point being in one class or the other varies smoothly.

In some circumstances, getting the most likely class from `predict()` is just what we want. In others, getting the more refined confidences from `decision_function()` (or `predict_proba()`) can be more useful.

Applying Pipelined Transformations

We promised to return to a gap in Figure B1-31, where we showed the fold going through a transformation, but not where the transformation came from. Let's address that now. This will also pay off our earlier promise to show how to use pipelines to correctly apply a transformation while doing cross-validation.

Let's continue with our previous example of using `PolynomialFeatures` followed by `RidgeClassifier`. The first step in our pipeline adds new polynomials to each sample, so it counts as a *transformation* of our training data, since it changes the samples that go into the classifier. In this example, `PolynomialFeatures` is including new features, rather than changing the features themselves like a scaling transform would. But the samples are changing, so we call it a transformation.

Remember our cardinal rule about transformations: anything we do to the samples in the training data must also be done to all other data.

Since the transformation of our training data (adding new features) is happening inside the pipeline, we have to pull that transformation out so we can apply it to the validation fold. Figure B1-38 shows a close-up of the pipeline in the cross-validation step, where step 1 (our `PolynomialFeatures` object) computes a transformation, and then that transformed data is applied to both the training data used by step 2, and the validation fold used to evaluate the result.

There's no leaked information here, because we're doing everything by the book. For each model, we extract a fold, compute the transformation on the remaining training data, and then apply that transformation to both the training data and the data in the fold.

This application of the transformation is carried out for us automatically by both the regular and random grid search objects, so we don't have to lift a finger, or change our code from the previous section in any way. In other words, scikit-learn applies the steps in the pipeline in just the right way, automatically.

We discussed this detail because it's a great illustration of how we have to always think about information leakage any time we touch our sample data. It also shows us a nice way to handle this issue. And finally, this example shows how our libraries make our lives easier by handling these issues for us.

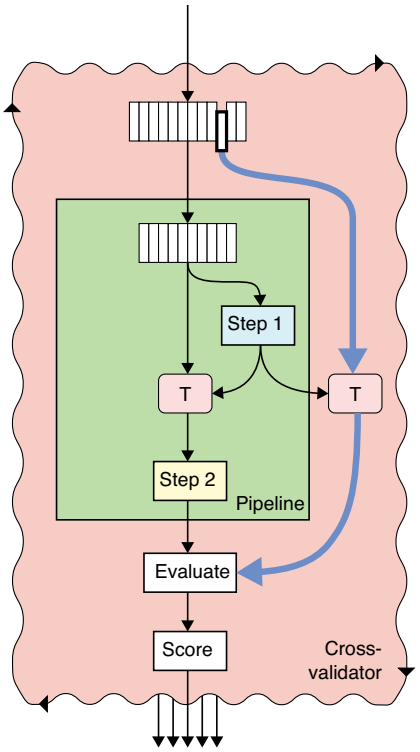


Figure B1-38: A close-up of the cross-validation step in Figure B1-31, filling in the missing source of the transformation to the fold.

We can use multiple transformations in our pipeline. For instance, we might add a `MinMaxScaler` after the `PolynomialFeatures` object. We might also then include a PCA object to reduce the dimensionality of the data. We can have as many of these data-transforming steps as we like, and they will all get applied in sequence to both the training and fold data.

Datasets

See the Jupyter notebook Bonus01-Scikit-Learn-8-Datasets.ipynb.

Scikit-learn provides a variety of datasets that are clean and ready for immediate use. Some of these are *real-world* data representing actual field studies, and some are *synthetic* data that is generated procedurally when we ask for it.

For example, we can easily import the famous *Iris* dataset, which describes the lengths of different petals of several species of iris flower. This is a classic database that's used in many discussions of categorization.

Scikit-learn also includes the *Boston housing* dataset that records the prices of homes in the Boston area for a period of years, along with other geographical information. This is often used in discussions of regression.

There are other famous datasets as well. For example, the *20newsgroups* dataset provides text data from online discussions, which is frequently used for text-based learners. The *digits* dataset contains small grayscale images of handwritten digits from 0 to 9. And the *Labeled Faces in the Wild*, or *LFW*, dataset provides labeled photographs of people that are useful for face detection and classifying images.

Most of these datasets are returned in NumPy arrays that are ready for immediate use, but always check the documentation to learn of any idiosyncrasies or exceptions.

To load a dataset, we usually need only to call its loader and save that routine's output in a variable. For example, we can load the Boston data as in Listing B1-35.

```
from sklearn import datasets

house_data = datasets.load_boston()
```

Listing B1-35: Loading the Boston dataset

The arguments for synthetic datasets are important because they help us control the size and shape of the data that's made.

Some of the most popular synthetic dataset creators are `make_moons()` which generates the two interlocking arcs we used earlier, `make_circles()` which creates a pair of nested circles, and `make_blobs()` which makes sets of points drawn from Gaussian distributions.

As an example, Listing B1-36 shows how we call `make_moons()`. We tell it how many points to make, and we can optionally add a `noise` parameter to break up the uniformity.

```
from sklearn.datasets import make_moons

(moons_xy, moons_labels) = make_moons(n_samples=800, noise=.08)
```

Listing B1-36: Using the `make_moons()` routine to generate 800 points of synthetic data.

Figure B1-39 shows these three types of synthetic data, using their defaults and with noise added. Note that each routine provides labels along with the point locations, so the data is ready for classification.

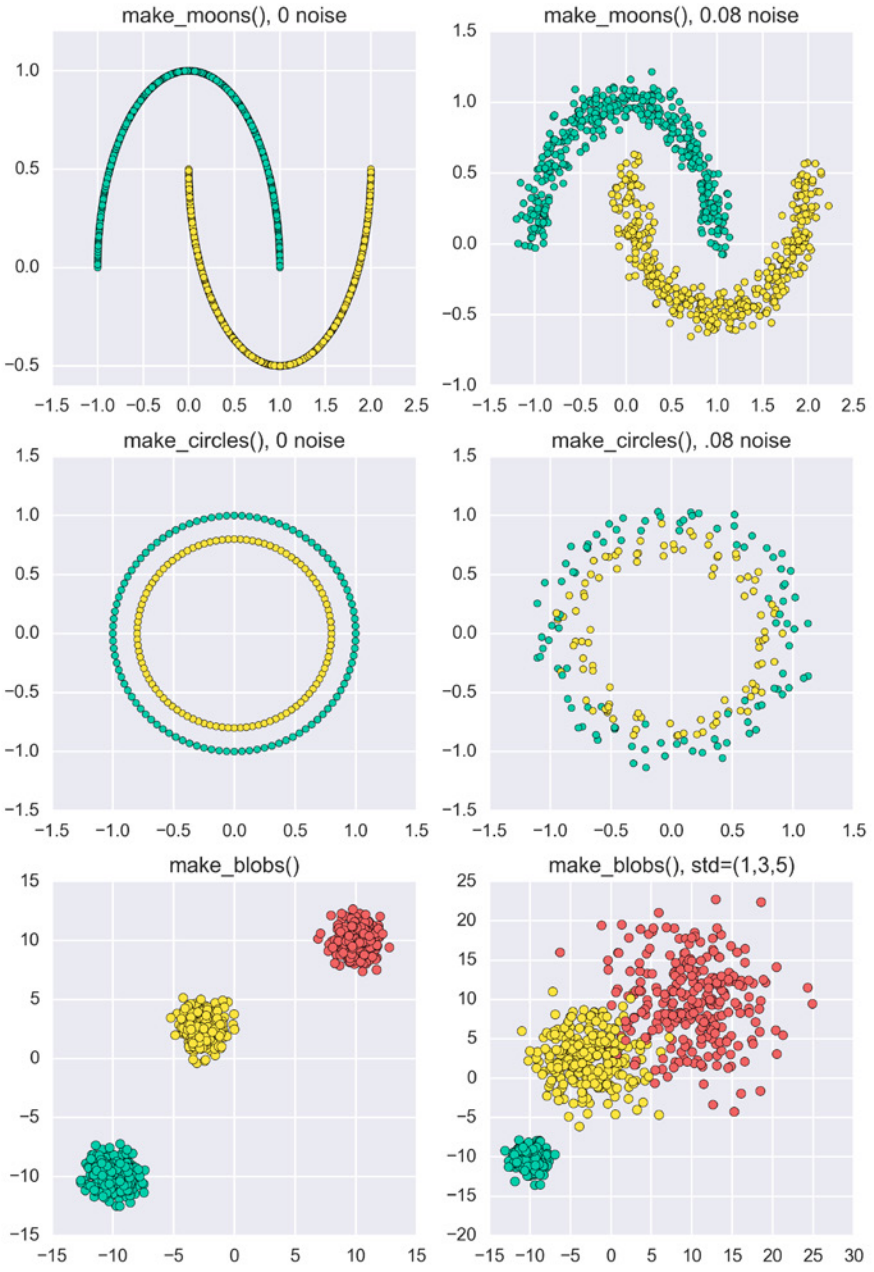


Figure B1-39: Synthetic datasets provided by scikit-learn. Top row: `make_moons` with 800 points, with no noise and with the `noise` parameter set to 0.08. Middle row: `make_circles` with 200 points, with no noise and with the `noise` parameter set to 0.08. Bottom row: `make_blobs()` with 800 points and 3 blobs, with the default standard deviation (all 1's) and with larger standard deviations to spread out the points more.

Utilities

See the Jupyter notebook Bonus01-Scikit-Learn-Utilities.ipynb.

Like any library, scikit-learn has its share of utility functions that we can collect together into a kind of grab-bag, or miscellaneous, category.

Perhaps one of the most popular of these is used to split a database into different pieces. The routine `train_test_split()` does just as it says: given a database, it splits it into two pieces that are typically used as a training set and a test (or validation) set. Among other useful arguments, `test_size` is a value from 0 to 1 that specifies the percentage of the database to place into the test set. By default, this has a value of 0.25. Listing B1-37 shows how this works with the nested-circles dataset.

```
from sklearn.model_selection import train_test_split

(circle_xy, circle_labels) = make_circles(n_samples=200, noise=.08)

samples_train, samples_test, labels_train, labels_test = \
    train_test_split(circle_xy, circle_labels, test_size=0.25)
```

Listing B1-37: Using `train_test_split()` to break up a dataset of nested circles into training and testing sets.

This produces the datasets shown in Figure B1-40.

Figure B1-40: Splitting up our original data into a training and testing set with `train_test_split()`. The two sets don't have any samples in common. Left: The starting data of 200 points. Middle: The training set of 150 points. Right: The test set of 50 points.

We've split up our starting data into two new sets. Notice that the correct labels have come along with their samples, as we'd like. What's not shown in the figure is that the order of the samples has been shuffled. That is, their order in the training and testing sets are not the same order as in the starting data.

To see why the algorithm takes this step, suppose our circle samples had been generated by starting at 3 o'clock and working clockwise. Then the first 75% of the data would be the samples from 3 o'clock to 12 o'clock, and last 25% would be from 12 to 3, as in Figure B1-41. The test data is

nothing like the training data. This would make a terrible training-test split! For this figure, we generated our own data as described above, starting at 12 o'clock and working clockwise, rather than using `make_circles()`.

Figure B1-41: We generated this data starting at 3 o'clock and then proceeded clockwise. Left: The original data of 200 points. Middle: The first 150 points for the training set. Right: The final 50 points for the test set.

We wouldn't have this problem with `make_circles()`, because it generates its samples more randomly, but other programs might not be so careful, and data we get from other sources might have been put into some kind of order before it came to us. To reduce the chance of such datasets producing bad training-test splits like Figure B1-41, `train_test_split()` shuffles the samples before assigning them to the train and test sets. The hope is that this will cause each set to be representative of the totality of the starting data, as it is in Figure B1-40.

Wrapping Up

As we promised at the start, this chapter has barely hinted at the huge variety of objects and functions offered by scikit-learn.

The scikit-learn online documentation is free and always available, but it's typically aimed at the working programmer who already understands the concepts and just needs to be reminded of syntax or argument names. There are some explanatory and tutorial articles, and some examples of use, but they, too, are often terse. For these reasons, the library's documentation is probably best used for reference, and not for explanations, though the FAQ can sometimes help clear up a question.

An easier way to dig deeper into the mechanics of this versatile library is the book by Müller and Guido [Müller-Guido16]. The book assumes familiarity with machine-learning algorithms, though there is some introductory and review material.

For help understanding a particular algorithm, or its implementations, great explanations in text or video are often just an internet search away.

A much more in-depth approach, with plenty of mathematical details, is offered in the book by Raschka [Raschka15].

References

[Domingos12]: Pedro Domingos, "A Few Useful Things to Know About Machine Learning," *Communications of the ACM*, Volume 55 Issue 10, October 2012. *https://homes.cs.washington.edu/~pedrod/papers/cacm12.pdf*

[Github14]: GitHub contributors, 2014. *https://github.com/mwaskom/seaborn/issues/229/*

[Jupyter17]: The Jupyter authors, "Jupyter," 2017. *http://jupyter.org/*

[Kassambara17]: Alboukadel Kassambara, "Determining the Optimal Number of Clusters: 3 Must Know Methods," STHDA blog [Statistical Tools for High-Throughput Data Analysis], 2017. *http://www.sthda.com/english/articles/29-cluster-validation-essentials/96-determining-the-optimal-number-of-clusters-3-must-know-methods/*

[Müller-Guido16]: Andreas C. Müller and Sarah Guido, "Introduction to Machine Learning with Python," O'Reilly Media, 2016.

[Raschka15]: Sebastian Raschka, "Python Machine Learning," Packt Publishing, 2015.

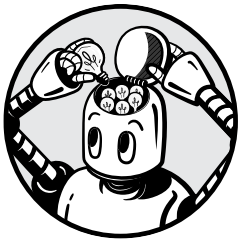
[scikit-learn20]: Scikit-learn authors, "Machine Learning in Python," version 0.24. *https://scikit-learn.org/stable/index.html*

[VanderPlas16]: Jake VanderPlas, "Python Data Science Handbook," O'Reilly Media, 2016.

[Waskom17]: Michael Waskom, "seaborn: statistical data visualization," Seaborn website, 2017. *https://seaborn.pydata.org/*

B2

KERAS PART 1



This chapter is a bonus chapter for my book *Deep Learning: A Visual Approach*. You can order the book from No Starch Press at *https://nostarch.com/deep-learning-visual-approach/*. The official version of this chapter can be found for free on my GitHub at *https://github.com/blueberrymusic/* (look for the repository "Deep-Learning-A-Visual-Approach"). All of the figures in this chapter, and all of the notebooks with complete, running implementations of the code discussed here, can also be found for free on the book's GitHub repository.

In this bonus chapter, we'll look at how to take many of the ideas in the book and actually implement them. That is, we'll build, train, and run deep learning systems.

In Bonus Chapter 3, we'll expand on these ideas to build more complex models.

There are many fine deep-learning libraries out there, and each has its advantages. Rather than try to cover many libraries, we'll focus on just one,

called *Keras*. This is a library deliberately designed to keep things simple, so it's a great start into building and training models.

Keras is part of the TensorFlow system. This means we're insulated from the details of that library while still enjoying the advantages of its highly-developed and efficient code.

Once you feel comfortable in Keras, you may want to stretch out into a more powerful and general-purpose library. Currently, the two most popular options are TensorFlow and PyTorch. They have many similarities, but also offer quite different experiences and supporting software. I suggest looking at both and choosing the one that best seems to fit your style. Another way to choose is to consider what project you'd like to start with, and then searching for GitHub repos that contain implementations that come close to your goals. Then you can use whichever library they're using, and modify their code for your own purposes.

Don't worry about which library you choose initially, because once you know one deep learning library, it isn't too hard to any of the others.

One of the nice things about working with Keras is that a typical session of building and training a machine-learning system requires very little routine Python programming. The actual deep learning code is often the easiest part of the program: we build the network with just a few lines, and train it with just one or two function calls. Most of the rest of the program is made of supporting tasks, such as getting the input data, cleaning it, structuring it for use in the network, writing routines for saving data and visualizing results, and so on.

In this chapter we'll start with simple networks, turn them into deep networks, and then continue into deep convolutional networks, and recurrent networks.

To keep things focused we'll stick to only the Keras routines and arguments we need. We'll need to discuss some principles and ideas along the way, but that will be kept to a minimum.

When we work with real code, we always have to deal with things like processing our data, manipulating it in various ways, setting up helper routines, and so on. As in the other bonus chapters, we leave this out of the text, referring the interested reader to the complete implementations in the associated Jupyter notebooks.

The Structure of This Chapter

This chapter is not a straight line.

Our goal in this chapter is give you the tools to design, build, train, and use a variety of deep learning networks. We will never lose sight of that purpose. But to get there, we will have to periodically stop and cover essential groundwork. That will often happen at the start of sections. It may sometimes feel like we take two steps forward and one step back. But that's just because we need to pause to lock in a new idea. The payoff will be all the sweeter because we will then see how that idea helps us build a working system.

Libraries, Programming, and Debugging

Before we dig in, it's fair to wonder why we're using a library at all? Surely it would be more educational to write all of our own code from scratch, implementing all the algorithms in this book on our own. That process would force us to learn essential details that we could otherwise overlook. That argument has a lot of merit. For in-depth understanding, writing our own implementations (even if they're just for toy networks) can't be beat.

But when it comes to actually building, training, and running deep networks, libraries are almost always the way to go whenever possible. To rival what today's libraries offer immediately, and for free, we'd have to spend enormous time and effort on issues like numerical stability, optimization, GPU programming, multi-threading, and much more. Though many of these are not essential to getting a toy system to function, when we start processing large amounts of data they become necessary to get good results in practical amounts of time.

As an analogy, consider that most people today use a high level language. In this chapter we'll be using Python. But Python is not executed directly by the computer. Any high-level language is ultimately turned into assembly code, which is the language of the CPU. Maybe we should be programming in assembly. But why draw the line there? Assembly code is just a way of controlling the low-level hardware of the processor, manipulating individual circuit hardware elements using a processor-specific language called machine code. Maybe we should program in machine code.

Of course, we don't use machine code because it would take us forever to write anything substantial. The value of working at higher and higher levels of abstraction is that we're freed up to think in more abstract terms, and we can spend our time working on how to structure a solution to our problem than on the mechanics of controlling the computer. For same reason, using a library like Keras lets us think abstractly in terms of deep learning ideas, without getting bogged down in the mechanics of their implementations.

Researchers are frequently publishing new and cleverer ways to teach deep networks with greater speed and efficiency. These approaches often require some custom programming, special architectures, new training methodologies, or all of these. In this book we'll stay focused on the basics. A strong foundation lets us more easily understand and implement more complex techniques, since they're usually built on their predecessors.

The word *model* deserves some special attention, because it's used by different authors and programmers to mean different things.

The Keras documentation in particular uses *model* in three ways. First, it refers to the architecture of a deep learning system. Second, it describes the combination of that architecture and the weights that it learns as a result of training. Third, it can refer to the set of library calls that we use to construct our system, also called an API (Application Program Interface). For brevity, and to match the Keras documentation, we'll use the word "model" in the same three ways. We will try to make the meaning clear from context.

Versions and Programming Style

Like the scikit-learn library we saw in Bonus Chapter 1, Keras is Python based.

A note on versions. In 2008, the Python language made a jump from version 2.7 to version 3. which is now the standard. We'll be using Python 3.7.6 in this chapter, but any release version of Python 3 will be compatible with our code. Happily, most of this code will run fine on Python 2.7 installations. The most common difference is merely that in Python 3, when we use `print`, we place the argument in parentheses, e.g., `print ('Hello')`, while in 2.7 we don't use parentheses for printing.

Just as Python receives updates, so too does the Keras library. In 2017, the Keras library went from version 1 to version 2. Many things remained the same, but there were changes.

One unfortunate consequence of this evolution of the library is that much of the example code that you can find online will not run under Keras 2. Sometimes it's easy to fix the problem, but sometimes it seems impossible.

For example, earlier versions of Keras (and programs written for it) referred to measurements of accuracy with the string `'acc'`. In Keras 2, that has become `'accuracy'`. It's a small change, but if you use the wrong string your program will crash! Usually the error message gives you some kind of hint that involves the string in either form.

If you find yourself unable to coerce a piece of Keras 1 code to run, it's often best to look for something more recent, or just dig into the documentation and write your own version from scratch, using the current rules. In this chapter we use Keras version 2.4.0, released in June 2020.

Python's own libraries are all managed independently, and they are changed and updated when the volunteers in charge of them feel it's appropriate. All of the notebooks in this chapter and the GitHub repo run without errors as of April 2021. They were tested with Python 3.7.6, using the latest versions of all libraries involved. Some of the most frequently-used libraries (beyond built-in libraries such as math and os) include: Keras 2.4.0, TensorFlow 2.4.1, NumPy 1.19.2, matplotlib 3.4.1, scikit-learn 0.24.2, scikit-image 0.18.1, and SciPy 1.6.3.

Python is a powerful language that has a lot of clever tricks up its sleeve. There are frequently tradeoffs between writing code that is clear, and code that is compact or efficient. Python code can also build on the more than 60,000 libraries that can be installed for the language [Ramalho16]. But this is not a book about Python, or how to write the shortest or fastest code.

For these discussions and their associated notebooks, I have preferred clarity and simplicity over compactness and even elegance. My goal was to offer code that can be understood, so I've used variable and function names that are longer than one might use in practice, and I've written out some expressions on multiple lines, even if they could be combined into a single step. I'll even sometimes use parentheses that are not strictly necessary, if they make it easier to visually grasp what a line of code is accomplishing.

As we build up our programs, we'll typically present small pieces of code one at a time. The idea is that the full program will be built by combining these pieces, usually just by entering them one after the next. By presenting the code in small pieces, it makes them easier to read and discuss.

Many programs need Python `import` statements to bring in libraries, such as NumPy or Keras itself. Our convention will be to include the `import` statement the first time we present a listing that includes a function that needs it, but to avoid repeating big blocks of boring `import` statements we won't repeat them in subsequent examples. Happily, there's no penalty for importing modules we don't need, or even importing the same module more than once. When developing a piece of code, we could simply copy and paste a chunk of text that imports every library that we commonly use. When we're done developing the code and we're cleaning it up, we can prune away any unnecessary or redundant `import` statements.

Python Programming and Debugging

Though this chapter presents a lot of code, we need to remember that this is like an art book showing final paintings, or an architecture book showing constructed buildings. Almost nothing starts out clean and nice. The code examples in this chapter were developed, one line at a time, debugged, improved, changed, debugged again, and so on.

Although the final results may appear simple and straightforward, they usually took a twisty and often error-producing path to get to that point. The code you see in this book was messy and ugly when I was developing it, and then once it was working I cut away the stuff that wasn't needed and cleaned up what was left. We should always expect to have to go through a similar process of incremental development with all programming, particularly when learning a new library such as Keras.

This process is much easier in Python than in many other languages because Python can be programmed interactively. That is, we don't have to write our program in a text editor, save it, compile it, then run it. We *can* do this if we want. But we can also choose to type our code one line at a time into an interpreter, getting immediate results. This greatly encourages and rewards experimentation.

The *Jupyter* system provides a very nice browser-based interactive system that is ideal for this kind of experimentation [Jupyter16]. One great thing about running Python in a browser is that we can have multiple, independent tabs open at once. We can use one tab as our main development environment, another for experiments, another for test runs, and so on. And there are lots of useful shortcuts that save time [Devlin16].

A great way to use Jupyter is to grow code one line or statement at a time. We can try lots of little experiments, checking everything along the way until we're convinced that we have all the details right. Then we can even wrap up that code with a function definition.

Debugging can be a challenge when using Keras, because the errors are often inscrutable. Keras assumes for the most part that we know what we're doing, and it doesn't do a ton of error checking on our code. When things

do go wrong, we often learn about it because some low-level routine that we've never heard of finds that it can't do its job. Understanding what went wrong in that routine is usually far from obvious. Having all the source code of Keras available can help, but debugging our code by reading through the library source requires a serious commitment of time and study.

An easier approach is to find the call we're making that triggers the problem, and then temporarily simplify it as much as possible until the problem goes away. If that fails, we can replace the call with a snippet of code from one of our other projects, or even an online example. Then we can transform the working code into our own code one step at a time, so we can discover just which step causes it to fail.

Some of this debugging can be done with little experiments in Jupyter. But other times we want to use a deeper and more fully-functioned modern debugger, equipped with features like breakpoints and single-step execution. We can find those tools in the development environment offered by PyCharm in the free *PyCharm Community Edition IDE* [JetBrains17]. Here one can do modern debugging like setting breakpoints, examining variables, and looking at a call stack.

Copying code back and forth between the two environments can be a bit of a hassle, but it's worth it to take advantage of Jupyter's immediate evaluation and feedback, and PyCharm's robust debugging tools.

In addition to Jupyter and PyCharm, there are many other Python development tools and environments to choose from. We used Jupyter and PyCharm for this book, but it's well worth the time to explore the alternatives out there and find the tools that best suit your style.

Running Externally

Running code on your own computer is pretty great. You can control everything, and save it all to your own hard drive. But when programs get big, they might start demanding more compute power or memory than your computer can easily provide.

Happily, there are free and paid online services that will let you run Jupyter notebooks on their big and powerful computers. Perhaps the two best-known free services as of early 2021 are Kaggle [Kaggle21] and Colab [Colab21]. They both let you run your own code and save the results. If your computer isn't giving you results as quickly as you'd like, these services are worth looking into. If you need even more power, the paid services offer you bigger and faster computers, more memory, and other amenities.

A Workaround Note

As of April 2021, there seems to be an issue with running Keras in Jupyter notebooks (which we use here) on at least some Mac computers. Programs run briefly, then seemingly out of nowhere the Python kernel exits (or dies), which means everything stops. This is pretty lousy.

There is a fix, which the documentation describes as "dangerous," but it's worked for me. I've included that fix in a cell containing two lines of

code in each notebook that uses Keras, right after the `import` statements. If you're not on a Mac, or not having this problem, you can ignore or delete that cell.

Here's that workaround. The first line is just an `import` statement.

```
import os
os.environ['KMP_DUPLICATE_LIB_OK']='True'
```

Overview

Keras is a library for creating, training, and using deep-learning networks [Chollet17b]. It's written in Python, so it's compatible with the scikit-learn library we saw in Bonus Chapter 1. In fact, it is deliberately intended to work alongside scikit-learn, and we'll be using both libraries freely in this chapter.

Keras makes it easy to create a deep-learning network by simply building up a stack of parameterized layers. This freedom of assembly is both a blessing and a curse.

We can make an analogy to most written languages. In English, we can build up a sentence by placing together words left to right in a sequence. As long as we follow the rules of sentence construction, we can choose our words blindly, and they will always form a valid English sentence. For instance, "Shoes and grapes sang clumsy windows" is a valid English sentence, but it's meaningless. Perhaps the most famous meaningless sentence is "Colorless green ideas sleep furiously" [Chomsky57]. In fact, if we just cobble together structurally sound sentences, the vast majority will be meaningless. Meaningful sentences are rare.

In the same way, we can assemble all kinds of deep-learning networks easily with Keras. But if we want a network that makes sense, meaning that it can learn from examples and make good predictions, we need to choose our layers, and their parameters, with care. Each layer has to make sense in the local context of the layers immediately preceding and following it, as well as the larger context of all the other layers in the network.

Much of the discussion in this chapter is to provide enough understanding of what's going on so that we can avoid the frustration of making the equivalent of "Pencils stumbling over burps never cook cooks." It's structurally correct, but it doesn't actually work as a sentence. The more we know about what Keras is doing, the better we'll be able to avoid building such oddities in the first place, and the better-equipped we'll be to fix them when we inevitably make them anyway.

So in this chapter and the next, we're going to carefully explain each step. The goal is that by the end of these chapters, you'll understand all the design decisions and choices, so you can design and implement new deep-learning networks with confidence.

Tensors and Arrays

We'll be working with data structures that have different numbers of dimensions, and we often give them distinctive names. For instance, we usually call a 1-dimensional list just a *list*, a 2-dimensional arrangement a *grid*, and a 3-dimensional arrangement a *block* or *volume*. In machine learning, grids and blocks must be *complete*. That is, there can be no pieces sticking out, and no holes. Each side is flat and every cell is filled in.

All of these arrangements belong to the category of *tensors*. In fact, a tensor can have any number of dimensions.

To mathematicians and physicists, the word "tensor" refers to a much more general idea. The machine learning version of a tensor isn't technically incompatible with the mathematical definition, but they are different. This is rarely a problem, but it's something to keep an eye open for when reading papers on machine learning that have a lot of physics in them, or vice-versa.

NumPy also works with tensors, but the NumPy documentation usually calls them *arrays*. Although to many programmers an "array" is a 1D list, remember that in NumPy, the word refers to a tensor that may have many dimensions.

Setting Up Keras

To install the latest version of Keras, just install TensorFlow as you would for any other Python library on your system. Keras will come along for the ride.

In this chapter (and its associated notebooks) we use Keras version 2.4.0. Earlier versions of Keras allowed us to actually run our networks using our choice of three deep learning libraries: *Theano* [Theano16], *TensorFlow* [TensorFlow16], or *CNTK* [CNTK17]. Keras called these *backends*, since they are "behind" the unified Keras interface, and provided the engines that actually created and ran our networks.

The current version of Keras only supports TensorFlow, but the notion of a backend as the system that really runs the code that Keras creates still exists, and we'll occasionally use backend functions here.

Shapes of Tensors Holding Images

An issue that can't quite be swept under the rug is how our data is organized. Particularly when we work with images, there are two popular but different ways to structure the tensors that hold our data.

Keras lets us use either approach, as long as we tell it which one we've chosen. We can do this by naming our choice in the configuration file. Let's look at this choice, and how we identify it.

Consider a single, RGB color image. The image has a width and height. There are also three *channels*, or slices, one each for red, green, and blue. As Figure B2-1 shows, we might imagine the images stacked from front to back, or left to right.

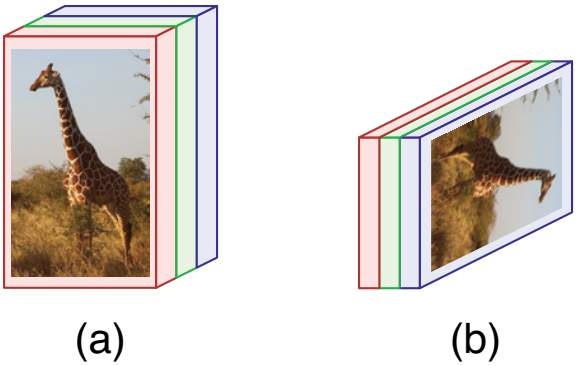


Figure B2-1: Two ways to stack images of size 100 wide by 200 high. We read the sizes of the blocks by the number of layers going away, then down, then right. (a) Stacking images from front to back. This block has dimensions 3 by 200 by 100. This is the `channels_first` *organization. (b) Stacking images from left to right. This block has dimensions 200 by 100 by 3. This is the* `channels_last` *organization.*

Suppose our image is 100 pixels wide and 200 pixels high, so in the order (rows, columns) we'd write this as (200, 100). We'll specify the dimensions of our 3D data structures in the order away, then down, then across. With this convention, Figure B2-1(a) places the number of channels first, creating a block with shape (3, 200, 100), and Figure B2-1(b) places the number of channels last, creating a block with shape (200, 100, 3).

Some libraries assume the data is in front-to-back form, and some assume it's in left-to-right form. If we don't match their assumptions, things can go very wrong. For example, if our library expects our data to be in the front-to-back order of Figure B2-1(a), but we're storing it in left-to-right order of Figure B2-1(b), the library will think that we have 200 images, each 100 pixels high by 3 pixels wide. This will not give us the results we want!

Keras hides these library-dependent preferences from us, and restructures the data as needed to make everything work. But we need to tell it which of these two approaches we're using. We do this by telling it whether our number of channels (in this case, 3) is the first dimension or the last when describing the block's size.

We usually provide this information in the Keras configuration file we mentioned above. In this text file, we identify how we're organizing our data by setting the parameter named `'image_data_format'` to either the string `'channels_first'` or `'channels_last'`.

It's always a good idea to make a backup of the configuration file before editing it. The file is plain text, so we can then open it with our favorite text editor and assign values to its variables, following the existing layout of the file. The values for most of the parameters will be strings that are named in quotes, and we need to preserve that.

As an example, Listing B2-1 shows a typical Keras configuration file. Note the opening and closing curly braces. Here we're setting

"image_data_format" to "channels_last", telling the system that our data is structured with the channels first. The other two options are untouched. These are the options we'll be using in this chapter and the next.

```
{
    "epsilon": 1e-07,
    "floatx": "float32",
    "image_data_format": "channels_last"
}
```

Listing B2-1: A typical Keras configuration file. We've set the `image_data_format` parameter. The other two are untouched.

When we import Keras into our Python code, Keras will read this configuration file.

If we're not working with images, then the setting of "image_data_format" is irrelevant.

There are two other entries in the configuration file that we haven't addressed. The parameter "epsilon" is used to control numerical calculations. Its default has been carefully chosen to match the system's internal algorithms, and it in normal use of the library it should not be changed.

The variable "floatx" tells the system what type of floating-point number it should expect the data to be stored in. This value is also rarely changed.

We can also read and write the values of these variables from our code. This way we can change them for a given program without modifying our configuration file. To access these values, we use `import` to bring in the Keras module `backend`, and then call one of the functions in Listing B2-2. Changing these defaults should be done before calling any Keras routines. The convention is to call these very soon or even immediately after any `import` statements at the start of a file.

```
from tensorflow.keras import backend as keras_backend

# read the values of epsilon, floatx, and image_data_format
ep_value = keras_backend.epsilon()
floatx_value = keras_backend.floatx()
idf = keras_backend.image_data_format()

# set the values of epsilon, floatx, and image_data_format
keras_backend.set_epsilon(0.0000001)  # rarely done
keras_backend.set_floatx('float32')   # rarely done
keras_backend.set_image_data_format('channels_last') # the important one
```

Listing B2-2: How to set Keras configuration values from code. Note that we cannot set the backend choice from code. Setting the values for "epsilon" or "floatx" is unusual, and should only be done by an expert.

Note that the first line in Listing B2-2 is an `import` statement that brings in the necessary module from Keras. If we forget this line, we'll probably get a `NameError` from Python when it runs this code.

GPUs and Other Accelerators

Many computers today come with a *Graphics Processing Unit*, or *GPU*. As the name suggests, these devices were originally designed to speed up the processing of 3D graphics typically used by games, scientific visualization, and other 3D-intensive applications. To accomplish this, the chips were designed to implement the mathematical steps commonly used to create these images. GPUs quickly became increasingly powerful, plentiful, and cheap.

In an unexpected surprise, machine-learning researchers realized that the feed-forward and backprop algorithms could be written in such a way that their mathematics looked a lot like the math that these chips were able to do so quickly, and in parallel. That is, not only could the calculation be performed faster than if it was done inside a "normal" computer, the chip could also do dozens or more of these calculations simultaneously.

The speed boost provided by using GPUs, particularly during training, had an enormous effect. Models that would have been impractical to train on a regular CPU were suddenly within reach.

But not all GPUs are the same. Different manufacturers design GPUs with different features and technologies. NVIDIA has put a lot of explicit support for machine learning into their chips, and offer great deal of support software, much of which is known collectively as CUDA [NVIDIA17]. As a result, most machine-learning libraries have targeted GPUs made by that company.

To provide an alternative, an open-source project called OpenCL is dedicated to producing a library that will enable authors to write GPU programs in such a way that they that will run on chips made by any manufacturer [Khronos21]. As of early 2021, OpenCL is in version 3.0, and can be used for some machine learning and deep learning tasks. This is a fluid situation that is changing fast. The most up to date information can be found online in blogs and discussion boards.

Another GPU alternative is the *tensor processing unit*, or *TPU* [Sato17]. This is a specialized chip designed for the kind of tensor processing needed by machine learning, and may be used instead of a GPU. As of early 2021, TPUs are rare on consumer-level hardware, though they are available through the free Google Colab system [Colab21].

Getting Started

The Keras documentation, while complete, can also be challenging. Much of it is written for experts. For example, the documentation will identify the options that are available for a given routine, but it might not describe what those options mean, the pros and cons of each, nor what criteria we should use for choosing one.

We can often fill the gap with online tutorials and examples. In extreme cases, we can dive into the publicly-accessible source code and, in

theory, work out exactly what every option does. To avoid that kind of internet microscopy and source-code spelunking, in this chapter we will motivate and explain all of our variable settings and choices.

Many Keras functions take optional arguments, some of which are broadly useful, while others are for very specific circumstances. To keep the discussion focused, we'll only talk about the functions and arguments that we use in this chapter.

Our first trek to a trained neural network will take us along three mountain tops before we get to the final peak, where we will reach our goal of a running network. We'll reach the first mountaintop when we've seen how to pre-process our data to make it ready for learning. We'll summit the second mountaintop when our network is built and ready to train. When we reach the third mountaintop we'll have seen how to train the network so it learns from our data. When we reach this final peak we'll have put it all together, taking us from an empty slate to a trained network that can make predictions on new data.

Let's climb!

Hello, World

The first program in the first book on programming in the C language demonstrated how to get the computer to print "hello, world" [Kernighan78]. Since then, printing "hello, world" has been used as the first program by innumerable books covering countless languages. The phrase "hello world program" has come to mean the first thing we learn in almost any programming language or computer system, even if it's not literally to print that phrase.

Machine learning has two "hello, world" examples that just about everyone starts with: the *iris dataset* and the *MNIST dataset*. They're both categorization problems, based on small, free data sets. Because they're so popular, Keras has special-purpose routines to let us read their data into our program with just a single line of code.

The *iris dataset* is a collection of information about 150 different iris flowers belonging to 3 different species [Wikipedia17]. Each sample contains 4 measurements, or features: the length and width of 2 different types of petals. Our job is to learn from this labeled data how to take in the 4 measurements of a new flower and predict which of the 3 types it belongs to. Listing B2-3 shows the first few rows of this data.

```
5.1, 3.5, 1.4, 0.2, Iris-setosa
4.9, 3.0, 1.4, 0.2, Iris-setosa
4.7, 3.2, 1.3, 0.2, Iris-setosa
4.6, 3.1, 1.5, 0.2, Iris-setosa
5.0, 3.6, 1.4, 0.2, Iris-setosa
```

Listing B2-3: The first few rows of the classic iris dataset. Each row holds the sepal length and width, petal length and width, and the name of the class that flower belongs to. We added some spaces for clarity.

We've seen the *MNIST dataset* in previous chapters. This is a big collection of tiny grayscale scans (28 by 28 pixels) of hand-written digits from 0 to 9 [LeCun13]. The database is separated into 60,000 images for training, and 10,000 for testing. Each image is accompanied by an integer from 0 to 9 that serves as its label, telling us what digit the image contains.

The drawings are diverse, with half coming from high school students, and half from employees at the US Census Bureau. The name MNIST stands for "modified NIST." NIST itself refers to the US National Institute of Standards (NIST), where the data originated. The modifications involved pre-processing such as cropping and scaling the images. An interesting quality of these images is that some are ambiguous, even to human observers. Figure B2-2 shows 10 randomly selected examples of each digit, chosen from the training data.

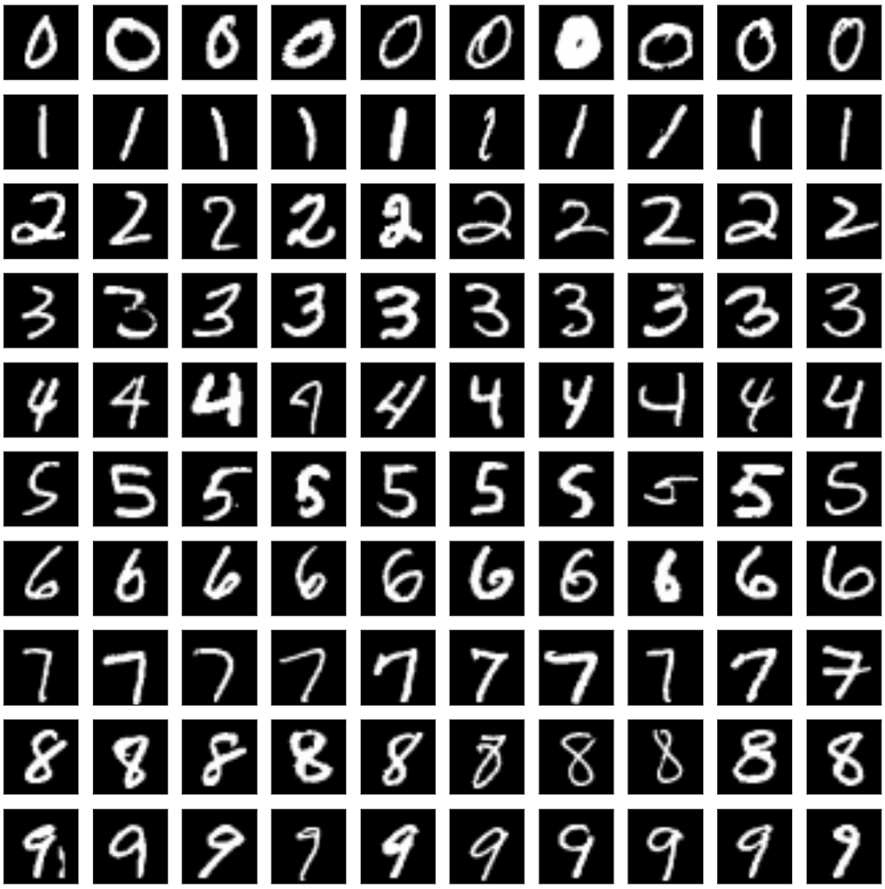


Figure B2-2: A random selection of images in the MNIST training set, organized by label

Notice the variation in thickness and style among the digits in the figure. A few details are worth noticing. The second 3 from the left almost disappears in places. The fourth 4 could be mistaken for a 9. The third 5

from the right could be called a 6 with an open loop. The rightmost 7 has a horizontal slash, which the other 7's do not share. The upper loop of several of the 8's is not closed. And the leftmost 9 has some extra artifacts.

Because the iris and MNIST datasets are the equivalent of "hello, world" for machine learning, they appear in almost every book and tutorial on the subject. This has both pros and cons.

The pros are substantial. One important advantage of using either of these well-known databases is that because so many people have studied them, they're known to be good test databases.

Another advantage of both data sets is that because they're very widely known, it's easy to find a variety of networks that people have already built and trained. The UCI Machine Learning Repository, which hosts the Iris flower dataset, calls it "...perhaps the best known database to be found in the pattern recognition literature" [UCI16]. The MNIST data is not far behind. Tables of scores for MNIST (and many other standard databases) are online, along with the architectures of the networks, so we can study and learn from them [Benenson16][LeCun13].

Another advantage of these datasets is that they have proven themselves to be excellent for developing skills in machine learning. They're small enough that our programs will run quickly, and they describe concrete, understandable phenomena. The datasets themselves are *clean*, meaning that they're free of typos, errors, and other details that can interfere with the learning process, for both humans and computers. And the MNIST database, with a total of 70,000 samples, is big enough to do some real training and experimenting.

The main downside of using these datasets is precisely that because they are so well-known, their use can become repetitive.

On balance, we feel that the risk of over-familiarity is worth the benefits of using such well-understood and useful datasets. For consistency we'll choose the MNIST dataset for our examples in this chapter.

Another substantial positive quality of the MNIST dataset is that we can draw pictures of it. Abstract data is great, but it can be challenging to interpret. Images are great because we can evaluate many things about them just by looking.

Preparing the Data

This section's notebook is Bonus02-Keras-1-Preparing-the-Data.ipynb.

According to the creators of MNIST, "The original black and white (bilevel) images from NIST were size normalized to fit in a 20x20 pixel box while preserving their aspect ratio. The resulting images contain grey levels as a result of the anti-aliasing technique used by the normalization algorithm. The images were centered in a 28x28 image..." [LeCun13]. So we know that the digits are all centered, the gray values range from black to white in each image, and looking at the data it's clear that they've all been scanned in so the digits are mostly upright. All of this makes our lives easier.

With most databases, we'd have to do this kind of pre-processing work ourselves to make our samples consistent and comparable with each other. We'd also have to weed out bad scans, correct mislabeled digits, and otherwise check and recheck (and recheck!) our database to make sure it was both complete and accurate. When all of this has been done, we say the database is *clean*. Cleaning a database can take a huge amount of time and effort, and another big advantage of using the MNIST data is that a lot of the cleaning has already been done.

We're going to go through the remaining pre-processing of the MNIST data slowly and carefully, one step at a time. We'll use tools from both Keras and scikit-learn. This is both to carefully demonstrate what we're doing, and to show the sort of thinking we go through when we think about pre-processing.

Our goal is not just to pre-process the MNIST data, but to present the flow of the process, so we can apply it to new databases in the future.

It's always important to get a good feeling for our data before we start to work with it. Visualization, statistics, and even direct examination of the data files can give us insights into the character of our data. These insights are always useful when we think about how to process and learn from our data.

Reshaping

In this chapter we're going to *reshape* our data several times. Rather than roll along for a while and then stop to discuss this operation, we'll cover it now so it will be familiar when we need it.

Reshaping can be a mysterious process for programmers who haven't worked with multidimensional arrays (or tensors), so here's a short overview of what's going on. Readers familiar with multidimensional arrays and reshaping them should at least skim this section, because it contains the conventions we'll be using to draw and refer to our data. We also introduce a few useful features of NumPy along the way.

Reshaping is a general programming idea, so the ideas covered here are applicable to any programming language or task, not just Python or machine learning.

We'll start by imagining a list of 12 objects, which we'll name with labels A through L. Figure B2-3 shows these items.

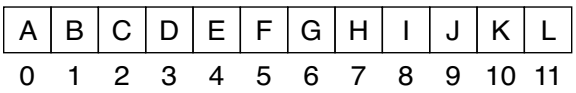


Figure B2-3: We have 12 items arranged in a one-dimensional list. Each element in the list is made up of just a single letter. Each element requires only a single index from 0 to 11 to identify it.

We call this a *one-dimensional list*, or simply a *list*, because we need only one *dimension*, or *index*, to identify which element we want. In a 1D list our convention will be to start at the left and count to the right. We always count indices starting with 0 [Dijkstra82].

So the cell at index 1 contains the label "B," and the label "H" is in the cell with index 7.

Here's the key point we're going to see in this section: we can tell the computer to think of this data arranged in different ways, *but we never change this list*. No matter how we re-shape it, the underlying data stays in a one-dimensional list and isn't affected. By re-shaping the data, all we're doing is telling the computer how to *interpret* the data when we read or write it. The data itself is not touched (as always, there are exceptions to this generalization, particularly when efficiency measures are applied. But those are usually invisible to us as users of a library).

NumPy offers a convenient routine that lets us *reshape* any input data into many different forms. For example, we can make a 2D grid that is 3 rows down by 4 columns across, as in Figure B2-4. Each entry now requires two indices to identify it, in the order down and then right. We place these indices in parentheses, separated by a comma. Starting in the upper left, we work our way right, then go down one row and start again from the left.

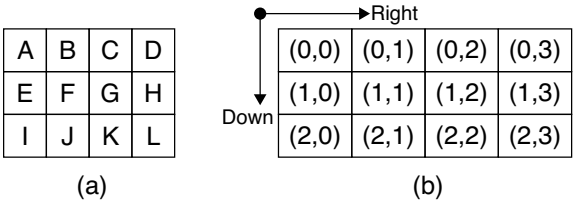


Figure B2-4: Our one-dimensional list of Figure B2-3 re-shaped into a 2D list of 3 rows and 4 columns

We call this is a *two-dimensional list*, or a *grid*, because we require two numbers to identify each element.

There's a possibility of confusion here that's worth addressing. In Figure B2-3 each element is drawn in a little box, and we called a horizontal row of those boxes a 1D list. But in Figure B2-4 we also drew our elements in little boxes, and called that arrangement a 2D grid. Couldn't we interpret Figure B2-3 as a 2D grid also, one that's 12 elements wide by 1 element high?

We definitely can, and we sometimes will. This is the source of the potential confusion we just mentioned: we can't tell just by looking at Figure B2-3 if it's a 1D array, or a 2D array of 1 row and 12 columns. We'll have the same problem later with 2D grids seen from the side, which might look like just the nearest slice of a 3D volume.

As humans looking at pictures, it's usually not a problem if we interpret of a row of boxes like Figure B2-3 as a 1D list, or a 2D grid with 1 row. But when we're programming, the distinction is critical. Most library routines are strict about their parameters, and they'll complain or even crash if they

get passed a variable with the wrong number of dimensions. If a routine expects a two-dimensional input, then it had better get a two-dimensional input, even if, to us humans, it's just a list of numbers.

When we get to the programming examples, we'll be careful to keep track of the number of dimensions in our data structures. In any discussion where the difference is important, we'll always be clear about how many dimensions make up any particular tensor.

Returning to our 2D grid of Figure B2-4, in a such a grid our convention is to use the first index to count down, and the second to count to the right. In brief, we index a 2D array as *(down, right)*.

This ordering is completely for our convenience. The computer cares about how the data is arranged, but it doesn't care how we *picture* the data's arrangement when we make diagrams for ourselves. But since we'd like to be able to draw pictures of our data, like Figure B2-4, and we want them to mean the same thing to everyone, we use the convention of listing the indices as down and then right.

Our (down, right) convention is popular, but not universal. We'll sometimes find pictures in documentation or other publications that interpret the data in some other order. It always pays to check.

Another convention is that we fill up the cells by starting at (0,0), then increment the rightmost index to get (0,1), then (0,2), and so on, until we reach the end of the row. Then we set the rightmost index back to zero and increment the index to its left, putting us at (1,0). We then continue to the right, with cells (1,1), then (1,2), and so on.

Using the down-then-over convention, we say that the layout of Figure B2-4 is arranged 3 by 4, meaning there are 3 rows and 4 columns. The cell at index (1,2) contains the label "G," and the label "J" is in the cell with index (2,1).

There are many other ways to arrange the 12 elements of our list into a 2D box. Continuing to use our convention of filling up the boxes left to right, then top down, Figure B2-5 shows a few other possibilities.

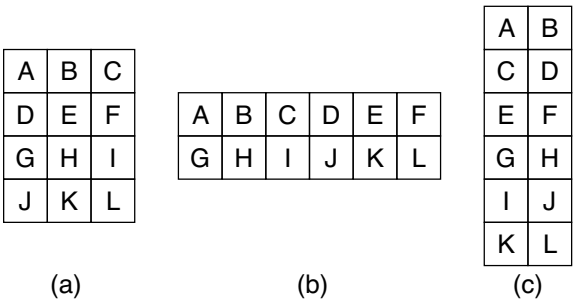


Figure B2-5: Three more ways to arrange our 12 items into a 2D list. From left to right, these grids have dimensions 4 by 3, 2 by 6, and 6 by 2.

We can even reshape our data into 3D. As in 2D, there is no universal convention for drawing data in 3D. Recall that in 1D, our one index told us how far to the right to move. When we needed a convention for 2D, we put

"down" in front of the 1D "right". For 3D, we'll put "away" in front of the 2D "(down, right)", giving us the order *(away, down, right)*. We start in the near, upper-left corner.

This has a nice analogy to reading a book. To identify a particular letter, we'd specify the page (away), the line of text (down), and the letter's position in the line (right).

Figure B2-6 shows this visually.

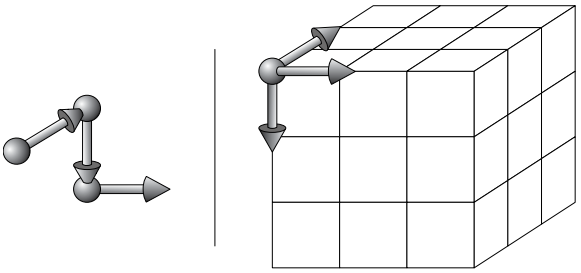


Figure B2-6: Our convention for identifying cells in a 3D block will be to start in the near, upper-left corner. We name cells in the order (away, down, right). (a) The three directions in sequence. (b) Finding a cell in a 3D volume. The first index tells us how far to move away, the second index how far to move down, and the third index how far to move right.

This fits nicely with our 2D convention as above. We think of our block as a collection of vertical slices arranged front to back. Each vertical slice is indexed in the order down and then right, just as in our 2D arrays above. In terms of our two arrangements in Figure B2-1 above, this is the `channels_last` organization.

A 3D block with indices is shown in Figure B2-7. Since there are 27 cells and only 26 letters, we placed a star at the end of the alphabet, in cell (2,2,2).

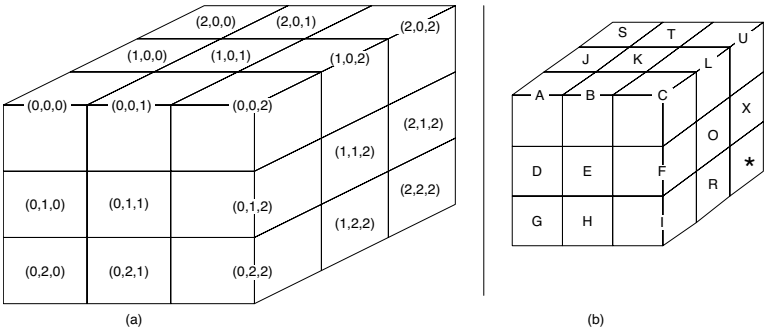


Figure B2-7: Identifying each cell in a 3 by 3 by 3 cube. Each cell requires 3 numbers to identify it. (a) Using our convention of Figure B2-6 we count away, down, and right. The rightmost index changes the fastest, then the middle index, and finally the left-most index. (b) Filling in the letters A-Z in order.

The closest vertical slice of nine cells are all indexed by their usual (down, right) values, with an "away" value of 0. The vertical slice in the middle has the same indices, but an "away" value of 1. And the farthest slice has an away value of 2.

Figure B2-8 shows three different ways to organize 12 entries into blocks.

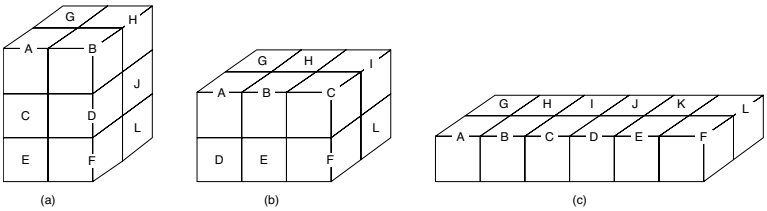


Figure B2-8: Three ways to organize our 12 elements into 3D blocks. From left to right, these have dimensions 2 by 3 by 2, 2 by 2 by 3, and 2 by 1 by 6.

We can change our arrangement of 12 items to any shape in Figure B2-8, and do so repeatedly, but remember that this operation only changes how the computer refers to the information. We never change the data itself. In other words, the computer does not move data around when we tell it to reshape it to some other shape. Reshaping simply tells the computer how we're going to name the elements: how many dimensions we'll use, and what values each dimension can take. It just saves those numbers, and then uses them when we actually read or write the data. So re-shaping a list of 12 elements is no faster than re-shaping a list of 12 million elements. The computer just remembers how many dimensions we have, and how big each one is, so it can locate the one we want when we provide a set of indices.

This principle is vital because it means we can repeatedly re-shape the data for different purposes, and it will always stay in order. So for example we can take our MNIST training samples, which arrive as a 3D box, and flatten them out, and then re-shape them in a 4D structure, and the data is never altered by these steps. In fact, we'll do just these sorts of things in the code below.

We just referred to a 4D data structure, meaning that we'll access our elements with 4 numbers. That's not easy to draw.

There's a nice way to visualize these *multidimensional lists* that works for any number of dimensions.

We think of our data structure as a *list of lists*. Instead of arranging our data spatially, as in the above figures, we draw the 1D list that represents the data in the computer's memory, and place the various pieces into a hierarchy of simple 1D lists, where each list is *nested* inside another.

In a 2D grid, there are 2 levels of nesting (each row is a list of elements, and the whole grid is a list of rows). In a 3D block, there are 3 levels (each row contains elements, each horizontal slice contains rows, and the whole block is a list of slices).

For example, recall the 3 by 4 grid in Figure B2-4. We can think of this as a grid of 3 rows of 4 items each, or as a list of 3 lists of 4 elements each, as in Figure B2-9.

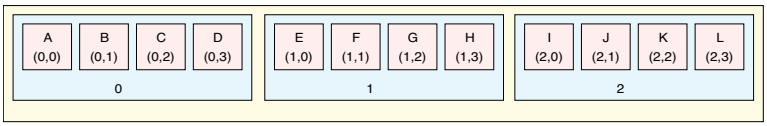


Figure B2-9: The 2D grid of Figure B2-4 has three rows of four elements each. We can show this as a hierarchy of 1D lists. Each set of four elements (that is, a row) is in a list. To identify any element, we first choose the list we want (the row), and then the element we want from that list (the column).

To find the element at cell (1,2) we go to list 1 (that's the second list, since we start counting at zero) and then select the third element. So element (1,2) is "G." We don't refer explicitly to the outermost list, since that's just a wrapper to keep everything together.

In the same way, we can nest our lists another level and represent the 3D blocks of Figure B2-8 as a set of nested lists. Figure B2-10 shows how this would look for the leftmost block of dimensions 2 by 3 by 2.

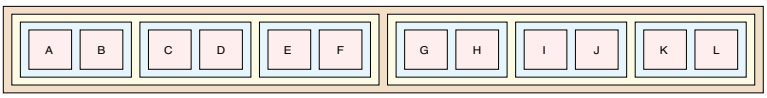


Figure B2-10: The 12 elements of our list arranged in the 3D block of size 2 by 3 by 2, as in Figure B2-8. To identify any cell, we need three numbers, corresponding to the indices in each of the nested lists.

We read the indices in the same way as before, starting with the outermost list and working inwards.

The element at index (1,0,1) is in the second outermost list, then the first list inside that, and then the second element of that list, giving us the label "H."

This offers another way to see that the data itself is never touched. The list-of-lists approach for the other blocks in Figure B2-8 are shown in Figure B2-11. We can see that the data is still just a simple, one-dimensional list of cells in order, and our reshaping simply tells the computer to group them together in different ways.

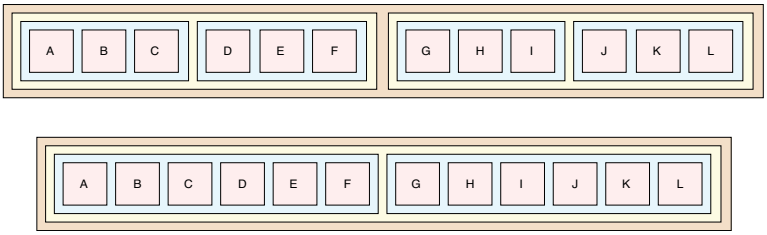


Figure B2-11: Interpreting the middle and right blocks of Figure B2-8 as lists-of-lists. Top: The block is 2 by 2 by 3. Bottom: The block is 2 by 1 by 6.

Note that in all of our examples, all of the lists at each layer have the same length. This is just another way of saying that our structures have no holes or extra bits sticking out in any dimension.

We reshape data using the NumPy function `reshape()`. Like many NumPy functions, we can call this in two different ways. Let's suppose we have data in a variable called `demoData`, arranged in a 2D grid like we saw above, with 3 rows and 4 columns. We'd like to rearrange this as a grid of 6 rows and 2 columns. We communicate the new shape we want by handing `reshape()` a list (or tuple) containing the new size along each dimension. For this example, we'd give it `(6, 2)`. We can assign the result back to `demoData` if we like, but let's save it in a new variable called `newData`.

If `demoData` is *not* a NumPy array, we need to call `reshape()` from the NumPy library. We give it the array we want it to reshape, and the list of the new dimensions. This is shown in Listing B2-4. The computer's output, here and in the rest of the chapter, is shown in red.

```
import numpy as np
demoData = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
newData = np.reshape(demoData, (6, 2))
print(newData)
 [[ 1  2]
 [ 3  4]
 [ 5  6]
 [ 7  8]
 [ 9 10]
 [11 12]]
```

Listing B2-4: Reshaping the array `demoData` by calling `reshape()` directly from NumPy

If `demoData` *is* a NumPy array, then we can call `reshape()` as a method of the array itself. To turn a Python array into a Numpy array, we can call Numpy's `array()` method. This will work for an array of any shape. That is, the input can be a tensor with any number of dimensions, and the output will be a Numpy array (or tensor) of the same shape. This version of reshaping is shown in Listing B2-5.

```
demoData = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
newData = demoData.reshape((6, 2))
print(newData)
 [[ 1  2]
 [ 3  4]
 [ 5  6]
 [ 7  8]
 [ 9 10]
 [11 12]]
```

Listing B2-5: Reshaping the array `demoData` by calling `reshape()` as a method of the array

The only rule is that the total number of elements in the tensor can't change. That is, if we multiply together all of the dimensions in the original shape of the tensor (here, 3 by 4), we must get the same value as when we multiply together all of the dimensions in the new shape (here, 2 by 6). Since 3 × 4 = 12 and 2 × 6 = 12, our examples worked.

If we try to reshape our data to an incompatible size, Python will complain. For example, Listing B2-6 shows the output from the interpreter when we try to reshape our 12-element array `demoData` to the shape `(5,15)`. Since we don't have 5 × 15 = 75 elements, Python reports an error.

```
demoData = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
demoData.reshape((5,15))
---------------------------------------------------------------------------
ValueError
Traceback (most recent call last)
<ipython-input-5-a51a5832a9f8> in <module>()
      1 demoData = np.array([[1, 2, 3, 4],[5, 6, 7, 8],[9, 10, 11, 12]])
----> 2 demoData.reshape((5,15))

ValueError: cannot reshape array of size 12 into shape (5,15)
```

Listing B2-6: Reshaping the array `demoData` to an incompatible size causes an error.

We'll make use of `reshape()` quite a bit.

Loading the Data

Now that we have re-shaping under our belts, let's return to our main goal of getting a neural network up and running. We'll begin by getting our hands on the data, and then prepare it for training.

Listing B2-7 shows how easy it is to load the MNIST set, since it's provided with Keras. To get it, we import the `mnist` module and then use its custom `load_data()` function to get the data. This returns two lists: the training data and the test data. Each list in turn contains two lists, holding the features (that is, the images), and the labels. We can use Python's convenient assignment mechanism to assign all four lists to our own variables with just one statement.

```
from tensorflow.keras.datasets import mnist

(samples_train, labels_train), (samples_test, labels_test) = \
    mnist.load_data()
```

Listing B2-7: Load MNIST data. It will be downloaded automatically if needed.

This is a good moment to point out that Keras functions (and their arguments) are pretty consistent about naming which kind of data set various objects belong to. Training data usually has the word `train` in there somewhere, test data has the word `test`, and validation data usually has the word `val` somewhere in its name.

As we saw in Chapter 8, when we use a technique like cross-validation we break down our input data into the *training set*, the *validation set*, and the *test set*. We teach many variations of the system using the training set, and then after each training we evaluate the performance with the validation set. When we're done searching, we select the model we want to deploy, we measure its performance with the test set. So the training and validation sets are used over and over, and the test set is used only once.

When we're not cross-validating, we need only the training set and the test set. The Keras documentation for `mnist.load_data()` identifies the returned data as belonging to these two categories, as in Listing B2-8.

```
# The definition of mnist.load_data() from the Keras documentation
(samples_train, labels_train), (samples_test, labels_test) = \
    mnist.load_data()
```

Listing B2-8: The routine `mnist.load_data()` returns a training set and a test set.

If the MNIST data has not been previously downloaded to this computer, then when we first load it Keras will automatically fetch a compressed form from the web, decompress it, and then save it in the directory that Keras maintains for these types of downloads (the exact location of this directory for each type of operating system can be found in the Keras documentation). If we request this data again on this computer, Keras will automatically grab the data already saved on the disk, saving us lots of time.

In Listing B2-8, the first pair of variables, `samples_train` and `labels_train`, holds arrays with the 60,000 images that form the training set, and their corresponding integer labels. The second pair of variables, `samples_test` and `labels_test`, holds arrays with the 10,000 images and labels that make up the test set.

Let's get a quick look at their shapes by printing them out in Listing B2-9. These arrays all come back to us from Keras already as NumPy arrays, so they all have a built-in `shape` attribute we can print.

```
print("samples_train shape = ",samples_train.shape)
print("labels_train shape = ",labels_train.shape)
print("samples_test shape = ",samples_test.shape)
print("labels_test shape = ",labels_test.shape)
samples_train shape =  (60000, 28, 28)
labels_train shape =  (60000,)
samples_test shape =  (10000, 28, 28)
labels_test shape =  (10000,)
```

Listing B2-9: The shapes of the MNIST data from Listing B2-7

This is telling us that `samples_train` is a 3D block of 60,000 layers. Each layer holds a 28 by 28 image. The `labels_train` variable is a 1D list of 60,000 elements (we'll see that each is a number from 0 to 9). The extra comma at the end of `(60000,)` is a Python convention to tell us that this is a list of 60,000 elements, and not just the number 60,000 surrounded by

parentheses [Wentworth12]). Similarly, `samples_test` is an array of 10,000 images, each 28 by 28, and `labels_test` is a list of integers with the test data's corresponding labels.

Although these variable names are perfectly fine, a common code convention is to use the capital letter `X` to refer to a data set's samples, and a lower-case letter `y` to refer to its labels. These letters were chosen to match the letters used in many deep-learning equations. The carry-over was natural in early programs that were written to closely match the equations, and the convention stuck. The lower-case `x` is also used for the samples, and the upper-case `Y` for the labels, though those are less common.

Using this convention, we'd write Listing B2-9 more succinctly as Listing B2-10.

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

Listing B2-10: Loading MNIST data, using `X` for samples and `y` for labels.

Using `X` and `y` is a nice convention once we're used to it, because these single letters can save us a lot of typing, and they are quickly understood by anyone who's used to this naming scheme.

There's no rule that says we have to use these cryptic variable names, even if they are conventional. The style of using `X` for features and `y` for labels is so frequently used that it's probably a good thing in the long run to use it, and we'll do so here. But every programmer should follow their own instincts for writing code that is clear and useful to themselves and others.

Looking at the Data

The first step in using any database is to look at it. We want to make sure that it's clean and organized in a useful way. We also want to generally get a feeling for what we're working with.

If the data needs to be modified before we use it for learning, we can use a combination of straight Python programming, and functions from libraries such as NumPy, SciPy, scikit-learn, and Keras itself. Such pre-processing is a vital step in making sure our network will work the way we want, and prevent errors. Happily, the MNIST dataset needs only a little bit of this work, so we can present it all here to get a flavor for the process.

There are at least two potential sources of problems to keep an eye out for. *Content problems* are numerical issues with the data itself, while *structural problems* are issues regarding how the data is organized.

Let's literally look at the data first. Figure B2-12 shows another random sampling of images from the training data. We can see that the examples are not all perfect.

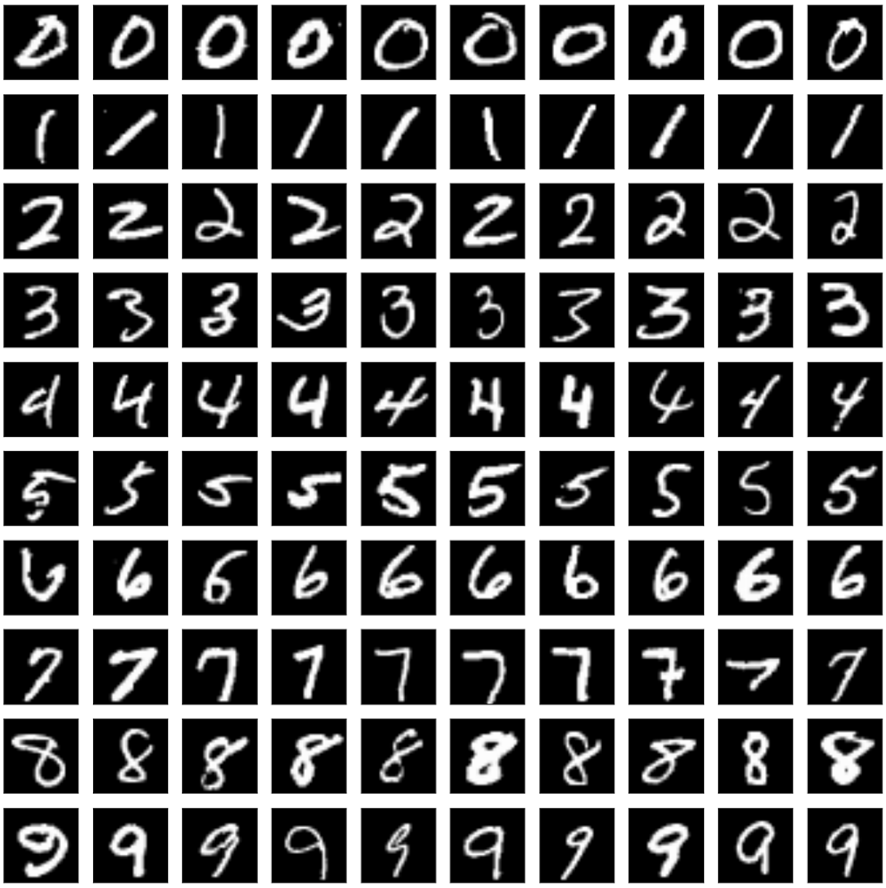


Figure B2-12: A random sampling of the images from the MNIST training set

There are four standout issues.

First, some of the images bleed very close to the edge of the 28 by 28 box, rather than sitting inside a relatively thick black border of 4 pixels all around that the original paper describes [LeCun13]. Some examples from the training set that have this quality are shown in Figure B2-13.

Figure B2-13: Some images from the MNIST training set that demonstrate a bleeding of the image very near, or right up to, the border. The numbers above each example shows its index in the training set, followed by its assigned label.

Second, some of the digits appear to have had pieces cropped away, substantially changing their shape. Figure B2-14 shows a few examples.

Figure B2-14: Some images from the MNIST training set that have been cropped, chopping away some of what seems very likely to have been drawn, and sometimes creating multiple, disconnected pieces.

Third, some of the images are noisy. Sometimes this means that lines thin out or disappear. More often there are spurious regions of white, perhaps due to errors during cropping or thresholding. These don't usually cause much confusion to human observers, but these artifacts have the potential to throw off a computerized network. Figure B2-15 shows some examples.

Figure B2-15: Some images from the MNIST training set that demonstrate noise artifacts. Some of these might be due to thresholding or cropping errors.

Finally, there are some examples that seem challenging to interpret, either because of how they were drawn, or how they were processed. Figure B2-16 shows a collection of some of these oddball training examples.

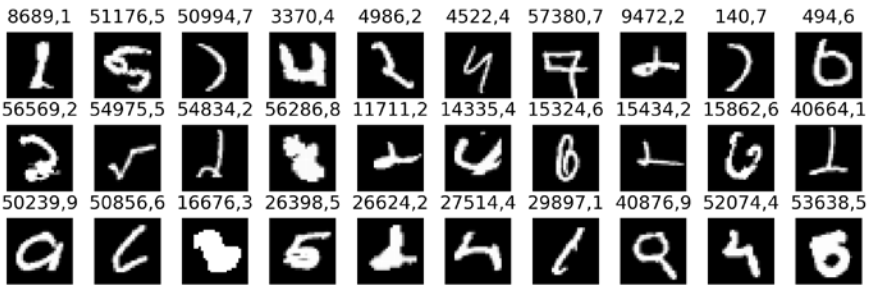


Figure B2-16: Some images from the MNIST training set that appear particularly challenging to categorize

We might be tempted to remove samples that have the artifacts we just looked at, but in fact as long as there aren't too many of them, they can make our system stronger. If our network can correctly identify these images despite their imperfections, then it has a robust quality that it wouldn't have without these stressful examples.

After browsing several random chunks of the data, we concluded that these problems were infrequent enough that we wouldn't bother to remove them. Even though we're taking no action, it was important to look the data over and reach this conclusion based on the data, rather than a hopeful guess.

Now we'll turn to the structure of the data, and see how it's organized.

Our main interest is in the shapes of the variables that we got from `mnist.load_data()`. Listing B2-11 recaps our starting objects using the shorthand `X` for samples and `y` for labels.

```
print('X_train shape:', X_train.shape,' y_train shape:', y_train.shape)
print('X_test shape:', X_test.shape,' y_test shape:', y_test.shape)
X_train shape: (60000, 28, 28)  y_train shape: (60000,)
X_test shape: (10000, 28, 28)  y_test shape: (10000,)
```

Listing B2-11: Printing shape information about our input data

Our training data, `X_train`, is in a 3D block. Using our (away, down, right) convention, it's 60,000 slices deep, where each vertical slice is 28 by 28 units. Figure B2-17 shows this shape.

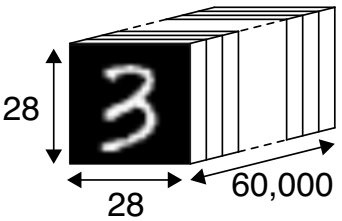


Figure B2-17: Our training data, `X_train`, has shape 60000 by 28 by 28. That means it's a stack of 60,000 objects, each an image that's 28 by 28 pixels.

The test data is set up the same way, except the stack is only 10,000 images deep.

We're going to reshape our data in the following sections, so let's stash the original height and width of each image in a variable. We'll also multiply them together and save that as the total number of pixels per image. Listing B2-12 shows how we'll save this data.

```
image_height = X_train.shape[1]
image_width = X_train.shape[2]
number_of_pixels = image_height * image_width
```

Listing B2-12: Saving the sizes of our input data for later use

This is a bit of overkill, since for this fixed data set we know every image is 28 by 28, but this more general approach will make it easier to later copy this code and adapt it to a new data set.

The labels are given to us as one-dimensional lists. The training label list `y_train` has, as expected, a length of 60,000, since it's providing one label for each sample in the training set. Let's look at the first few elements in Listing B2-13.

```
print('start of y_train:', y_train[:15])
start of y_train: [5 0 4 1 9 2 1 3 1 4 3 5 3 6 1]
```

Listing B2-13: The first few elements of the labels in `y_train`

So each entry in `y_train` is an integer. We expect it to be the label of the corresponding image in `X_train`. It always pays to check, so let's look at the first 15 images in X_train, shown in Figure B2-18.

Figure B2-18: first 15 images in `X_train`. These match the labels in `y_train`, shown above each sample, so we're good.

Great, the labels in `y_train` match the corresponding images in `X_train`. Since the MNIST data is so well known we can stop here, but with less familiar data sets we'd probably want to make at least several of these spot checks throughout the data to make sure that the two lists stay in sync.

Now let's look at the data itself. In Listing B2-14 we print an arbitrary little rectangle from within the first image of `X_train`. A handy bit of Python to keep in mind is that by simply typing the name of a variable to the interpreter (rather than using a print statement), we sometimes get more information about the variable.

```
X_train[0, 5:12, 5:12]
array([[  0,   0,   0,   0,   0,   0,   0],
       [  0,   0,   0,  30,  36,  94, 154],
       [  0,   0,  49, 238, 253, 253, 253],
       [  0,   0,  18, 219, 253, 253, 253],
       [  0,   0,   0,  80, 156, 107, 253],
       [  0,   0,   0,   0,  14,   1, 154],
       [  0,   0,   0,   0,   0,   0, 139]], dtype=uint8)
```

Listing B2-14: A small rectangle from the first training image in `X_train`

The variable `dtype` at the end tells us that this is a NumPy array, represented by the data type `uint8`, which means an unsigned 8-bit integer. Checking `X_test` reveals the same structure. As we might expect from grayscale image data, all of the values are between 0 and 255 (more on that below).

Are the labels also NumPy arrays? Listing B2-15 shows a piece of the `y_train` array.

```
y_train[:15]
array([5, 0, 4, 1, 9, 2, 1, 3, 1, 4, 3, 5, 3, 6, 1], dtype=uint8)
```

Listing B2-15: A piece of the y_train array. The input is on the first line, the rest is output.

Yup, this is a 1D NumPy array of unsigned 8-bit integers. That's pretty restrictive for training labels, since such numbers can't go above 255. But here we're only storing the labels 0 to 9, so the range from 0 to 255 is plenty.

To use this data for training with Keras, we need to turn the training and test sample data into normalized floating-point numbers, and turn the labels into one-hot encodings (as discussed in Chapter 10).

But before we do that we'll take a quick pause. The MNIST data is conveniently already split into training and test sets. What if it wasn't? There's a nice utility that will split our data for us. Let's look at it now.

Train-test Splitting

Most data sets require us to manually split them into training and test sets. The MNIST data has already been split for us, but for completeness, let's see how we'd do the job if we had to.

The easiest and most common approach is to use scikit-learn's `train_test_split()` function to do all the work for us. Suppose that the MNIST data came to us as only two tensors, called `samples` and `labels`, and we want to split it into a training set and a test set. A typical test set is often around 20% or 30% of the starting data, so let's go down the middle with 25%.

We just call `train_test_split()` with our data and the split size, and it returns four arrays, as in Listing B2-16.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = \
    train_test_split(samples, labels, test_size=0.25)
```

Listing B2-16: Splitting data into a training set and a test set using `train_test_split()` from scikit-learn

Figure B2-19 shows this operation visually. The function `train_test_split()` doesn't simply cut the input data in one place as shown in the figure, but shuffles a copy of the data first so it's more likely that each of these two pieces will contain a good mix of all the samples.

Note that Listing B2-16 gives us back four arrays, not the two arrays of two elements each that were returned by `mnist.load_data()` They're also in a slightly different order compared to in Listing B2-10. These kinds of minor inconsistencies between libraries can be a hassle until we get used to them.

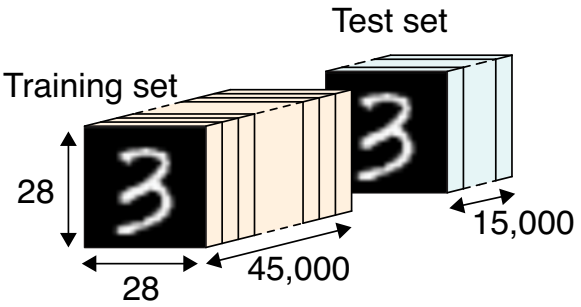


Figure B2-19: Splitting a dataset of 60,000 images. We're using 25% of the data for the test set and the other 75% for the training set.

One way to catch these inconsistencies before they become major debugging problems is to go slowly and build up our code one line at a time in an interactive Python environment, as we discussed earlier. When we do something wrong, we'll get an immediate error that we can investigate more closely by printing things out, and comparing what we're doing with what the library documentation describes we should be doing.

When learning a new library, lots of little experiments can help us write good code from the start.

Fixing the Data Type

As we saw in Listing B2-14, the sample data we get from `mnist.load_data()` is returned to us as integers. Though this is efficient and reasonable for storing the data, Keras wants to work with floating-point numbers. To prevent making incorrect assumptions, Keras won't automatically *cast*, or convert, number types for us. That's our job, and it's mandatory. Keras expects floats and it better get them, or it will either go haywire at some point, or more usually, report an error and stop.

In fact, Keras expects the specific type of floats that match its internal `floatx` parameter. We saw above in Listing B2-1 that we can assign that parameter to different data types in the keras configuration file by assigning a new value to `floatx`, or in our code by calling `set_floatx()` in the Keras backend, as in Listing B2-2.

By default, `floatx` has the value `float32`, meaning a 32-bit floating-point value. Unless we change the configuration file, or call a backend function to change this in our code, this is the type that Keras expects.

Switching this to another data type (such as `float64`) is easy to do, but knowing when such a choice makes sense is complex and dependent on one's specific hardware and software, so we'll stick with `float32`.

Now that we know the format Keras expects for our floating-point numbers, we can return to our job of converting our samples into that form. The easy way to do this is to use the function `cast_to_floatx()` from the Keras backend, which takes a tensor as an argument and casts every element of that a tensor into the type specified by the current value of `floatx`. The routine doesn't even care about the shape of the tensor. From a 1D list to some giant

tensor with a thousand dimensions, the routine will simply crank through every entry and convert it to our desired type of data. Note that the last word in this routine's name is not `float`, but rather `floatx`, referring to the configuration variable. Listing B2-17 shows how to use it.

```
from tensorflow.keras import backend as keras_backend

X_train = keras_backend.cast_to_floatx(X_train)
X_test = keras_backend.cast_to_floatx(X_test)
```

Listing B2-17: Using the Keras backend to change our array types to the value it expects.

We might be tempted to cast the `y_train` and `y_test` arrays to the `floatx` type also, but that's not necessary. We'll be converting these arrays into their one-hot forms below using another utility routine, and that routine expects a list of integers as input. This is yet more of the kind of details that make for slow going when first getting used to a new library.

Now that our features have the right type, we can move on to making sure they have the most useful range of values.

Normalizing the Data

Another important step in preparing data is *normalizing* it. This can mean slightly different things in different contexts, but it always means changing the data itself, rather than simply re-shaping it.

The networks that we'll be building in this chapter to categorize the MNIST data will use convolution layers near the start, and those will work best with data that has been normalized so that each feature has been scaled to fit the range 0 to 1.

Note that normalization is just for the features, and not the labels. The labels need to refer to the 10 different classes from 0 to 9, and we don't want to change those values.

Listing B2-14 showed us that our feature data in `X_train` and `X_test` is originally made of integers in the range 0 to 255, This is a common range for a channel of image data. We've just converted these values to 32-bit floats, so we could say that they're now in the range 0.0 to 255.0.

We said above that we need to normalize our data to the range [0.0, 1.0]. As we saw in previous chapters, this helps to keep neuron outputs in the same range, which helps with regularization and delaying the onset of overfitting. And if we're using an activation function like a sigmoid, it keeps our functions from saturating.

We could accomplish this normalization with a full pre-processing step. We'd examine the values of the pixels in the training data, build a transformation to scale them to [0,1], and then apply that transformation to the training data, the test data, and any future data. We could create one of scikit-learn's transformation objects, train it, and then apply it to our data.

That's a perfectly good way to proceed, but when we're working with image data like that in the MNIST data set, we almost always transform our data with a simpler and more direct approach.

We know that our pixels in the training and test data are in the range [0, 255]. All we want is to rescale all the pixels in the same way, compressing them from the range [0, 255] to the range [0,1]. Conceptually, this is like converting measurements in millimeters into kilometers, or vice-versa.

We can scale our input data with Numpy's `interp()` routine, which is designed for exactly this job. It takes an array (or tensor), an input range, and an output range. For each entry it will find its location in the first range (0 to 255) and find its corresponding position in the second range (0 to 1). Listing B2-18 shows the code.

```
X_train = np.interp(X_train, [0, 255], [0,1])
X_test = np.interp(X_test, [0, 255], [0,1])
```

Listing B2-18: Scaling pixels from [0, 255] to [0,1].

This works perfectly, but since we know our data is in the range 0 to 255, we can accomplish the same thing just by dividing all the pixels by 255.0, as in Listing B2-19.

```
X_train /= 255.0
X_test /= 255.0
```

Listing B2-19: Rescaling our pixels to [0,1] by dividing them by 255

Listings B2-18 and B2-19 do exactly the same job, Although the second approach is a little less explicit about what's going on, it's both shorter to write and ever-so-slightly faster to execute than the version that uses interpolation.

These reasons are probably why Listing B2-19 is the common idiom for scaling images. Keeping with that convention, we'll use it here as well.

Let's gather everything we've seen so far in one place. We'll import the modules we need, read in the data with Listing B2-10, save the sizes with Listing B2-12, convert it to floating-point with Listing B2-17, and scale it to [0,1] with Listing B2-19. This is all bundled together in Listing B2-20.

```
from tensorflow.keras.datasets import mnist
from tensorflow.keras import backend as keras_backend

# load MNIST data and save sizes
(X_train, y_train), (X_test, y_test) = mnist.load_data()
image_height = X_train.shape[1]
image_width = X_train.shape[2]
number_of_pixels = image_height * image_width

# convert to floating-point
X_train = keras_backend.cast_to_floatx(X_train)
X_test = keras_backend.cast_to_floatx(X_test)

# scale data to range [0, 1]
X_train /= 255.0
X_test /= 255.0
```

Listing B2-20: Reading in our data, saving the sizes, converting to floats, and scaling to [0,1].

Our training and test samples are now in floating-point format and scaled from 0.0 to 1.0.

This is the end of pre-processing for the samples. We need to remember in the future that if we get any new samples that we want to evaluate with this network, they too need their pixel data to be converted to 32-bit floats and divided by 255.

There's a subtle point that's important to note. Any new images we get after training is complete should *not* be simply scaled to the range [0,1]. Instead, we need to apply the identical pre-processing that we applied above, meaning that the new image data needs to be divided by 255. If for some reason there are values in that image less than 0 or greater than 255, then they will turn into floating point values less than 0 or greater than 1. That might be inconvenient in some way, but we can't avoid it, because we must use the same transformation on the new data that we used on the data we train with.

Now let's pre-process the labels so that they're ready for use.

Fixing the Labels

We know that the MNIST data contains images of digits from 0 to 9. So in our network we'll create an output layer with 10 neurons, one for each digit. Each neuron will produce a probability that the image it's just been fed corresponds to that digit. The neuron with the highest value will be the network's final prediction for the input.

We'd like to compute an error value that tells us how close these 10 values are to the values we want. To make this comparison easy, we represent the label for each image using *one-hot encoding*, as we discussed in Chapter 10. For an input of a 3, it's a list of 10 elements, where all are 0 except for 1 in slot 3. Figure B2-20 shows the idea visually.

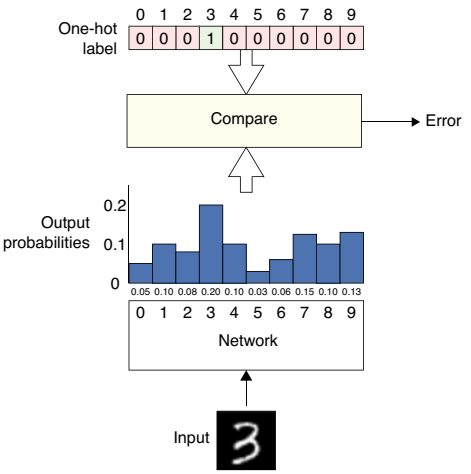


Figure B2-20: Computing the error. We feed an image (here a picture of a 3) to our network, and we get back a probability from 0 to 1 for each possible label from 0 to 9. We compare these 10 numbers with the 10 values in the one-hot label. The more the prediction is like the label, the smaller the error.

In this imaginary example, the network has given the value 3 the greatest probability, but it's given each of the other digits some chance of being right, too. A perfect answer from the network would be a probability of 1 that the input is a 3, so all other choices would have a probability of 0. In other words, a perfect prediction would be the same as the label. The more the two are different, the higher the error. The one-hot form of the label simplifies this comparison of the output and the label.

It might seem like one-hot encoding is superfluous, since a network could do this operation on the fly when it's needed. That's true, but that step would have to be repeated for every sample during training. If we trained for only one epoch (that is, every sample is used once) then it wouldn't matter if we used a pre-processed label or created it only when we needed it. But if we train for, say, 200 epochs, then we'd have to repeat the on-the-fly encoding of every sample 200 times. It's faster to encode the values just once before we start training. Providing pre-encoded labels also lets us create labels with values other than just 0 and 1, if we prefer.

So we'd like to turn the integers we get back in the variables `y_train` and `y_test` into one-hot encoded versions.

Turning each integer in a list into a one-hot encoding is such a common task that Keras provides a utility for it. The routine `to_categorical()` looks through an array of integers and finds the largest value, so it knows how many 0's are needed to represent all the values that need to be encoded. It then makes a one-hot encoding for each integer in the list. The output of `to_categorial()` is a list of these encodings, which are themselves lists of 0's and 1's.

Let's see one-hot encoding in action. Listing B2-21 shows the first 5 entries of the original `y_train` array before and after they've been one-hot encoded.

```
from tensorflow.keras.utils import to_categorical

# print the first 5 entries of the original y_train array
y_train[:5]
array([5, 0, 4, 1, 9], dtype=uint8)

# encode the y_train array as one-hot lists
y_train = to_categorical(y_train)

# print the new first 5 entries of y_train, now one-hot encoded
y_train[:5]
    (array([[ 0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.],
            [ 1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
            [ 0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.],
            [ 0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
            [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.]]),
    dtype('float64'))
```

Listing B2-21: Before and after one-hot encoding the `y_train` array with the `to_categorical()` utility function.

As we can see, the output is a 2D grid with one row for each input. Every entry is 0 except for a single 1, located at the index corresponding to the original `y_train` value for that row.

The one-hot values produced by `to_categorical()` are in 64-bit floating-point form. Happily, floating-point is just fine, since Keras will be comparing these floating-point values with the floating-point values coming out of our neural network. It's a bit strange that Keras doesn't use the default `floatx` type when it produces this data, but the 64-bit floats work fine when training our network.

We might be tempted to simply pass `y_train` and then `y_test` to `to_categorical()` in succession and move on, but that could introduce a subtle bug. The problem is that the largest value in one list might be different than the largest value in the other, giving us lists of different sizes.

For instance, suppose that the test data was missing any images of the digit 9. That means that `y_test` will contain only the digits 0 to 8. When we use `to_categorical()` we'll get back a list that has only 9 items. This will cause trouble later when we want to compare it to the values in our output layer, which has a score for each of 10 categories.

We don't have to worry about this problem with the MNIST data, because it has examples for every image in both sets, but it might come up in other data sets.

There's an easy, general solution that will always avoid this problem. It involves using an optional argument to `to_categorial()` that overrides its scanning step. This argument, called `num_classes`, tells the routine to always make lists of the given length. The prefix `num_` is a common convention which is read as "number of," so `num_classes` stands for "number of classes."

The value of `num_classes` has to be at least big enough to encode all the possible values, or we'll get an error. If `num_classes` is bigger than necessary, that's fine, and the extra values at the end will always be 0.

To make sure both encodings will be the same size for any two lists of labels, we will combine all the labels into one big list and extract its largest value. Since we're starting with 0, we'll add 1 to the result, and that's the smallest size of the list that can encode all the values in all the labels.

Listing B2-23 shows how to use `to_categorical()` to turn our list of integer labels into a list of one-hot encodings in a general way.

```
# combine the input lists to find largest value in either, then add 1
number_of_classes = 1 + max(np.append(y_train, y_test))

# encode each list into one-hot arrays of the size we just found
y_train = to_categorical(y_train, num_classes=number_of_classes)
y_test = to_categorical(y_test, num_classes=number_of_classes)
```

Listing B2-22: The label arrays are replaced with one-hot encodings.

Sometimes we want the original list of integers somewhere else in the program, as we'll see later when we do cross-validation. We can "undo" the one-hot encoding in two ways. If the one-hot encoding is represented as a

regular Python list (that is, not a NumPy array), we can use Python's built-in index() method, as in Listing B2-23.

```
one_hot = [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
print("one-hot represents the integer ",one_hot.index(1))
one-hot represents the integer  3
```

Listing B2-23: Using Python's index() method to "undo" one-hot encoding.

If the one-hot version is a NumPy array, then we can't use index(), because NumPy doesn't support that method. There are several ways to use NumPy to find the index of a single 1 in list of 0's. Listing B2-24 shows one way to do it. This uses NumPy's argmax() method, which returns the index of the largest value in a list.

```
one_hot_np = np.array([0, 0, 0, 1, 0, 0, 0, 0, 0, 0])
print("one_hot_np represents the integer ",np.argmax(one_hot_np))
one_hot_np represents the integer  3
```

Listing B2-24: Using Python's index() method to "undo" one-hot encoding.

Rather than use either of these methods to find the integer versions of the one-hot encodings, we'll just save the original integer lists before we call to_categorical(), as in Listing B2-25.

```
# save the original y_train and y_test
original_y_train = y_train
original_y_test = y_test
```

Listing B2-25: Saving the labels in their original format as lists of integers.

Just for reference, Listing B2-26 provides a Python one-liner that will undo one-hot encoding, for those times when we're given the data already in one-hot form.

```
original_y_train = [np.argmax(v) for v in y_train]
original_y_test = [np.argmax(v) for v in y_test]
```

Listing B2-26: Turning our one-hot encoded targets back into lists of integers.

Because one-hot encoding is so common, scikit-learn also offers a tool to perform it. It's in the preprocessing module, and is called OneHotEncoder().

Pre-Processing All in One Place

We've just reached the first mountaintop! It's been a long way, but we've done a lot. Starting from an empty slate, our data is now ready for training.

To recap, we began by reading in (and possibly downloading) the MNIST data, and then prepared each image for Keras by changing it from integers to floats, and then normalized it. Then we created one-hot encodings of our labels.

Listing B2-27 brings all of these pre-processing steps together in one place. We've also added a line to seed NumPy's random number generator.

This means that any random numbers we get from NumPy will always be the same from one run to the next. Though we're not using random numbers yet, we will be using them later. Forcing our random numbers to always come out the same in each run makes debugging a lot easier.

```
from tensorflow.keras.datasets import mnist
from tensorflow.keras import backend as keras_backend
from tensorflow.keras.utils import to_categorical
import numpy as np

random_seed = 42
np.random.seed(random_seed)

# load MNIST data and save sizes
(X_train, y_train), (X_test, y_test) = mnist.load_data()
image_height = X_train.shape[1]
image_width = X_train.shape[2]
number_of_pixels = image_height * image_width

# convert to floating-point
X_train = keras_backend.cast_to_floatx(X_train)
X_test = keras_backend.cast_to_floatx(X_test)

# scale data to range [0, 1]
X_train /= 255.0
X_test /= 255.0

# save the original y_train and y_test
original_y_train = y_train
original_y_test = y_test

# replace label data with one-hot encoded versions
number_of_classes = 1 + max(np.append(y_train, y_test))
y_train = to_categorical(y_train, num_classes=number_of_classes)
y_test = to_categorical(y_test, num_classes=number_of_classes)
```

Listing B2-27: Combining the fragments above to create a complete pre-processor.

Part of the appeal of using libraries like scikit-learn and Keras is that there is remarkably little fiddling about with additional Python code to get things done. Almost every line of Listing B2-27 is either doing a specific pre-processing step, or saving variables that we'll use again later.

In this code we're repeatedly over-writing the data in `X_train` and `X_test`, and the labels in `y_train` and `y_test`. This is a common approach during pre-processing, because we don't care about the starting or intermediate values. The upside is a degree of simplicity. The downside is that if we want to access the original data, we either have to save it (as we do here for the labels), or load a fresh copy of the data.

Making the Model

This section's notebook is Bonus02-Keras-2-Making-the-Model.ipynb.

Now that our data is ready for use, let's build our deep learning model.

The beauty of model-making in Keras is that creating the structure of our model (that is, our neural network's architecture) is streamlined. There are only two steps.

First, we name the layers we want in the order we want them. This is called *specifying* the model.

Second, we tell Keras how to use this model to learn. We tell it which loss function and optimizer to use, and what data we'd like it to collect along the way. This is called *compiling* the model. The compilation step converts our specification into code that runs on TensorFlow.

Our first model for classifying MNIST data will be simple. It will have an input layer (which is implicit in every network), a single hidden layer, and an output layer. The hidden and output layers will both be fully-connected, or dense, layers. Figure B2-21 shows our first deep-learning system.

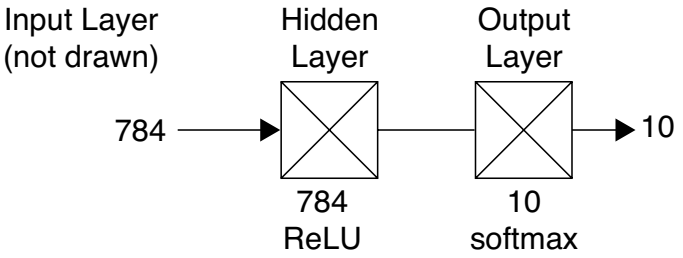


Figure B2-21: Our first, very simple deep-learning model consists of a fully-connected layer of 784 neurons (one for each input pixel) followed by a fully-connected layer of 10 neurons (one for each output class).

Recall that in drawings like Figure B23.21, we don't draw the input layer, because it's just a memory buffer. By convention, data flows left to right, as we're doing here, or sometimes instead bottom to top. The labels at the ends show the size and shape of the data going into and coming out of the network.

We've decided to set up our first layer to have a single neuron for each pixel. This is a common way to configure the first layer, but it's definitely not required. We could use 5 neurons or 5000 if we thought that would produce better results.

Using this "one neuron per input pixel" approach for our 28 by 28 images, our first layer requires $28 \times 28 = 784$ neurons.

Wait a second. We saw above that our input is a list of 2D grids, each 28 by 28. Why are we setting up our network to expect a flat list, rather than a 2D grid?

We're not doing that on purpose. A fully-connected layer can only take in a 1D list. There's no processing inside of a dense layer would let it figure out how to get at the pixels in a 2D data structure. We'll see later that convolution layers have that processing, so we can give them grids directly. But right now we're using a dense layer, and the input to a dense layer is a list.

So we need to convert each input sample of 28 by 28 pixels into a 1D list of 784 values.

Turning Grids into Lists

There are at least two ways to do this. The first is to build it right into our neural network, using the `Reshape` utility layer provided by Keras. The second is to reshape the data ourselves before training.

The first approach has simplicity going for it. We just make a `Reshape` layer and stick it ahead of the `Dense` layer and we're done. The downside is that every sample will get reshaped every time it's evaluated, and that will take some time. Since we expect to be running all the training samples through the network multiple times (that is, we'll train for multiple epochs), it's more efficient to pre-process it ourselves once. Recall that this is the same logic that led us to pre-process our labels into one-hot versions.

To convert our images into a list, we'll convert our starting 3D input data into a 2D grid. Each row of the grid is one sample, made up of a list of 784 features. The result is shown in Figure B2-22.

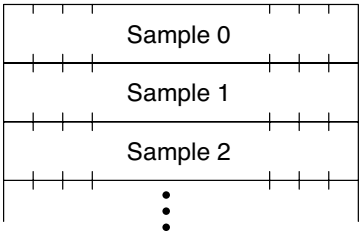


Figure B2-22: Turning our 3D input into a 2D grid, containing one long row of pixels for each image.

This is easy to do using Numpy's `reshape()` function, discussed above. We'll tell it to re-interpret `X_train`, which it is thinking of as a 3D block with dimensions 60,000 by 28 by 28, instead as a 2D array with dimensions 60,000 by 748.

As we discussed above, there are two ways to use `reshape()`. Let's first use the version where we call it from Numpy and pass it the array we're reshaping as the first argument.

The second argument to `reshape()` is a list with the new dimensions. In this case, the second argument is the list `[60000, 748]`. To make it easier to re-use this code for other projects later, we'll get these numbers from the data rather than typing them in directly. Recall that `number_of_pixels` has been set in our pre-processing step of Listing B2-27 to be the size of each input image, or 784.

For simplicity, we'll continue to over-write our values of `X_train` and `X_test` with these new versions. Listing B2-28 shows the code.

```
# reshape samples to 2D grid, one line per image
X_train = np.reshape(X_train, [X_train.shape[0], number_of_pixels])
X_test = np.reshape(X_test, [X_test.shape[0], number_of_pixels])
```

Listing B2-28: Flattening our images into a 2D grid, so each sample is just a single list of numbers. This is the format we need for a dense layer, like our first layer in Figure B2-21.

As we discussed, the other way to call `reshape()` is to call it as a method on the object being reshaped. In this case, the only necessary argument is the list containing the new dimensions. Because this is also common, we demonstrate this in Listing B2-29.

```
# reshape samples to 2D grid, one line per image
X_train = X_train.reshape([X_train.shape[0], number_of_pixels])
X_test = X_test.reshape([X_test.shape[0], number_of_pixels])
```

Listing B2-29: Another way to reshape our images into a 2D grid. The results are identical to those of Listing B2-28.

Both of these variations produce the same results, so we can use whichever one we prefer. We'll use the shorter, second version in the following discussion.

This re-shaping step is properly part of the pre-processing section, because we only need to do it once, so we'll place it there in the listings below.

We'll see later that other types of layers, such as convolution layers, will want their data to be shaped in other ways. Getting the data into the right structure is an essential step in training neural networks.

Creating the Model

Now that our data is fully processed, we can build the model.

We start by telling Keras the overall architecture of our model. Our choices are basically "a list of layers," and "anything else."

The "list of layers" architecture is called the `Sequential` model. That's perfect for us, since our architecture of Figure B2-21 is just two dense layers one after the other. In other words, they can be described as a 2-element list starting with the hidden layer and ending with the output layer.

The "anything else" architecture is called the `Functional` model. This is more flexible than the `Sequential` model, but requires a little more work from us. We'll come back to the `Functional` model later.

We build a model in the `Sequential` style using the *Sequential API*, which is a collection of library calls designed to make this process easy. The beauty of the `Sequential` API is that to create our model we just name our layers in order from start to finish. This lets Keras automatically work out how each layer connects to the one before and after, so it can manage the flow of data from one layer to the next automatically. This is a great time-saver both in programming and debugging.

To build our model, we create a variable to hold a `Sequential` object. This is initially an empty layer of lists. Then we add our layers to that object.

The first time we add a layer to our model, Keras will automatically create an input layer for us to hold the incoming data. Then it places our new layer after that. We could stop right there if we wanted, and that would be a 1-layer neural network (remember that we usually don't count the input layer, since it doesn't do any processing).

But we can keep going, and add as many more layers as we like. Each new layer takes its input from the most recently added layer. The last layer we add in is implicitly our output layer. We never explicitly say that we're starting or ending. We just add in layers until we're done.

Listing B2-30 shows the first step, where we create the `Sequential` object and save it in a variable.

```
from tensorflow.keras.models import Sequential

model = Sequential()
```

Listing B2-30: Creating an empty deep-learning architecture in the `Sequential` style.

A quirk of this approach is that the layers appear in the code in exactly the opposite order that we normally draw them. As we've seen, the drawing convention is to show the layers going rightwards or upwards. But in the source code, each new layer appears *under* the one that precedes it, so reading the code downwards corresponds to reading the figure rightwards or upwards. This can take a little getting used to, but eventually the mental flip becomes second nature.

Let's start building our model. The first layer is always the input layer. But recall that the input layer is implicit. We don't usually draw it, or count it, and in the `Sequential` model we usually don't even explicitly make it.

This is fine, because the input layer does nothing but hold the feature list for a sample. So the only thing we need to tell Keras about the input layer is how big that list should be, and it will make the appropriate storage for us.

We tell Keras the size of the input layer with an optional argument called `input_shape`. We pass a value to this argument in the first layer only. In other words, this argument *must* be included when we make our first layer, but must *not* be in any others. Every type of layer that can serve as the first layer in a sequence (including the fully-connected layer we'll be using), takes `input_shape` as an optional parameter.

Let's make our first layer.

Our diagram of Figure B2-21 specifies that our first layer is a fully-connected layer.

Keras calls a fully-connected layer a *dense* layer. Note that here the word "dense" refers to how the layer connects to the layer that *precedes* it. In other words, every neuron in this layer will be connected to every output in the previous layer. We are saying nothing at all about what happens to the outputs of the neurons on this layer. Keras will only discover where they go and how they get used when we specify the *next* layer in the description. If there is no next layer, then the outputs of this layer are the outputs of the whole system.

Since most layers are in the midst of a stack, we usually refer to the neurons receiving data from neurons in the "previous" layer. In the special case when the previous layer is the input layer, those neurons get their data from the values of the input saved on that layer.

Figure B2-23 shows a dense layer in schematic form. When we create this layer, we're only declaring the nature of its connections to the layer before it, and we're saying nothing about what happens to its outputs.

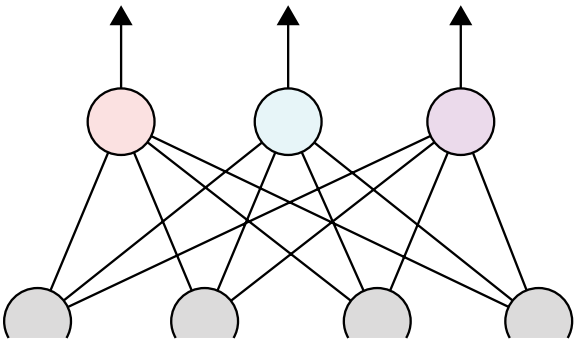


Figure B2-23: A schematic view of a `Dense` layer. The three colored neurons make up the dense layer. Each of them connects to every neuron in the preceding layer (in gray).

To add a dense layer to our model, we create a `Dense` object and then append it to the end of our model's sequence of layers. Although the `Dense` object has many arguments, we'll only use three of them right now. In standard Python convention, the first argument (which is mandatory) is not named, but the others are named and may appear in any order.

The necessary first argument is the *size* of the layer. This is just the number of neurons. This can be, and often is, different from the number of nodes in the preceding layer. For instance, the previous layer (whether it's the input layer, or has neurons) might have 4 outputs. Our `Dense` layer could have fewer than that, or the same amount, or more, as shown in Figure B2-24.

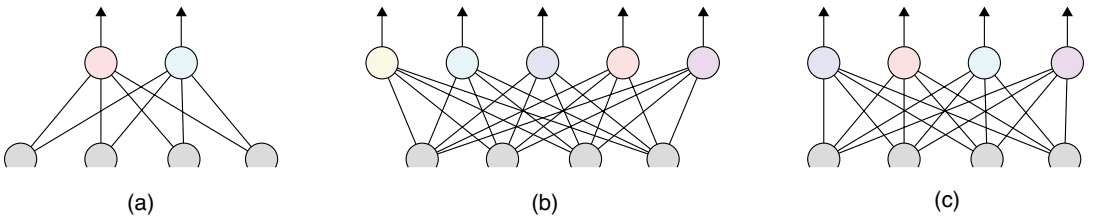


Figure B2-24: Our fully-connected layer is shown with colored neurons, connecting to a previous layer with gray neurons. The number of neurons in the fully-connected layer is independent of the number of neurons in the layer that precedes it.

As we discussed above, for our first classifier we'll use the same number of neurons as there are pixels in the inputs. This is a common way to set up an image classifier, but we might later find that the system learns better with fewer nodes in this layer, or more. In the back of our minds we can consider this a variable to play with later on, to see what value gives us the best performance.

The first optional argument we'll use tells Keras which activation unit to place after each neuron in the layer. We can specify any one of the functions built into Keras (and, as usual, listed in the documentation) by supplying a string. Common choices are `'relu'` and `'tanh'` for the ReLU and tanh functions in hidden layers, and `'softmax'` or `'sigmoid'` for the output layer. The default is `'None'`, or the linear activation function, so for internal layers we'll almost always want to specify one of the other choices.

The second optional argument we'll use is `input_shape`, which defines the size of each dimension in the input. As we saw above, we use this *only* for the very first layer in a model. The value of this argument is a list that tells Keras to build an input layer of the given shape and size, which must match the shape and size of each sample we'll be providing.

Since each of our samples (after processing) is a 1D list of 784 numbers, we'll tell Keras that our `input_shape` is a 1D list of 784 numbers (using the variable `number_of_pixels` that we saved during pre-processing).

Listing B2-31 shows how to create our first `Dense` layer.

```
from tensorflow.keras.layers import Dense

# create the Dense layer
dense_layer = Dense(number_of_pixels, activation='relu',
                    input_shape=[number_of_pixels])
```

Listing B2-31: Creating our first `Dense` layer. We need to import the `Dense` object from `keras.layers` to access it. Because this is the first layer in the model, we provide a value for `input_shape`.

Once we've made our `Dense` layer, how do we add it to our model? Curiously, although Python has a built-in operation called `append()` that adds one element to the end of a list, Keras doesn't use that name for this operation, which is conceptually the same. Instead, it uses the ambiguous name `add()`, in its colloquial sense of "add another log to the fire," rather than its numerical sense of "add 2 and 4." It may be useful to think of the Keras `add()` routine as though it had a more descriptive name such as "append."

Listing B2-32 shows the code for appending our layer to the list of layers in `model`.

```
# append our layer to the list of layers in model
model.add(dense_layer)
```

Listing B2-32: Appending a new layer to our model

Using the two listings above one after the other is perfectly fine. It's clear and it works right. Listing B2-33 shows the sequence.

```
dense_layer = Dense(number_of_pixels, activation='relu',
                    input_shape=[number_of_pixels])
model.add(dense_layer)
```

Listing B2-33: Creating a `Dense` layer, and adding it to our model, in two steps

But it's conventional to create the layer and add it to the model in a single line, as in Listing B2-34. This means that the layer doesn't get a variable that holds it, but we rarely need that (Keras does provide a mechanism for getting at the layer later, if we really need it).

```
model.add(Dense(number_of_pixels, activation='relu',
                input_shape=[number_of_pixels]))
```

Listing B2-34: A more efficient and common way to create a `Dense` layer and then add it to our model

Now we can add the next layer of our model. This will be another `Dense` layer, but with 10 neurons.

As we mentioned before, we don't explicitly tell Keras that this is our output layer. We just make it and add it to the growing list of layers. When we use the model, Keras will treat it as the output layer simply because it's the last one on the list.

We create our next `Dense` layer much like the previous one, but with a few changes. In particular, we leave out the `input_shape` argument, since that is only for the very first layer.

As always, the first argument, which is un-named and mandatory, is the number of neurons. Since we're categorizing our images into 10 classes, we'll have 10 neurons, one for each class. We'll use the variable `number_of_classes` that we saved during pre-processing.

As discussed in Chapter 13, we often use softmax to process the outputs of a final dense layer in a classifier in order to turn them into probabilities. Let's do that here. We need only name it as a string, and Keras will take care of the rest.

Using the standard style of creating and appending the layer in one step, our next line is shown in Listing B2-35.

```
model.add(Dense(number_of_classes, activation='softmax'))
```

Listing B2-35: Adding our second `Dense` layer, which will work as the output layer

Keep in mind that because this layer is fully-connected to the previous layer, each of these 10 nodes receives inputs from all 784 nodes in the hidden layer.

That's the whole thing. We've built a deep-learning model! Listing B2-35 brings it all together.

```
model = Sequential()
model.add(Dense(number_of_pixels, activation='relu',
                input_shape=[number_of_pixels]))
model.add(Dense(number_of_classes, activation='softmax'))
```

Listing B2-36: All the code needed to create our deep-learning model.

That's all there is to it. Our model is complete!

We can ask Keras to print out the model in text form. This isn't terribly revealing for our simple example, but it can come in useful for much

larger models with tens or hundreds of layers. We call the model's `summary()` method, as in Listing B2-37. This printout lists the layers in the order they were placed into the network, so we read it top-down. This summary is rather terse, doesn't include information like the activation functions we've chosen for each layer.

```
model.summary()
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 784)               615440
_________________________________________________________________
dense_2 (Dense)              (None, 10)                7850
=================================================================
Total params: 623,290
Trainable params: 623,290
Non-trainable params: 0
_________________________________________________________________
```

Listing B2-37: Our model summary from Keras

Keras automatically numbers the layers, such as `dense_1` and `dense_2` here. During an interactive session, these numbers will increase over time, so if we build our model again and again we'll see something like `dense_3` and `dense_4`, and so on. Keras gives every layer it builds a unique label so they don't get mixed up if we build our model over and over in a given session.

The column labeled "Output Shape" tells us the shape of the tensor that comes out of each layer, in the form of a list of dimensions. When we see `None` as an entry here, this is a placeholder for the number of samples that are provided as a mini-batch during training. For example, if we have a mini-batch size of 64, then the first layer will process 64 of our samples in one shot (using the GPU if it can). The output will be a list containing 64 rows, each with 784 elements. But since right now Keras doesn't know the size of the mini-batch, it uses `None` to stand for "Not Yet Known."

The summary also tells us how many parameters, or weights, are used by each layer, and then it adds those up to tell us the total number of parameters in the model. We can see that `dense_1`, the first `Dense` layer, has 784 neurons, each of which reads the value of each of the 784 inputs. Since each connection has a weight, there are $784 \times 784 = 614{,}656$ weights. Each neuron also has a bias term, so adding the 784 bias terms to the number we just got gives us the 615,440 in the table. That's a lot of weights! Similarly, the second layer has 10 neurons, each with a connection to each of the 784 neurons in the previous layer. Remembering to add the 10 bias terms, we get $(10 \times 784) + 10$, or 7,850 parameters.

The final line adds these numbers together, telling us that the complete model has over 600,000 parameters.

This is food for thought. Our tiny two-layer model involves well over a half-million weights that need to be adjusted on every update step. Bigger networks can easily have tens or hundreds of millions of parameters. For

example, the VGG16 network we used in Chapter 17 to classify images uses almost 140 million parameters [Lorenzo17]. No wonder the efficient backprop algorithm is so popular, and accelerating it on a GPU is so attractive.

Compiling the Model

This section's notebook continues Bonus02-Keras-2-Making-the-Model.ipynb.

So far, our model is nothing more than a list of specifications. It's a *potential* model, but it's no more a real model than blueprints for a house are a real house. That house has to be built from the blueprints. In our case, we need to turn our description into running code. We call this *compiling* the model. When our model is compiled, it's ready for training.

The act of compiling turns our layer descriptions into code that will run on our computer (and GPU, if available). This is where Keras writes programs for us in TensorFlow. When we train and use our model, we'll be using that code.

To compile the model, we need to give Keras at least two pieces of information.

First, we have to tell Keras how to measure the error for each sample (that is, how to put a number to any difference between the network's output and the target we want it to produce). Second, we have to tell it which optimizer it should use to update the weights to reduce that error. Let's look at these in turn.

To measure the quality of the weights we need a loss (or cost) function. When we discussed backprop in Chapter 14 we used a simple measurement of error based on the differences between the output value(s) and the label value(s). But there are alternatives.

Loss functions are interesting to think about, because they give the "why" of our network. The neurons, dropout layers, activation functions, and so on are the "what" of our network, providing the individual pieces, like the gears in a mechanical clock. The computation of a result, followed by backprop and weight updates, are the "how," like the way the gears of a clock are connected to and propel one another.

But the error function tells us *why* we're doing it all. Is it to find one perfect label? Is it to find three equally-likely labels? Is it to predict a floating-point value? The name for a face in a photo? The best stock to buy tomorrow? A phrase translated from one language to another? Or perhaps it's something more esoteric.

Every neural network has a purpose, and the loss function in some sense defines that purpose, because it drives the whole enterprise. The network's goal is to make the loss, or error, as small as possible. So the loss function is driving the whole show.

Because of their versatility and importance, loss functions can get complicated in a hurry. And that usually means a lot of mathematics.

The good news is that most of the basic things that we will be doing here fall into just a few typical applications, and each one has a ready-made loss function already programmed into Keras for just that job. We need only name the one that was designed for our purpose. Since we're building

a multi-category classifier, rather than, say, a network to perform regression or binary classification, we'll tell Keras to use the pre-built loss function appropriate for a multi-category classifier.

That function will compare the one-hot label with the outputs from our final layer. This comparison uses the idea of *entropy* from Chapter 6 to determine how close our match is. The name of the loss function we want combines these two ideas into the long string `'categorical_crossentropy'`.

If we have just two categories, and we're using one output to decide between them (perhaps setting it to a value near 0 for one category and a value near 1 for the other), the function that evaluates the error for that case is named `'binary_crossentropy'`.

There are a bunch of other error functions, all listed in the Keras documentation, which are useful when doing regression or a variety of other specific tasks. And if the perfect loss function isn't already there, we can write our own in Python and tell Keras to use it instead.

Happily, our goal here is basic categorization using multiple outputs, so we can use the pre-built `'categorical_crossentropy'` loss. That tells the network that we want the network's outputs to match the numbers in our one-hot label as closely as possible.

With the loss function selected, our next job is to pick the optimizer. Once the error has been computed, Keras gives it to the optimizer, which uses that error to update the weights. We saw a variety of optimizers in Chapter 15, with names like SGD, RMSprop, and Adagrad. Once again, they're all implemented for us already, so we only need to tell Keras which one we want it to use by providing its name.

There are many other optional pieces of information we can give to Keras when we compile our model. One of the most common is to provide a list of measurements, called `metrics`, telling Keras what we'd like it to measure as the model learns. We can think of these metrics as supplemental error or loss functions, but they're only computed and returned to us as helpful information for understanding and monitoring the learning process, and are not used to update the model. There are many metrics available to choose from. If we don't see the quantity we wish to measure, we can create a function to compute a custom metric which will be evaluated for us. Though the metrics are always a list, we usually provide a list of just one element, requesting it to record the accuracy, using the string `'accuracy'`.

We compile our model by calling our model's `compile()` method. This builds everything that the model needs to actually run on our computer with TensorFlow. Because this information is saved along with the model object, we don't have to save anything ourselves. When `compile()` returns, the model is ready to learn.

Listing B2-38 shows how to call `compile()` with a loss function, an optimizer, and a list of metrics. In this case we're using the `'categorical_crossentropy'` loss function, which as we discussed above is the appropriate choice for a classification problem with multiple outputs. We've picked the `'adam'` optimizer, just because it's usually a good place to start, and we've specified the common choice of `'accuracy'` for the metrics to be measured once we start learning (note that earlier versions of Keras used the string

'acc' rather than 'accuracy'; for Keras 2 you must use the longer version, 'accuracy').

```
model.compile(loss='categorical_crossentropy',
              optimizer='adam', metrics=['accuracy'])
```

Listing B2-38: Compiling a model with its `compile()` method and our arguments. We're choosing the `'categorical_crossentropy'` loss function and the `'adam'` optimizer. Using these strings is a shorthand for creating the corresponding objects with their defaults. We're also telling it that we'll want it to measure and return the `'accuracy'` once we start training.

Our initial choices of the loss function and optimizer are, as usual, guided by experience. We pick something we hope is reasonable, see how it goes, and then make changes to improve on the performance we get.

If we think we're close but things could be better, we might decide to create a custom optimizer and set some of the parameters to something other than the defaults.

For example, the Keras documentation says that Adam's learning rate argument is called `lr` (a lower-case L and R), and its default value is 0.001. Maybe we have a hunch that a smaller starting value could improve our results. When we create our optimizer using the string `'adam'`, as in Listing B2-38, we're asking for an instance of the Adam optimizer with all of its default values. To set some of those values ourselves, we make our own instance of an `Adam` object where we specify whatever parameters we want to give values to, leaving all the others at their defaults. We then hand that object to `compile()`, instead of giving it a string. Listing B2-39 shows how.

```
from tensorflow.keras import optimizers

slow_adam = optimizers.Adam(lr=0.0001)
model.compile(loss='categorical_crossentropy',
              optimizer=slow_adam, metrics=['accuracy'])
```

Listing B2-39: Compiling our model using a custom object for the `Adam` optimizer

The Keras documentation lists all the optimizers and their instance names, their parameters, and all the defaults.

The loss functions don't take parameters, so unless we're using a custom function that we wrote ourselves, we usually provide a string naming one of the built-in functions.

We've gone through a lot in this section, but it boils down to the one function call of Listing B2-38 (or the more customized version of Listing B2-39). Calling `compile()` with a loss function and optimizer gives Keras enough information to convert our network specification into real code that we can run.

Model Creation Summary

This section's notebook continues Bonus02-Keras-3-Model-Creation-Summary.ipynb.

We've just summited our second mountain.

We started out with how to create a new model. We began by creating an empty `Sequential` object. Then we added a dense, or fully-connected, hidden layer that also specified the shape of the input layer. We finished with another dense layer that produced 10 outputs, one for each category.

Then we compiled our model to turn it from blueprints into reality. We told Keras how to measure the loss, how to update the weights, and what data we'd like it to measure along the way for us.

Putting this all together, Listing B2-40 shows how to create and compile our model. We've merged everything into a little function that returns the compiled model. This way our code can contain multiple models, and we can pick the one we want just by calling the appropriate function. In this summary, we're assuming that we've already run Listing B2-27, so the variables `number_of_classes` and `number_of_pixels` have are available to us (for simplicity, we're using them as global variables, but they could be passed in as parameters).

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

def make_one_hidden_layer_model():
    # create an empty model
    model = Sequential()
    # add a fully-connected hidden layer with #nodes = #pixels
    model.add(Dense(number_of_pixels, activation='relu',
                    input_shape=[number_of_pixels]))
    # add an output layer with softmax activation
    model.add(Dense(number_of_classes, activation='softmax'))
    # compile the model to turn it from specification to code
    model.compile(loss='categorical_crossentropy', optimizer='adam',
                  metrics=['accuracy'])
    return model

model = make_one_hidden_layer_model()   # make the model
```

Listing B2-40: Summarizing how to create and compile our first network

Combining the data-loading and pre-processing steps in Listing B2-27 with the model creation steps in in Listing B2-40 takes us from a blank slate to a model that's ready to learn.

Building our model took only three lines of code. Compiling it took only one. And now we'll see that training the system also takes only one line. But as we've seen, each of these lines packs in a lot of information.

Now we're ready to hand our prepared data to our compiled model and start learning.

Let's start training!

Training the Model

Now that our data is set up for learning, and we have a model defined and compiled, it's time to give the data to the model and let it learn.

This is where a library like Keras really shines. All the machine-learning work of managing the data flow, calculating gradients with backprop, applying weight update formulas, and the rest, is all handled for us.

In a nod to scikit-learn, where we used a routine named `fit()` to train our objects, the Keras training routine is also named `fit()`. This one function call takes our data and model, and runs the entire learning process for us, soup to nuts. We just call it and go get a cup of coffee, or sleep overnight, visit friends for the weekend, or take a vacation for a few weeks, depending on our network, data, and the computing resources available. For our little 2-layer model of Listing B2-40, running on MNIST, a quick break is all that's called for. It takes about 2-3 seconds per epoch on a 2014 iMac, running TensorFlow without a GPU. We'll see that we get good results after 20 epochs, so that's less than a minute.

To keep an eye on the learning process, we can ask `fit()` to print out intermediate progress after each epoch. This lets us see if things are going well, and potentially interrupt the process if the network isn't learning. If we do let it run to completion, `fit()` returns an object of type `History`. This contains all the data that Keras measured after each epoch, such as the model's accuracy and loss. We can use that history to make plots and graphs to visualize the system's performance.

The terminology used by the documentation describing `fit()` deserves a moment's attention.

The training data is now simply called `x` and `y`, though since they're the first two arguments we don't have to explicitly provide those names. The data that's then used to evaluate the system is called the *validation* data, not the test data. The reason for this is that Keras will evaluate our model after each epoch. Thus we hold the test data aside, to evaluate the final model just before deployment. We use the validation set for measuring performance while training.

With the terminology in place, let's look at how we call `fit()`.

The first two arguments, which are both mandatory, are the training samples and the training labels, in that order. As we just saw, they're called `x` and `y`, though following Python convention, these mandatory first arguments are usually not explicitly named when we call `fit()`.

During training, `fit()` will periodically evaluate the model using the validation data. We can choose to either provide that data explicitly, or we can tell `fit()` to extract a validation set from the input data.

If we have our own validation data, we provide the validation samples and their labels in a little 2-element list as the value of the optional argument `validation_data`.

If we don't have our own validation set, `fit()` can make one for us if we give it a value for the argument `validation_split`, in the form of a floating-point number from 0 to 1, telling it what percentage of the training data to use as validation data. This is like using scikit-learn's `train_test_split()` routine, but on the fly. Generally speaking, it's better to provide our own validation data, since we have more control over what it contains.

As we saw in Chapter 15, we typically train models in *mini-batches*. Since we rarely train with the full batch at one time, many people refer to mini-batches as simply "batches." Keras does this as well, using parameter names like `batch_size` for what is more properly a "mini-batch size." Since using "batch" for "mini-batch" is so common, we'll use that language here as well.

When learning in batches, `fit()` will pull off a batch-sized chunk of samples from our training set, learn from it, update the weights, and then take another chunk. It's our job to tell `fit()` how big those chunks should be with the optional argument `batch_size`. This argument defaults to 32, but we can set it to any value we like. If we're using a GPU, we typically set this to a power of 2 (like 32 or 128) that makes our data fit best into the GPU we're using, so it can process an entire batch in one parallelized operation. When we're training on a CPU only, we often use a larger batch size, perhaps even a few hundred samples, since our computer has more memory available.

In this chapter we'll be demonstrating results without a GPU, so we'll usually use a pretty big batch size like 256.

Another important argument is how many epochs the training should run for. Recall that one epoch means one complete pass through the training set (taken in batches, as above). That is almost never enough to train the system fully, so then the system runs through all the data again, for another epoch, repeating the process over and over. The downside of telling `fit()` how many epochs to use before we've even started training is that we could be wildly off. Maybe we need far more epochs than we ask for, so we end up stopping training too soon, or we pick a number far larger than we need, wasting a lot of time training a network that's no longer learning (or worse, overfitting). We will see cures for both problems later. For now, we'll just pick a number and hope that it's about right. The name of the argument is `epochs`, short for "number of epochs." We'll pick 3 just to make sure everything's working right, and then crank that number up later.

The last argument we'll use is `verbose`, which tells the system how chatty to be after each epoch (grammatically, we might prefer "verbosity" for the name of this argument, but `verbose` it is). If we set this to 0 it doesn't print out anything. A value of 1 prints an animated progress bar that shows the system chugging its way through the samples in each epoch. A value of 2 just shows a single summary line of text after each epoch.

Let's train our model with our own validation data, for 3 epochs. Since we're training on a CPU, we'll use a large batch size of 256 samples per batch. This will give us smoother graphs when we plot our data, compared to the results for the smaller batch sizes we'd usually use if we were training on a GPU. We'll set `verbose` to 2 so that we'll get a line of information after each epoch. Listing B2-41 shows the one line that does it all.

```
# call fit() to train the model, and save the history
history = model.fit(X_train, y_train, validation_split=0.25,
              epochs=3, batch_size=256, verbose=2)
```

Listing B2-41: Finally, we're training our model!

When we enter this line, the system will start to train. With only 3 epochs, this should run in well under a minute on most any modern computer.

This is the top of the third mountain! Let's put it all together.

Training and Using Our Model

This section's notebook is Bonus02-Keras-4-Train-and-Run.ipynb.

We've reached the peak of the final mountain. Starting from scratch, we've got a (barely) trained neural network for classifying MNIST digits.

This is a good time to pause, enjoy the view, and look back on how far we've come.

Listing B2-42 combines the pre-processing of Listing B2-27, the model building of Listing B2-40, and the training of Listing B2-41 into one place.

In a scant 53 lines, including comments and blank spaces, this code starts with nothing, gets our data, pre-processes it, builds and compiles a deep-learning model, and then trains it for 3 epochs.

```
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.import backend as tensorflow.keras.backend
from tensorflow.keras.utils import to_categorical
import numpy as np

random_seed = 42
np.random.seed(random_seed)

# load MNIST data and save sizes
(X_train, y_train), (X_test, y_test) = mnist.load_data()
image_height = X_train.shape[1]
image_width = X_train.shape[2]
number_of_pixels = image_height * image_width

# convert to floating-point
X_train = keras_backend.cast_to_floatx(X_train)
X_test = keras_backend.cast_to_floatx(X_test)

# scale data to range [0, 1]
X_train /= 255.0
X_test /= 255.0

# save the original y_train and y_test
original_y_train = y_train
original_y_test = y_test

# replace label data with one-hot encoded versions
number_of_classes = 1 + max(np.append(y_train, y_test))
y_train = to_categorical(y_train, num_classes=number_of_classes)
y_test = to_categorical(y_test, num_classes=number_of_classes)

# reshape samples to 2D grid, one line per image
```

```
X_train = X_train.reshape([X_train.shape[0], number_of_pixels])
X_test = X_test.reshape([X_test.shape[0], number_of_pixels])

def make_one_hidden_layer_model():
    model = Sequential()
    model.add(Dense(number_of_pixels, activation='relu',
                    input_shape=[number_of_pixels]))
    model.add(Dense(number_of_classes, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam',
                  metrics=['accuracy'])
    return model

# make the model
one_hidden_layer_model = make_one_hidden_layer_model()

# call fit() to train the model, and save the history
one_hidden_layer_history = one_hidden_layer_model.fit(X_train, y_train,
    validation_split=0.25, epochs=3, batch_size=256, verbose=2)
```

Listing B2-42: Combining the pre-processing of Listing B2-27, the model building of Listing B2-40, and the model training of Listing B2-41 into one place

Looking at the Output

The final line of Listing B2-42 actually trains our model, learning from the training data in `X_train`. Listing B2-43 shows the summaries we asked it to print after each epoch. These numbers may be slightly different on different computers, but they should be pretty close.

```
Epoch 1/3
2s - loss: 0.3467 - accuracy: 0.9020 - val_loss: 0.1857 - val_accuracy: 0.9489
Epoch 2/3
1s - loss: 0.1473 - accuracy: 0.9577 - val_loss: 0.1307 - val_accuracy: 0.9629
Epoch 3/3
1s - loss: 0.0970 - accuracy: 0.9725 - val_loss: 0.1110 - val_accuracy: 0.9684
```

Listing B2-43: The output of Listing B2-42. Note that different backends may produce slightly different values.

The system prints out `Epoch 1/3` when it starts the first epoch, and prints the summary line when it has run through every sample in the epoch. The first piece of information is the time consumed. Here, it took about 3 seconds (again, on a CPU only) to train our simple model on every sample in the training set (that is, one epoch). The system then prints out the loss and accuracy (`loss` and `acc`) for the training set. Unfortunately, these are not explicitly labeled as being for the training set. But we can see that the next two results are for the validation set (`val_loss` and `val_acc`), so that helps remind us that the unlabeled versions are for the training data.

How'd we do? At a glance, the results for the training data look promising. The test loss is dropping after each epoch, and the test accuracy is improving. This suggests that we have everything wired up sensibly, and that the system is learning.

A moment of exuberance would not be out of place.

The validation data also looks good. Again, the loss is dropping every epoch, and the accuracy is climbing. After just 3 epochs of training, it's already up to over 97% accuracy! That's not nearly as good as the best scores anyone's found [LeCun13], but it's pretty amazing that with a tiny network containing just one hidden layer, after only 3 epochs of learning, and no tuning at all, we are recognizing handwritten digits correctly 97.25% of the time!

Training for 3 epochs isn't really enough for almost any network, even one this simple. Let's run this for 20 epochs and see how it does. All we have to do is change the argument to `epochs` to 20 and let it go; see Listing B2-44.

```
history = model.fit(X_train, y_train, validation_split=0.25,
              epochs=20, batch_size=256, verbose=2)
```

Listing B2-44: Finally, we're training our model for real! We're giving it 20 epochs to learn.

The first few and last few lines from the output of Listing B2-44 are shown in Listing B2-45.

```
Epoch 1/20
176/176 - 2s - loss: 0.3404 - accuracy: 0.9057 - val_loss: 0.1838 -
   val_accuracy: 0.9473
Epoch 2/20
176/176 - 1s - loss: 0.1436 - accuracy: 0.9591 - val_loss: 0.1309 -
   val_accuracy: 0.9622
Epoch 3/20
176/176 - 1s - loss: 0.0944 - accuracy: 0.9733 - val_loss: 0.1158 -
   val_accuracy: 0.9655
Epoch 4/20
176/176 - 1s - loss: 0.0674 - accuracy: 0.9811 - val_loss: 0.1005 -
   val_accuracy: 0.9699
Epoch 5/20
176/176 - 1s - loss: 0.0511 - accuracy: 0.9858 - val_loss: 0.0899 -
   val_accuracy: 0.9713
```

Listing B2-45: The start and end of the output from Listing B2-44

The output in Listing B2-45 looks fantastic. Our score on the training set is 100% accuracy. That's perfection!

The testing set score is not perfect, but it's very respectable for such a simple model. Our model is misclassifying only 287 out of all 10,000 test samples.

As we mentioned earlier, one of the nice things about using image data is that we can look at it. Let's look at some examples that were misclassified.

In Figure B2-25 each row shows images whose given label is that row's number. That is, every image in the top row was originally assigned the label 0 in the data set, every image in the second row was originally assigned the label 1, and so on. But here we're showing images that were classified incorrectly by our network. The column shows the label that the

system assigned to that image. For example, in the top row, there's a picture in the fifth position. It's in the top row, so the MNIST data tells us that this should be a 0, but it's in the fifth column, so our system predicted that this was a 4. That seems like a pretty odd mistake.

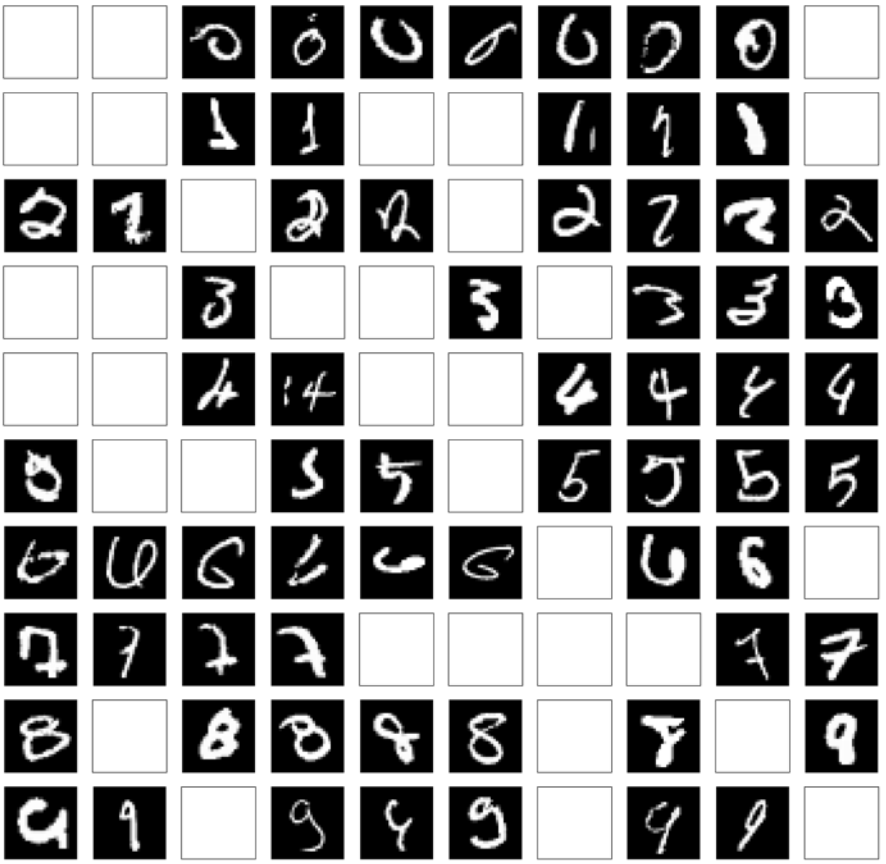


Figure B2-25: A visualization of the test-set images whose predicted value did not match their label

The top row of Figure B2-25 contains images that were labeled 0 in the original MNIST data. The next row contains images that were labeled 1 in the original data, and so on. Each column tells us what label our network assigned. For example, the top row shows images that should all have been labeled 0. Empty boxes mean no images fell into that position. The image shown in each position is randomly selected from all the images that belong to that cell.

A more sensible mistake can be seen in the third row. The second image from the left was labeled a 2 in the data, but our system categorized it as a 1. It's hard to tell what it should be. In the sixth row, the first entry was labeled a 5 in the data, but our system called it a 0. That doesn't seem unreasonable.

While some of these errors seem surprising (the leftmost 8 in Figure B2-25 seems pretty clearly an 8 and not a 0), it's important to keep in mind that these errors are rare. Out of 10,000 test images, the system only disagreed with the given labels 206 times.

Figure B2-26 shows a "heat map" of our errors, where each cell tells us how many images fell into that cell. The range runs from black (for no images at that cell) through reds and then yellows to white.

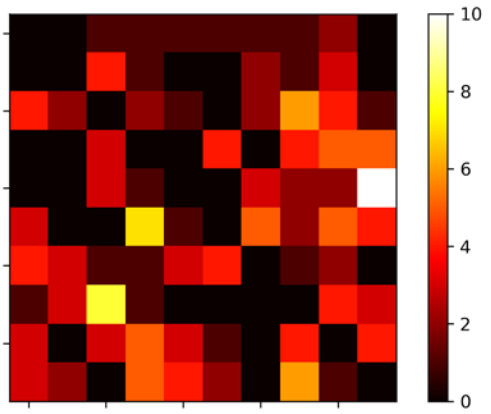


Figure B2-26: A heat map of the population of the error visualization of Figure B2-25, telling us how many images landed in each of the cells. Black indicates an empty list, while brighter reds, then yellows, and finally white represent longer lists.

The white box in the 9th position of the 5th row shows that on this training run, the biggest mistake the system consistently made was with digits that had been labeled as a 4, but which the system identified as a 9. Out of the 10,000 digits in the validation set, 9 images were misclassified this way. Other common mistakes were digits labeled as 5 being called 3, and digits labeled 7 being classified as 2.

Prediction

This section's notebook is Bonus02-Keras-5-MNIST-Photo-Prediction.ipynb.

Our validation data is getting some impressive numbers, but what if we looked at some digits that weren't made by Census Bureau workers or high-school students?

Let's take the model we just trained, and deploy it. We'll give it some new images that it's never seen before, and see how it does.

Figure B2-27 shows four photographs taken on a winter's day in the Seattle area. We have a sign in a coffee-shop window, some spray-painted marks on the ground near a construction site, a number painted onto the side of a dumpster, and a parking-lot stall number.

Figure B2-27: Four photos from the Seattle area. Left to right: a sign in a coffee shop window, spray-painted marks on the ground near a construction site, a building number on the side of a dumpster, and a 3 digit parking stall number.

The stenciled numbers in the parking lot stall are not hand-drawn, and they have gaps, so they're really not appropriate for our system. They're included just for fun, and to see what our deep-learning system comes up with.

When we extract these digits, rotate them to be upright, and prepare them in the same way as the original MNIST data [LeCun13], we get Figure B2-28.

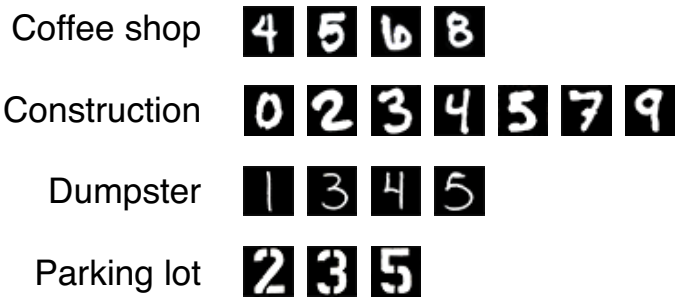


Figure B2-28: Extraction of the digits from Figure B2-27. Each digit has been rotated to be upright and then processed like the original MNIST images.

How well will our system do? It's important to emphasize before we get into this that this is not a fair test. The test set has 10,000 images for a good reason, but here we've only got 18 images. This is far too small a sample set to have any statistical validity. Even worse, the parking lot images are not hand-drawn, and they each have prominent gaps. So all we're going to get here is some anecdotal evidence, rather than anything we can use to reliably characterize our system's performance. That, after all, is exactly what the test set is for. Nevertheless, they make a fun and interesting test, and along the way we'll see how to ask our model for predictions, so let's dig in.

Just to clarify which set we're working with, let's make four test sets, one for each group of images. For instance, we'll arrange the coffee shop data into a grid that has 4 rows and 784 columns, as in Figure B2-29.

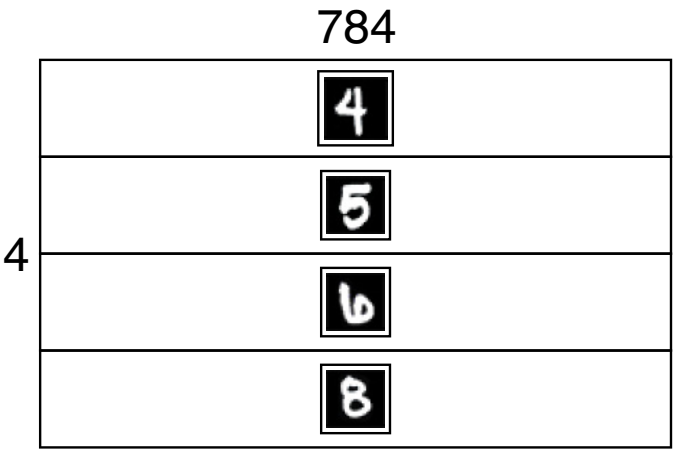


Figure B2-29: Arranging the four images of our coffee shop data into a 2D grid, one row per image.

The construction data has 7 rows, the dumpster data has 4 rows, and the parking lot data has 3 rows.

As always, we need to pre-process our data. We've already got it into a 28 by 28 shape, but that's not enough. Just like the MNIST data, we need to convert the input pixels into the current Keras floating-point form, and then *we must apply the same pre-processing that we applied to the training data we used to train our model.* So we'll use `cast_to_floatx()` again to get the data into the right type, and then we'll divide every pixel by 255, just as before. The processing step for the coffee shop data is shown in Listing B2-46.

```
CoffeeShopDigits_set = keras_backend.cast_to_floatx(CoffeeShopDigits_set)
CoffeeShopDigits_set /= 255.0
```

Listing B2-46: Pre-processing of the coffee shop images. We set them to the current floating-point type, and then use the same normalizer *we used when training.*

Now we're ready to give these images to the model and ask it to identify, or predict, each digit. We're testing our deep-learning system on new data!

To get a prediction, we hand one or more samples to our model and call its `predict()` method. For each sample, we'll get back the outputs from the final layer. In our case, this means a list of 10 values that come out of the softmax step after the last dense layer. Listing B2-47 shows the code.

```
coffee_probs = model.predict(CoffeeShopDigits_set)
coffee_probs.shape
(4,10)
```

Listing B2-47: We give our model a set of samples, and ask for predictions with `predict()`. The result is an array with one row for each sample, and one entry in that row for each class.

In Listing B2-47 we've printed out the dimensions of the probabilities for the coffee shop data. As expected, there are four rows (since there are four images in the data), and each row holds 10 numbers, one for each class.

We wrote a little loop to print out these values, abbreviated to just two significant digits so that we can see them all easily. Listing B2-48 shows the output.

```
digit 0 probabilities: 0.00 0.00 0.00 0.00 0.22 0.00 0.00 0.00 0.01 0.78
digit 1 probabilities: 0.00 0.00 0.00 0.00 0.00 1.00 0.00 0.00 0.00 0.00
digit 2 probabilities: 0.00 0.00 0.00 0.00 0.00 0.00 1.00 0.00 0.00 0.00
digit 3 probabilities: 0.18 0.00 0.00 0.00 0.00 0.00 0.01 0.00 0.80 0.01
```

Listing B2-48: The values of `coffee_probs` printed out neatly.

Figure B2-30 shows this data graphically.

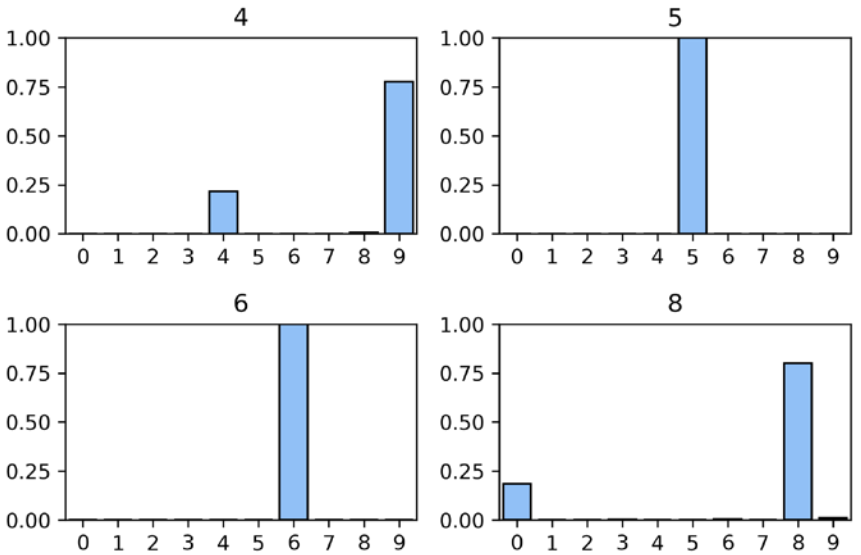


Figure B2-30: Plots of the probabilities from Listing B2-2. The system was pretty sure the first digit was a 9, but thought it might also be a 4. It was very sure of the 5 and 6, but a little less confident about the 8.

Figure B2-30 shows us that the system got the first digit wrong (though it did give a 22% chance to the correct result of 4). It was all but certain about the 5 and 6, and pretty sure about the 8 (though there was some chance of it being a 0).

So our network correctly classified three of the four images, which isn't too bad. In its defense, the 4 does look a lot like a 9 with the top left open a little.

Looking at all the data in Listing B2-50 or Figure B2-30 gives us a full picture of what the network predicted, and it lets us see the difference between when it was unsure versus when it was essentially certain. But sometimes we need just one answer, and in those cases we usually select the class with the highest probability.

We can get that with the Numpy function `argmax()`. This looks through a list and returns the index of the entry with the largest value. We'll tell it to search horizontally rather than vertically by giving the optional `axis`

argument the value `1` (rather than `0`, which would cause it to search column by column, rather than row by row). Listing B2-49 shows the call, and the result.

```
print('CoffeeShop:', np.argmax(coffee_probs, axis=1))
CoffeeShop: [9 5 6 8]
```

Listing B2-49: The index of the largest value in each row of `coffee_probs` tells us the digit with the highest probability.

We can run this on each of the other three data sets, getting the output in Listing B2-50.

```
Construction: [0 2 3 4 5 3 8]
Dumpster: [1 3 4 5]
Stencil: [2 3 5]
```

Listing B2-50: The highest probability results for the other three small data sets.

They're all correct, except for the last two from the construction photos.

The parking-lot stencil digits are not hand-drawn, and they all have multiple gaps. This isn't the sort of thing the model was trained on at all, so there's no reason to think it would interpret these images well. Yet it nailed all three digits. We'd want to see a few hundred more stencil results at least before making any claims, but it's encouraging that it got these three correct.

Our tiny model with just two layers, and just 20 epochs of training, did a great job, correctly classifying 15 out of 18 of our images. Again, this was a totally unfair test, because we gave the system data unlike what it was trained on. But in the real world, people frequently give systems unexpected variations on the data they were trained on.

It's good to stress test a system in this way (though of course on a larger scale) to get a sense of whether it's robust in the face of these kinds of unexpected inputs, or brittle and error-prone. That knowledge can help guide how we deploy it and make it available to others.

Analysis of Training History

This section's notebook is Bonus02-Keras-6-MNIST-Training-History.ipynb.

Our system seems to be doing pretty well, particularly for something so simple.

We mentioned before that `fit()` returned some history information that tells us how the training went. Let's investigate that now and see what we can learn.

To gather lots of data, this time we'll train for 100 epochs, even though we know the system hits 100% on the training data after just 20 epochs.

The history information is returned by `fit()`, so we can just assign the output of that method to a variable, as in Listing B2-51.

```
one_hidden_layer_history = model.fit(X_train, y_train,
    validation_split=0.25, epochs=100, batch_size=256, verbose=2)
```

Listing B2-51: Saving the history returned by `fit()`

Here we're saving the history in a variable we've given the name `one_hidden_layer_history`. This contains a bunch of fields that summarize the training process (like how many epochs it ran for, and what parameters we used). The field that's most interesting to us right now is called `history`. It's a Python dictionary object that contains the accuracy and loss values for both the training and validation sets after each epoch.

The training accuracies are in this dictionary as a list stored under the key `'accuracy'`, so we'd get them from `one_hidden_layer_history` with the expression `one_hidden_layer_history.history['accuracy']` (that's a lot of typing!). The training loss uses the key `'loss'`. Similarly, the validation accuracy and loss are stored with the keys `'val_accuracy'` and `'val_loss'`.

Note that as in many other places in Keras, information related to the training set has no prefix, so those lists use the keys `'accuracy'` and `'loss'`. Information related to other data sets is prefixed by a descriptor, so here we have validation data saved with the keys `'val_accuracy'` and `'val_loss'`.

Using the list of numbers retrieved by using each of these keys, Figure B2-31 plots the accuracy and loss of our training data graphically.

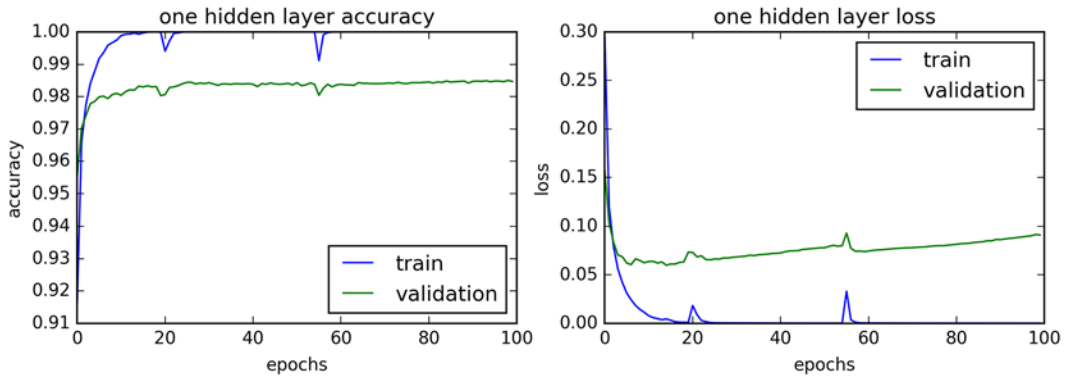


Figure B2-31: The accuracy and loss of our one-layer network plotted against the number of epochs

There are some surprises here.

First, it's worth noting the scales of the data. The accuracy graph *begins* at about 0.91 (and tops out at 1.0). That means that after just one epoch of training, our system was up to 91% accuracy. That's far from perfect, but it's pretty amazing for such a tiny network and one epoch of training. The loss plot has a correspondingly small range, from 0 to just 0.3.

Both graphs show some spikes. This is probably due to a time when the samples arrived in just the right order so that some systematic errors were able to accumulate. The system righted itself nearly immediately in both cases.

The training loss quickly drops to 0 by about the 20th epoch, and except for the spikes, it stays there. But the validation loss is slowly increasing. In other words, the training loss and validation loss are *diverging*. This is a picture of *overfitting*. As we discussed in Chapter 9, overfitting means that the system has learned how to identify the training set by honing in on its idiosyncrasies, not its general principles.

Learning during overfitting is actually reducing our performance on the validation data, as the system fruitlessly learns more and more about the training set, sharpening its rules and memorizing details. This is a complete waste of effort, and it comes at the expense of losing generality, with each epoch causing even more harm to the network's accuracy on new data. Though it doesn't look like the accuracy is dropping in these graphs, the increasing validation loss suggests that that time may come, if we kept on training.

To prevent this overfitting, we might be tempted to stop training where the loss or accuracy curves cross one another, but this would be too early. The validation accuracy is still improving, and the validation loss is still generally dropping. The best place to stop would be when our validation loss or accuracy stop improving. That is, when the loss starts to increase or the accuracy starts to drop. Of these two choices, we usually use increasing loss on the validation set as our trigger to stop training.

Below we'll see how to detect that situation automatically, and stop training at that point. This will help us avoid overtraining our model. It will also solve our problem of having to guess the right number of epochs to train for, and hope we don't guess either too high or too low. We'll just pick a huge number, and let the system stop itself when it starts to overfit.

Before we get into that, let's see how to save and load our hard-won trained networks to a file. After all, if we've spent hours (or days or weeks) training a model, we'd surely like to be able to save all of those precious weights. Then the next time we want to use that model we can just load the weights from a file and our fully-trained model will be ready to go, and we won't have to teach it all over again from scratch.

Saving and Loading

This section's notebook is Bonus02-Keras-7-Save-and-Load.ipynb.

After we've gone to all the trouble of training a model, we certainly want to save it so we can use it again later. There are several options.

Saving Everything in One File

The easiest way to save our model and weights is to call a built-in method belonging to our object that tells it to write itself to a file. The method is, sensibly enough, called `save()`. When we call this method, the model will write a file that contains both its architecture and weights.

The model is saved in a format called HDF5, which conventionally uses the extensions `.h5` or `.hdf5` [HDF517]. We can save our model with just one line, as in Listing B2-52.

```
model.save('my_model.h5')
```

Listing B2-52: Saving a complete version of our model to a file

Later, we can read this file back in with the `load_model()` function. Unlike `save()`, we need to import a new Keras module to access `load_model()`. That's because when we load a model, we might not yet have an object whose methods we can call.

Suppose we want to load the model that we saved in Listing B2-54. We can do it with Listing B2-53.

```
from tensorflow.keras.models import load_model

model = load_model('my_model.h5')
```

Listing B2-53: Loading the complete version of our model from a file

Just like that, the `model` variable now contains a complete version of the model we saved, with all the weights we'd learned as of the time the file was written.

We can now use that model to predict new results. Because Keras also saves the state of the optimizer in the file, if we want to train the model some more, we can just pick up training from where we left off.

Saving Just the Weights

If we only want to save the weights (probably to save a little space on our hard drive), the method `save_weights()` will do the job, as in Listing B2-54.

```
model.save_weights('my_model_weights.h5')
```

Listing B2-54: Saving just the weights to a file.

If we want to use these weights later, then we have to first build a model to receive them. The most common case is when our model has the same architecture as the model we used to save the weights. Then the weights just pour right back in to where they had been, as shown in Listing B2-55.

```
# create a model just like the one we saved the weights from
model = make_model()  # a pretend function to make our model

# now read the weights back from a file and fill up the model
model.load_weights('my_model_weights.h5')
```

Listing B2-55: Loading the weights only

Saving Just the Architecture

Saving both the model and its weights is the most convenient way to save our work, since we have everything we need in one place. Saving just the weights is useful if we want to share our trained model with people using different libraries that aren't set up to read the Keras architecture information.

Much less frequently we'll want to save just the architecture without the weights.

If we need to save just the architecture of the model, Keras supports two different formats: JSON [JSON13] and YAML [YAML11]. These formats are both designed to save data structures to text-only files. YAML is a superset of JSON, meaning that it can do everything JSON can do and more, but if we're just saving and loading a model architecture that extra power is moot. Since both standards are text-based, so it's easy to open and read files in either format with a text editor if we want.

The technique for saving an architecture in both cases is to use Keras to convert the model into a big character string, and then write that string to a file.

To get the architecture back, we read the string from the file, and then use Keras to turn the string into a model.

To turn a model into a YAML string, we use the `to_yaml()` method that is part of the model. Then we can write that to a file, as in Listing B2-56.

```
import yaml

filename = 'my_model_arch.yaml'

yaml_string = model.to_yaml()
with open(filename, 'w') as outfile:
    yaml.dump(yaml_string, outfile)
```

Listing B2-56: Saving our model architecture, without weights, as a YAML file.

To read our architecture back, we can use Listing B2-57.

```
import yaml
from tensorflow.keras.models import model_from_yaml

filename = 'my_model_arch.yaml'

with open(filename) as yaml_data:
    yaml_string = yaml.load(yaml_data, Loader=yaml.FullLoader)

model = model_from_yaml(yaml_string)
```

Listing B2-57: Reading our model architecture, without weights, from a YAML file.

Using Pre-Trained Models

The ability to save and load our models is useful when we're developing and testing models of our own. But it also allows us to build on the work of others.

Some deep learning models can have dozens of layers, and may have been trained for days or weeks on mountains of data that we don't have access to. But if the authors of the model have released the structure and weights, then we can instantly use their model and all the hard work that went into it. That's just what we did when we used the VGG16 model in Chapters 16 and 17.

We often *fine-tune* these *pre-trained models* by training them on our own data, helping them specialize on the tasks we need to do. This is sometimes called *transfer learning* [Karpathy16].

We might even modify the architecture, such as by adding a few layers of or own to the end of the pre-trained model. We "protect" the existing model by telling Keras not to change their weights during training. We say that such layers are *frozen*. This means that only our new layers get updated weights as we train.

To freeze a layer, we set the layer's optional parameter `trainable` to `False`. We can later "thaw" a frozen layer by setting this parameter to `True` and compiling it again.

An alternative to adding more layers to the end of a model is to freeze all but the last few layers. We typically then train the model with our new data with a very small learning rate. The idea is that we're just tweaking, or fine-tuning, the weights that came with those layers so that they're more amenable to our data [Gupta17].

A list of pre-trained models in Keras can be found in the documentation at *https://keras.io/applications/*.

Saving the Pre-Processing Steps

We've seen how to save the architecture, the weights, and both combined. But as we know, any time we use a model we must pre-process our new data in the exact same way that the training data was processed.

For example, in our pre-processing of MNIST data in Listing B2-20, we divided all of our pixel data by 255. In the VGG16 model we used in Chapter 17, the color images used as samples must be pre-processed by subtracting a specific number from every channel of every pixel [Lorenzo17].

The key point is that in order to properly use a saved network, we want to also save and load the data pre-processing steps, so that we can apply them to new data.

Unfortunately, as of Keras 2, there's no standard way to do this. Part of the problem is that we can do the pre-processing any way we like. We might use a library function, or a function of our own, or we could just explicitly modify the data, the way we did when we divided it by 255. Without some kind of standard, there's no way to capture those kinds of operations.

The general solution is to document our pre-processing steps as well as we can. That usually means writing comments into the code or in a text file, and then try to make sure that the description stays with the model somehow. We also have to figure out how to alert people that it's there, and encourage them to read it.

It's a messy situation.

But it's a situation that we must address somehow, because we need to apply the same pre-processing that we used on our training data to any new data. Unfortunately, at the moment we must manage the documentation and implementation of sample pre-processing on a case-by-case basis.

The important thing to remember is that whether we're sharing a trained model of our own, or using someone else's, we need documentation on how

the training data was pre-processed. As authors, it's our job to write that documentation and make it available in some reasonable format. As adopters, it's our job to find that information and follow it when preparing our own data.

Callbacks

This section's notebook is Bonus02-Keras-8-Callbacks.ipynb.

Now that we can save our models, let's get back to the issue of bringing our training to a halt when the validation loss starts to climb and we begin to overfit.

Recall that the `fit()` function runs the data through one batch at a time, for epoch after epoch.

After each epoch it computes values such as loss and accuracy, as well as the values we asked for in the `metrics` argument. It also consults a list of *callback* procedures that we supply. Keras then calls each of those procedures for us, and they can do anything we want.

We tell Keras what functions to call by handing them to `fit()` as the value of an optional argument called `callbacks`. These callbacks can be a combination of functions we've written ourselves, and functions built into Keras.

In this section, we'll focus on three of the callbacks provided to us by Keras: one to *checkpoint* (or save the weights), one to control the *learning rate* over time, and one to perform *early stopping* (or cease training when we appear to start overfitting).

Checkpoints

A popular use for callbacks is to *checkpoint* our model during training. This means saving out the model (or, if we prefer, just the weights) to a file. We can save a checkpoint after every epoch if we like, but usually we only do this after every few epochs.

Having checkpoints means that if we're training a system that takes hours or days, and we lose power or for any other reason the training stops, we can pick up again by loading the most recently saved model file.

To tell Keras to make checkpoints we'll create a `ModelCheckpoint` object, and then hand it to Keras when we call `fit()`.

The first argument to `ModelCheckpoint`, which is mandatory and unnamed, is the path to the file that will be written. This file is in the HDF5 format, so we typically give it an extension of either `.h5` or `.hdf5`.

This filename is special, because it can include Python string-formatting instructions that include values for variables that Keras knows about. It always keeps track of the epoch, so a string like `{epoch:03d}` means that the braces and everything between them will be replaced by a 3-digit decimal number holding the current epoch.

We can tell Keras to include one other value of our choice. By default, that value is `val_loss`, or the loss on the validation set. So to include that value in the name of the output file, we can use a string like `{val_loss:0.3f}`.

In this case the fragment will be replaced with a 3-digit floating-point value of the current loss (when that value is less than 1, Python inserts a `0.` at the start for us).

A typical filename is in Listing B2-58. Here we're placing our files into a pre-existing folder named `SavedModels`.

```
filename = 'SavedModels/model-weights-{epoch:02d}-{val_loss:.03f}.h5'
```

Listing B2-58: A filename that Keras will use for checkpointing. It will include the epoch number and validation loss with the given format when the file is created.

This will create checkpoints with names like those in Listing B2-59.

```
model-epoch-000-val_loss-0.156.h5
model-epoch-001-val_loss-0.102.h5
model-epoch-002-val_loss-0.080.h5
model-epoch-003-val_loss-0.072.h5
```

Listing B2-59: Names of the first few checkpoint files written out using the filename of Listing B2-58

To get Keras to make these files, we need to create the function that builds and saves them. We do this by making an instance of the built-in `ModelCheckpoint` object. It takes one mandatory argument providing the filename, formatted as we just saw. Listing B2-60 shows how we build this object, leaving all of its other options at their default values.

```
checkpointer = ModelCheckpoint(filename)
```

Listing B2-60: Creating an instance of `ModelCheckpoint` with our desired filename

The only thing left is to provide this object to Keras when it trains the model. In our call to `fit()`, we include the optional argument `callbacks`, which expects a list of callback objects. Since we just have this one, we'll wrap it up in square brackets to make a list that's just one element long. Using our call to `fit()` from the previous section, our call using checkpointing is shown in Listing B2-61.

```
history = model.fit(X_train, y_train,
                    validation_split=0.25,
                    epochs=100, batch_size=256, verbose=2,
                    callbacks = [checkpointer] )
```

Listing B2-61: Calling `fit()` with our single-element list of callbacks

There are some useful options to `ModelCheckpoint` that can make it more useful.

Writing out the complete model after every epoch may take up more disk space (and computer time)

than we want to use. We can cut down on the size of the file by saving just the weights. To do this, set the optional argument `save_weights_only` to `True` (the default is `False`, so every files contains both the architecture and the weights).

We might not need even the weights written out after every epoch. We can tell it to write out a file only periodically by setting the optional argument `period` to some value (the default is 1, meaning the file is written after every epoch). For example, if we set `period` to 5, then the file is only produced every 5th epoch.

By default, the value that Keras can insert into the file name is the validation loss, `val_loss`. But we can ask it to use the validation error `val_err`, the training loss `loss`, or the training error `err`. We just use the name we want in the checkpoint file.

For example, we can save the training accuracy by setting up the filename as in Listing B2-62.

```
filename = 'SavedModels/model-weights-epoch-{epoch:03d}-' + \
           'accuracy-{accuracy:0.3f}.h5'
checkpointer = ModelCheckpoint(filename, monitor='accuracy',
                               save_weights_only=True, period=10)
```

Listing B2-62: Saving the training accuracy in the checkpoint file name

When we run the code, we'll get filenames like those in Listing B2-63.

```
model-weights-epoch-009-accuracy-1.000.h5
model-weights-epoch-019-accuracy-1.000.h5
model-weights-epoch-029-accuracy-1.000.h5
```

Listing B2-63: File names created by Listing B2-63

We can easily accumulate a lot of these checkpoint files in a long training run. We can tell `ModelCheckpoint` to only write a new file if some measurement is better than that in any previously-saved version. We do this by setting two parameters.

First, we tell it that we want this mode by setting `save_best_only` to `True` (the default is `False`).

Second, we tell it which parameter it should use to determine if this epoch's results are "better" than any that has been already saved. As usual we can choose from the training accuracy `'accuracy'`, training loss `'loss'`, validation accuracy `'val_accuracy'`, and validation loss `'val_loss'`. We pass the variable we want it to keep track of using the optional parameter `monitor` (the default is `'val_loss'`).

The system knows that the best loss is the smallest one, and the best accuracy is the largest one.

For example, to only write a new file if it has better validation accuracy than any that have come before, we could use Listing B2-64.

```
checkpointer = ModelCheckpoint(filename,
                               save_best_only=True, monitor='val_accuracy')
```

Listing B2-64: Saving only the checkpointing file with the best validation accuracy

When the training is complete, the most recently-written file will be the one corresponding to the best value of the validation accuracy over the whole training run. Note that since we're only printing three digits of the

value, it might not be obvious that the accuracy has improved. For example, if it goes from 0.9353 to 0.9354, both files will list the accuracy in the file name as 0.953. By looking at the time stamps of the files, we can infer that the more recently written one is better.

Learning Rate

Another popular use of callbacks is to change the *learning rate* over time. As we saw in Chapter 15, many modern optimizers automatically adjust the learning rate adaptively (they usually have names that begin with "Ada" for "adaptive learning rate"). But if we choose to use something like SGD, then we'll need to manage the learning rate ourselves.

In Chapter 15 we saw a variety of strategies for adjusting the learning rate over time. For example, we might start with a large learning rate, and then shrink it either on every epoch, or in stair step fashion after each group of some fixed number of epochs. To pull off these strategies, or any others we might prefer, we use the built-in callback routine named `LearningRateScheduler()`

The `LearningRateScheduler` callback is really just a little connection function between Keras and a function that we write. The `LearningRateScheduler` calls our function, and it returns the value that our function returns. The function we write must take one argument: an integer with the epoch number that just finished as an input (this starts at 0). It must return a new floating-point learning rate as an output.

Listing B2-65 shows the idea. We start by compiling the model with the non-adaptive SGD optimizer. We've written a little scheduling routine that we've called `simpleSchedule()`. If we wanted to use checkpointing here as well, we could create a `ModelCheckpoint` object as in the last section, and include it in the list we provide to `callbacks`. The order in which the callback routines are named in this list makes no difference.

```
from tensorflow.keras.callbacks import LearningRateScheduler
from tensorflow.keras.optimizers import SGD

# make the model but don't compile it
model = make_model()

sgd = SGD(lr=0.0, momentum=0.9, decay=0.0, nesterov=False)
model.compile(loss='categorical_crossentropy',
              optimizer=sgd, metrics=['accuracy'])

def simpleSchedule(epoch_number):
    return max(.1, 1-(0.01*epoch_number)) # start at 1 and drop to 0.1

lr_scheduler = LearningRateScheduler(simpleSchedule)

history = model.fit(X_train, y_train, validation_split=0.25,
                    epochs=100, batch_size=256, verbose=2,
                    callbacks=[lr_scheduler])
```

Listing B2-65: Setting up and using a learning-rate scheduler

Early Stopping

Another popular use of callbacks is to implement *early stopping*. Recall from Chapter 9 that this involves watching the performance of our network and looking for signs of overfitting. When we see overfitting, we stop training.

So we stop "early" in the sense that we probably would have kept going if not for this intervention, but in fact we're stopping at the right time to prevent overfitting.

The built-in routine provided by Keras implements this idea by monitoring a statistic of our choice. When that value stops improving, it stops training.

Early stopping is often used with checkpointing. We might tell our system to train for a ridiculous number of epochs, like 100,000 of them, and then go to lunch (or to sleep), leaving the computer to run, checkpointing the model every few epochs (or saving the best one according to some measurement). We count on the early stopping callback to stop training when our monitored statistic stops improving. Then when we return to the computer, we look through our saved files. Since the most recently-written file is usually the best-trained model, that's the one we use from then on.

Our callback is made by creating an instance of an `EarlyStopping` object. Let's look at four of its useful options.

First, we tell the system which value it should be watching. As usual, we can specify the training accuracy `'accuracy'`, training loss `'loss'`, validation accuracy `'val_accuracy'`, or validation loss `'val_loss'`. We hand our choice to the parameter named `monitor`.

Second, we provide a value to a floating-point parameter called `min_delta`. The word "delta" refers to the Greek letter δ (delta), which mathematicians often use to refer to the idea of "change." In this case, `min_delta` is the minimum amount of change to the monitored value for `EarlyStopping()` to notice. Any change less than this amount is ignored. By default, this value is 0, so every time the monitored value changes, `EarlyStopping()` checks to see if we need to stop. That default is usually a good place to start. We might increase this value if we're getting way too many files.

Third, we provide a value to an integer called `patience`. As the system watches our chosen parameter from one epoch to the next, there might be some ranges of time where it doesn't improve, or even gets a little worse. We don't want to give up as soon as this happens, because it might just be a temporary effect. As we've seen, the accuracy and loss curves are often a bit noisy and jump around a little. We only want to call a halt if the value we're watching is really getting worse over the long term. The value we assign to `patience` tells the routine how long the "long term" is. It's the number of epochs to wait for things to get better before deciding that `fit()` should stop training. The default value of `patience` is 0, which is usually too aggressive. This is a parameter that's best set after a bit of experimentation to see how noisy the results are.

Finally, we can also set a value to `verbose` to have it print out a line of text if it decides to stop training, so we can look at the output and know that it intervened.

Listing B2-66 shows how to set up and use this callback. We'll watch the validation loss, set `patience` to 10 epochs, and `verbose` to 1 so we get a notice when `EarlyStopping()` decides we should indeed stop.

```
from tensorflow.keras.callbacks import EarlyStopping

early_stopper = EarlyStopping(monitor='val_loss', patience=10, verbose=1)

history = model.fit(X_train, y_train, validation_split=0.25,
                    epochs=100, batch_size=256, verbose=2,
                    callbacks=[early_stopper])
```

Listing B2-66: Setting up and using `EarlyStopping()` to stop training when the validation loss stops dropping for more than 10 epochs.

Let's run this and see what happens.

Listing B2-67 shows the result. At epoch 23 we see that `EarlyStopping()` has decided we need to stop. Since we set `patience` to 10, and we're monitoring the validation loss, this tells us that the validation loss hasn't improved since epoch 13. So training ends after the 23rd epoch and `fit()` returns. It's just as if we'd interrupted the training process ourselves.

```
...
Epoch 22/100
3s - loss: 5.8155e-04 - acc: 1.0000 - val_loss: 0.0636 - val_acc: 0.9834
Epoch 23/100
3s - loss: 4.4813e-04 - acc: 1.0000 - val_loss: 0.0631 - val_acc: 0.9825
Epoch 24/100
3s - loss: 4.0089e-04 - acc: 1.0000 - val_loss: 0.0647 - val_acc: 0.9828
Epoch 00023: early stopping
```

Listing B2-67: The last few epochs of training with early stopping. At epoch 22 the system decides we ought to stop training.

The accuracy and loss graphs for this run are shown in Figure B2-32.

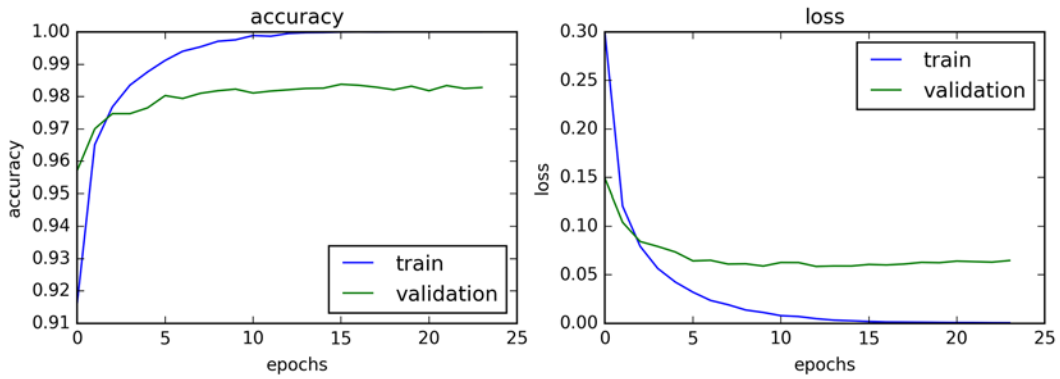


Figure B2-32: Accuracy and loss for our early-stopping run. Note that the validation loss settles down at about epoch 13. The early stopping routine we set up waits another 10 epochs for improvement, and then halts training at epoch 23.

The validation loss that we're monitoring seems to stop improving at around epoch 13. We can be sure of that, because our early stopping callback halted training 10 epochs later, at epoch 23. The validation loss might be starting to rise just a little bit, but we've definitely avoided the rising slope of the overfitting curve we saw in Figure B2-31 when we trained for 100 epochs

Experimenting with the `patience` value allows us to tune the performance of the `EarlyStopping()` routine to our network and data. As we mentioned above, we can always use an early stopping algorithm of our own and use that instead [ZFTurbo16].

Early stopping is the solution we promised earlier to the problem of picking the wrong value for `epochs` when calling `fit()`. With early stopping in place, we can always pick a ridiculously large number for `epochs`, and let the computer automatically stop training at the right time.

Wrapping Up

This brings us to the end of our first visit with Keras. We've seen a lot of code and a lot of the library, and we've built and trained deep learning systems.

In Bonus Chapter 3 we'll continue on from here, diving deeper into Keras and exploring more of the tools it provides for building even more complex and interesting systems.

References

[Benenson16]: Rodrigo Benenson, "What is the class of this image?" 2016. *http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html*

[Bengio17]: Yoshua Bengio, "MILA and the future of Theano," email thread on theano-users Google Group, September 28, 2017. *https://groups.google.com/forum/!topic/theano-users/7Poq8BZutbY*

[Chollet17a]: François Chollet, "Keras Documentation," 2017. *https://keras.io/ and https://github.com/fchollet/keras*

[Chollet17b]: François Chollet, *Deep Learning with Python*, Manning Publications, 2017.

[Chomsky57]: Noam Chomsky, "Syntactic Structures," Mouton and Co., 1957.

[Colab21]: Colab authors, "Google Colab," 2021. *https://colab.research.google.com*

[CNTK17]: Microsoft, "The Microsoft Cognitive Toolkit," Microsoft Cognitive Toolkit, 2017. *https://docs.microsoft.com/en-us/cognitive-toolkit/index*

[Devlin16]: Josh Devlin, "28 Jupyter Notebook tips, tricks, and shortcuts," Dataquest.io, 2016. *https://www.dataquest.io/blog/jupyter-notebook-tips-tricks-shortcuts/*

[Dijkstra82]: Edsger W. Dijkstra, "Why Numbering Should Start at 0," August 1982. *https://www.cs.utexas.edu/users/EWD/ewd08xx/EWD831.PDF*

[Gupta17]: Dishashree Gupta, "Transfer Learning and The Art of Using Pre-trained Models in Deep Learning," Analytics Vidhya blog, 2017. *https://www.analyticsvidhya.com/blog/2017/06/transfer-learning-the-art-of-fine-tuning-a-pre-trained-model/*

[HDF517]: The HDF5 Group, "What Is HDF5?" HDF Group Support Page, 2017. *https://support.hdfgroup.org/HDF5/whatishdf5.html*

[JetBrains17]: Jet Brains, "Pycharm Community Edition IDE," 2017. *https://www.jetbrains.com/pycharm/*

[JSON13]: JSON Contributors, "Introducing JSON," ECMA-404 JSON Data Interchange Standard Working Group, 2013. *https://www.json.org*

[Jupyter16]: The Jupyter team, 2016. *http://jupyter.org/*

[Kaggle21]: The Kaggle team, "Kaggle", 2021. *https://www.kaggle.com/*

[Karpathy16]: Andrej Karpathy, "Transfer Learning," Stanford CS 231 Course Notes, 2016. *https://cs231n.github.io/transfer-learning/*

[Kernighan78]: Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice Hall, 1978.

[Khronos21]: The Khronos Group, "The open standard for parallel programming of heterogeneous systems," Khronos Group Website, 2021. *https://www.khronos.org/opencl/*

[LeCun13]: Yann LeCun, Corinna Cortes, Christopher J. C. Burges, "The MNIST Database of Handwritten Digits," 2013. *http://yann.lecun.com/exdb/mnist/*

[Lorenzo17]: Baraldi Lorenzo, "VGG-16 pre-trained model for Keras," GitHub, 2017. *https://gist.github.com/baraldilorenzo/07d7802847aaad0a35d3*

[NVIDIA17]: NVIDIA Corp, "CUDA Home Page", NVIDIA Website, 2017. *http://www.nvidia.com/object/cuda_home_new.html*

[Ramalho16]: Luciano Ramalho, *Fluent Python: Clear, Concise, and Effective Programming*, O'Reilly Media, 2016.

[Sato17]: Kaz Sato, Cliff Young, and David Patterson, "An in-depth look at Google's first Tensor Processing Unit (TPU)," Google Cloud Big Data and Machine Learning Blog, 2017. *https://cloud.google.com/blog/big-data/2017/05/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu*

[TensorFlow16]: Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon

Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2016. *http://tensorflow.org*

[Theano16]: Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," *https://arxiv.org/abs/1605.02688*. For online documentation, see *http://deeplearning.net/software/theano/index.html*.

[Wentworth12]: Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers, *How to Think Like a Computer Scientist: Learning with Python 3*, Chapter 9, "Tuples," 2012. *http://openbookproject.net/thinkcs/python/english3e/tuples.html*

[Wikipedia17]: Wikipedia authors, "Iris Flower Data Set," Wikipedia, 2017. *https://en.wikipedia.org/wiki/Iris_flower_data_set*

[YAML11]: YAML Contributors, "YAML Home Page," 2017. *http://yaml.org/*

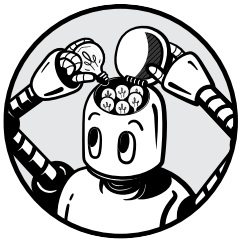
[ZFTurbo16]: ZFTurbo, "How to tell Keras stop training based on loss value," Stack Overflow, 2016. *http://stackoverflow.com/questions/37293642/how-to-tell-keras-stop-training-based-on-loss-value*

Image Credits

Figure B2-1, Giraffe, *https://pixabay.com/en/giraffe-wildlife-safari-africa-2868936*

B3

KERAS PART 2



This chapter is a bonus chapter for my book *Deep Learning: A Visual Approach*. You can order the book from No Starch Press at *https://nostarch.com/deep-learning-visual-approach/*. The official version of this chapter can be found for free on my GitHub at *https://github.com/blueberrymusic/* (look for the repository "Deep-Learning-A-Visual-Approach"). All of the figures in this chapter, and all of the notebooks with complete, running implementations of the code discussed here, can also be found for free on the book's GitHub repository.

In Bonus Chapter 2 we introduced the Keras library and looked at how to build and train basic models.

Now we'll expand our horizons. We'll see how to improve our models, incorporate search routines from the scikit-learn library (discussed in Bonus Chapter 1), and build more complex models such as CNNs and RNNs.

Improving the Model

This section's notebook is Bonus03-Keras-1-Improving-the-Model.ipynb.

In Bonus Chapter 2, we explored a lot of the features Keras offers using a tiny, 2-layer model. As we saw, after only about 20 epochs of training this model was able to accurately classify about 98% of the images in the MNIST test set.

Let's see if we can improve that. How might we build a better model?

The answer is not obvious. It's made more difficult by the sheer number of choices that we can try out. Even in this extremely simple model with one hidden dense layer, we've made many choices, all of which influence how well and how fast our network learns.

Though sometimes a change to our model can bring about a big improvement in accuracy, much of the time improving a model's performance is a game of accumulating a sequence of tiny improvements.

Counting Up Hyperparameters

Before we start modifying the hyperparameters of our model, let's get a clearer picture of just how many choices we've made. Note that we've not counting up the weights, which aren't under our control. We're just looking at all the places where we could make a different decision in the design of our model.

Many of the routines we've used take multiple optional arguments that we've ignored. In a sense, we've chosen values for those arguments by letting them stay at their defaults. So let's count those, too.

Each of our two `Dense` layers took 2 arguments: the number of neurons, and the activation function. Consulting the Keras documentation, there are at least 7 more arguments that we could reasonably experiment with.

Then there's the choices we made when we compiled the model. We chose a loss function and an optimizer, giving us 2 more options to adjust.

Given that we chose the `adam` optimizer, there are 5 optional arguments that we can use to tune its behavior.

Finally, we supply a host of options when we call `fit()` to train the model. We have choices for the batch size and the number of epochs (we could argue that the number of epochs doesn't matter if we use early stopping, but then we'd have to set a value for that algorithm's patience). So we have at least 2 arguments here.

So in this casual tour of our choices, we've got 9 layer-level choices on each of 2 layers for a total of 18 choices, 2 choices when we compile, 5 more choices for our optimizer, and at least 2 choices when we fit. That's a total of 27 hyperparameters for this tiny model.

The number goes up fast as we add more layers.

Figuring out what changes to these choices will make the model better is a daunting task. Imagine sitting at a control panel with 27 sliders, switches, and knobs. This is just to control basic performance, and doesn't include controls for additional options like adjusting the learning rate schedule.

We might set the controls, push the big red button to train the network, wait for a while, and eventually look at the numbers that report how we did.

Then we could adjust one or more controls in the hopes of making things better, and repeat.

Complicating the problem is that many of the hyperparameters interact. So if we increase one value, we might only see an improvement if we simultaneously decrease two or three other values, and increase one or two others.

Things get even harder when we want to improve larger and deeper models with dozens of layers. The number of choices and their possible settings becomes enormous.

This is why we've gone through so many chapters of information to get here. The only chance we have of improving our model's performance is to draw from our knowledge about what the network is doing, and why, and what all of our choices do. When we understand what's happening inside, we have a fighting chance of learning from our experience and developing the intuition and hunches that are essential to building great deep-learning networks.

Though we almost always have to run experiments and see what happens, our knowledge and experience improve our chances of making things better.

Changing One Hyperparameter

A frequent rule in experimentation of all sorts is to change only one thing at a time and see what happens. This is a good plan if the values involved are largely *decoupled*, meaning that they don't affect one another. As an analogy, suppose we're adjusting the sound of our car radio, and boosting or cutting the highs and lows. The results of these choices combine, but they're independent: generally speaking, adding more treble doesn't change how much bass sound is delivered, and vice-versa.

Unfortunately, the hyperparameters of most real systems, and most deep-learning systems, are not decoupled. If we increase the amount of hyperparameter A and find things get better, and then increase the amount of hyperparameter B, we may find that we now have to *decrease* the value in A to make further progress. The connections are complex.

But still, changing one hyperparameter at a time is usually a good way to start. We can explore what that value does, find a good value for it, and then choose another hyperparameter to adjust, and so on, searching for a good combination by fine-tuning one hyperparameter at a time. If we have to go back, then our experience with each value can help guide us to select which one to adjust again, and by how much. We can also build up a sense of which values are related to which others, so we can anticipate their interactions.

Let's try that now, arbitrarily picking the batch size as our first hyperparameter to experiment with. We said above that when we're using a GPU we pick a batch size that best fits our particular hardware. But on a CPU we can pick almost any value we like. We've been using a batch size of 256, but like most of our initial choices for each of the 27 hyperparameters we just counted up, it was really just a shot in the dark. Let's try cranking that up and down and see what happens, if anything.

Figures B3-1 through B3-4 show the results of setting this hyperparameter to 2048, 512, 64, and finally 8. We used the same code for every run,

changing only the batch size. Note that the vertical scale on the graphs is not the same from one graph to the next. This allows us to show all the data, though it means we can't compare them equally at a glance.

Three things jump out from these figures.

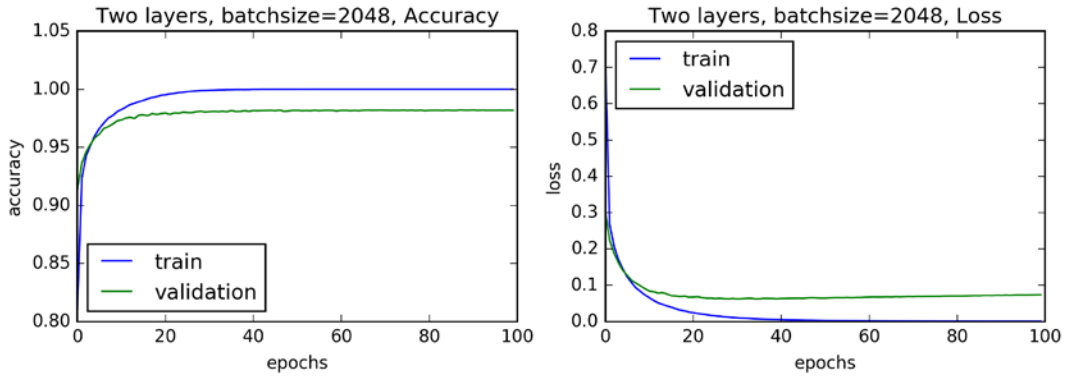


Figure B3-1: Training our two-layer model with a batch size of 2048

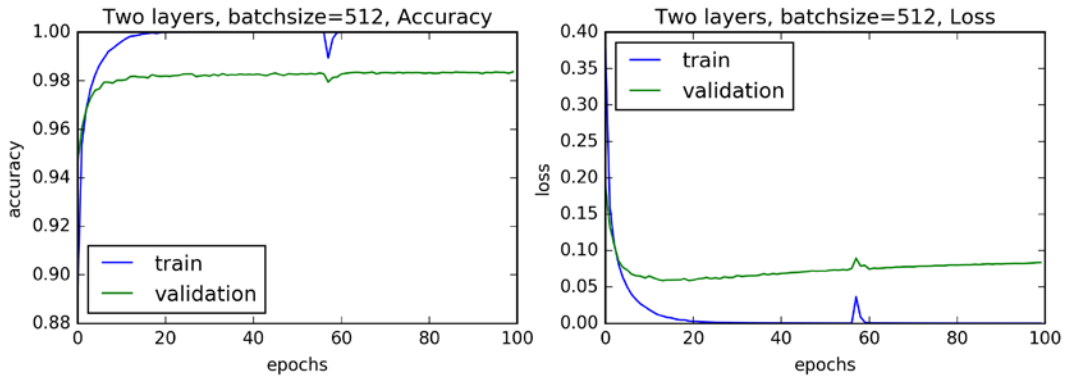


Figure B3-2: Training our two-layer model with a batch size of 512

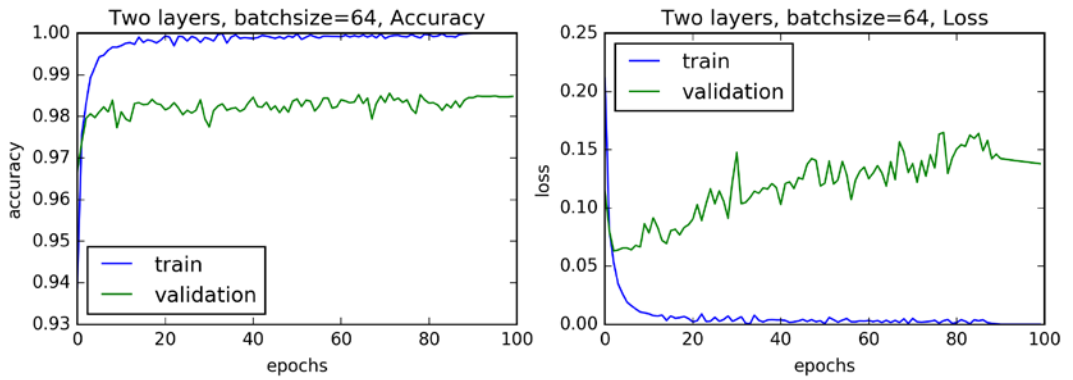


Figure B3-3: Training our two-layer model with a batch size of 64

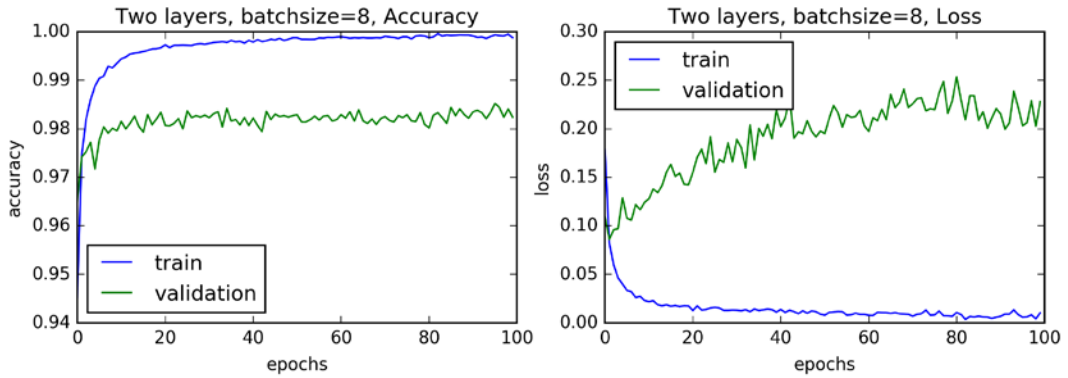


Figure B3-4: Training our two-layer model with a batch size of 8

First, as the batch size gets smaller, the results get more jittery, or noisy. This is because each new update is working with fewer samples, so it's responding to whatever happens to be in that batch. Larger batches tend to become more representative of the dataset as a whole, and give us smoother results. Smaller batches give us a lot of jumping around.

The second thing is that the training accuracy is about 98% on all the models, so the batch size didn't affect that accuracy very much.

The third thing is that although all of the models are overfitting, as demonstrated by the diverging training and validation losses, as the batch size gets smaller the divergence of the training and validation error increases. In other words, the amount of overfitting increases.

Smaller batches mean that epochs take longer, because we need to perform backprop and update the weights more frequently. Figure B3-5 shows the clock time, in seconds, for each of the above batch sizes, plus the other powers of 2 between them.

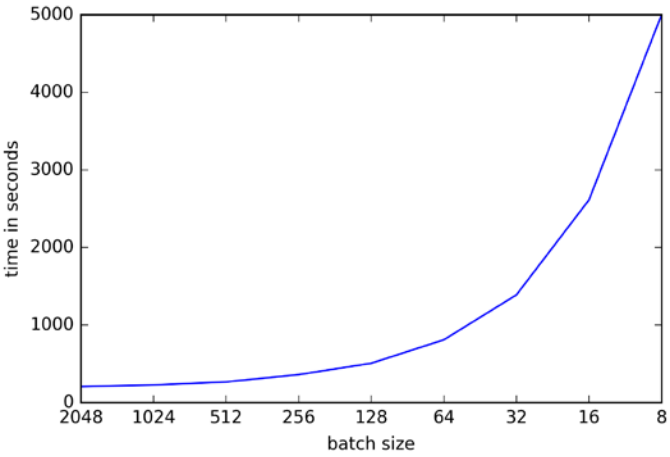


Figure B3-5: The timing results (running time in seconds, on a late 2014 iMac) taken by the experiments whose data are shown in Figures B3-1 through B3-4, as well as other intermediate batch sizes. Note that the vertical scale is linear while the horizontal scale is not.

The curve in Figure B3-5 confirms that on these CPU-only runs, as the batch size went down, we're running more backprop and update steps, so the total training time went up.

The above experiments tell us a lot about how the batch size affects training for this dataset on this model. They suggest that for this model and data, large batch sizes are more desirable than small ones.

Other Ways to Improve

When we seek to improve a model, it can help to keep in mind that we'll be unlikely to find *the very best* set of parameters for the training speed and accuracy that we're after. Instead, we look for parameters that come close enough.

It also helps to have one goal in mind at every step in the search. We might be looking to reduce overfitting, or drive down the test loss, or increase the test accuracy, or speed up training time, or fit best onto the GPU, or use the least computer memory, and so on. We're unlikely to be able to improve all of these at once.

So we typically pick out just one or two things to improve, and then modify some of our variables until they're as good as we can get them. Then we move on to another group of criteria, and look for the variables that will help with those, and so on.

For the MNIST problem, let's aim to improve accuracy while reducing overfitting.

Rather than continue to adjust hyperparameters, let's try something radically different: adding a second `Dense` layer.

In order to keep everything comparable with the models earlier in this chapter, we'll return to a batch size of 256, and leave out early stopping.

Adding Another Dense Layer

Let's add a second dense hidden layer, just as big as the first. After all, having more neurons means more ability to learn, right?

Not really. We've seen that our single-layer model is already too capable of learning the idiosyncrasies in the training data. That's why it's overfitting. If we throw in yet more neurons without making any other structural changes, then this overfitting should get worse, faster.

Let's try it and see.

Figure B3-6 shows the architecture for a model with two dense hidden layers, each arbitrarily given as many neurons as there are input elements (that is, each layer has 784 neurons), followed by a 10-neuron output layer with softmax on the output.

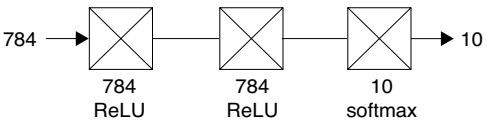


Figure B3-6: The architecture of our three-layer model. Each of the hidden layers is a dense layer with 784 neurons, one for each input. Though our convention is that dense layers have a ReLU activation function by default, for completeness we're listing it here explicitly.

We can make this model in Keras by adding just one line to our model-making routine, creating the second dense hidden layer. This line looks just like the one above it except that we don't include the `input_shape` argument, since that's only used by the first layer in the model. Listing B3-1 shows the code.

```
def make_two_hidden_layers_model():
    model = Sequential()
    model.add(Dense(number_of_pixels, input_shape=[number_of_pixels],
                    activation='relu'))
    model.add(Dense(number_of_pixels, activation='relu'))  # new layer
    model.add(Dense(number_of_classes, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam',
                  metrics=['accuracy'])
    return model
```

Listing B3-1: Create and compile the network of Figure B3-6, with two identical `Dense` layers in a row.

Figure B3-7 shows the accuracy of the training and validation sets over 100 epochs of learning (we did not use early stopping).

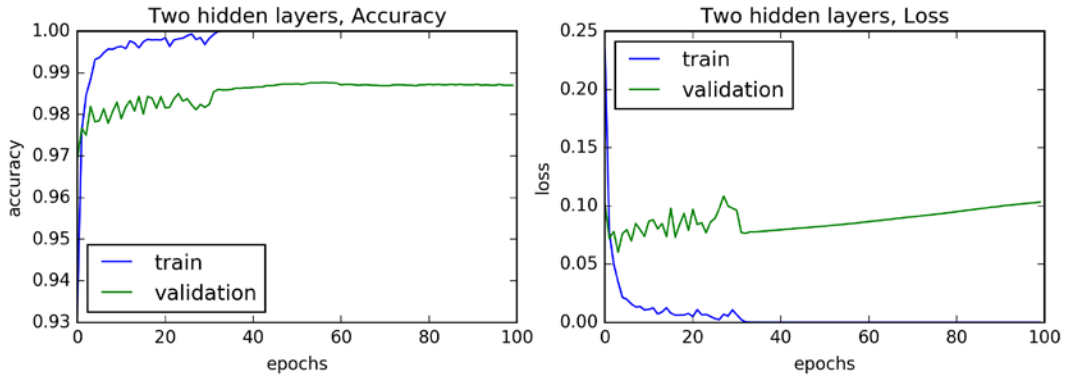


Figure B3-7: The accuracy and loss of our two-layer network plotted against the number of epochs

Compared to our previous results, the new curves are wigglier in the starting epochs, suggesting that the bigger network took more time to settle down. And just as we suspected, the system overfit the training data just like before, but it did so even more quickly, driving up the validation loss faster than before.

So just throwing more neurons at the problem did not make everything better. Validation accuracy improved a touch, but the loss is looking much worse, and we're still overfitting considerably.

Less Is More

Having too many neurons has made our network *too capable*. It had more than enough power for this task, so it used its extra abilities to extract more and more idiosyncratic detail from the training set, and thus overfit.

We generally want the smallest, simplest network that will get us the results we're after. A simpler network not only trains and predicts faster, but it's less prone to overfitting because there's less superfluous computational power to get distracted by irrelevant details in the training data.

Let's go back to our single dense layer, but make it far smaller, with only 64 neurons. This gives us roughly one neuron for every 12 input pixels. The new architecture is shown in Figure B3-8.

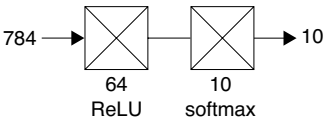


Figure B3-8: A new two-layer network where we'll only use 64 neurons in the first, fully-connected hidden layer

We'll just change the line that defines this layer to give it 64 neurons rather than 784. Listing B3-2 shows the change.

```
def make_smaller_one_hidden_layer_model():
    model = Sequential()
    model.add(Dense(64, input_shape=[number_of_pixels],
                    activation='relu'))
    model.add(Dense(number_of_classes, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam',
                  metrics=['accuracy'])
    return model
```

Listing B3-2: Building a model where the first (and only) hidden layer has just 64 neurons

The accuracy and loss results for 100 epochs are shown in Figure B3-9.

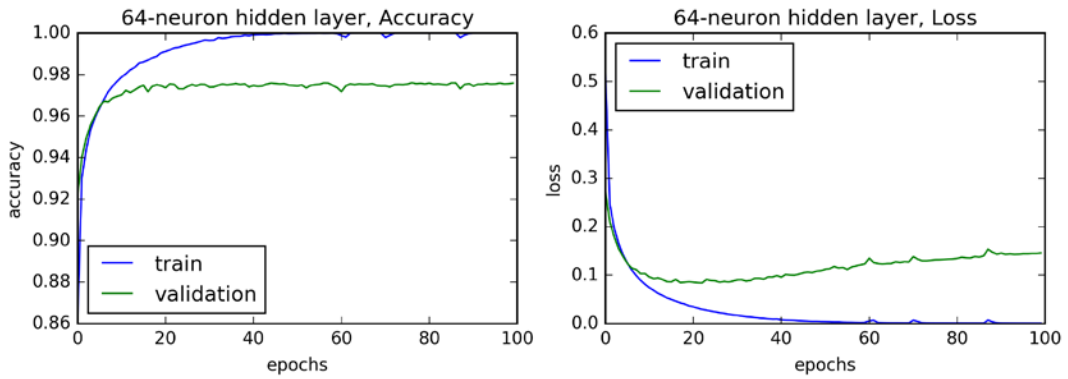


Figure B3-9: The accuracy of our model of Figure B3-8

Our network is giving us a bit less accuracy than the one with 784 neurons in the first layer, but even so, it's still overfitting. What to do? Let's try using our idea from the last section and use two hidden layers instead of one, but we'll keep the same number of neurons and split them evenly. In other words, we'll have two hidden layers of 32 neurons each, as shown in Figure B3-10.

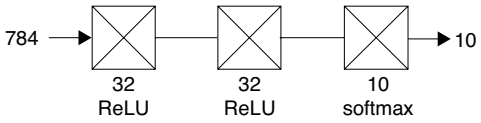


Figure B3-10: A deeper model with three layers. We're splitting up the 64-neuron hidden layer of our previous model into two separate, fully-connected 32-neuron hidden layers.

The accuracy and loss results for 100 epochs of training are in Figure B3-11.

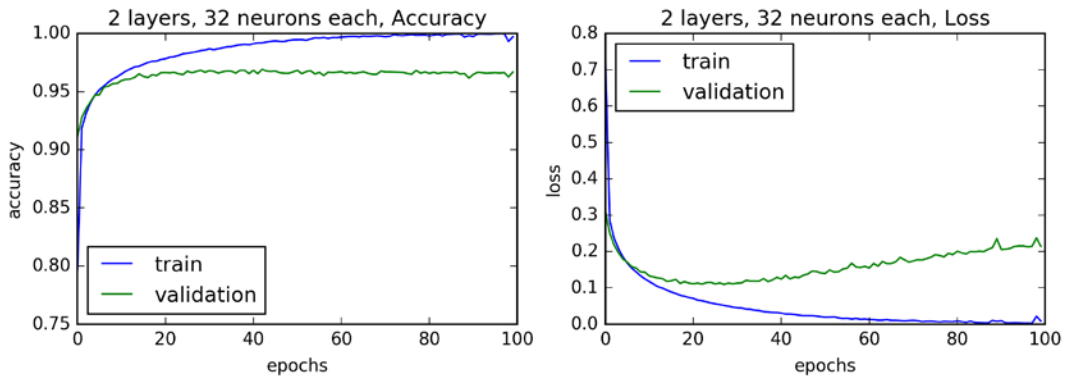


Figure B3-11: The accuracy of the model in Figure B3-10

The validation accuracy after 100 epochs has decreased a little bit from about 97.5% for our single 64-neuron layer to about 97% for our new, two-layer network. The loss has increased, too, and we seem to be overfitting even more rapidly than before.

We've taken a step backwards. A small step, granted, but the measurements are worse and we're still overfitting.

Though chopping up layers into multiple, smaller pieces can sometimes work, it didn't help us much in this example. This is often the way it goes: we try one thing and another, following up on ideas that work and setting aside those that don't make things better.

Before we give up on these two small layers, let's see if we can try another trick to get the overfitting under control.

Adding Dropout

Since overfitting is a problem for this model and data, let's try using *dropout*. As we discussed in Chapter 15, this is a regularization technique explicitly

designed to address overfitting. Dropout temporarily removes a random selection of neurons before each epoch, and puts them back in at the end. The intuition is that our neurons will be less likely to specialize (and potentially over-specialize), since they all need to be able to compensate for randomly missing neurons.

To apply dropout in Keras, we create a new *dropout layer* and add it to the growing stack, just after the layer we want to affect. When dropout is applied, randomly-chosen neurons are isolated from the network for one epoch, so they don't contribute to predictions, and they don't learn when the network's weights are updated. When the epoch is done, the neurons are restored, and before the next epoch, a new random collection gets disconnected. This process occurs only during training.

It might seem a little weird that dropout is included as a layer. It doesn't have any neurons, and it doesn't participate in backprop or computation, so how can it be a layer? Calling this a layer is really just a conceptual device. We'd like to apply dropout to not just `Dense` layers, but other types of layers, like the convolution and recurrent layers we'll cover later in this chapter. Rather than build dropout into each layer, Keras lets us specify this kind of "supplemental" layer that doesn't do any computing, but tells Keras about something we want it to do. Thinking of operations like dropout as implemented by their own layers lets us keep our conceptual view of our model simple and clean. We just have a big stack of layers. Some layers have neurons, and others perform operations on other layers or on data.

In this case, the dropout layer says to Keras, "apply dropout to the preceding layer." If there are, say, 3 `Dense` layers preceding this dropout layer, only the most recent one is affected. If we wanted to apply dropout to all three `Dense` layers, we'd have to follow each one individually with its own dropout layer.

The dropout layer in Keras takes only one parameter, and it's mandatory. It's a floating-point number between 0 and 1 that describes the percentage of neurons that will be temporarily removed after each batch. A value of 0 disables dropout, while a value of 1 would make the preceding layer effectively disappear. The authors of the original paper on dropout advise a value of 0.2, and that's generally a good place to start [Srivastava14].

The authors also advise constraining the magnitude of the weights on the dense layers that are affected by dropout. Speaking generally, the concern is that when some nodes are removed, the others might overcompensate by cranking their weights up very high. Without getting into the math, we can take their advice by setting an optional parameter on the `Dense` layer that's going to experience dropout. The parameter is called `kernel_constraint`, and the advice of the authors of the paper cited above is to set that to the value 3, so we'll do just that. We only need to add this option to `Dense` layers that will have dropout applied to them, as we'll see in a code listing just below.

The complete model specification for our two-layer model, with dropout, is shown in Figure B3-12. Here we're applying dropout to both of our two hidden layers.

In Figure B3-12 we show our schematic symbol for a dropout layer, which is a slanted line crossing the line carrying data, suggesting that some of the data is being struck out, or removed.

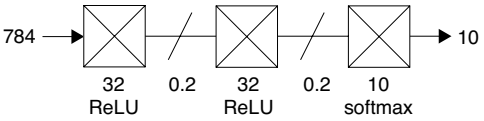


Figure B3-12: We'll change our model in Figure B3-10 to add dropout layers after each 32-neuron, fully-connected, hidden layer. Here we're using our symbol for dropout: a diagonal slash through the line connecting two layers. Each dropout layer applies to the layer preceding it.

The code for making this model is in Listing B3-3. There are a few new things happening in this code.

```
from tensorflow.keras.layers import Dropout
from tensorflow.keras.constraints import MaxNorm

def two_layers_with_dropout_model():
    model = Sequential()
    model.add(Dense(32, input_shape=[number_of_pixels],
              activation='relu', kernel_constraint=MaxNorm(3)))
    model.add(Dropout(0.2))
    model.add(Dense(32,
              activation='relu', kernel_constraint=MaxNorm(3)))
    model.add(Dropout(0.2))
    model.add(Dense(number_of_classes, activation='softmax'))
    # compile the model to turn it from specification to code
    model.compile(loss='categorical_crossentropy', optimizer='adam',
                  metrics=['accuracy'])
    return model

model = two_layers_with_dropout_model()
```

Listing B3-3: Two dense layers of 32 neurons each are both followed by `Dropout` layers. We use a dropout percentage of 0.2. We also set `kernel_constraint` in the `Dense` layers.

First, of course, we're adding dropout layers. The argument 0.2 tells the layer to use the 20% dropout rate suggested by dropout's creators.

As we mentioned above, the original paper on dropout also suggested imposing a technical condition on the weights in the layer that's experiencing the dropout, and that advice is widely followed. In Listing B3-3 we do this by adding the optional argument `kernel_constraint` to the argument list for each layer that will be affected by dropout, and setting that parameter's value to `MaxNorm(3)` (note that we have to import `MaxNorm()` in order to use it). The thinking behind this step, which explains what this `MaxNorm()` thing is doing, is explained in the original paper [Srivastava14]. It's reasonable to just think of it as a mechanism to keep the weight values from getting too big.

Training this model for 100 epochs produces the results in Figure B3-13.

We've conquered over overfitting problem! The losses are no longer diverging. Dropout has done a great job for us.

The accuracy is a bit weird, since we're getting better accuracy on the validation data than the training data. The validation accuracy seems to

have taken a small hit, too, since it's not up to the 98.3% from before. We might be able to tweak our accuracy upwards by reducing the dropout rate a bit, or adding a few more neurons to our dense layers.

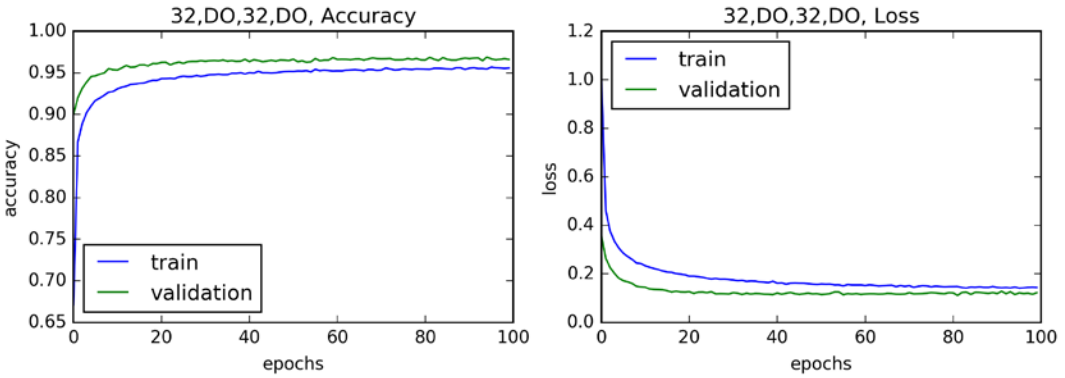


Figure B3-13: Accuracy and loss for the model of Figure B3-12

The dropout paper also recommends that that we configure our optimizer to use a learning rate 10-100 times larger than we normally would. We can tell Adam to start with any specific learning rate by setting its optional argument `lr` (that's a lower-case letter L followed by a lower-case letter R, standing for "learning rate"). This value defaults to 0.001.

To pass this argument to Adam, we have to make an `Adam` object as we did earlier, and pass it our new value to the learning rate parameter. Listing B3-4 shows how we'd set the initial learning rate to 0.1. This would replace the line previously calling `model.compile()`.

```
from tensorflow.keras.optimizers import Adam

# make our own Adam object
adam_optimizer = Adam(lr=0.1)

# optimizer gets our object, rather than a string
model.compile(loss='categorical_crossentropy',
              optimizer=adam_optimizer, metrics=['accuracy'])
```

Listing B3-4: We can provide our own optimizer object when we compile, rather than rely on a default. Here we make an `Adam` with our own choice of learning rate.

A shorter way to write this is shown in Listing B3-5, where we create the `Adam` object and assign it, without needing a temporary variable to hold it.

```
model.compile(loss='categorical_crossentropy',
              optimizer=Adam(lr=0.1),
              metrics=['accuracy'])
```

Listing B3-5: We don't need to store our new `Adam` object in its own variable. This shorter approach is more common.

The surprising results are shown in Figure B3-14.

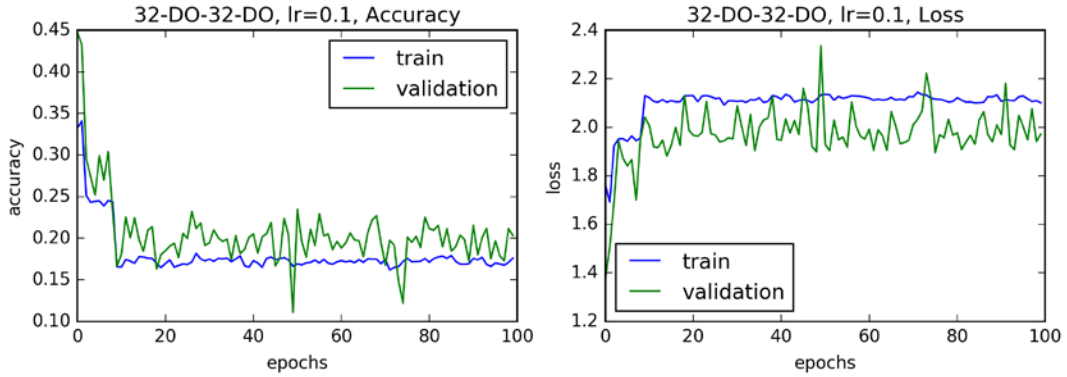


Figure B3-14: The accuracy and loss for our model with dropout when we set Adam's initial learning rate to 0.1

Wow. These graphs are as bad as they look.

For this data and architecture, starting Adam with a learning rate of 0.1 was much too aggressive. The training accuracy plummeted to about 0.18, which is terrible. The validating accuracy seems to be fluttering around 0.2, but it's got a lot of noise. The loss was also terrible, more than 10 times worse than before.

If we drop the learning rate down to 0.01, we get much more encouraging performance, as shown in Figure B3-15.

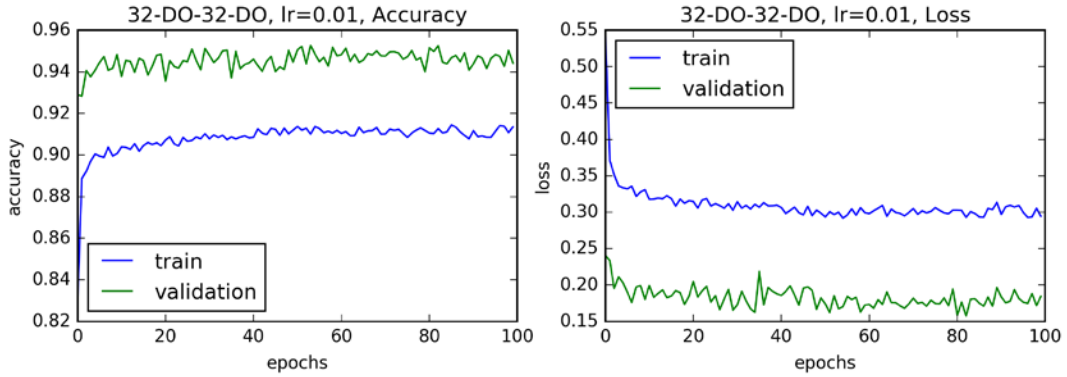


Figure B3-15: Accuracy and loss for our dropout model with Adam's initial learning rate set to 0.01

These results aren't nearly as good as what we got with the default learning rate of 0.001, but it was worth a shot. Things are much calmer, and our accuracies are both above 90%. And we're still not overfitting.

We could try out a variety of learning rates to try to see what value works the best for this model and data, but that would be a lot of typing and waiting.

It would be really nice if we could automate this search, so the computer could try out a variety of learning rates for us while we do other things. In an upcoming section, we'll see how to do just this using tools from scikit-learn.

Observations

We've only begun the process of tuning and refining our model. Tinkering with a practice model like this one in search of the best results is time well spent, since it hones our intuition and can help guide our choices with other, larger, databases and models in the future.

This is one of those times when saying "the exercise is left to the reader" is completely appropriate. There's no substitution for sitting down with a deep learning model and adjusting its structure and hyperparameters to get a feeling for how that model behaves on that data.

When we work with models that we will actually be deploying in the real world, we want the best-performing models we can develop. When models become so big that they can take days (or even weeks) to train, it's important to have a good sense of what's likely to work, since it lets us start much closer to the finish line than just assembling a model at random. Even if we automate the parameter search, we still want to focus our search where it's going to pay off the most.

We found that for this data a single giant fully-connected hidden layer was overkill. It started overfitting almost immediately, and the training accuracy went flat. Worse, the training loss was steadily climbing, which would eventually cause the training accuracy to drop. We were throwing a network with too many computing resources at the problem, and it used those resources to overfit, learning far too much about the idiosyncrasies of the training data even after it had perfect accuracy.

By significantly reducing the size of that layer we got away from overfitting, but our accuracy dropped.

Then by splitting that single layer into two pieces, and adding dropout, we got performance up, and stopped overfitting.

There are many other things left to try. Using more layers is always something to consider. Perhaps each layer could be smaller than the one before it, forcing the system to look for larger patterns. Or perhaps we could have one small "choke" layer between two larger ones. We might try applying dropout only on some of the layers, or applying it more aggressively (that is, raising the number of neurons we're suppressing).

Using Scikit-Learn

This section's notebook is Bonus03-Keras-2-scikit-learn.ipynb.

So far we've been searching our hyperparameters by hand. It's been illuminating, but it also required a lot of manual effort.

We saw in Bonus Chapter 1 that the scikit-learn library offers us routines to cross-validate our model (to estimate how good it is), and grid-search its hyperparameters (to find the best-performing combination).

Keras doesn't offer either of these tools directly, because it offers a way to use the ones already in scikit-learn.

Let's pause a moment to think about what we might be asking for from these tools. If a model takes 3 hours to train, then running cross-validation with 10 folds will take about 10 times longer, or 30 hours. If we're

grid-searching over, say, three hyperparameters with 5 values each (which is not a very large search), then it will take 125 times longer than that, or more than five months!

Is there any way to cut this down?

A popular approach is to extract a tiny piece of the data set, carefully selected to be representative of the whole, and search on that. Then each training run will be much faster.

By cross-validating and grid-searching one or more of these little proxy databases, we can get some guidance for what models and hyperparameters are worth exploring on a larger scale. Then we can take that knowledge and work with larger and larger pieces of the dataset, tuning the hyperparameters at each step. The hope is that by the time we reach the full database, we'll have a great set of hyperparameters to train on and we'll need only a little searching, or perhaps even none at all.

Keras Wrappers

It would be nice to use scikit-learn's cross-validation and grid-search tools directly on our Keras models. Happily, Keras "knows" about scikit-learn, and how to communicate with it.

In particular, Keras knows how scikit-learn expects its estimators to behave. With that, Keras can dress up one of its models to act like a scikit-learn estimator, bridging the gap.

This act of camouflage lets us place a Keras model into scikit-learn, and then do cross-validation, grid search, or any other operation we like. From scikit-learn's perspective, this object is just some custom estimator that we wrote and gave to it. It doesn't know that there's a deep network hiding inside.

We pull off this trick by embedding our Keras model in an object of type `KerasClassifier` or `KerasRegressor`, depending on the job it does. These objects are called *wrappers*, since they "wrap" our Keras model in a disguise that makes it look and act like a scikit-learn estimator. We don't have to modify our network in any way to wrap it. We just make a wrapper object, place our network inside, and we're done.

Since both wrappers work identically, we'll choose `KerasClassifier` as an example so we can stick with the MNIST classifiers we've been discussing so far.

We don't actually hand our model to the wrapper function. Instead, we hand it the name of a function that builds the model and returns it. This makes sense when we think about it. The searching process, for instance, may create many versions of our model with different hyperparameters. If we gave it a built and compiled model, there wouldn't be any way for it to make different versions. By giving it a function that builds the model, the searching program can call the function, and when doing so, to set various parameters as it desires.

The other arguments to the wrapper creator are arguments that get passed on. Some are given to the model-making function, and others get passed to scikit-learn.

Let's dig in, starting with the model-making function.

This argument is named `build_fn`, short for "build function." Its value is a function that we've written which will construct, compile, and return a Keras model, just like we've been seeing in listings like Listing B3-3, where the function `two_layers_with_dropout_model()` made a model, compiled it, and returned it.

There are some advanced options, but usually we assign this argument the name of a function in our code, such as `two_layers_with_dropout_model`. Note that we leave off the parentheses, since we're not calling the function, but only providing its name. Often we'd like to parameterize this function with arguments. For instance, we might be searching for the best number of neurons to use in the first two layers. So we make those numbers arguments that we use when we make the model.

As we said before, this model making-function will be called automatically by scikit-learn when the model is required. When we're grid searching, the model will usually be built over and over again at the start of each new step of the search.

So we have our model-making function that takes arguments, and a wrapper, and scikit-learn which is going to call our function. How do we get scikit-learn to include the arguments we want when it calls the model-making function?

Happily, the mechanism is easy. The trick is in the naming of our arguments. Recall from Bonus Chapter 1 that when we create a search using scikit-learn, we provide it with a dictionary that names each parameter we want it to search on as a key, with values to be tried as the value. Python could then match up those dictionary names with the names of parameters in the functions it called.

In the case of a wrapper, things are even easier. Thanks to Python's ability to "know" what the parameter names are in functions, we don't even need the dictionary. We can just name the parameters we want to assign values to, along with the values we want them to have.

For instance, if our model-making function takes a parameter to control the number of neurons it makes, perhaps called `number_of_neurons`, then we can place that into the wrapper's argument list with a value, just as if we were assigning a value to it when calling the function. Any arguments in our model-making function whose names match arguments in the wrapper-making step will be given the assigned values.

To see this in action, let's start with a model-making function that takes parameters. Listing B3-6 shows an example, building on our previous listings that imported and processed the MNIST data.

```
def make_model(number_of_layers=2, neurons_per_layer=32,
               dropout_ratio=0.2, optimizer='adam'):
    model = Sequential()

    # first layer is special, because it sets input_shape
    model.add(Dense(neurons_per_layer, input_shape=[number_of_pixels],
        activation='relu', kernel_constraint=MaxNorm(3)))
```

```
    model.add(Dropout(dropout_ratio))

    # now add in all the rest of the dense-dropout layers
    for i in range(number_of_layers-1):
        model.add(Dense(neurons_per_layer,
                        activation='relu', kernel_constraint=MaxNorm(3)))
        model.add(Dropout(dropout_ratio))

    # finish up with a softmax layer with 10 outputs
    model.add(Dense(number_of_classes,  activation='softmax'))

    # compile the model and return it
    model.compile(loss='categorical_crossentropy',
                  optimizer=optimizer, metrics=['accuracy'])
    return model
```

Listing B3-6: Our model-making function `make_model()` takes four optional parameters. Two are integers, one is a float, and one is a string. It creates as many dense layers as we request, appending a dropout layer after each one.

Listing B3-7 shows how to pass parameters to our new `make_model()` function which takes arguments. There are ways to make this code smaller (such as by using Python's `*kwargs` technique), but we've chosen clarity over conciseness.

```
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier

kc_model = KerasClassifier(build_fn=make_model,
                           # parameters for the model-making function
                           number_of_layers=2, neurons_per_layer=32,
                           optimizer = 'adam',
                           # parameters for scikit-learn
                           epochs=100, batch_size=256, verbose=0)
```

Listing B3-7: We wrap up `make_model()` in a `KerasClassifier` along with defaults for all the variables we need to create the model and control scikit-learn's training process. When scikit-learn calls `make_model()`, it will assign the parameters of that function the values we provided when we made our `KerasClassifier`.

In effect, the wrapper just takes the values we provide to it and passes them to the model-making function arguments of the same name. The syntax is a little confusing because it looks like `KerasClassifier` is taking these arguments for itself, but it works because Python allows this kind of clever use of parameters.

It's a common convention to assign the wrapped-up object to a variable called `model`, but that can be confused with the more typical use of `model` to mean a normal, or unwrapped, neural network. To emphasize that this is not a straightforward Keras model, we're calling it `kc_model` for "Keras classifier model." This way we can speak of the wrapped object (`kc_model`) and the model that gets created using its `build_fn` function, which we'll continue to simply call a "model."

If we hand `kc_model` to scikit-learn for cross-validation, `make_model()` will be called and passed the value 2 for `number_of_layers`, the value 32 for

the argument `neurons_per_layer`, and the string `'adam'` for `optimizer`, just as though we'd assigned them ourselves Since we're not giving a value for `dropout_ratio` to `KerasClassifier`, it doesn't assign any value to that parameter, so `make_model()` will use its default value for that argument.

If we use this `kc_model` for grid searching, the searcher can assign its own values to any of these parameters when it calls the function to make the model. Any arguments we don't explicitly re-assign will use the default values specified when we make the wrapper.

The last set of three arguments to `KerasClassifier()` (named `epochs`, `batch_size`, and `verbose`) are not for our model, but are intended for scikit-learn. They get passed to the cross-validator's `fit()` routine to control the training process.

In addition to giving parameters for `fit()`, we can also name parameters that get passed to `predict()`, for use if and when that function gets called.

As long as we keep all of our parameter names distinct, Python will correctly pass the desired value to every function involved in the cross-validation and grid searching.

This is a flexible system. For example, it means that we could tune the value of `batch_size` when searching a grid, or the size of our network by searching trying different values of `number_of_neurons`, or different optimizers by trying different strings for `optimizer_choice`.

The key thing to remember is that the wrapper is basically remembering what values should be used for the arguments in the model-making function, and it will use those by default. It also remembers a few values that get passed on to scikit-learn. As long as the names we're assigning to in the wrapper match the names in the model-making routine, everything will be automatically matched up.

Cross-Validation

Let's use a Keras wrapper to do cross-validation on our recent three-layer deep learning system of Figure B3-12 and Listing 2-3, with two 32-neuron dense layers (each with dropout) and a 10-neuron dense output layer.

Applying cross-validation may seem pointless. After all, we already have an excellent, large testing set. What more are we going to learn from cross-validation that we haven't already seen by using our validation data?

In this case, not much. We should expect the results of cross-validation to be very close to what we saw above.

But having our own high-quality validation set coming along with the data is a luxury we can't always count on. Sometimes there is no validation set. Sometimes we have one but we're not sure it's very good. For instance, consider a new deck of cards. Ignoring any jokers or other cards, one typical arrangement for the new cards is to run from ace of hearts to king of hearts, then ace to king in the suit of clubs, then again for diamonds and then spades. This is called "New Deck Order" [Cain13]. Suppose someone opened up a new deck and took away the bottom 25%, calling it the validation data. This set is definitely not representative of the rest of the deck,

because it contains no red cards of any suit, and the remaining cards have no spades.

Another challenge of validation sets comes when we're working with a small version of an original dataset. If this dataset is small, as we saw in Chapter 8, cross-validation is a great way to evaluate it without making the training set even smaller by making a dedicated validation set.

So although the MNIST data makes a great validation set, we'll proceed as though that isn't the case, so we can see how to approach the problem of evaluating the quality of a trained model in the general case.

We'll first do something simple but incomplete, just to get a feeling for the process. Then we'll add in the missing step.

Cross-validation requires training and then validating our entire model over and over again with slightly different data. We'll be using 10 folds, so each session of the cross-validator will take 10 times longer than the training sessions earlier in this chapter.

Let's get going, since there's almost nothing to it. We'll just make our model and then run cross-validation with scikit-learn as in Bonus Chapter 1.

As we mentioned above, we'll repeat our model of Listing B3-7 and use 2 layers of 32 neurons, each with dropout. We'll stick with the default Adam optimizer, though we could make our own `Adam` object as before and use that here instead.

To make our model, we'll supply our generalized model-making routine `make_model()` in Listing B3-6 with the parameters that make our desired network, as shown in Listing B3-8.

```
kc_model = KerasClassifier(build_fn=make_model,
                           number_of_layers=2, neurons_per_layer=32,
                           optimizer='adam',
                           epochs=100, batch_size=256, verbose=0)
```

Listing B3-8: Placing our model inside of a Keras wrapper

In Listing B3-8 we built a Keras wrapper of type `KerasClassifier` with our new routine `make_model()`. We gave two arguments (`number_of_layers` and `neurons_per_layer`) that matched the arguments in `make_model()`, so they will get passed in when the model is built. We also set the parameters that we want to give to `fit()` when it gets called (`optimizer`, `epochs`, `batch_size`, and `verbose`). Now `kc_model` can be used inside of scikit-learn like any other estimator.

Before we actually do that, there are a couple of loose ends to clean up.

One issue is that scikit-learn's cross-validation function `cross_val_score()` doesn't want the one-hot encoded version of our label data. It wants the original versions that contain lists of integers. It so happens that we've been saving the original labels all this time in their own variables, so we have just what the routine needs. What a coincidence!

The other issue has to do with what data we pass to the cross-validation system. As we've done before, we'll simply pretend that we don't have a

validation set, and treat the training data as if it was our entire dataset. We'll let the cross-validator manage the train-validation split for us.

Now let's get this cross-validation going. There are two tasks to perform. First, we'll make the object that drives the cross-validation process. Let's use our old friend `StratifiedKFold()` from Bonus Chapter 1 with 10 splits. We'll shuffle the data, and we'll set the optional `random_state` variable to the value of `random_seed` that we already have around. That's useful for debugging.

We can use Listing B3-9 to make the `StratifiedKFold` object.

```
from sklearn.model_selection import StratifiedKFold

kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=random_seed)
```

Listing B3-9: Creating the `StratifiedKFold` object that will build the cross-validation training and test sets.

Now we're ready to go. Using the same techniques that we saw in Bonus Chapter 1, we just tell scikit-learn to run the cross-validator and track the scores by calling `cross_val_score()` with our model, our training data and original labels, and our folding object. Listing B3-10 shows the code.

```
from sklearn.model_selection import cross_val_score

results = cross_val_score(kc_model, X_train, original_y_train,
                          cv=kfold, verbose=0)
```

Listing B3-10: Running cross-validation from scikit-learn, using `kc_model`, our Keras model in a wrapper.]

Putting it all together, we get Listing B3-11.

```
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras.constraints import MaxNorm
from tensorflow.keras import backend as tensorflow.keras_backend
from tensorflow.keras.utils import np_utils
from tensorflow.keras.models import load_model
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier
from tensorflow.keras.utils import to_categorical

from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import cross_val_score
import numpy as np

random_seed = 42
np.random.seed(random_seed)

# load MNIST data and save sizes
(X_train, y_train), (X_test, y_test) = mnist.load_data()
image_height = X_train.shape[1]
image_width = X_train.shape[2]
number_of_pixels = image_height * image_width
```

```
# convert to floating-point
X_train = keras_backend.cast_to_floatx(X_train)
X_test = keras_backend.cast_to_floatx(X_test)

# scale data to range [0, 1]
X_train /= 255.0
X_test /= 255.0

# save y_train and y_test for use when cross-validating
original_y_train = y_train
original_y_test = y_test

# replace label data with one-hot encoded versions
number_of_classes = 1 + max(np.append(y_train, y_test))
y_train = to_categorical(y_train, num_classes=number_of_classes)
y_test = to_categorical(y_test, num_classes=number_of_classes)

# reshape samples to 2D grid, one line per image
X_train = X_train.reshape(X_train.shape[0], number_of_pixels)
X_test = X_test.reshape(X_test.shape[0], number_of_pixels)

def make_model(number_of_layers=2, neurons_per_layer=32,
               dropout_ratio=0.2, optimizer='adam'):
    model = Sequential()

    # first layer is special, because it sets input_shape
    model.add(Dense(neurons_per_layer, input_shape=[number_of_pixels],
                    activation='relu', kernel_constraint=MaxNorm(3)))
    model.add(Dropout(dropout_ratio))
    # now add in all the rest of the dense-dropout layers
    for i in range(number_of_layers-1):
        model.add(Dense(neurons_per_layer, activation='relu',
                        kernel_constraint=MaxNorm(3)))
        model.add(Dropout(dropout_ratio))
    # finish up with a softmax layer with 10 outputs
    model.add(Dense(number_of_classes, kernel_initializer='normal',
          activation='softmax'))
    # compile the model and return it
    model.compile(loss='categorical_crossentropy',
                  optimizer=optimizer, metrics=['accuracy'])
    return model

kc_model = KerasClassifier(build_fn=make_model,
                           number_of_layers=2, neurons_per_layer=32,
                           optimizer='adam',
                           epochs=100, batch_size=256, verbose=0)

kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=random_seed)
results = cross_val_score(kc_model, X_train, original_y_train,
                          cv=kfold, verbose=0)
print("results = {}\nresults.mean = {}".format(results, results.mean()))
```

Listing B3-11: Doing cross-validation with our MNIST data

Running this code gives us the output shown in Listing B3-12.

```
results =[ 0.95221445  0.95019157  0.95617397  0.9525  0.95116667
           0.96166028  0.95999333  0.95415903  0.95797899  0.95346898]
results.mean=0.9549507265070032
```

Listing B3-12: Our first results running cross-validation on our simple Keras model

So the cross-validation run is telling us that on the original dataset of 60,000 images, we got a performance of a bit more than 95% accuracy. That's a just about the same as what we saw graphically for this model way back in Figure B3-13, where the validation accuracy was just a smidge better than 95%.

That's reassuring. It says that this whole wrapping and cross-validating scheme is producing the same results that we got when we trained and tested the model ourselves.

Although cross-validation didn't gain us anything in this example, since we already had a great validation set, now we know how to evaluate a model if we don't have such a test set handy.

Cross-Validation with Normalization

Earlier we said that something was missing from this process. The thing we left out was normalizing the data before each run of cross-validation.

We got away with it in this case because we already normalized the training data to the range [0,1] when we divided it by 255. So when cross-validation grabs a random 90% of these samples and trains on them, it's likely to get samples that run from 0 to 1.

But that's only because we've already normalized our data and things are very simple. In general, the data that's going to get chosen from our database and used for cross-validation won't be normalized to the range [0,1]. It's up to us to get that normalization in there, and then apply that same transform to the part of the data that was set aside for testing in that run.

Happily, that's easy. We just build a pipeline.

As we saw in Bonus Chapter 1, we can normalize the particular piece of training data that's built for each pass through cross-validation by building a `Pipeline` object composed of two steps: a normalizer followed by our model.

Let's do this by first making our objects, and then assembling them into a `Pipeline` object. For demonstration purposes our pipeline will contain a `MinMaxScaler` from scikit-learn, followed by our model. The `MinMaxScaler` is attractive because there are no parameters to set or options to pick. `MinMaxScaler` isn't a perfect choice for this case, since it adjusts each pixel independently, which could lead to bright or dark spots. But it should be okay for demonstration on this data. Listing B3-13 shows how to build the pipeline.

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import MinMaxScaler

estimators = []
```

```
estimators.append(('normalize_step', MinMaxScaler()))
estimators.append(('model_step', kc_model))
pipeline = Pipeline(estimators)
```

Listing B3-13: Making a two-step pipeline with named components

Constructing a pipeline this way is useful when we want to later refer to the individual steps. We'll need to do that soon when we use grid searching.

But for this cross-validation step, we don't need that kind of access. We'll often see code that builds the pipeline in one line, using the shortcut `make_pipeline()` function. Listing B3-14 shows the code.

```
pipeline = make_pipeline(MinMaxScaler(), kc_model)
```

Listing B3-14: Making a pipeline using the shorthand notation, without names for the individual steps

These two `pipeline` objects are the same. The only difference is that we've given our own names to the steps in the first version.

To use our `pipeline` object, we just give it to `cross_val_score()` in place of a model (or wrapped model). Scikit-learn will recognize that it's a pipeline and take care of all the rest. So each time through the loop, `cross_val_score()` will select one of folds as a validation set. The rest of the data will be the training set for that run. It will give that training data to the `MinMaxScaler()` (using all the default arguments). Once the data has been analyzed, the transformation found by the `MinMaxScaler` will be applied to both the current training data and validation data. The model will then learn from the training data. When training is done, the system will run the transformed validation set through the model, predict its categories, compare those to the labels, and compute error scores.

That's a huge amount of work, all from one function call! That call is in Listing B3-15, where we simply replace `kc_model` in Listing B3-11 with `pipeline`.

```
results = cross_val_score(pipeline, X_train, original_y_train,
                          cv=kfold, verbose=0)
```

Listing B3-15: We cross-validate with our pipeline just as we do for a model.

Putting these new lines together, we get a new block of code that replaces the last few lines at the end of Listing B3-11. The new code, and the output we get from running the process with it, is shown in Listing B3-16.

```
pipeline = make_pipeline(MinMaxScaler(), kc_model)
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=random_seed)
results = cross_val_score(pipeline, X_train, original_y_train,
                          cv=kfold, verbose=2)

print("results = {}\nresults.mean = {}".format(results, results.mean()))
```

```
results =[ 0.95454545  0.9508579   0.95534078  0. 0.95283333
    0.963994  0.95749292  0.95415903  0.95781224  0.95380253]
results.mean=0.9552838183370085
```

Listing B3-16: Setting up and calling the cross-validator with our pipeline

This value of about 0.954 matches our previous average accuracy.

In this case, the extra work of building the pipeline with the normalizer didn't pay off with any new benefits or accuracy. That's probably because randomly removing a batch of samples from the training set and then normalizing probably had little effect on the samples, since they were already normalized.

But that's definitely not going to be true for all datasets, and we should never take it for granted. Unless we are certain about the input data and its statistics, using a pipeline and processing our data is usually worth the extra effort on our part. It takes little extra computing time to compute and apply most common transformations, compared to training and testing.

We picked a `MinMaxScaler` here pretty much arbitrarily, but as we know, different data sets require different types of pre-processing. Using the pipeline mechanism, we can apply whatever steps we need.

Cross-validation is a great way to get a handle on the quality of our model. It's not so great when training times start to push our patience, since every fold is essentially a brand-new full-length training and testing process. Using 10 folds requires training and then testing our model 10 times in a row.

The time required can add up fast. But if we don't have a good validation set, then running cross-validation tests on small bits of our database, using different parameters, can teach us a lot about what's going on in our data. That knowledge can in turn help us design an efficient larger network to process the whole database.

Hyperparameter Searching

This section's notebook continues Bonus03-Keras-2-scikit-learn.ipynb.

We've been using some arbitrarily hand-picked numbers in this chapter. For instance, we've settled on an architecture with 2 layers of 32 neurons each, but not for any particular reason.

Going forward, we'll often refer to "parameters," rather than the more awkward "hyperparameters." As well as being more readable, this makes sense when we consider that many of these values are provided to the system as parameter, or arguments, of functions.

We can use the grid searching algorithms offered by scikit-learn to help us out. With those routines, we can automatically try out all the different combinations of multiple settings for multiple parameters. We could do this ourselves with some nested loops, but it's easier to relax and let scikit-learn do the driving.

The grid searching object `GridSearchCV` will try out every combination of the parameters we give it, and measure each model's performance using cross-validation. By default, it uses 3 folds to save time, but we can increase that with an optional argument.

We think of this as "searching" because we imagine that each combination of parameters is a point in some very high-dimensional space, called the *search space*. Each point in search space represents some combination of parameters, and the value of that combination (that is, the accuracy or loss that results from training a model with those parameters) is the value associated with that point. The intuition is that we're searching through this space, wandering from point to point and region to region, looking for the point that has the highest performance.

Figure B3-16 shows an example of a search space with two dimensions. The idea really comes into its own when we're dealing with many more dimensions. Though we can't visualize them, we can make the analogy to something like Figure B3-16 and talk about two sets of parameters being close or far apart, and even talk about regions of the space where it looks like we're finding good results.

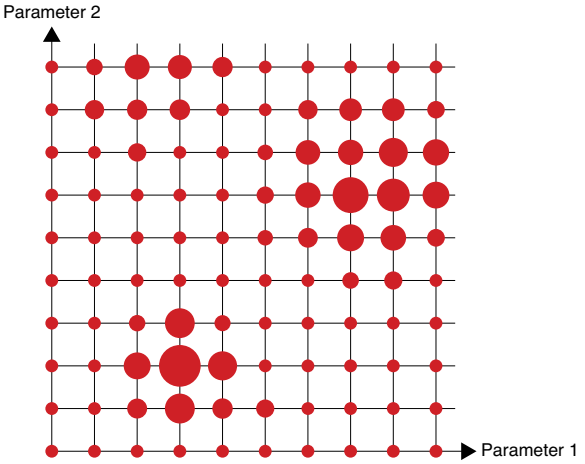


Figure B3-16: A two-dimensional search space

For each pairing of values of two parameters in Figure B3-16, the size of the circle shows the quality of the result, with larger circles better than smaller ones. It looks like there's a small but high-quality region in the lower left, and two other promising regions in the upper right and upper left. Now that we know where the good results can be found, we can investigate those areas with a finer resolution to find the best combination of parameters.

As we discussed earlier, often we'll use just a subset of our training data when searching, so that it runs more quickly. When we've found the best values for our model's parameters using this smaller database, we can use a larger version and work our way up to the full dataset.

To keep things simple, for this discussion we'll continue to use MNIST, and the whole `X_train` database while searching.

We'll want to use our normalizing pipeline again, since in general that will be necessary, or at least a very good idea.

As we saw in Bonus Chapter 1, when we prepare a pipeline for grid searching, we need to tell `GridSearchCV` where each parameter it's searching

through ought to be routed. This means we need to identify different steps in the pipeline. That's easy if we use the pipeline-building method of Listing B3-13, where we gave a name to each step.

As we saw in Bonus Chapter 1, referring to parameters inside pipelines is somewhat baroque. Let's recap briefly.

We build a dictionary, where each key is the name of a parameter to one of the steps in our pipeline, and its value is a list of all the values we'd like to explore. Each name is formed by combining the step name in our pipeline and the name of the parameter with *two* underscore characters, as in `step__parameter`. With some typefaces the two underscores look like just one big underscore, which is unfortunate, but that's how it is. For comparison, here is `one_underscore` and here are `two__underscores`.

Let's build a dictionary to search through three of our model's parameters: the number of dense layers (each with dropout), the number of neurons per dense layer, and, just for curiosity's sake, two different optimizers. Listing B3-17 shows this dictionary. Note that the keys are not strings.

```
param_grid = dict(model__number_of_layers=[ 2, 3, 4 ],
                  model__neurons_per_layer=[ 20, 30, 40 ],
                  model__optimizer=[ 'adam', 'adadelta' ])
```

Listing B3-17: A dictionary of parameters that we'd like to use for searching. Each key is a parameter named by gluing together the name of the pipeline step with the name of its parameter, with two underscores in between. Each value is a list of settings to be tried.

Now that we have our dictionary, we can use the pipeline we created above in Listing B3-14 and create our searching object, as in Listing B3-18.

```
grid_searcher = GridSearchCV(estimator=pipeline,
                             param_grid=param_grid, verbose=2)
```

Listing B3-18: Creating a `GridSearchCV` object that will go through our parameter grid, assemble a model for every combination of options, and cross-validate that model

Now we're ready to roll. We just call the searcher's `fit()` routine with our data, and let it run. Listing B3-19 puts together the searching code. We're going to suffix each variable with a `1` because we're going to run another grid search below, which will be version 2.

```
from sklearn.model_selection import GridSearchCV

param_grid1 = dict(model__number_of_layers=[ 2, 3, 4 ],
                   model__neurons_per_layer=[ 20, 30, 40 ],
                   model__optimizer=[ 'adam', 'adadelta' ])
grid_searcher1 = GridSearchCV(estimator=pipeline,
                              param_grid=param_grid1, verbose=2)

search_results1 = grid_searcher1.fit(X_train, original_y_train)
```

Listing B3-19: Combining the grid construction with `GridSearchCV` object construction, and then calling `fit()` to run the search.

Warning! Grid searches are *slow*.

The total number of full 3-fold cross-validation runs it will perform is reported by the searcher as soon as it starts up. Listing B3-20 shows what we'd see for the search we just defined.

```
Fitting 3 folds for each of 18 candidates, totaling 54 fits
```

Listing B3-20: As this output from the grid-search `fit()` shows, this exhaustive cross-validation will call `fit()` on our model 54 times.

The number 54 comes from multiplying the number of folds (3 by default) by the number of combinations of variables. In our case, we have three lists of variables, with lengths 3, 3, and 2. The total number of possibilities is found by multiplying these together: $3 \times 3 \times 2 = 18$. Since each possibility has to go through three steps of cross-validation, we will call `fit()` a total of $3 \times 18 = 54$ times.

If it takes a minute to train and evaluate the model, it will take about an hour to run this search.

The variable `search_results1` we get back contains a lot of information. One of the objects in `search_results1` is a dictionary called `cv_results_` (recall that all of scikit-learn's internal variables are suffixed with an underscore). The `cv_results_` dictionary contains detailed information on the cross-validation results.

Since we're interested in finding the best combination of parameters, two dictionary items are of particular interest. The `'params'` item tells us which set of parameters corresponds to each score. The `'mean_test_score'` item tells us the average value that came out of the cross-validation for each set of parameters.

Let's look first at the `'params'` entry, shown in Listing B3-21.

```
search_results1.cv_results_['params']
{'neurons_per_layer': 20, 'number_of_layers': 2, 'optimizer': 'adam'},
{'neurons_per_layer': 20, 'number_of_layers': 2, 'optimizer': 'adadelta'},
{'neurons_per_layer': 20, 'number_of_layers': 3, 'optimizer': 'adam'},
{'neurons_per_layer': 20, 'number_of_layers': 3, 'optimizer': 'adadelta'},
{'neurons_per_layer': 20, 'number_of_layers': 4, 'optimizer': 'adam'},
{'neurons_per_layer': 20, 'number_of_layers': 4, 'optimizer': 'adadelta'},
{'neurons_per_layer': 30, 'number_of_layers': 2, 'optimizer': 'adam'},
{'neurons_per_layer': 30, 'number_of_layers': 2, 'optimizer': 'adadelta'},
{'neurons_per_layer': 30, 'number_of_layers': 3, 'optimizer': 'adam'},
{'neurons_per_layer': 30, 'number_of_layers': 3, 'optimizer': 'adadelta'},
{'neurons_per_layer': 30, 'number_of_layers': 4, 'optimizer': 'adam'},
{'neurons_per_layer': 30, 'number_of_layers': 4, 'optimizer': 'adadelta'},
{'neurons_per_layer': 40, 'number_of_layers': 2, 'optimizer': 'adam'},
{'neurons_per_layer': 40, 'number_of_layers': 2, 'optimizer': 'adadelta'},
{'neurons_per_layer': 40, 'number_of_layers': 3, 'optimizer': 'adam'},
{'neurons_per_layer': 40, 'number_of_layers': 3, 'optimizer': 'adadelta'},
{'neurons_per_layer': 40, 'number_of_layers': 4, 'optimizer': 'adam'},
{'neurons_per_layer': 40, 'number_of_layers': 4, 'optimizer': 'adadelta'}
```

Listing B3-21: The contents of `search_results1.cv_results_['params']`. The prefix `model_step__` before each parameter has been removed to make the list fit and easier to read.

We can see that the searching algorithm checked every combination, but it proceeded in a different order than in our `param_grid1` dictionary. The outer loop ran through the 3 values of `neurons_per_layer`, the loop nested inside of that ran through the 3 values of `number_of_layers`, and finally the innermost loop ran through the 2 values of `optimizer`. Because Python dictionaries are not guaranteed to return their results in any particular order, we won't be able to predict in what order the search will proceed before we run it.

The numerical data that describes our cross-validation test performance is in `mean_test_score`, as shown in Listing B3-22.

```
search_results1.cv_results_['mean_test_score']

array([ 0.92901667,  0.91761667,  0.93081667,  0.91146667,  0.92288333,
        0.90051667,  0.9472    ,  0.93373333,  0.94561667,  0.93166667,
        0.94333333,  0.92621667,  0.95566667,  0.9424    ,  0.95376667,
        0.94185   ,  0.95413333,  0.9402    ])
```

Listing B3-22: The cross-validation scores for our search

Using NumPy's utility `argmax()` we can find the index of the largest value in this list, and then extract the corresponding element from the `'params'` item, so we can see which set of parameters gave us the best score. Listing B3-23 shows this step.

```
best_index1 = np.argmax(search_results1.cv_results_['mean_test_score'])
print('best set of parameters:\n  index {}\n  {}\n'.format(
    best_index1, search_results1.cv_results_['params'][best_index1]))

best set of parameters:
  index 12
  {'model_step__optimizer': 'adam',
   'model_step__neurons_per_layer': 40,
   'model_step__number_of_layers': 2}
```

Listing B3-23: A snippet of code that finds the best test score from our cross-validation results, and prints the parameters that gave us that score. The output was slightly reformatted to fit better.

So our best combination used 2 layers, with 40 neurons per layer, and the Adam optimizer. But how much better was this than the other combinations? Let's plot all the values of `mean_test_scores` so we can see how every combination performed, as in Figure B3-17.

Guided by the labels, we can interpret this graph as having three major sections, one each for 20, 30, and 40 neurons. Within each section we have three pairs, one pair each for 2, 3, and 4 layers. Finally, each pair of values shows the performance for Adam and then Adadelta.

Each of these innermost pairs slopes downwards, so we can say that Adam consistently outperformed Adadelta.

The general trend in each second-level group is also downwards, so adding more layers usually caused a loss of performance.

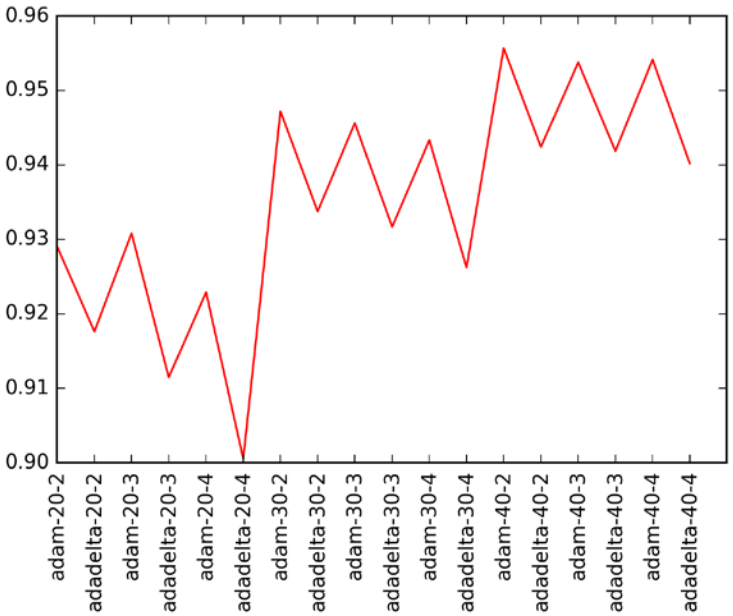


Figure B3-17: The values of `mean_test_scores` resulting from searching for the best cross-validation results for every combination of the parameters in our dictionary of Listing B3-19. The best accuracy came from 2 layers of 40 neurons each, using the Adam optimizer.

The general trend in the largest groups is going upwards, suggesting that more neurons are better than fewer.

As Listing B3-23 reports, the best set of parameters used 2 layers of 40 neurons each, optimized with Adam.

It's interesting to note that the worst performance by far was the result of 4 dense-dropout layers of 20 neurons each. So that's a structure to avoid for this data.

Keep in mind that we're always referring to "layers" as our combination of dense and dropout layers, using our default dropout rate of 0.2.

Let's search around this area of the parameter space. Since it seems fewer layers are working better than more, let's try searching for models with 1 or 2 layers. And since more neurons are performing better than fewer, we'll try going up from 40 and look at some larger values.

We could explore other optimizers, but this time around we'll stick with Adam.

There's no hard and fast rule for making these choices. We need to use our judgment based on our knowledge of our model and data, coupled with the results of our experiments, to guide our search strategy. If we search with too fine a grid we can waste a lot of time, but if we use too coarse a grid we could miss a big spike in performance. Generally speaking, searching for performance is a task that rewards both intuition and analysis.

Listing B3-24 shows the dictionary for our second search.

```
param_grid2 = dict(model_step__number_of_layers=[ 1, 2],
                   model_step__neurons_per_layer=[ 50, 80, 110, 140, 170 ])
```

Listing B3-24: The dictionary for our second parameter search.

The results are plotted in Figure B3-18.

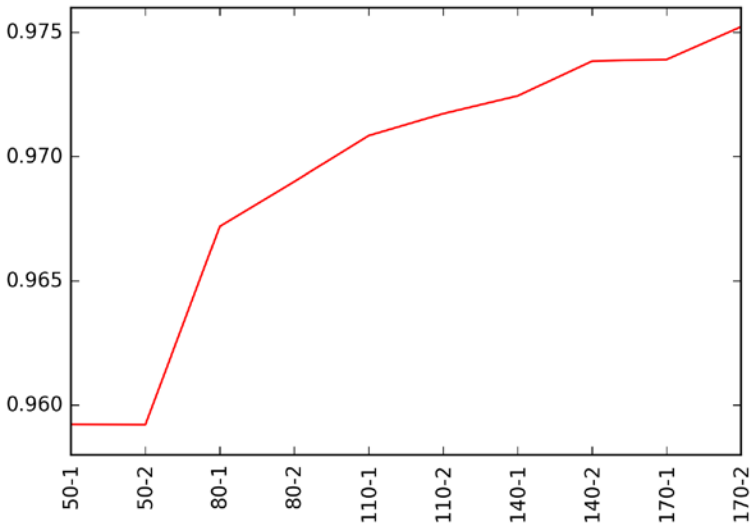


Figure B3-18: Searching 1 and 2 layer networks for layers with increasing numbers of neurons. The results alternate with the 1-layer version, then the 2-layer.

This tells us that more neurons keep working better, suggesting that our hunch was right. And 2-layer models are looking consistently better than 1-layer.

Let's crank up both the number of neurons and the search range quite a bit. Listing B3-25 shows the dictionary for our third search.

```
param_grid3 = dict(model_step__number_of_layers=[ 1, 2 ],
                   model_step__neurons_per_layer=[ 180, 280, 380, 480, 580 ])
```

Listing B3-25: The dictionary for our third parameter search.

The results are in Figure B3-19.

Figure B3-19 shows us that the best performance came from 2 layers of 280 neurons each. As we added more neurons, performance started to decline, though slowly. Perhaps this is due to overfitting, though we'd have to look more closely to be sure.

Notice the size of the vertical scale. Our first graph in Figure B3-17 showed an improvement of about 0.055 from the worst performer to the best, while in our most recent graph Figure B3-19 the difference is only about 0.0035, which is about 1/15 the size.

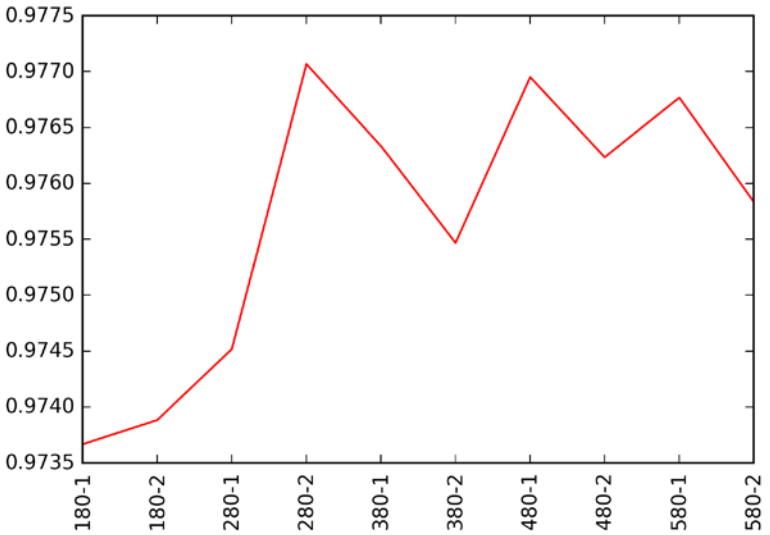


Figure B3-19: Searching 1 and 2 layer networks for layers with large numbers of neurons. The results alternate with the 1-layer version, then the 2-layer.

The overall feeling of the curve, though it's jumping around when we're zoomed in so much, seems to be flattening out. We might be pretty close to the best choice for this set of parameters.

We'll stop searching here, but we could continue to try different values for all of these parameters, or some we didn't even try (like any of the `Normalization` object's parameters, or `dropout_ratio` in our own model).

The time required to do a full grid search can quickly become impractical, because the searcher exhaustively tests every possible combination of the search parameters. So it's always a good idea to use the smallest number of combinations we can get away with, and to search with the smallest amount of data that will give us a reasonable prediction of the model's performance.

A good strategy is to start with searches that cover broad ranges with just a few values. When we see where the model is performing best, we can then run another, denser search to explore the area around that zone. This is called *multiresolution searching*, and it's just an algorithmic version of what we do when we look for something in the real world. Say we're looking for a book in the library. We have a call number, so we wander around to the right section of the library, then the right stack, the right shelf, and so on, using a series of ever-narrower searches until we find our book.

We do the same thing with ever-smaller searches until we find a combination of parameter values that work the best.

A useful alternative to the exhaustive search performed by the grid searcher is provided by scikit-learn's `RandomizedSearchCV` algorithm. As discussed in Bonus Chapter 1, this variation on the grid search picks an

unexplored random combination of the search parameters on each run. We could for instance just search a third of the total combinations. We'd get our answer back 3 times sooner than a complete grid search, but it would be incomplete. It would be incomplete in a nice way, though, giving us a roughly equal scattering of points throughout the parameter space. This might be enough to guide our choice for a smaller, more focused search.

Convolution Networks

This section's notebook is Bonus03-Keras-3-CNN.ipynb.

Let's build some *convolutional neural networks*, also called *convnets*, or more commonly, *CNNs*.

In Chapters 16, 17, and 23 we surveyed some of the remarkable things we can do with convnets. Recall that each convolution layer holds a collection of *filters*, or *kernels*, which are rectangles of numbers (often a small square that is 3, 5, or 7 elements on a side). When we use 2D convolution layers with images, each filter in the first layer is applied in turn to every pixel in the input. The output of the filter becomes the value of that element in a new tensor produced at the layer's output. If there are multiple filters, then the output tensor contains multiple *channels*, just like the red, green, and blue channels of a color image.

Although the original input to the first convolution layer of a model is often an image, the data that flows through the rest of the network is not. If a layer has 32 filters, for example, then the output will have 32 channels. It may have the same width and height as the input (though we'll see those measures often change as well), but it's not really an "image" any more. So while it's tempting to casually speak about convolution layers as processing "images" made of "pixels," it's a better idea to refer to them as tensors, made of elements.

Recall from Chapter 16 that we can characterize a convolution layer by the number of dimensions in which the filters are moved. If the filter is moved in just one dimension (for example, down), then we call it a 1D convolution layer. Typically, when we work with images, we slide our filters over the 2D width and height of the tensor, so we usually use 2D convolution layers for image processing. Keras also offers 3D convolution layers for working with volumetric data.

In this chapter, we'll focus on images, so we'll be using 2D convolution layers.

In the following sections we'll see how to build and train our own CNNs. As we'll discuss later, in practice we don't often build and train a new CNN from the ground up. Instead, we usually try to start with an existing network whenever possible, and specialize it for our task by perhaps modifying it, and then training it some more with our own data. Such *transfer learning* is appealing because we get to start with an existing architecture that is known to work well, and we save the time (sometimes days or weeks) that was invested in training the model we're building upon. We also get

the benefit of the data that network was trained on, which might not be available to us.

But it's important to know how to build our own from scratch. This lets us start fresh when we need to, and gives us the tools to modify an existing network when we want to. Whether we're working with our own model or one we've adopted, knowing what's going on inside will help us diagnose problems and get the best performance out of our model.

Let's begin by setting up a few basic ideas that will be helpful when we build our convnets.

Utility Layers

We'll briefly recap some of the utility layers that we saw throughout the book, focusing on those that are useful in convnets.

Like the dropout layer, these aren't fully-fledged computational layers. Instead, they're often "supplemental" layers that tell Keras how to process or manipulate data that flows through the layer, or how to affect a previous layer. Keras structures these as layers so that we can think of our model consistently as a stack of layers.

Most of the layers described below are available in 1, 2, and 3 dimensional forms. When working with images, we almost always use the 2D versions, so that's all we'll cover here.

A *flatten* layer takes a tensor of any number of dimensions and lines up all of its contents into a single one-dimensional list. It always does this in the same order, so we can predict where each element in the tensor will appear in that list (we usually don't care what in what order the elements are listed, as long as it's consistent from one sample to the next). Keras calls this layer `Flatten`.

A *pooling* layer looks at elements that make up a block in the input, calculates a single value from them, and saves that single value to the output in place of all the input elements. The most common use of pooling is to reduce the size of its input. For example, if the blocks are 2 by 2, and they don't overlap, the output will be half the width and height of the input. Keras offers two types of pooling layers. The `MaxPooling2D` layer finds the largest value in each block. We can tell it the *size* of the block to use, and the *stride*, or how many elements to move horizontally and vertically after each block. Commonly we use blocks that are 2 by 2 with a stride of 2 in each dimension. The `AveragePooling2D` layer works the same way, but instead computes the average value of each block.

As we discussed in Chapter 16, pooling layers after convolution layers are falling out of favor, replaced by *striding* within the convolution layer itself to achieve a similar result. The striding and pooling approaches produce similar but different results. Usually we don't care about that difference, but sometimes it matters, so pooling layers are still sometimes used.

A *cropping* layer removes a tensor's outermost elements, leaving just the inner rectangle. The Keras layer called `Cropping2D` takes arguments which let us describe how many elements to remove from each of the four sides.

An *upsampling* layer is designed to make the input tensor larger. Each element is just repeated horizontally and vertically by the given number of times. Keras calls this layer `UpSampling2D` (note the upper-case S in the middle of the name).

As we mentioned in Chapter 16, an alternative to an explicit upsampling layer after a convolution layer is to use transposed convolution (or fractional striding) in the convolution layer itself. Like normal striding and max pooling, transposed convolution produces a similar but different result compared to upsampling.

A *batchnorm* layer performs regularization on each batch of data flowing through it, giving it an average of 0 and a standard deviation of 1. This helps keep weights from growing too large.

A *noise* layer adds some random noise to every element in the tensor. This is rarely used, but can be helpful if some neurons seem to be overly-aggressive in matching specific features that are not ultimately important.

Finally, a *zero padding* layer places 0's around the perimeter of the input. This is usually so that convolution kernels will not "fall off the edge" and try to access non-existent data. Keras calls this the `ZeroPadding2D` layer (note the upper-case P). Because Keras now offers this feature in the convolution layers themselves, explicit zero-padding before convolution is rare in Keras 2 models.

A recap of our schematic symbols for the major types of layers is shown in Figure B3-20.

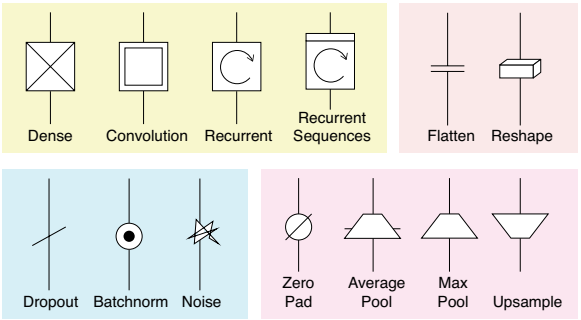


Figure B3-20: Schematic symbols for different layers

Preparing the Data for A CNN

We'll continue to use MNIST for our example data set.

We prepare our MNIST data for a convnet with *almost* the same process as we've been doing so far when the first layer was a `Dense`, or fully-connected, layer.

The difference is in the shape of the feature data. So far, we've been shaping our feature data in a 2D grid, with one row per image. Each row held all the pixels for that image.

Something important happened when we flattened out our image to make that grid: we lost the spatial information that tells us which pixels are near one another vertically (technically, it's still there, but definitely not in a structure that's easily useful). A great thing about CNNs is that they work with inputs as multidimensional tensors, not long 1D lists. For instance, the receptive field for a filter covers a group of spatially-related elements.

When working with CNNs there's no need to flatten out input 2D grids of pixels. We'll maintain them instead as three-dimensional volumes, where each input image has a height, width, and depth.

One important use of this third dimension is to bundle together the channels of data that represent color images. A typical digital color image has three channels, one each for red, green, and blue. So if we stack these up, we'll get a block of 3 layers. On the other hand, a picture that has been prepared for printing usually has four channels: cyan, magenta, yellow, and black. This requires a block of 4 layers. Figure B3-21 shows this idea.

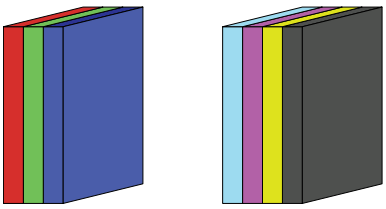


Figure B3-21: Multiple 2D grids of pixels are often used to represent richer types of images. (a) A typical digital image in color uses three channels, one each for red, green, and blue. (b) A file prepared for printing may have four channels, for cyan, magenta, yellow, and black.

The MNIST data is black and white, so we have just a single channel of pixel data. But we still have to explicitly tell Keras that we have just that one channel, by making it one of the dimensions of our input tensor.

The order in which we name our dimensions depends on whether we're using the `channels_first` or `channels_last` option, as we discussed at the start of Bonus Chapter 2.

We'll use `channels_last` in this section.

By adding a channel dimension, each MNIST image will become a 3D block with dimensions 28 by 28 by 1. Our input data structure will contain 60,000 of these 3D blocks. That means the complete tensor will have a first dimension of 60,000, followed by the shape of each image.

Using the `channels_last` convention, this tensor will have dimensions of 60,000 by 28 by 28 by 1.

We can't draw a 4D tensor, but we can show lots of 3D tensors in a list. Figure B3-22 uses that approach to picture our dataset.

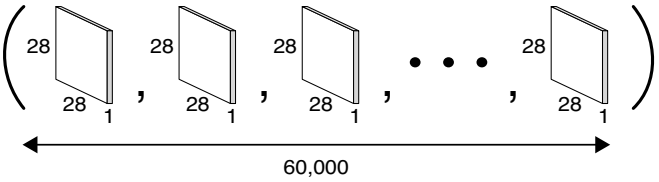


Figure B3-22: Each image in the input will be reshaped as a 3D block with dimensions 28 by 28 by 1, since we're using the `channels_last` convention. Then we stack all 60,000 of these 3D blocks together to make a 4D tensor of shape 60,000 by 28 by 28 by 1, which will serve as input to our CNN.

As we discussed earlier, it can be easier to think of this not as a 4D structure but instead as a sequence of nested lists: the outermost list contains 60,000 images, each image contains 28 rows, each row contains 28 elements, and each element has 1 channel.

This means that each pixel is named with *four* numbers: the image number, y position, x position, and channel number, in that order.

Convnets work best with input data scaled from –1 to 1 [Karpathy16b]. This means we can't just divide every pixel by 255. Instead, we'll use the NumPy function `interp()` to convert each input value in the range [0,255] to the range [–1,1], shown in Listing B3-26.

```
X_train = np.interp(X_train, [0, 255], [-1,1])
X_test = np.interp(X_test, [0, 255], [-1,1])
```

Listing B3-26: Using NumPy's `interp()` routine to convert all the input values from [0,255] to [–1,1]

Now we'll re-shape the data into the shape we just discussed. We just tell NumPy how to take our original version of `X_train`, which was 60,000 by 28 by 28, and reshape it into a 4D tensor that's 60,000 by 28 by 28 by 1. We're not changing the total number of elements, so this is a valid request. We just hand it the dimensions we want it to use. Listing B3-27 shows the code.

```
# reshape sample data to 4D tensor using channels_last convention
X_train = X_train.reshape(X_train.shape[0], image_height, image_width, 1)
X_test = X_test.reshape(X_test.shape[0], image_height, image_width, 1)
```

Listing B3-27: How to transform our input MNIST data into the 4D tensor that our CNN expects.

We'll place these re-shaping lines right after the scaling step. For completeness, Listing B3-28 shows all the pre-processing in one place. This includes all the `import` statements we'll be needing going forward. Aside from the `import` statements, and the final re-shaping, this pre-processing is identical to what we've been doing so far. A CNN is, after all, just another deep neural network, but jazzed up with some new types of layers.

Note that we're assuming that the `channels_last` option has been selected in the Keras configuration file. If that's not the case, either change the file, or import the backend and include a call to set the value of `image_data_format`.

```
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers.core import Dense, Dropout, Activation, Flatten
from tensorflow.keras.layers.convolutional import Conv2D, MaxPooling2D
from tensorflow.keras.constraints import MaxNorm
from tensorflow.keras.optimizers import Adam, SGD, RMSprop
from tensorflow.keras import backend as tensorflow.keras_backend
from tensorflow.keras.utils import np_utils
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.utils.np_utils import to_categorical
import numpy as np

random_seed = 42
np.random.seed(random_seed)

# load MNIST data and save sizes
(X_train, y_train), (X_test, y_test) = mnist.load_data()
image_height = X_train.shape[1]
image_width = X_train.shape[2]
number_of_pixels = image_height * image_width

# convert to floating-point
X_train = keras_backend.cast_to_floatx(X_train)
X_test = keras_backend.cast_to_floatx(X_test)

# scale data to range [-1, 1]
X_train = np.interp(X_train, [0, 255], [-1,1])
X_test = np.interp(X_test, [0, 255], [-1,1])

# save original y_train and y_test
original_y_train = y_train
original_y_test = y_test

# replace label data with one-hot encoded versions
number_of_classes = 1 + max(np.append(y_train, y_test))
y_train = to_categorical(y_train, num_classes=number_of_classes)
y_test = to_categorical(y_test, num_classes=number_of_classes)

# reshape sample data to 4D tensor using channels_last convention
X_train = X_train.reshape(X_train.shape[0], image_height, image_width, 1)
X_test = X_test.reshape(X_test.shape[0], image_height, image_width, 1)
```

Listing B3-28: The pre-processing step for our CNNs to categorize MNIST data

Shaping the feature data into these 4D tensors is a necessary pre-processing step. It puts the data into the structure that is expected by the convolution layer that will sit at the start of our convnet.

Convolution Layers

Let's look more closely at how we define a convolution layer. We saw that Keras offers convolution layers in 1, 2, and 3 dimensions. We'll pick the 2D version, since that's what we usually use for image data like our running example of MNIST digits.

The layer has the name `Conv2D`, and we access it by importing it from the module `tensorflow.keras.layers.convolutional`.

The `Conv2D` layer takes two unnamed, mandatory arguments at the start of its argument list, followed by a variety of optional arguments.

The first mandatory argument is an integer specifying the number of filters the layer should manage. Recall from Chapter 16 that each filter is applied to the input independently, and produces its own output. So if our input has one channel (as our input does), and we use 5 filters in a convolution layer, the output will have 5 channels, as shown in Figure B3-23.

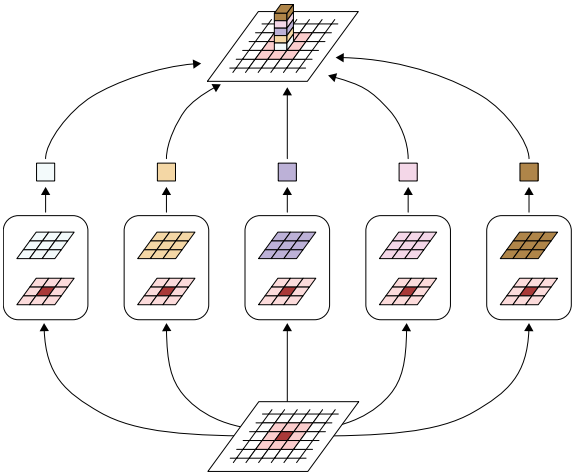


Figure B3-23: If we have a 1-channel input and 5 filters, the output tensor will have 5 channels, one for the output of each filter. Here we show the operation of the 5 filters on a single element of the input, producing an output with 5 channels.

The second argument to `Conv2D` is a list that gives the dimensions of the filters on this layer. In Keras, as in many libraries, all of the filters in a given layer are of the same size, so we only specify one filter size for the entire layer. Continuing our example from before, if we have 5 filters and each is 3 by 3, then these arguments would be `5,[3,3]`. This tells the layer to automatically allocate and initialize 5 volumes, each of shape 3 by 3 by 1 (the trailing 1 is the number of channels).

In practice, we almost always use square kernels, often of 3 or 5 elements on a side. Experience has shown that these sizes, coupled with reduction in the size of the input (either by pooling or striding), represents a good tradeoff of computation and results. Larger kernels are sometimes used, but they're not as common.

Keep in mind that these filter kernels are 3D volumes, since there's one channel in the kernel for each channel in the input. For example, suppose that our input image to the first layer is a color image, and that our layer is using 5 by 5 kernels. Since there are three channels, each filter is created as a block that is 5 by 5 by 3. This block is moved over the image in 2 dimensions (hence the name `Conv2D`), and at each element the 75 values from the input are multiplied with their corresponding 75 entries in the kernel, the results are all added together, and that's the output of this kernel at this element, as shown in Figure B3-24.

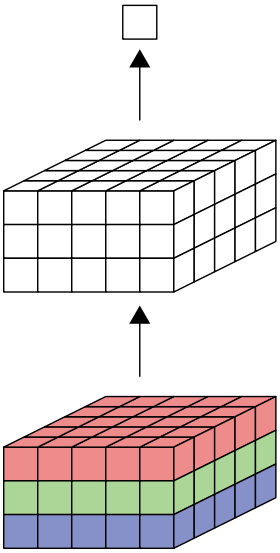


Figure B3-24: Each filter automatically holds as many channels as there are in the input.

In Figure B3-24, a 5 by 5 filter is being applied to a 3-channel input, so the system automatically gives the filter 3 channels as well. Each of the 75 values in the input (bottom) is multiplied by its corresponding value in the filter (middle), and all of those products are added together to produce a single number (top), which then passes through an activation function, producing the output of that filter for that location of the input.

If we have several of these filters in a given convolution layer, then we'll produce several outputs, just as in Figure B3-23, except now for a multi-channel input. Figure B3-25 shows the idea.

Keeping track of all of these shapes would be an administrative challenge, so Keras manages them for us. As a result, we can create sequences of convolution layers by doing nothing more than telling Keras how many filters we want to use on each layer, and what their footprint should be. Keras keeps track of the number of channels coming from the previous layer, and makes the filters of the necessary size, with no effort from us.

For example, let's suppose that we've made a convolution layer with 5 filters, each 5 by 5. Then every output it produces will have 5 channels. If the next layer is also a convolution layer, and we say that we want 2 filters

that are 3 by 3, Keras will automatically know to make each filter 5 channels deep, since that's what's coming out of the previous layer. In short, the number of channels in each filter is equal to the number of channels in the input, which in turn is the number of filters used in the previous convolution layer. Figure B3-26 shows this idea visually.

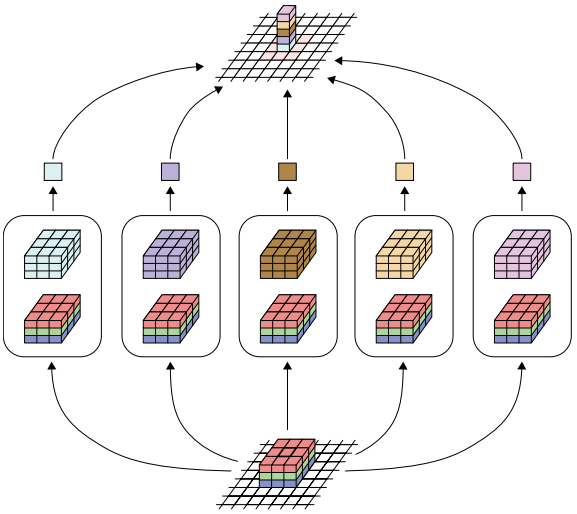


Figure B3-25: If we apply several filters to a multi-channel input, then each filter will also have multiple channels. The number of channels in the output is given by the number of filters that were used.

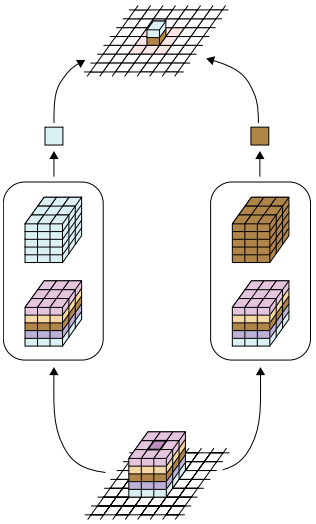


Figure B3-26: When one convolution filter follows another, the filters in the second layer are automatically configured to have as many channels as there are filters in the preceding layer.

This automatic sizing of filters is one of the great pleasures of using a library such as Keras. We don't have to manage any of this book-keeping ourselves, because the library already has everything it needs to know to always keep things straight.

Let's make a convolution layer with 15 filters, each 3 by 3. Listing B3-29 shows the code for making the layer and placing it into a model.

```
convolution_layer = Conv2D(15, (3, 3))
model.add(convolution_layer)
```

Listing B3-29: How to make a convolution layer with 15 filters that are each 3 by 3

In practice, we usually do this in one step, as we did with other layers earlier in the chapter. The single line that's usually used for this job is shown in Listing B3-30.

```
model.add(Conv2D(15, (3, 3)))
```

Listing B3-30: The more usual way to add a convolution layer to a model

The `Conv2D` layer accepts many optional arguments, all of which are described in the documentation. We'll discuss just the ones that we'll be needing.

We'll start with the arguments that are unique to convolution layers, and then touch on the ones we've already seen when using `Dense` layers. For convenience, we'll continue to refer to input tensor elements as "pixels," though as we discussed earlier that doesn't really make sense past the first layer.

A nice optional feature of the convolution layer is that we can include zero-padding inside the layer itself if we want it, rather than building an explicit zero-padding layer into the model. Even better, Keras can automatically compute how much zero padding we need if we want our output to have the same shape as our input. It uses the sizes of the filters, and our striding choices if necessary, and adds enough 0's around the outside to guarantee that the filters will never "fall off the edge" of the input.

To apply zero padding, we set the optional argument `padding` to the string `'same'`, meaning "make the output the same size as the input." The default value of padding is the string `'valid'`, which means "only place the filter where there is valid data available." This is a long-winded way of saying, "no padding."

We saw in Chapter 16 that striding allows us to move the filter by any distance on each step. We set the stride amounts with the optional parameter `strides`. This accepts a list of 2 numbers, giving the number of pixels to move horizontally and vertically. This list defaults to `(1,1)`. For example, if we set the stride values to `(2,2)`, then our output will be half the width and height as the input. Note that number of output channels is not affected by the stride, since that comes from the number of filters.

As a little convenience, we can set `strides` to just a single number, and it will use that for both directions. So instead of the list `(2,2)`, we can just give the single value `2`.

The last option we'll look at now is `activation`. Just like our previous discussion of `Dense` layers, every value produced by a convolution layer goes through an activation function. The default is a linear function, which in effect does nothing. We can set an activation function by providing its name as a string. All of the functions we discussed in Chapter 13 are available, plus several others (see the Keras documentation). Common choices for hidden convolution layers are `'relu'` and `'tanh'`.

Recall that a `Dense` layer required an argument to `input_shape` if and only if that layer was the first layer in the network. Convolution layers work the same way, and require a value to be assigned to `input_shape` if they're the very first layer.

The `input_shape` argument we applied to `Dense` layers was a list with a single value: the length of the list that represented an image. Just as with those layers, a CNN layer wants the `input_shape` to describe not the shape of the whole data set, but *just one sample*. We know that in our MNIST example we have 60,000 samples, each 28 by 28 by 1. Therefore, the value of `input_shape` is the list `(28,28,1)`, describing one image.

Now that we've covered all the background, let's construct a 2D convolution layer. This will be the first layer in our model, so we need the `input_shape` argument. Let's say that we want 16 filters, each 5 by 5. For the sake of demonstration, we'll pick the `relu` activation function, zero-padding so that the output is the same size as the input, and a stride of 2 in both X and Y. Listing B3-31 shows how we'd add this to a model named `model`.

```
model.add(Conv2D(16, (5, 5), activation='relu',
                 strides=(2, 2), padding='same',
                 input_shape=(image_height, image_width, 1)))
```

Listing B3-31: Adding a 2D convolution layer to our model. The first argument is the number of filters, followed by the width and height of each filter. The other, named arguments (except for `input_shape`) are optional. Here, we set the activation function to `relu`, choose zero-padding to make the output the same sizes as the input by setting `padding` to `same`, set the stride to (2,2), and we specify the shape of the input tensor using the `channels_last` convention.

After all that discussion, the mechanics end up to be pretty short, even with the optional arguments.

That covers the basics of using convolution layers. Everything else is just like before: we create our model, add in layers, compile it, and then train it. Later we can ask it for predictions.

Keras takes care of all the work of creating filter kernels of the right size, initializing them with good values, and improving them with backprop.

Now that we know how to create a convolution layer, let's build a fully-functional convnet.

Using Convolution for MNIST

Let's build a CNN for categorizing MNIST images. To start with, we'll make a simple convnet with just one convolution layer, a flattening layer, and a fully-connected output layer.

Our pre-processing step is unchanged from what we saw in Listing B3-28. It reads in our data, normalizes it, re-shapes the features to the 4D channels-last tensor, and makes one-hot encodings for the labels.

Our model will have the architecture of Figure B3-27.

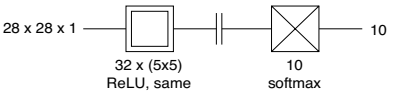


Figure B3-27: The architecture for a tiny convolutional neural network. We have one convolutional layer with 32 filters, each a square of 5 by 5 elements. Following that is a flatten layer, and then a 10-neuron fully-connected layer to present our categorization outputs.

To create our model, we start just as before, by creating a new object of type `Sequential`, then then adding layers one at a time.

We'll start with a 2D convolution layer with 32 kernels, each 5 by 5. We'll use a `relu` activation function, and set `padding='same'`, which will give the layer a temporary ring of 0-padding, so that the output will have the same horizontal and vertical sizes as the input. Since the Dense layer at the end takes a list as input, we'll use a Flatten layer to turn the 28 by 28 by 32 tensor into a list of 28*28*32 = 25,088 elements. We use `softmax` on that final layer, just as before.

As usual, we'll wrap all of this up in little function which creates the model, compiles it, and returns the final result.

Listing B3-32 shows the code.

```
def make_simple_cnn_model():
    model = Sequential()
    model.add(Conv2D(32, (5, 5),
                     activation='relu', padding='same',
                     input_shape=(1, image_height, image_width)))
    model.add(Flatten())
    model.add(Dense(number_of_classes, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam',
                  metrics=["accuracy"])
    return model
```

Listing B3-32: The code to make our first CNN for classifying MNIST digits

As far as Keras is concerned, this is just a `Sequential` object like any other. We can train this model just as we always have, by calling `fit()` with all the necessary parameters. Just for completeness, Listing B3-33 shows the code. Like our experiments in the last section, we'll run this model for 100 epochs, using a batch size of 256.

```
simple_cnn_model = make_simple_cnn_model()

simple_cnn_history = simple_cnn_model.fit(X_train, y_train,
                     validation_split=0.25,
                     epochs=100, batch_size=256)
```

Listing B3-33: To train our CNN, we only need to call its `fit()` method with the usual parameters.

The results of our this training session are shown in Figure B3-28.

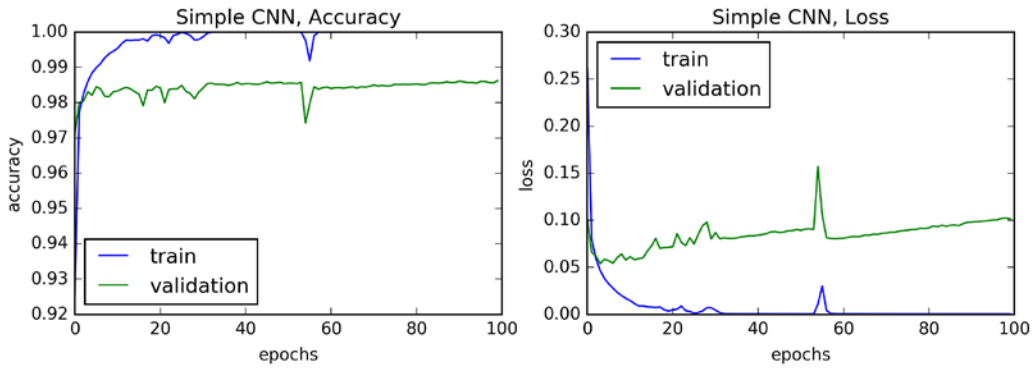


Figure B3-28: The accuracy and loss for our simple convnet

The numbers from the final epoch are shown in Listing B3-34.

```
Epoch 100/100
66s - loss: 2.7044e-04 - acc: 1.0000 - val_loss: 0.1016 - val_acc: 0.9862
```

Listing B3-34: The final values from Figure B3-28

The good news about these results is that everything seems to be working pretty well. Our system learns the training data and does a good job predicting the classes of the validation data, getting about 98.6% accuracy.

On the other hand, these curves aren't really looking great. The training accuracy gets up to about 1.0 within 35 epochs or so, and the validation accuracy seems to plateau about there as well. That's okay, but the loss curves tell a different story. The system starts overfitting before even 10 epochs are done, and it just gets worse as time goes on.

As usual, to cure problems like this we need to follow our hunches. Let's guess that maybe we would see better performance if we used a deeper model. We'll increase the number of convolution layers from 1 to 3, and to control overfitting we'll add dropout after each one.

Figure B3-29 shows our new, deeper architecture.

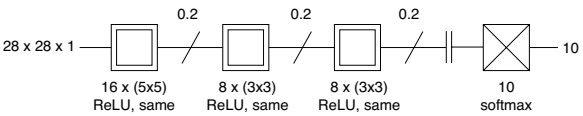


Figure B3-29: A bigger CNN using multiple convolution and dropout layers

The first convolution layer uses 16 filters of 5 by 5, and then we follow that with two layers that use 8 filters of 3 by 3. All of these numbers are more or less arbitrary, resulting from an initial guess and then some trial and error. We follow each convolution layer with a dropout layer, and at the end we flatten the result and feed it to a 10-neuron dense layer using softmax, as usual.

Because we're using the `border_mode='same'` option, and the default stride of 1 by 1, the output of each convolution will be the same width and height as the input.

Listing B3-35 shows a function to make this new model. Note that we're setting the optional argument `kernel_constraint` to the value `MaxNorm(3)`, just as we did with the `Dense` layers earlier. For convolution layers it prevents the values in the filters from getting too big, in the same way that it prevented the weights in the `Dense` layers from getting too big.

```
def make_bigger_cnn_model():
    model = Sequential()
    model.add(Conv2D(16, (5, 5), activation='relu', padding='same',
                     kernel_constraint=MaxNorm(3),
                     input_shape=(1, image_height, image_width)))
    model.add(Dropout(0.2))
    model.add(Conv2D(8, (3, 3), activation='relu', padding='same',
                     kernel_constraint=MaxNorm(3)))
    model.add(Dropout(0.2))
    model.add(Conv2D(8, (3, 3), activation='relu', padding='same',
                     kernel_constraint=MaxNorm(3)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(number_of_classes, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam',
                  metrics=["accuracy"])
    return model
```

Listing B3-35: The code to make the model of Figure B3-29

Listing B3-36 shows how we'd call this function to make a new model.

```
bigger_cnn_model = make_bigger_cnn_model()

bigger_cnn_history = bigger_cnn_model.fit(X_train, y_train,
                     validation_split=0.25,
                     epochs=100, batch_size=256)
```

Listing B3-36: Training the model of Listing B3-35

The results are shown in Figure B3-30.

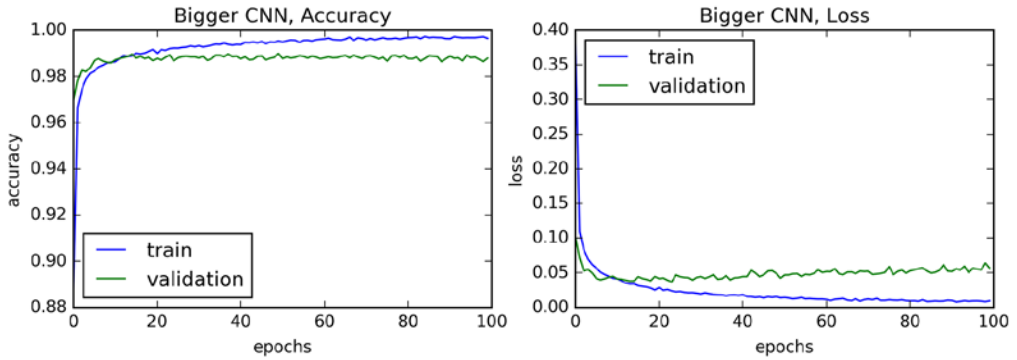


Figure B3-30: The accuracy and loss from training our model of Figure B3-29 for 100 epochs

The numbers from the end of this run are in Listing B3-37.

```
Epoch 100/100
127s - loss: 0.0094 - acc: 0.9965 - val_loss: 0.0565 - val_acc: 0.9879
```

Listing B3-37: The numbers from the final line of training our bigger CNN

We've pretty much eliminated overfitting, though the validation accuracy is just a bit lower than before. As far as validation accuracy goes, it seems we could have stopped after about 40 epochs.

There might be a small amount of overfitting going on, which we can try to reduce by adjusting hyperparameters, as we did before.

On a late-2014 iMac, without GPU support, using TensorFlow, this model took a little more than 2 minutes for each epoch. That's about double the time required by our first model with just one convolution layer.

Now that we have our feet wet, we can think about looking for better performance. But where do we start?

Progress on deep-learning architectures often comes from building on results other people have developed and published. Looking at the MNIST page [LeCun13], we can see architectures for convnets that have worked well on this data set. Most of these have some advanced or experimental features, but we can still emulate their basic structure.

Let's try making the image smaller and smaller as it works its way through the network. This way each layer gets to work with larger pieces of the original image.

We'll do this first using pooling layers. Though we've noted pooling layers are falling out of favor in convnets, we'll use them now because they let us explicitly show how the tensor gets reduced in size as it flows through the network.

Each pooling layer will look at 2 by 2 non-overlapping boxes, and return the largest value in that group of 4 input elements. As a result, the output of each of these layers will have half the width and height of its input. The input images are 28 by 28, so the output of the first max pooling layer will be 14 by 14, and the output of the second will be 7 by 7. As usual, the depth of each of these tensors is given by the number of filters in the preceding convolution layer.

We'll include three dense layers at the end, also of decreasing size. This is basically just working on a hunch that because we have fewer inputs arriving at the final dense layers (just $7 \times 7 = 49$ values in all), we could benefit from more processing of those values. And what the heck, let's include dropout after each convolution layer.

Figure B3-31 shows a diagram of this architecture.

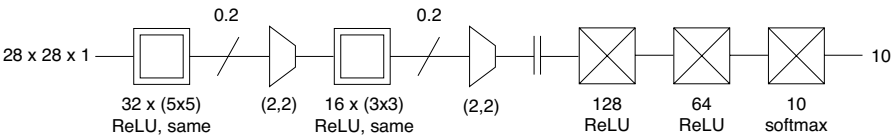


Figure B3-31: A CNN with dropout and pooling layers

Listing B3-38 shows a function to make this new model.

```
def make_pooling_cnn_model():
    model = Sequential()
    model.add(Conv2D(30, (5, 5), activation='relu', padding='same',
                     kernel_constraint=MaxNorm(3),
                     input_shape=(1, image_height, image_width)))
    model.add(Dropout(0.2))
    model.add(MaxPooling2D(pool_size=(2, 2), padding='same'))
    model.add(Conv2D(16, (3, 3), activation='relu', padding='same',
                     kernel_constraint=MaxNorm(3)))
    model.add(Dropout(0.2))
    model.add(MaxPooling2D(pool_size=(2, 2), padding='same'))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(64, activation='relu'))
    model.add(Dense(number_of_classes, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam',
                  metrics=["accuracy"])
    return model
```

Listing B3-38: Making the model in Figure B3-31

Listing B3-39 shows how we'd call this function to make a new model.

```
pooling_cnn_model = make_pooling_cnn_model()

pooling_cnn_history = pooling_cnn_model.fit(X_train, y_train,
                          validation_split=0.25,
                          epochs=100, batch_size=256)
```

Listing B3-39: Code to train the model of Figure B3-31

The results are shown in Figure B3-32.

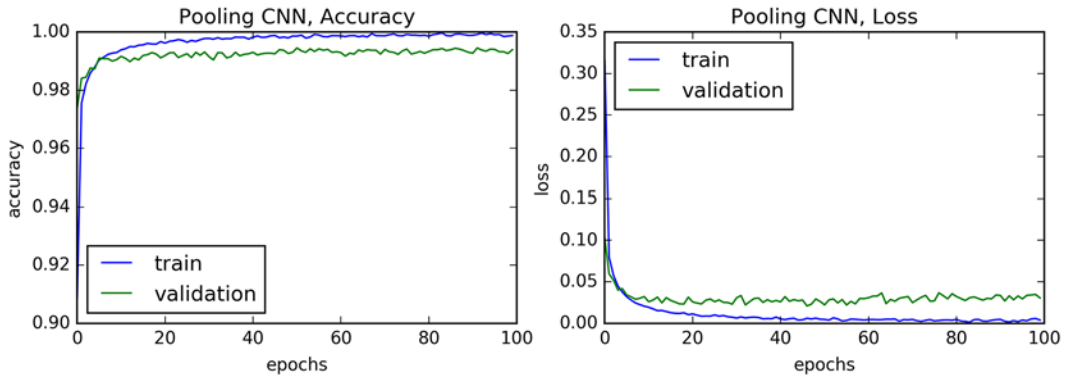


Figure B3-32: Results for training the model of Figure B3-31 for 100 epochs

The numbers from the end of this run are in Listing B3-40.

```
Epoch 100/100
147s - loss: 0.0038 - acc: 0.9988 - val_loss: 0.0304 - val_acc: 0.9939
```

Listing B3-40: The final line of data from training the model of Figure B3-31 for 100 epochs

We've picked up a small but meaningful increase in validation accuracy, from 0.9879 to 0.9939. As we mentioned earlier, progress at this point often proceeds in tiny steps. This is actually pretty large, since we've shaved off almost half of the distance to 1.

And we seem to have no overfitting. In fact, after about the 50th epoch everything looks pretty much settled.

We mentioned that pooling layers after convolution are being replaced these days with striding in the convolution layers themselves. So let's duplicate this most recent architecture, but instead of 2 by 2 max pooling layers, we'll use 2 by 2 striding in the convolution layers. The resulting sizes will the same as before. The results will be a little different, because the process of striding the convolution kernels is not identical to moving them by single steps and then pooling, but we'd expect a rough similarity at least.

Figure B3-33 shows a diagram of this architecture.

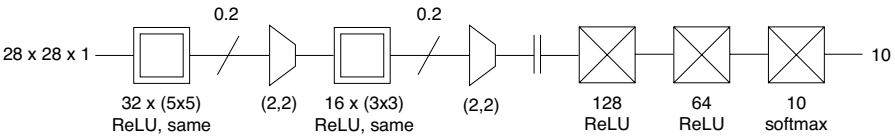


Figure B3-33: Our model of Figure B3-31, where we've replaced the pooling layers with striding in the convolution layers

Listing B3-41 shows a function to make this new model.

```
def make_striding_cnn_model():
    model = Sequential()
    model.add(Conv2D(30, (5, 5), activation='relu', padding='same',
                     strides=(2, 2), kernel_constraint=MaxNorm(3),
                     input_shape=(1, image_height, image_width)))
    model.add(Dropout(0.2))
    model.add(Conv2D(16, (3, 3), activation='relu', padding='same',
                     strides=(2, 2), kernel_constraint=MaxNorm(3)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(64, activation='relu'))
    model.add(Dense(number_of_classes, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam',
                  metrics=["accuracy"])
    return model
```

Listing B3-41: Code to make the striding CNN of Figure B3-33

Listing B3-42 shows how we'd call this function to make a new model.

```
striding_cnn_model = make_striding_cnn_model()

striding_cnn_history = striding_cnn_model.fit(X_train, y_train,
                            validation_split=0.25,
                            epochs=100, batch_size=256)
```

Listing B3-42: Code to build the striding CNN of Figure B3-33

The results are shown in Figure B3-34.

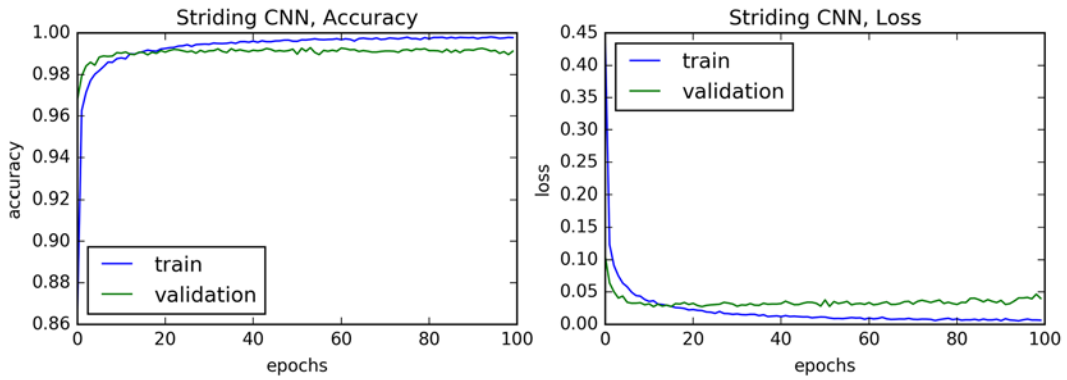


Figure B3-34: Accuracy and loss from training the striding CNN of Figure B3-33 for 100 epochs

The numbers from the end of this run are in Listing B3-43.

```
Epoch 100/100
36s - loss: 0.0062 - acc: 0.9978 - val_loss: 0.0400 - val_acc: 0.9912
```

Listing B3-43: The final lines from training the model of Figure B3-33

We've lost a very tiny bit of accuracy in both the training and validation sets, but otherwise these numbers and their graph look pretty much like what we saw from using explicit pooling layers.

But what has changed a lot is the timing. As we can see in Listing B3-40, the pooling model required about 147 seconds per epoch on a late-2014 iMac without GPU, while the striding version took only 36 seconds per epoch. The striding epochs took only about 25% of time required by the epochs that used explicit pooling. Clock times can be misleading, but it's hard not to like getting nearly the same performance with only 25% of the time and effort.

Can we increase performance even more?

We can play with any aspect of our network. We can add or remove filters at each layer, change their size, increase the dropout percentage, add more convolution layers, and so on. Just for variety, let's try replacing the dropout layers with batchnorm layers. Both are designed to reduce overfitting, so we can see which of the two techniques works best for this network and this data.

Figure B3-35 shows a diagram of this architecture.

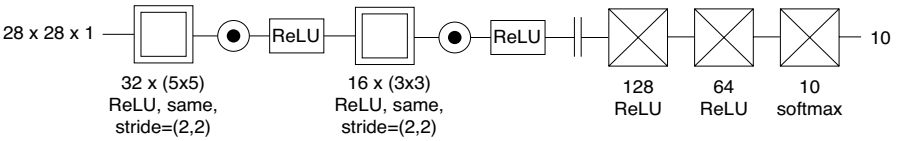


Figure B3-35: Our model of Figure B3-33, where we've replaced each dropout layer with a batchnorm layer.

Listing B3-44 shows a function to make this new model. We're setting the `activation` parameter in the convolution layers to `None`, because, as we saw in Chapter 15, batchnorm operates *between* a layer's output and its activation function. So we follow each convolution layer with a `BatchNormalization` layer, and then a layer to apply the `ReLU` activation function (recall that batch normalization was designed to take place after a layer's outputs, but before the activation function).

```
def make_striding_batchnorm_cnn_model():
    model = Sequential()
    model.add(Conv2D(30, (5, 5), activation=None, padding='same',
                     strides=(2, 2),
                     input_shape=(1, image_height, image_width)))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(Conv2D(16, (3, 3), activation=None, padding='same',
                     strides=(2, 2)))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(64, activation='relu'))
    model.add(Dense(number_of_classes, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam',
                  metrics=["accuracy"])
    return model
```

Listing B3-44: Code to make the striding-batchnorm CNN of Figure B3-35. Note that we place the `BatchNormalization` layer between the convolution layer and its `relu` activation function, placed on its own layer.

Listing B3-45 shows how we'd call this function to make a new model.

```
striding_batchnorm_cnn_model = make_striding_batchnorm_cnn_model()

striding_batchnorm_cnn_history = striding_batchnorm_cnn_model.fit(
                          X_train, y_train,
                          validation_split=0.25),
                          epochs=100, batch_size=256)
```

Listing B3-45: Code to build the striding-batchnorm CNN of Figure B3-35

The results are shown in Figure B3-36.

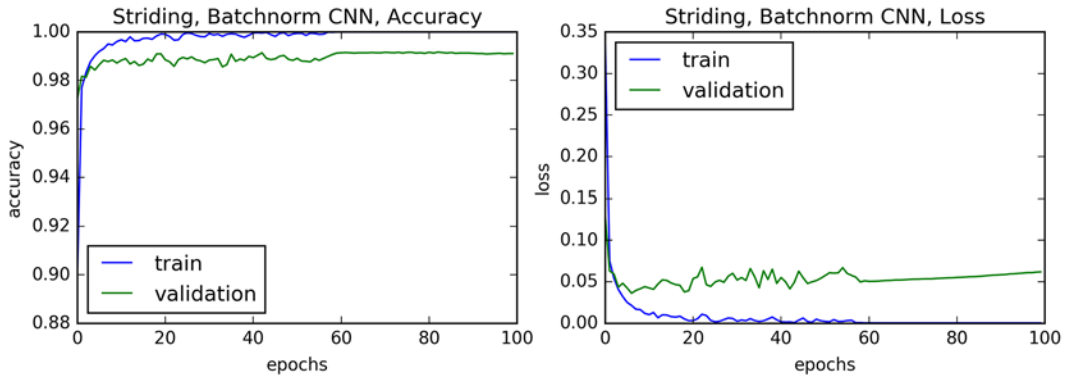


Figure B3-36: Accuracy and loss from training the striding-batchnorm CNN of Figure B3-35 for 100 epochs.

The numbers from the end of this run are in Listing B3-46.

```
Epoch 100/100
45s - loss: 2.6886e-04 - acc: 1.0000 - val_loss: 0.0618 - val_acc: 0.9911
```

Listing B3-46: The final lines from training the model of Figure B3-35

The validation accuracy is about the same as we got before, but we seem to have picked up a small amount of overfitting. The epochs also take about 25% longer to run.

The noise in the curves is a matter of some concern. We'd want to be careful when we stop training, to make sure we're not in one of those peaks in the validation loss (or the corresponding valleys in the validation accuracy). This is one of those times when it makes a lot of sense to keep multiple checkpoints, and then choose one based on looking at the performance graphs.

Overall, this variation doesn't seem to have given us anything better than before. This is the value of experimenting: until we try, we can't be sure how a network and a particular data set will behave.

Patterns

Historically, many CNNs were assembled by repeating a few types of recognizable clusters of layers [Karpathy16a]. Such a cluster is a set of convolution layers, followed by a pooling layer. This cluster is then repeated several times, perhaps with different parameters to the convolution layers. After that comes a series of fully-connected layers. Figure B3-37 shows an example of a such an architecture.

The pooling layers usually tile the input with 2 by 2 blocks. That is, the receptive field is 2 by 2, and we use a stride of (2,2) so that we produce a tiling with no overlaps or holes. This makes an output that has half the width and height of the input.

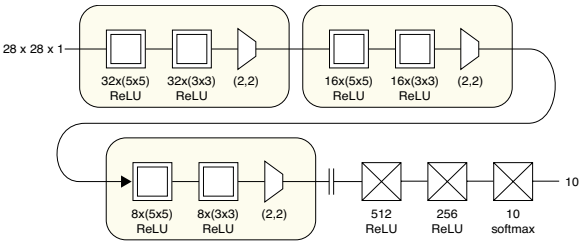


Figure B3-37: CNNs are often made up of repeating patterns. One popular pattern is a few convolution layers followed by a pooling layer, which we see here repeated three times. Typically, the parameters of the convolutions change from one repeat of the block to the next.

This network is a simplified version of the VGG16 network we've seen before, drawn here to demonstrate the idea of repeated units, but just for fun, let's run this on the MNIST data The results are shown in Figure B3-38.

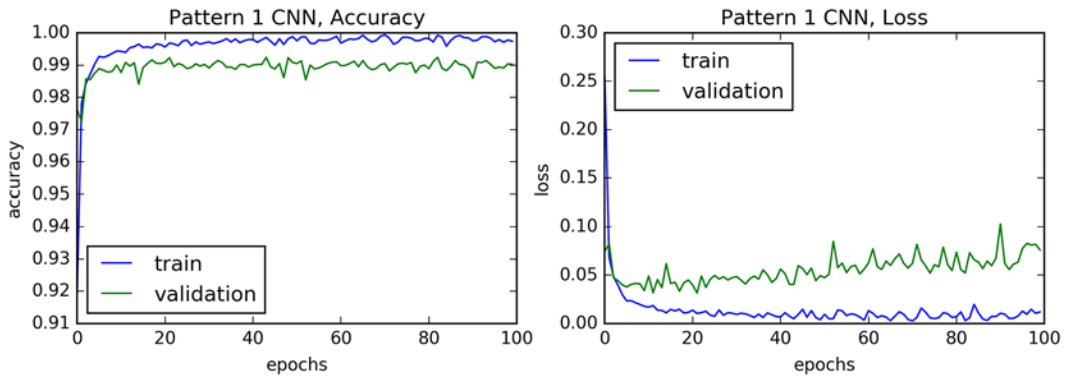


Figure B3-38: The architecture of Figure B3-37 evaluating MNIST data

The final lines from training are in Listing B3-47.

```
Epoch 100/100
339s - loss: 0.0119 - acc: 0.9974 - val_loss: 0.0760 - val_acc: 0.9902
```

Listing B3-47: The final lines from training the model of Figure B3-37

This isn't the best data we've seen, and there's some overfitting, but it's not bad for an essentially arbitrary network. It does take quite a while to run each epoch, as we might expect from all those layers.

As we did before, let's replace the pooling layers with striding in the final convolution layer of each set, as in Figure B3-39. We usually leave the stride of the other layers at 1. This is becoming a more attractive option as omitting the pooling layers gives us a smaller and faster network, and

when things are well-tuned there seems to be no loss of performance [Karpathy16a][Springenberg15]. The stride sizes are often (2,2), as they were for the pooling layers we're replacing.

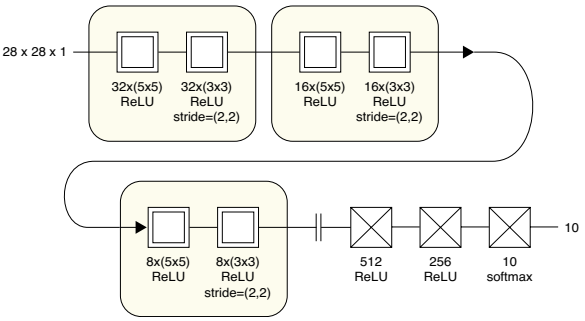


Figure B3-39: Replacing the pooling layers of Figure B3-37 with striding in the convolution layers

The results are shown in Figure B3-40.

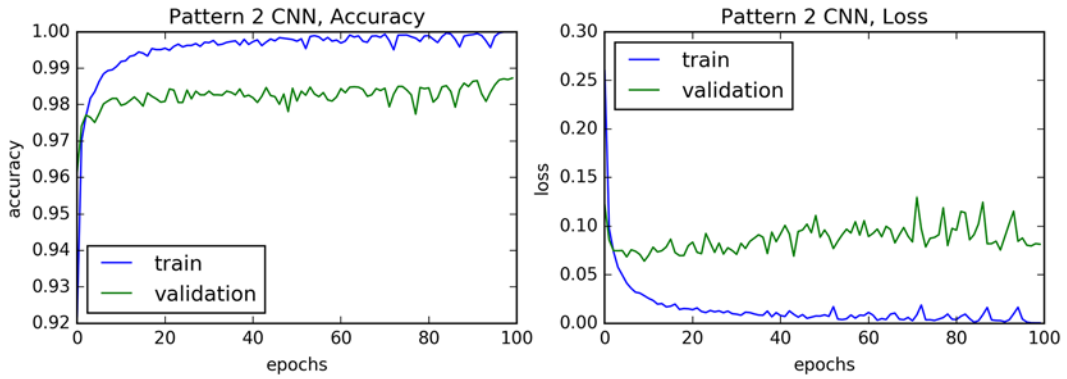


Figure B3-40: The architecture of Figure B3-39 evaluating MNIST data.

The final lines from training are in Listing B3-48.

```
Epoch 100/100
186s - loss: 2.8056e-04 - acc: 1.0000 - val_loss: 0.0815 - val_acc: 0.9873
```

Listing B3-48: The final lines from training the model of Figure B3-39

The results are just a little noisier than before, but they land in roughly the same place. We even cut the time per epoch roughly in half, suggesting that striding in the convolution layer is significantly faster than subsequent pooling. This makes sense, since convolution with striding means we apply the convolution filter less frequently, and we can skip the subsequent post processing step of pooling altogether.

Image Data Augmentation

One of the best ways to improve performance of any model is to give it as much training data as we can, while avoiding overfitting.

When we're working with images, we can easily create lots of new data by simply manipulating the images we already have, creating a wide variety of variations of the original. We could move each image left, right, up, or down, make it a little smaller or larger, rotate it clockwise or counter-clockwise by some amount, or perhaps flip it horizontally or vertically. Figure B3-41 shows some of these variations on an image of a Eurasian Eagle Owl.

Figure B3-41: Augmenting an image of a Eurasian Eagle Owl with rotations, flips, and scaling. The original is in the upper left. These transformations are deliberately extreme to show their effect.

The process of enlarging a dataset by creating variations is called *data amplification*, or *data augmentation*.

When we're working specifically with images, Keras provides a built-in object to perform data augmentation. It's called the `ImageDataGenerator`, and it performs all of the modifications we just mentioned, and a few others besides.

As its name suggests, this object is a "generator," which is a specific kind of object in the Python language [PythonWiki17]. In a nutshell, a generator can be thought of as a function that runs an internal loop, typically carrying out calculations and producing data. When that loop reaches a `yield` statement, the generator returns control to the routine that called it, with the argument to `yield` set to its value, just like a `return` statement. But if we call that function again, the loop picks up where it most recently stopped, and continues as though it had never been interrupted.

The `ImageDataGenerator` is set up this way because we can configure it to produce large numbers of variations on each of our input images. This can take a lot of time and a lot of computer memory. So rather than compute all the variations ahead of time and hold on to them until they're needed, the generator creates batches of images on demand. Each time we call the generator, it will produce and return another batch of images. The `fit()` method can use the generator as the source of training data, rather than

tensors that we pass in. The routine calls the generator over and over, each time it needs more data.

The transformations we applied to the images in Figure B3-41 were deliberately exaggerated to show their effects. In practice, we want to make new data that is close enough to the input to be plausible. After all, there's no reason to learn from distorted inputs that are not representative of the data we expect to see. In fact, that could hurt our ultimate performance, since some of the network's power would be uselessly directed to processing those inputs.

If we want to use our generated data again later, we can save time by telling the generator to read and write its images in a given directory. Then each time we ask for another batch of images, it reads and returns the saved, transformed files if they're available, or else generates them, saves them, and then returns them. This feature is also useful when we want to look at the generated files, to make sure we've selected the right options to get the sort of variations we're after. If disk space is precious and we're not pressed for time, we can just always skip the disk files altogether and make the transformed images fresh, on demand.

The `ImageDataGenerator` is a workhorse, capable of applying all sorts of transformations to our images. We only need to list the image-transforming operations we want when we build the object, and it will apply them all. We'll demonstrate the process with just a few transformations. The Keras documentation provides the complete list of all available options.

The overall process of setting up and using the generator takes only two steps. First, we create our `ImageDataGenerator` with the options we want. Second, we train our model. But instead of using `fit()` to start training, we use `fit_generator()`. These both take the same arguments with one exception: The first argument to `fit_generator()` is a function that returns batches of samples.

The usual function that we provide to `fit_generator()` is a function called `flow()`, which is automatically created for us as part of our `ImageDataGenerator` object. The metaphor is that the generator is producing a flow of data on demand, like water flowing out of a faucet when we turn the handle. Calling `flow()` provides a burst of training samples for our use.

Together, `fit_generator()` and `flow()` manage the production of batches of images, and presenting them to our model for training. Listing B3-49 shows a typical use of `ImageDataGenerator`.

```
# create the image generator with rotations and flips
image_generator = ImageDataGenerator(
                        rotation_range=100, horizontal_flip=True)

# fit our model using images produced by the image generator
model.fit(image_generator.flow(X_train, Y_train, batch_size=256),
                    seed=42, epochs=100,
                    samples_per_epoch=len(X_train))
```

Listing B3-49: Using an `ImageDataGenerator` to produce transformed images. Each image might be randomly flipped horizontally, and/or rotated up to 100 degrees in either direction.

A few images produced by an `ImageDataGenerator` using just a single sample from the MNIST training set are shown in Figure B3-42. In practice, for this data, we'd probably use much less rotation, and we wouldn't allow flips, since a mirror-image 2 isn't really a 2 at all. Some noise is visible around the edges where the low-resolution original image has been resampled.

Figure B3-42: Using an `ImageDataGenerator` to produce variations on a single MNIST sample for the number 2. For demonstration purposes we've allowed horizontal flipping, and a large range for rotation.

Normally, each run of this code will produce different results. For testing and debugging, it's often useful to get back the same sequence every time. We can force this by setting the `seed` argument when we call `fit()`, as we did above. This has the same purpose as setting the seed for a random number generator, which sets it up to always produce the same sequence of pseudo-random values.

In Listing B3-49 we also told `fit()` how many samples we want per batch, how many samples make up an epoch, and how many epochs we want.

It might seem odd to have to specify the number of samples per epoch, since until now the library has been able to infer that from the size of the input tensor. But the generator will just keep cranking out variations as long as we keep calling it, so there's really no sense of having run through "all the data," which is what we normally call an epoch. Yet epochs are important. For example, it's at the end of an epoch when statistics get collected and our callbacks are invoked. So we tell `fit()` how many images to generate before it simply declares that an epoch has passed. The value of `samples_per_epoch` has to be a multiple of `batch_size` or we'll get an error.

Synthetic Data

This section's notebook is Bonus03-Keras-4-CNN-Synthetic-Data.ipynb.

We usually build networks because we want to deploy them for real-world use, so we train them on real-world data. But testing and practice datasets are useful for helping us experiment with architectures, pre-processing strategies, and hyperparameters.

A great way to create an environment where we control everything is to train on our own data, which we generate on demand. Then we can make the data we want, rather than search for something out there that comes close.

We use the phrase *synthetic data* to describe data that we create ourselves, usually on the fly with an algorithm. We saw synthetic data in Bonus Chapter 1, when we used scikit-learn's built-in algorithms for making half-moons and blobs.

The great thing about generating synthetic data is that we can make as much of it as we want or need, and then train with it as usual.

It's conceptually easy to do this. Rather than modify an `ImageDataGenerator` object, which can require a lot of work, we subclass Python's built-in `Sequence` object, and equip it with a few standard routines such as `__len__` and `__getitem__`.

To demonstrate the idea, we've written a little routine that draws an image in a 64 by 64 square. There are five types of images: a Z shape, a plus sign, three vertical lines, a squared U, and a circle. Each time we draw one of these shapes we wiggle the points around a little, so that no two shapes are the same. The function returns the image it drew and the label. The label is a number from 0 to 4 that identifies which type of shape is in the image.

Figure B3-43 shows a random collection of these images. Note that these variations are performed on the points that make up the image, and are inherently different than what we get by applying the types of deformations (scaling, rotating, etc.) with an `ImageDataGenerator` object.

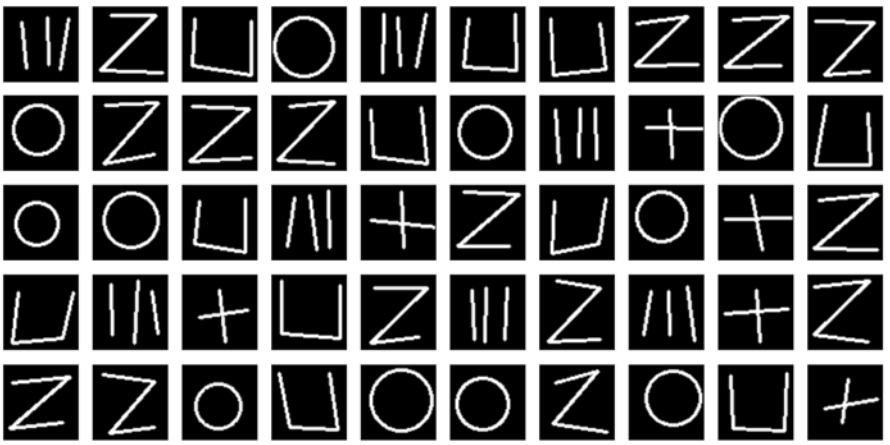


Figure B3-43: Synthetic images produced by a little routine. Each of the 5 types of images uses points (and in the case of the circle, a radius) that have been randomly perturbed from their starting positions.

We used this to train the simple convnet shown in Figure B3-44.

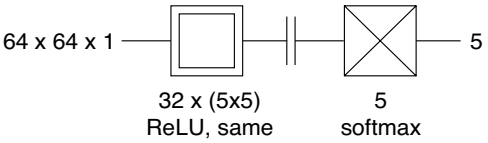


Figure B3-44: A simple convnet for classifying our synthetic data

The results are shown in Figure B3-45.

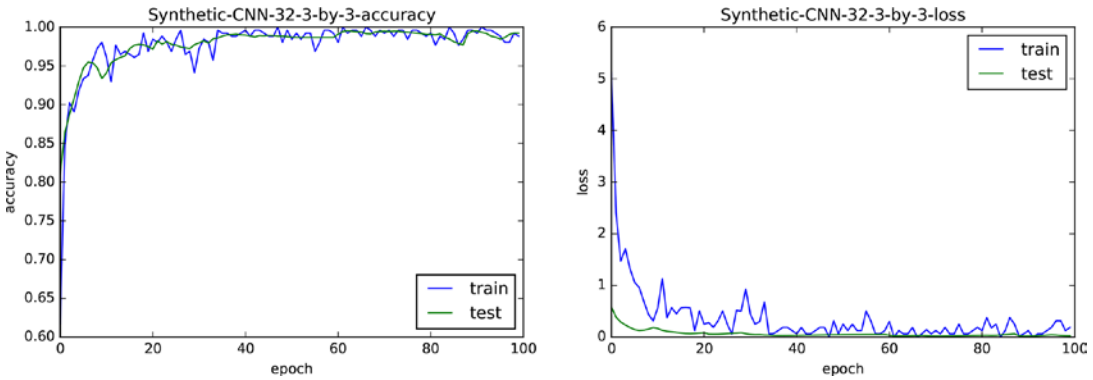


Figure B3-45: Accuracy and loss for our simple model on our synthetic data, generated on the fly

Despite being about as tiny as a classifier convnet can be, the network manages impressive accuracy on the test data, without obvious overfitting.

Parameter Searching for Convnets

Deep convnets can take a long time to train, particularly if we use big data sets. If we use the `GridSearchCV` or `RandomizedSearchCV` objects we used earlier to search for hyperparameters, we might not have enough compute power to produce results in a reasonable time.

There are faster alternatives. Unfortunately, they take some work to set up and use, so we won't go into them here. A good place to start for automatic parameter searching is the Spearmint project [Snoek16a][Snoek16b].

RNNs

This section's notebook is Bonus03-Keras-5-RNN.ipynb.

As we saw in Chapter 19, recurrent neural networks, or RNNs, are great for *sequential data*. The MNIST image data we've been using is not sequential, because there's no order to the images.

Sequential data, on the other hand, is inherently ordered. Classic examples are daily temperatures, the daily price of a stock, and the hourly height of a tide. There's also data that's ordered, but not in time, such as children lined up by height, shelved library books, and the colors of the rainbow.

In all of these phenomena, we want to use the information in the sequence of inputs to help us produce new output.

In RNN terminology, we still have a dataset made of samples, where each sample contains multiple features. But now each feature contains multiple values, called *time steps*. Recall that we can also think of "time steps" as "series of measurements for a given feature." Our example in Chapter 19 imagined a weather station on top of a mountain, taking multiple measurements every hour during 8 daytime hours. Each day's results make up a sample, and each type of measurement (such as temperature and wind speed) is a feature. Each feature contains 8 time steps, with one value for each hour.

Throughout this chapter we've been classifying the MNIST data. But the MNIST data has no sequential qualities, and our goal now is not classification, but prediction of the next value in a sequence. So in the next section we'll generate some new data of our own that we can use to show how to set up and run RNNs.

Generating Sequence Data

There are lots of sequential datasets available, but some of them are complicated or hard to draw. So let's make our own simple dataset that we can easily draw and interpret.

We'll just add up a bunch of sine waves, such as that at the top of Figure B3-46. Each sine wave has a frequency, an amplitude, and a phase (or offset). We'll write a routine that takes lists of these values, called `freqs`, `amps`, and `phases` respectively, and uses them to add up all the waves at many points. Figure B3-46 shows the idea.

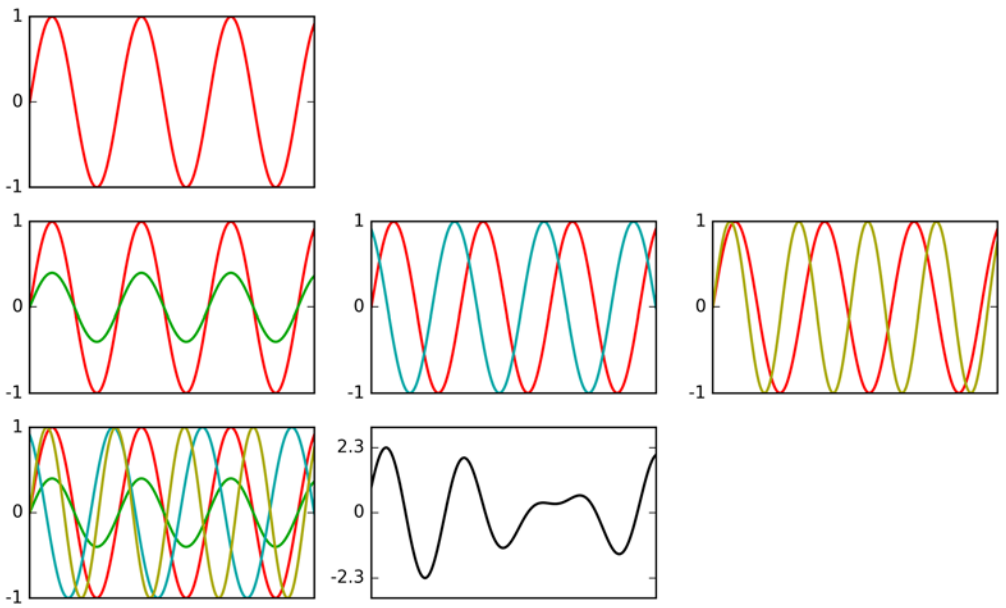


Figure B3-46: By adding up lots of sine waves we can make an interesting curve.

The top row of Figure B3-46 shows a single sine wave. On the left of the second row, we see our original wave in red, and another wave in green with smaller amplitude. In the center of the second row, we see our original wave in red, and another wave in blue with a different phase. And at the right end of the second row, we see our original wave in red, and another wave in yellow with a different frequency. The left image in the bottom row shows all four waves superimposed, and the one to its right shows all four waves added together. This is sine-wave-ish, but more interesting.

Our little curve-building routine takes three other arguments. The first is an integer called `number_of_steps` which tells it how many points to generate. The second is a float called `d_theta` that tells the routine the spacing of the samples (the name comes from thinking of the sine wave as based on an angle, which is often written with the lower-case Greek letter θ (theta). Finally, `skip_steps` is an integer that provides an offset to the starting point, so we don't always begin at 0. This is useful for creating the test data, which can start far to the right of the training data.

The routine `sum_of_sines()` is shown in Listing B3-50. We wrote it to emphasize clarity. Since this is plain Python programming, and nothing specific to machine learning, we won't go into the details.

```
def sum_of_sines(number_of_steps, d_theta, skip_steps, freqs, amps, phases):
    '''Add together multiple sine waves and return a list of values that is
    number_of_steps long. d_theta is the step (in radians) between samples.
    skip_steps determines the start of the sequence. The lists freqs, amps,
    and phases should all the same length (but we don't check!)'''
    values = []
    for step_num in range(number_of_steps):
        angle = d_theta * (step_num + skip_steps)
        sum = 0
        for wave in range(len(freqs)):
            y = amps[wave] * math.sin(freqs[wave]*(phases[wave] + angle))
            sum += y
        values.append(sum)
    return np.array(values)
```

Listing B3-50: A little routine to create a list that holds the sum of multiple, different sine waves

We'll use this routine to generate two different data sets, which we'll call data set 0 and data set 1. The values we used to construct them were found by trial and error until they produced one graph that felt "calm," so it wouldn't be too hard to predict, and one that felt "busy," for a harder challenge.

We'll consider the data we use for evaluation to be a test set, which is used just once, rather than a validation set, which can be used multiple times when evaluating different forms of the network.

Data set 0 is a gentle sum of two waves. We made one wave twice the speed of the other by setting `freqs` to `(1,2)`, the second wave twice as high as the first by setting `amps` to `(1,2)`, and started both waves at 0 by setting `phases` to `(0,0)`, Our training data came from using 200 steps (`number_of_steps` = 200), a step of about 0.057 radians (`d_theta` = 0.057), and no offset (`skip_steps` = 0). We chose the weird step size by eye so that 200 samples produced what we felt was a good amount of data for an easy test case.

The training set is 200 samples long, starting at 0. The test set is another 200 steps, starting far to the right of the training set. Listing B3-51 shows the calls to make this data set.

```
train_sequence_0 = sum_of_sines(200, 0.057, 0, [1, 2], [1, 2], [0, 0])

test_sequence_0 = sum_of_sines(200, 0.057, 400, [1, 2], [1, 2], [0, 0])
```

Listing B3-51: Creating data set 0

The resulting training and test data are shown in Figure B3-47.

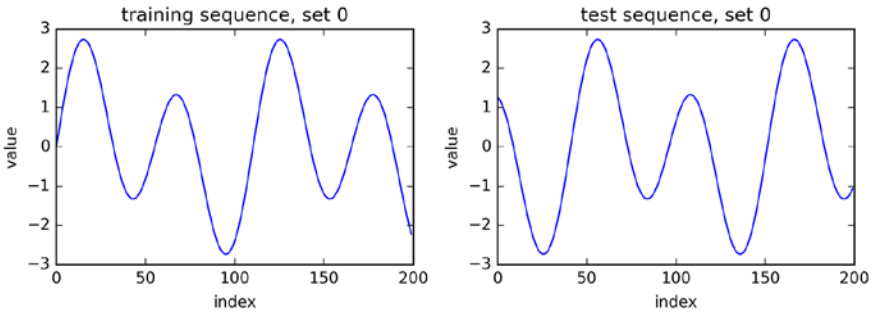


Figure B3-47: The training and test data for our first set of sine waves

Data set 1 is a harder challenge that uses 4 waves. For this set, we set `freqs` to `(1.1, 1.7, 3.1, 7)`, `amps` to `(1, 2, 2, 3)`, and we again left all the phases at 0, so `phases` is `(0,0,0,0)`. The weird frequencies are chosen so that the pattern won't repeat for tens of thousands of samples. The other variables are the same as for data set 0. Listing B3-52 shows the calls to make the data.

```
train_sequence_1 = sum_of_sines(200, 0.057 ,0, [1.1, 1.7, 3.1, 7],
                                [1, 2, 2, 3],  [0, 0, 0, 0])

test_sequence_1 = sum_of_sines(200, 0.057, 400, [1.1, 1.7, 3.1, 7],
                               [1, 2, 2, 3],  [0, 0, 0, 0])
```

Listing B3-52: Creating data set 1

The results are shown in Figure B3-48.

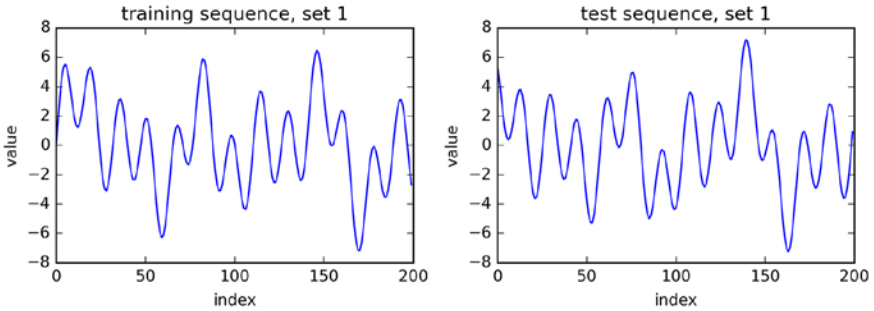


Figure B3-48: The training and test data for our second, busier set of sine waves

RNN Data Preparation

The mechanics for preparing data for RNNs in Keras are a little more complicated than what we've been working with so far, because we have to carry out a couple of reshaping steps in order to use all the library routines we want. We also need to extract our little windowed sublists, which we have to do ourselves since there aren't any library routines to do it for us.

Let's dig into the steps and knock them down one by one in order.

As before, we want to normalize our data to get it into the range [0,1]. The `MinMaxScaler` from scikit-learn is the perfect tool for the job. But recall from Bonus Chapter 1 that this routine expects our features to be arranged vertically, as in Figure B3-49.

	Temperature	Rainfall	Wind Speed	Humidity
June 3	60	0.2	4	0.1
June 6	75	0	8	0.05
June 9	70	0.1	12	0.2
	↓	↓	↓	↓
	[60, 75]	[0, 0.2]	[4, 12]	[0.05, 0.2]
	↓	↓	↓	↓
new June 3	0	1	0	0.33
new June 6	1	0	0.5	0
new June 9	0.66	0.5	1	1

Figure B3-49: The `MinMaxScaler`, like most feature-wise normalizers, reads all the values for each feature, finds the minimum and maximum, and re-scales the data to the range [0,1]. Each feature (a column in this example) is scaled independently (this is a variant of Figure 12-37).

Our sine wave data has only one feature, with many time steps, and it's a 1D list (that is, it's not a column as `MinMaxScaler` is expecting). So let's reshape our data into a column. In Python, that means making a 2D grid that is as tall as our collection of measurements, and just one element wide, as in Figure B3-50.

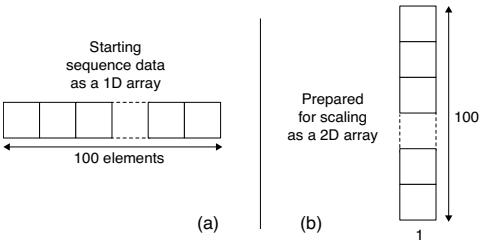


Figure B3-50: Reshaping our list of sine wave values as a 2D grid with one column

We can use `reshape()` to make this, as shown in in Listing B3-53, where `train_sequence` and `test_sequence` can be the corresponding variables from either data set 0 or 2.

```
train_sequence = np.reshape(train_sequence, (train_sequence.shape[0], 1))
test_sequence = np.reshape(test_sequence, (test_sequence.shape[0], 1))
```

Listing B3-53: Our input data contained in two 1D lists called `train_sequence` and `test_sequence`. To prepare them for `MinMaxScaler` we turn each list into a column of elements.

Now that we have our data in the right format to give to `MinMaxScaler`, we'll make an instance of that object and then call its `fit()` routine on the training data. It will find the minimum and maximum values, and remember them. Then, as usual, we apply the transformation to both the training and test data by calling the scaler's `transform()` method, as in Listing B3-54. To keep things clear, we'll give the results new names, prefixed with `scaled_`.

```
from sklearn.preprocessing import MinMaxScaler

min_max_scaler = MinMaxScaler(feature_range=(0, 1))
min_max_scaler.fit(train_sequence)
scaled_train_sequence = min_max_scaler.transform(train_sequence)
scaled_test_sequence = min_max_scaler.transform(test_sequence)
```

Listing B3-54: We make our transformation from the training data, and then apply it to both the training and test data, which we save in new variables.

Note that, as usual, we first fit the scaling object to the training data, and then applied that transformation to the test (or validation) data. Our object `min_max_scaler` remembers its transformation, so we'll later be able to apply its inverse to the output of our network, giving us a result in the same range as the input.

Now that our data is normalized, it's time to create the little windowed sublists that make up our training and test data. Figure B3-51 shows the idea.

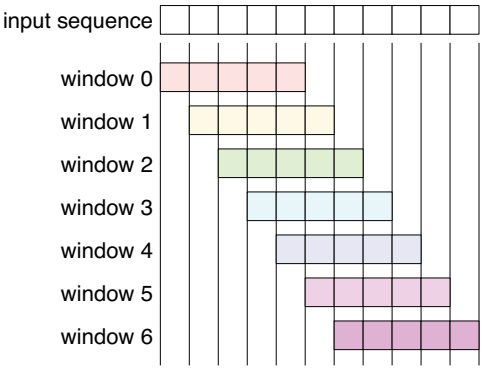


Figure B3-51: Chopping up an input sequence into a series of sub-lists. Each sub-list has the same length, called the window length. In this example, the windows are overlapping, with each one starting just one element to the right of the previous window.

Once we have the windows, we can split them into the sample, which will be everything but the final value, and the target, which is that final value, as in Figure B3-52.

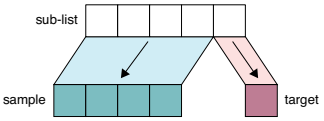


Figure B3-52: We break up each windowed list into all the elements but the last, which make up our sample, and the last element, which is the target.

Building these windows is a common operation when working with RNNs. There are probably a million ways to write a routine to do this job. We'll present a little version that emphasizes clarity. It moves the starting point of a window of the given size from the start of the sequence to a bit before the end, stopping at the last position where the whole window still fits inside the input sequence. Because this is straight Python programming, and doesn't have anything to do with machine learning, we won't go deeper into the details. Listing B3-55 shows our routine.

```
def samples_and_targets_from_sequence(sequence, window_size):
    '''Return lists of samples and targets built from overlapping
    windows of the given size. Windows start at the beginning of
    the input sequence and move right by 1 element.'''
    samples = []
    targets = []
    for i in range(sequence.shape[0]-window_size): # i is starting position
        sample = sequence[i:i+window_size]         # sub-list of elements
        target = sequence[i+window_size]           # next element
        samples.append(sample)                     # append sample to list
```

```
        targets.append(target[0])                  # append target to list
    return (np.array(samples), np.array(targets))  # return as Numpy arrays
```

Listing B3-55: A Python routine to convert a list of values and a window size into two new lists. The first contains multiple overlapping sub-sequences from the original list. Each sub-sequence is 1 less than the given value of `window_size`. The second list contains the next value in the original sequence, which will be our target when training and testing.

We can now create our training and test data just by handing our scaled sequences to this routine. Listing B3-56 shows the code. As before, we'll assign the windowed training data to `X_train` and `y_train` and the windowed testing data to `X_test` and `y_test`. We'll assume that the integer variable `window_size` has been set.

```
(X_train, y_train) = samples_and_targets_from_sequence(
                                  scaled_train_sequence, window_size)
(X_test, y_test) = samples_and_targets_from_sequence(
                                  scaled_test_sequence, window_size)
```

Listing B3-56: How to create our training and test data using the utility function in Listing B3-55

Now that we have our data, we have to make sure it has the necessary *shape*.

Getting the shape of the data into the right form for the network is as essential for RNNs as it is for all the other types of networks we've seen. If we organize the data in a way that doesn't match what the network is expecting, we'll typically either get an error, or our network will go haywire and produce crazy results.

The good news is that we've already accomplished that mission, because we wrote the routine `samples_and_targets_from_sequence` to return its data in the shapes that we need for RNN training in Keras.

Let's look at those shapes, so it's clear how the data is structured.

The easy part is the targets, which we're saving in `y_train` and `y_test`. These are just 1D lists.

The training and test data that we're saving in `X_train` and `X_test` are 3D blocks. The `X_train` block is as deep as the number of windows that we were able to make (that is, the number of samples), and as tall as the window size itself (that is, the number of time steps). The block is as wide as the number of features we're learning. Since we have only 1 feature in this dataset, the block is only 1 element wide.

This is the structure that we want for training RNNs in Keras. The depth of the block tells us the number of samples we'll train with. The time steps are arranged vertically, and the features horizontally. Figure B3-53 breaks down this shaping for an imaginary data set with 3 samples, each made of 2 features, each with 7 time steps.

In Figure B3-53, we have 3 samples, each containing 2 features, which in turn hold 7 time steps. In part (a) we see the `X_train` data set ready for learning by an RNN. Part (b) shows that the first sample from `X_train` is the slice of the block that is closest to us. Here it's the elements labeled A through G. This can be represented as a 2D grid. Part (c) shows that the

first feature in this sample is located in the leftmost column. In part (d), we see that the elements inside that column are the time steps corresponding to that feature.

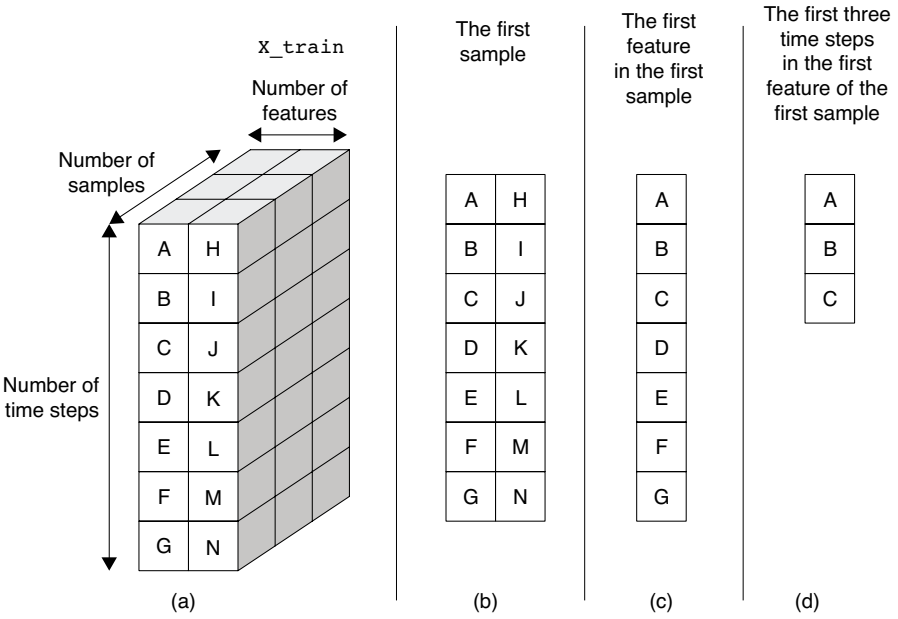


Figure B3-53: The structure of data prepared for RNN training

As we mentioned, we set up our pre-processing so that we now have our data in the proper shape, as shown in Figure B3-54.

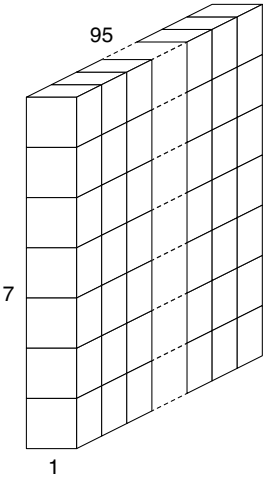


Figure B3-54: The shape of our training data, assuming we have 95 samples and a window size of 7. The test data has a similar shape, though fewer samples.

This wraps up the pre-processing of our data so it's ready for training.

Building, Compiling, and Running the RNN

Now that our data is properly normalized and structured, we can create the network and train it.

We'll create an extremely simple RNN that runs quickly, yet still demonstrates all the basic principles. We'll have one recurrent layer, followed by one dense layer. Figure B3-55 shows the architecture. Note that the dense layer has no activation function listed. That's because we don't want it to modify the value it computes, since that value is our prediction. We can either say that we've left off the activation function, or we've set it to the linear function.

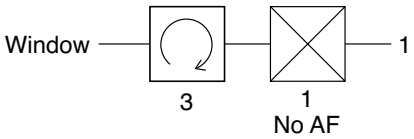


Figure B3-55: Our first RNN will have 1 recurrent layer with 3 elements of memory, followed by a fully-connected layer with one unit. The input contains as many time steps as are in our window. The dense layer has no activation function.

Recall from Chapter 19 that there are two standard types of building blocks used in RNNs: the LSTM (Long Short-Term Memory), and GRU (Gated Recurrent Unit). Keras supports both of these, and the layers are named for the unit they use.

Let's use an LSTM. An RNN icon, as in Figure B3-55, represents an LSTM unit unless explicitly marked otherwise.

To create an LSTM layer, we just create an `LSTM` object with the options we want, and then put it into our model with `add()` as usual.

How much memory should be in the state for this layer? Let's arbitrarily start with just 3 elements.

When we make an `LSTM` layer, we specify the number of cells we want. Because this will be our first layer of the network, we also have to supply the input dimensions. As usual, we set the argument `input_shape` to the shape of one sample. From our discussion above, we know that each sample is a 2D grid whose height is the number of time steps (that's the height of our window), and whose width is the number of features (we have just 1), as we saw in Figure B3-53.

Listing B3-57 shows how to create this layer.

```
lstm_layer = LSTM(3, input_shape=[window_size, 1])
```

Listing B3-57: How to create an `LSTM` layer object. The first argument is the number of LSTM cells in the layer. The second argument, `input_shape`, tells the size of a single sample.

We follow this up with a dense layer of a single neuron. If we don't specify an activation function in a `Dense` layer, Keras defaults to `None`. This is good for us in this situation, because we don't want the output to be squashed down to the range [0,1] or [–1,1] or any other range. Just to be explicit, we'll include a redundant assignment of `None` to the activation function.

The complete model is shown in Listing B3-58.

```
# create and fit the LSTM network
model = Sequential()
model.add(LSTM(3, input_shape=[window_size, 1]))
model.add(Dense(1, activation=None))
```

Listing B3-58: How to create our RNN model. We just create the model as a `Sequential` object as usual, add in our RNN layer (in this case, an `LSTM` object), and then we place a one-neuron `Dense` layer at the end.

That's it for building our model. Now we just compile it and run.

As usual, to compile the model we need to supply a loss function and an optimizer. We've been using the Adam optimizer in this chapter and it's been working great, so let's keep using it. For the loss function, we don't want to use the same categorization function we used earlier, because we're no longer doing categorization. What we want is something that will compare the single value that comes out of our network with the target value for that sample. Consulting the list of loss functions in the Keras documentation, we can see that the `mean_squared_error` loss function does the job, so let's use that.

The compilation step is shown in Listing B3-59.

```
model.compile(loss='mean_squared_error', optimizer='adam')
```

Listing B3-59: How to compile our RNN. We use the `'adam'` optimizer as before. We've chosen to use the `'mean_squared_error'` loss function which is appropriate for this RNN.

Now that our model is compiled, we train it just like every other model by calling `fit()`. The only change we'll make here is to use a batch size of 1, because some experimentation showed that for this tiny network and this small dataset, that produced the best results. Listing B3-60 shows our command to train the network.

```
history = model.fit(X_train, y_train, epochs=number_of_epochs,
                    batch_size=1, verbose=2)
```

Listing B3-60: Training our RNN with `fit()`. This is like all of our other training steps, except we've set `batch_size` to 1.

Bringing these steps together gives us the code in Listing B3-61. This builds our RNN, and then trains it for whatever number of epochs we choose to save in the variable `number_of_epochs`.

```
# create and fit the LSTM network
model = Sequential()
model.add(LSTM(3, input_shape=[window_size, 1]))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')

history = model.fit(X_train, y_train, epochs=number_of_epochs,
                    batch_size=1, verbose=2)
```

Listing B3-61: Building and training our RNN

Now that we have our model trained, we can ask it for its predictions. In Listing B3-62 we compute predictions for both the training and test data.

```
y_train_predict = model.predict(X_train)
y_test_predict = model.predict(X_test)
```

Listing B3-62: We get predictions from our model by handing the sample data to the model's `predict()` method, as usual.

Before we look at the results, we should mention that when we use the network to make predictions, we rarely use the results directly.

The issue is that we've applied the `MinMaxScaler` to our input data (both training and test), transforming it from our original values to a range more suitable for training. That means that *the network's predictions are also transformed*. For example, if our original data was in the range [–6,6], then after the transformation it will be in the range [0,1]. This means that the predictions will be roughly in the range [0,1] as well.

Because of this, we cannot directly compare our predictions with our original data. In the case of a simple scaling, like we're doing here, that's not a major issue. But if we perform a more complicated processing step, then it could be very hard to mentally interpret the predicted data.

As we saw in Chapter 10, the general solution is to *inverse-transform* the predicted data. Like most of scikit-learn's transformation routines, `MinMaxScaler` come with a method called `inverse_transform()` that does just this.

In Listing B3-63 we use the `inverse_transform()` routine of `min_max_scaler` (our `MinMaxScaler` object) to un-transform our predictions. We'll also invert the transform on our training and test targets.

```
# inverse-transform original targets
inverse_y_train = min_max_scaler.inverse_transform([y_train])
inverse_y_test = min_max_scaler.inverse_transform([y_test])

# inverse-transform predictions
inverse_y_train_predict = min_max_scaler.inverse_transform(y_train_predict)
inverse_y_test_predict = min_max_scaler.inverse_transform(y_test_predict)
```

Listing B3-63: We invert both the original target data and the predicted targets for both the training and test sets. This undoes the scaling operation we performed using the `transform()` method of `min_max_scaler()`.

Inverting (that is, un-transforming) the original targets `y_train` and `y_test` may seem wasteful. Why not simply save the original targets in their un-transformed form, and use them here?

Let's look again at the last two lines of Listing B3-54, repeated here as Listing B3-64.

```
scaled_train_sequence = min_max_scaler.transform(train_sequence)
scaled_test_sequence = min_max_scaler.transform(test_sequence)
```

Listing B3-64: A repeat of the last two lines of Listing B3-54, where we applied transformations to our data

We can see that we transformed the entire windowed sequence before we split it into a sample and a label, so we never really had the labels `y_train` and `y_test` sitting around in non-transformed variables before.

We structured the code this way for clarity and simplicity. It's certainly reasonable to extract and save the labels before they're transformed. Then we wouldn't need to undo the transformation here. Either way works. We'll stick with the version we just presented.

Now that we have the predictions back in the original range of the data, we can plot them with the original data and see how good our predictions are. We can also use them to get a quick numerical summary of accuracy using a measure called the *root mean squared error*, or *RMS error*. This is a standard way to measure error that lets us compare apples to apples when we look at multiple networks. It's close to what the `'mean_squared_error'` loss function is computing, except that we include a square root. To compute this, we use the square-root routine `sqrt()` from the `math` module, and the `mean_squared_error()` routine from scikit-learn. Listing B3-65 shows the steps.

```
from sklearn.metrics import mean_squared_error

trainScore = math.sqrt(mean_squared_error(inverse_trainY[0],
                                          inverse_y_train_predict[:,0]))
print('Training RMS error: {:.2f}'.format(trainScore))

testScore = math.sqrt(mean_squared_error(inverse_testY[0],
                                         inverse_y_test_predict[:,0]))
print('Test RMS error: {:.2f}'.format(testScore))
```

Listing B3-65: Computing and reporting the root-mean-squared (RMS) error for our training and test predictions.

The funny indexing comes from the different shapes of the original targets and their predictions. In our code, `inverse_y_train` and `inverse_y_test` are 2D grids with one row (for example, if there are 30 targets in each set, their shapes would be 1 by 30), so we get a list containing the data stored in the first (and only) row by selecting `inverse_y_train[0]`. On the other hand, the predictions coming back from `model.predict()` are 2D grids that have one column (so they would be 30 by 1). We get a list of the data in the first (and only) column by selecting `inverse_y_train_predict[:,0]`.

Issues like getting these indices right are frequently hard to anticipate, so we often discover them only when we write some code that seems reasonable but then messes up. Using interpreted Python interactively lets us examine the shapes of our variables step by step and line by line, and develop the proper adjustments and selections to select the data we want at each step.

Analyzing RNN Performance

Let's try out our sine wave data on this tiny RNN. We'll start with the easier, first data set in Figure B3-46, shown here again as Figure B3-56.

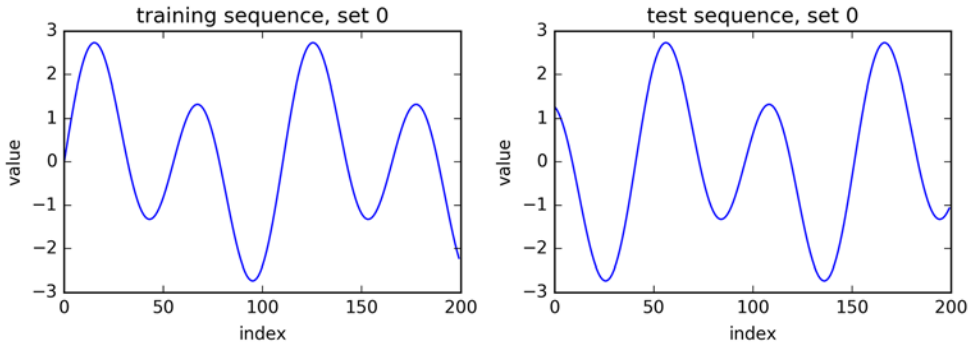


Figure B3-56: The "calm" data set. This figure is a repeat of Figure B3-46. There are 200 data points in this set.

Recall that our data-generating routine with the ungainly name `samples_and_targets_from_sequence()`, takes an argument called `window_size` that lets us specify how many time steps are to be used in each sample.

Let's arbitrarily start with a window size of 3 steps. This means each sample will have 3 values, and we'll ask the network to predict the one that comes after. We'll always provide that value as the target, so during training the system can learn to match that value, and during testing we can see how well we did.

As usual, we'll train for 100 epochs. After each epoch, we get back a single number that tells us our loss, measured by the difference between the value we predict and the target.

The loss is plotted in Figure B3-57.

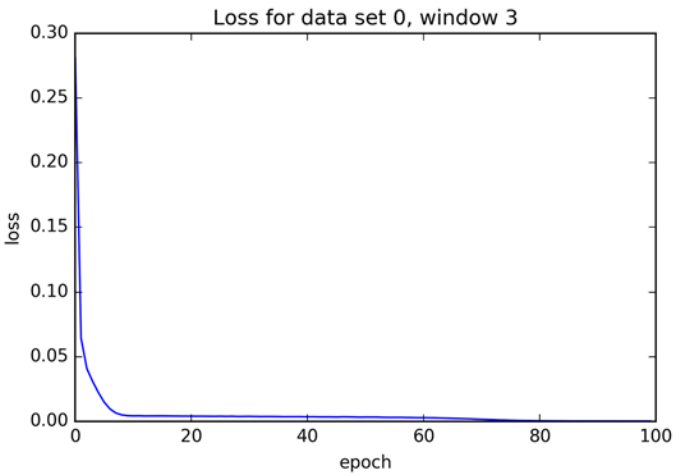


Figure B3-57: The loss from training 100 epochs of our RNN on data set 0 with a window of 3 time steps

The loss drops quickly to about 0.06, then more gently to something close to zero around epoch 8, and then over the next 60 epochs or so it continues to drop, finally hitting a value visually hard to distinguish from zero at around epoch 80.

Let's draw our predictions on top of our data so we can eyeball our performance.

Figure B3-58 shows our training data in black, and the predictions in red. Recall that we're using 200 samples in our training set.

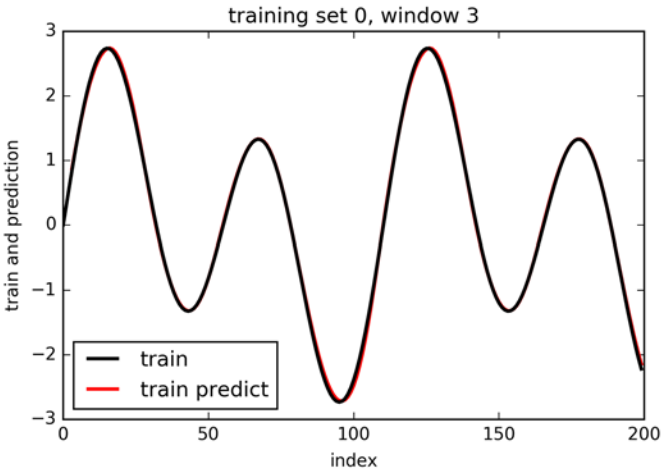


Figure B3-58: Our training data for set 0 is shown in black. Overlaid on that are the predictions from our RNN after 100 epochs of training with a window of 3 time steps.

This is pretty spectacular for a network with only 1 LSTM unit with 3 elements of state and 1 neuron after that. The match between the predictions and the real values isn't perfect, as we can see near the tops of the hills and bottoms of the valleys, but it's pretty great.

The predictions start slightly after the start of the training data, because the first prediction is the 4th element of the training data. This is hard to see in this figure, but will be easier to spot in later results (such as Figure B3-70).

Let's now looks at the test data. Figure B3-59 shows our test data in black, and the predicted data in red. Recall that we're also using 200 samples in our test set.

The test data looks similar to the training data because it's made from the same repeating sine waves, just located later in the sequence. The test predictions are close, again messing up near the extremes of the hills and valleys.

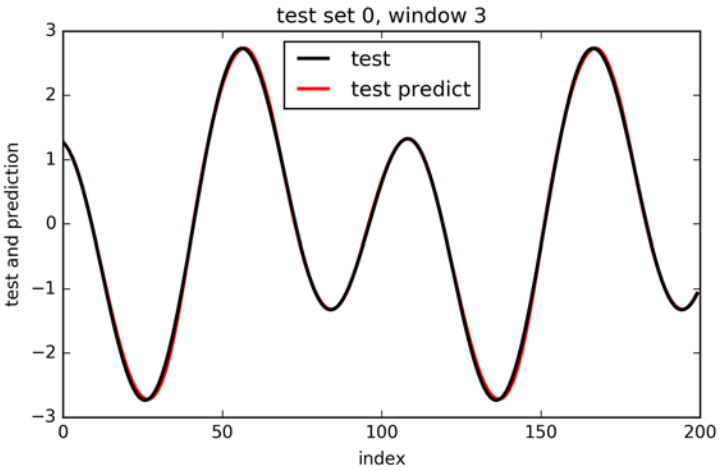


Figure B3-59: Our RNN's predictions for the test data of set 0, after 100 epochs of training with a window of 3 time steps

Maybe we don't even need 3 samples in our window. What if we try a window of just 1 sample? So we give the network a single value, and ask it to predict the next one. This only has even a hope of working because our dataset probably doesn't have any numbers that repeat exactly. So if it can learn the value that comes after each value in the training set, it should be able to reproduce those numbers. The loss is plotted in Figure B3-60.

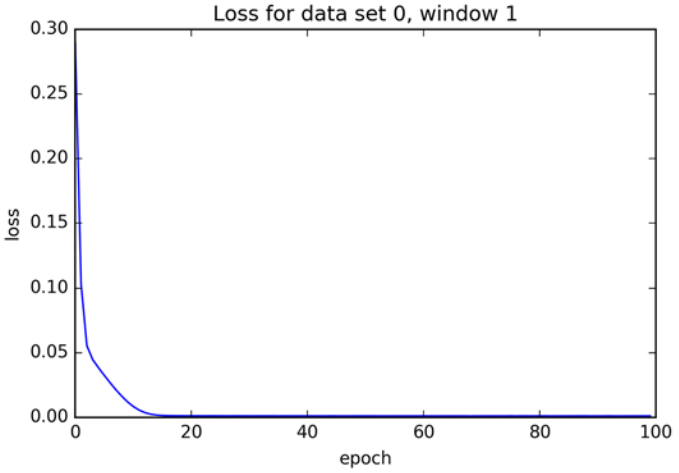


Figure B3-60: The loss from training 100 epochs of our RNN on data set 0 with a window of 1 time step

Figure B3-61 shows its predictions on our test data.

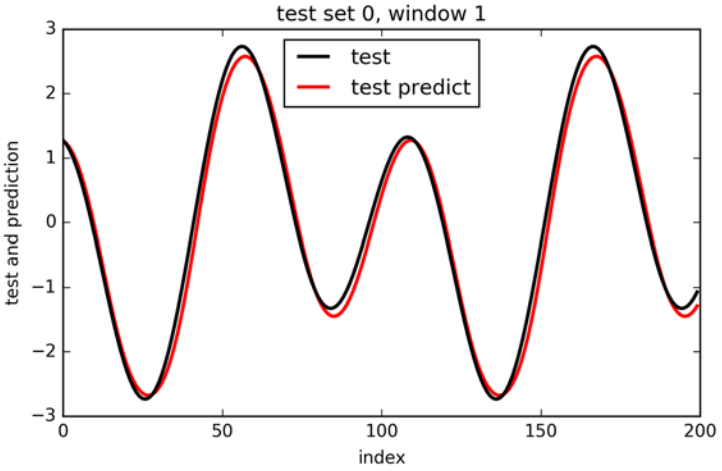


Figure B3-61: Our RNN's predictions for the test data from set 0, after 100 epochs of training with a window of 1 time step

We've clearly lost considerable performance, but the network is doing a great job at memorizing input/output pairs.

Even though 3 steps did a good job predicting our test data, let's go the other way and crank up our window size up to 5 time steps. Since we're just trying to get a feeling for things now, rather than carry out a detailed analysis, we'll skip the curve showing the training predictions, and go straight to the loss curve and test predictions. Figure B3-62 shows the loss for a window of 5 time steps.

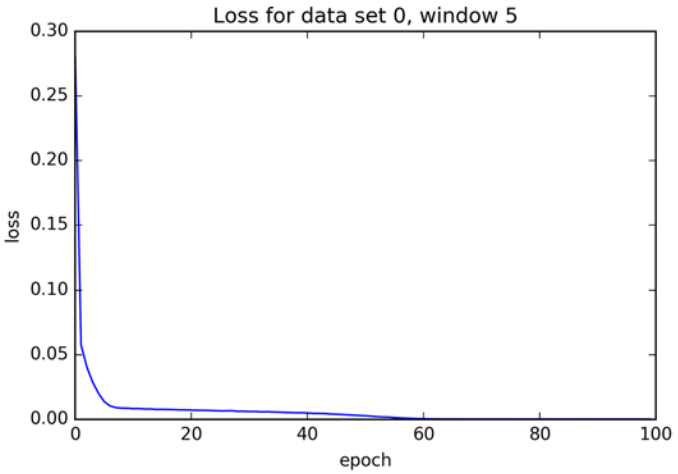


Figure B3-62: The test set loss from training 100 epochs of our RNN on data set 0 with a window of 5 time steps

Figure B3-63 shows our test predictions for a window of 5 steps.

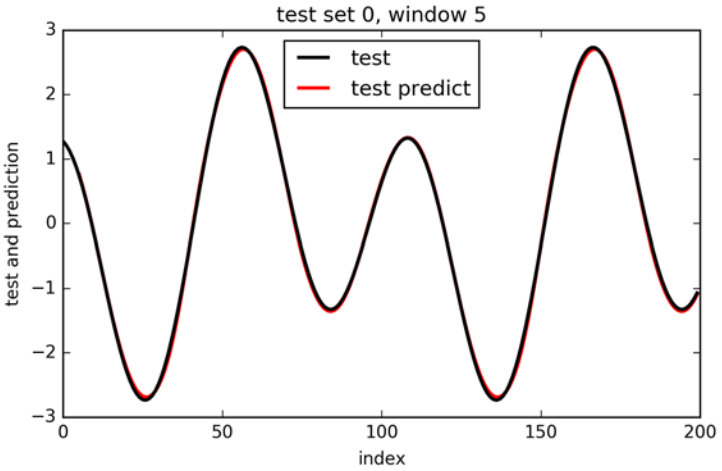


Figure B3-63: Our RNN's predictions for the test data from set 0, after 100 epochs of training with a window of 5 time steps

Visually, this looks a lot like our window of 3 steps in Figure B3-59. Perhaps a window of 3 pieces of data was enough for the network to do a really good job of predicting the 4th.

Let's try the more complicated data in our second test set, shown in Figure B3-48, repeated here in Figure B3-64.

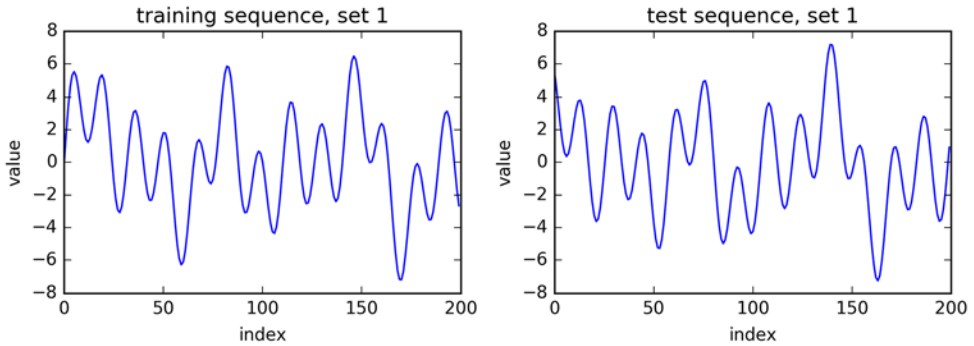


Figure B3-64: Data set 1. This figure is a repeat of Figure B3-48.

Since a window of 3 worked well for the simple data, let's try that again. The test set loss is plotted in Figure B3-65.

As with the first data set, the loss plunges at the start and then slows its descent. There's a knee around epoch 4 and a more gradual one around epoch 50, until the hits zero around epoch 80. This suggests that this test set is harder for our tiny network to learn than the last one, which makes sense.

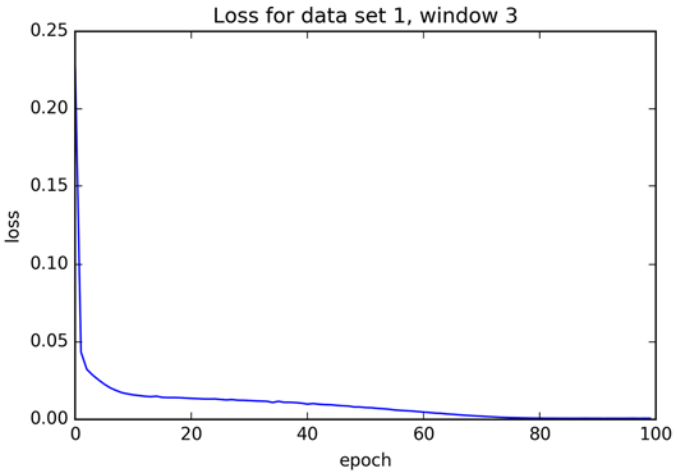


Figure B3-65: The test set loss from training 100 epochs of our RNN on data set 2 with a window of 3 times steps

Let's look at how well this window of 3 matches the test data. Figure B3-66 shows our results.

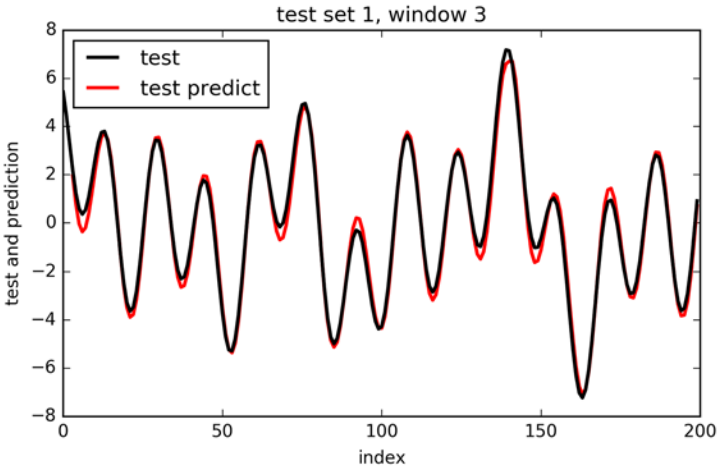


Figure B3-66: Our RNN's predictions for the test data of set 2 after 100 epochs of training with a window size of 3 time steps

This is a pretty great match for such a tiny network and such a wiggly set of data, particularly with such small windows. Let's try a window of 5 elements, shown in Figure B3-67.

Wow. Increasing the window from 3 to 5 seems to have mostly backfired, though in some places, like the peak around epoch 140, the match improved.

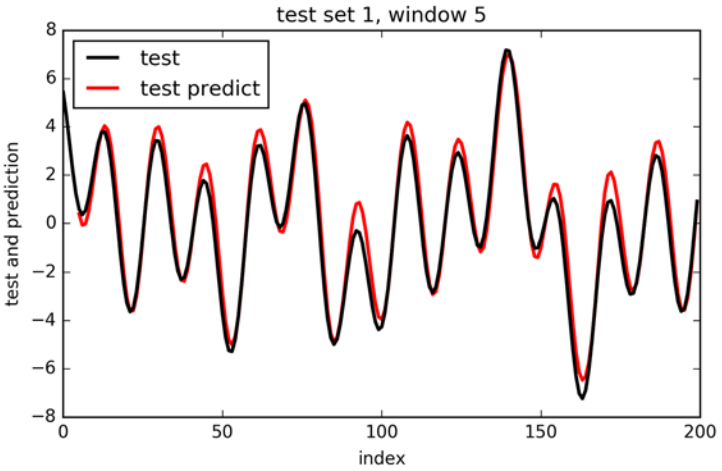


Figure B3-67: The predictions for the complex dataset with a window of 5 steps

Looking back over these results, our tiny network of just 1 LSTM unit with 3 steps of memory, and 1 final neuron, did a great job on both test sets. The window size made a difference. For these simple datasets, a window of 3 or 5 seemed to usually do a very good job. As we'll see later, more complicated datasets will often need larger windows.

A More Complex Dataset

We used only one recurrent layer so far because it worked so well. But we can build *deep recurrent networks* by simply adding in more recurrent layers. Depending on the data, it may be best to have just a few recurrent layers, each with lots of units of state memory, or it might be better to have many layers, each with just a small amount of memory.

We can make a small change to our sine-wave data to make it much more challenging for an RNN to learn: any time the curve is heading downwards, we'll flip it around the X axis so that it's heading upwards. Our curve will change from being smooth to choppy, with abrupt jumps. This is a completely arbitrary operation that we're applying to create a more challenging dataset for our networks.

Figure B3-68 shows this operation applied to the training data for our second set of waves, creating a third set of data, which we'll call data set 2.

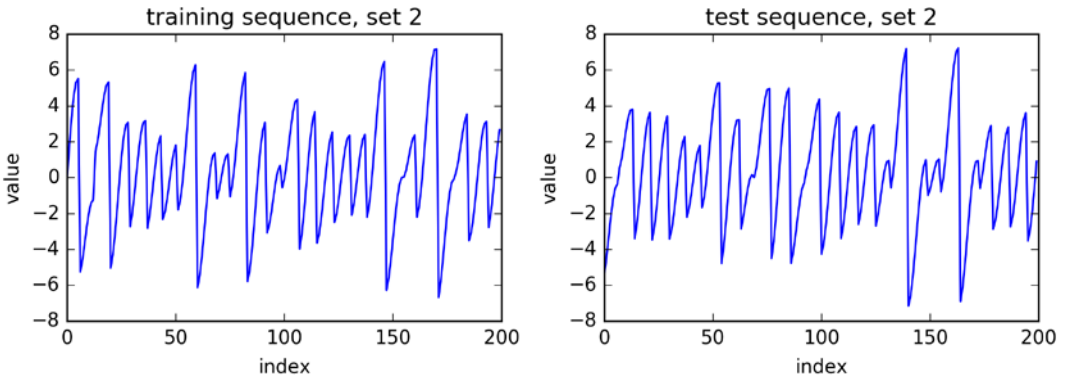


Figure B3-68: Creating a third set of data by starting with our second set of training data, only any time the curve starts to head downwards, we reflect it around the X axis. This is the data we'll use to train our deep RNNs.

Naturally enough, we call the modified data-making routine `sum_of_upsloping_sines()`, and present it in Listing B3-66.

```
def sum_of_upsloping_sines(number_of_steps, d_theta,
                           skip_steps, freqs, amps, phases):
    '''Like sum_of_sines(), but always sloping upwards'''
    values = []
    for step_num in range(number_of_steps):
        angle = d_theta * (step_num + skip_steps)
        sum = 0
        for wave in range(len(freqs)):
            y = amps[wave] * math.sin(freqs[wave]*(phases[wave] + angle))
            sum += y
        values.append(sum)
        if step_num > 0:               # are we past the first sample?
            sum_change = sum - prev_sum  # find direction we're headed in
            if sum_change < 0:         # are we going downward?
                values[-1] *= -1       # if so, flip the last value
                if step_num == 1:      # is this second sample in the set?
                    values[-2] *= -1   # if so, flip the first value, too
        prev_sum = sum      # remember the sum we found, without any flips
    return np.array(values)
```

Listing B3-66: A modification of `sum_of_sines()` where we flip the curve upside down any time it's heading downwards. The new lines are in the `if` statement and the assignment just after it.

Let's run our new dataset through the same tiny network as in our most recent experiment, keeping the window size of 5. The loss results are shown in Figure B3-69.

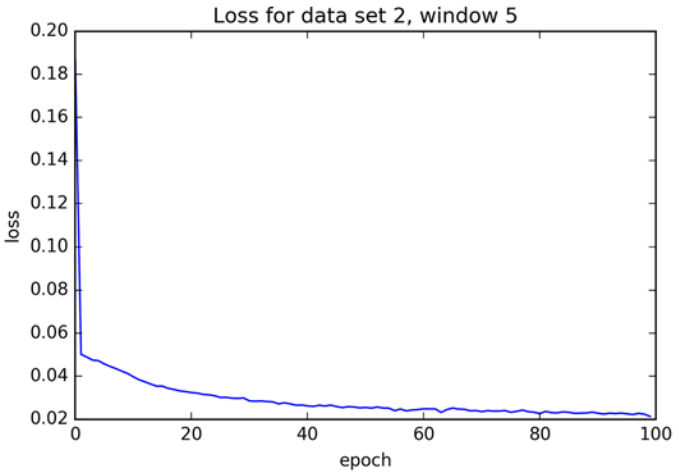


Figure B3-69: The test loss from running our upsloping test data through our 3-unit RNN with a window of 5 time steps

The loss drops fast at the very start, and then improvement slows down a lot. Around epoch 60 it seems to either have settled down, perhaps improving just a tiny bit from then on.

The predictions by this model to our modified test data are shown in Figure B3-70.

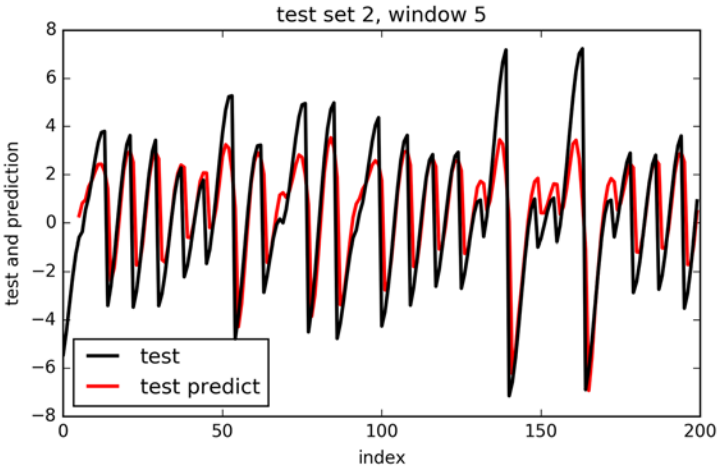


Figure B3-70: The quality of the predictions of our model to the modified test data

The late start (where the first 4 values have no prediction) is easier to see here at the far left. This result is pretty bad. The predicted values do tend to generally track the straighter sections of the test data, but the predictions frequently overshoot and undershoot the peaks and valleys.

Our tiny network worked surprisingly well up until now, but we've finally asked too much of it.

Deep RNNS

Let's add a second recurrent layer, also made of 3 LSTM units.

Figure B3-71 shows our new architecture.

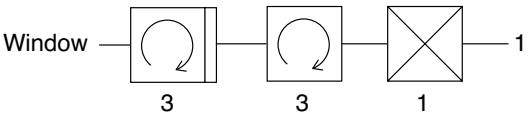


Figure B3-71: The diagram for our 2-layer RNN. The small box on the output end of the first LSTM indicates that it has `return_sequences` set to `True`.

Listing B3-67 shows how to build our 2-layer deep RNN model, using 2 layers of 3 LSTM units each.

The first LSTM needs us to include a new argument, `return_sequences`, which we need to set to `True`. We indicate this with a small box on the right of the LSTM icon (or, when the network is drawn bottom to top, on the top). We'll discuss what this `return_sequences` is about soon, but for now we can treat it as something that has to be included any time we create an LSTM that is followed by another LSTM.

Here's the code, with `return_sequences` set to `True` in the first LSTM.

```
model = Sequential()
model.add(LSTM(3, return_sequences=True, input_shape=[window_size, 1]))
model.add(LSTM(3))
model.add(Dense(1))
```

Listing B3-67: Building a deep RNN just means adding more recurrent layers. All recurrent layers that precede another must have their optional argument `return_sequences` set to `True`.

As far as Keras is concerned, this is just another `Sequential` model, so we can train this model and get predictions from it just as before.

Let's see how this performs with our new data.

Figure B3-72 shows the loss during training.

The loss gets down to a little more than 0.02, which is roughly what we saw before. So we shouldn't get too optimistic about the predictions.

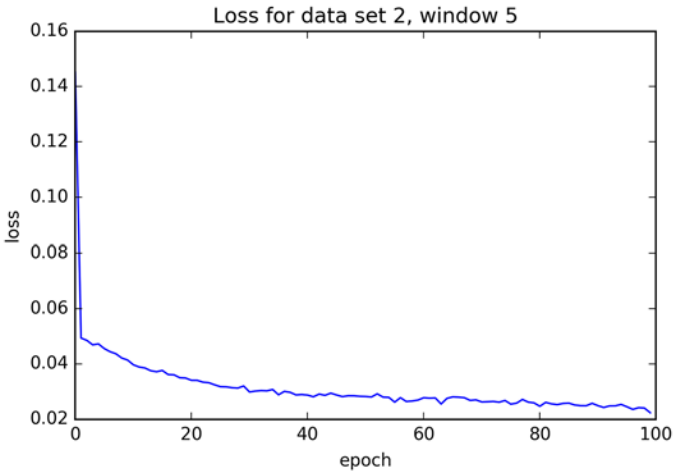


Figure B3-72: Training our model with 2 recurrent layers of 3 LSTM units each produces these loss results.

Figure B3-73 shows the predictions on the test data.

This is pretty bad. Around epochs 130 to 180 it seems to lose track of the value of the data, though it does roughly mimic its rising and falling. It seems that adding a second layer has made things a lot worse.

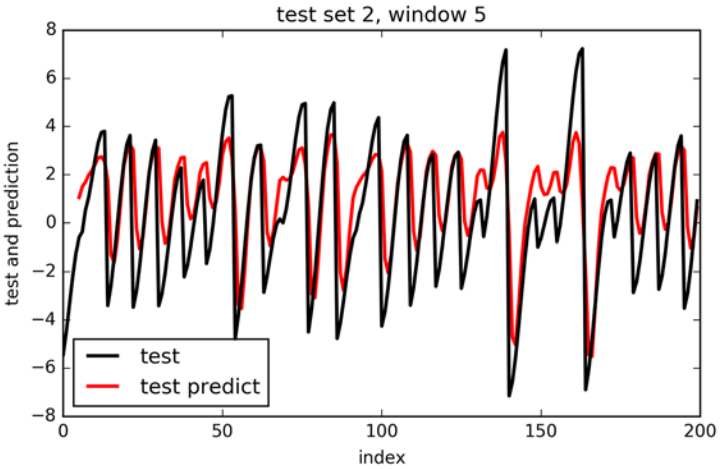


Figure B3-73: Training our model with 2 recurrent layers of 3 LSTM units each produces these predictions to the test data.

Let's see if we can get something better with an even deeper model. Let's make 3 LSTM layers of decreasing sizes, with 9, 6, and 3 units respectively. Figure B3-74 shows this architecture.

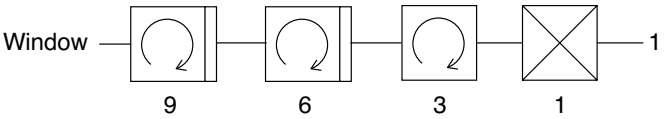


Figure B3-74: The diagram for our 3-layer RNN

Listing B3-68 shows the code to build the model.

```
model = Sequential()
model.add(LSTM(9, return_sequences=True, input_shape=[window_size, 1]))
model.add(LSTM(6, return_sequences=True))
model.add(LSTM(3))
model.add(Dense(1))
```

Listing B3-68: Making an even deeper RNN with 3 recurrent layers of decreasing numbers of LSTM units

Once again, each LSTM that feeds another LSTM has to have `return_sequences` set to `True`, indicated by the small box at the top of the icon.

Figure B3-75 shows the loss of our 3-layer model during training.

This is encouraging, because while the loss at epoch 100 is still only about 0.025, like in previous runs, the loss is still dropping, while the previous loss graphs were flat. If we kept learning, we ought to expect further improvements. It's not a dramatic improvement, though.

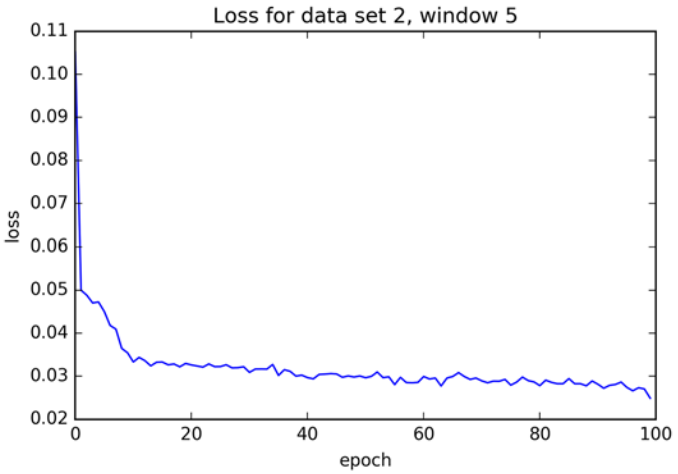


Figure B3-75: The loss from training our model with 3 recurrent layers of 9, 6, and 3 LSTM units

Figure B3-76 shows the prediction results for this deeper RNN.

The improved numerical loss is encouraging, but the network is still not doing a good job visually. This might even be worse than before.

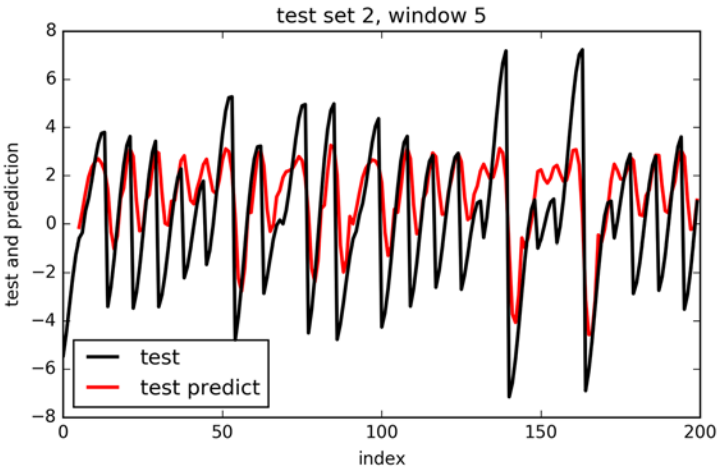


Figure B3-76: The test data predictions made by our deep RNN with 9, 6, and 3 LSTM units on successive layers

The Value of More Data

Remember our general principle that more data is usually better than fancier algorithms. So rather than continue to tweak the network, let's get more data.

One of the pleasures of working with synthetic data is that we can make as much of it as we want. The values of the frequencies in the test 2 dataset don't repeat for a long time, so we can crank out a lot of data (over 40,000 samples) without repeating. Let's increase the size of our training set from 200 samples to 2000. To match the 10-fold increase in the number of training samples, let's increase the window from 5 to 13, leaving all the other parameters the same.

The loss during training is shown in Figure B3-77.

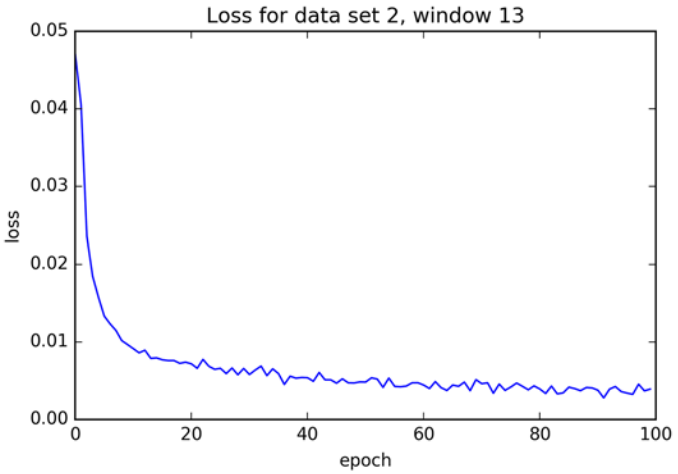


Figure B3-77: Loss from training our 3-layer network with 2000 samples and a window size of 13 time steps

This is a huge drop in loss. In Figure B3-75 the loss got down to about 0.025 after 100 epochs. Here, the loss seems to be about 0.004.

If we plot all 2000 samples they'll jam up and give us a black rectangle, so let's look at the just the first 200 samples. This has the added benefit that we're already familiar with them. Figure B3-78 shows the predictions on our test data.

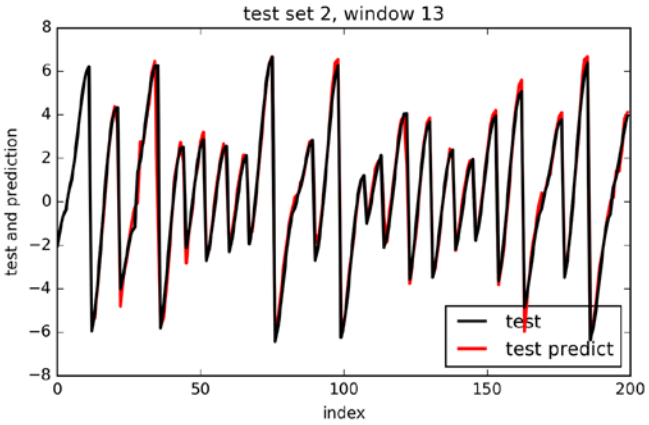


Figure B3-78: Test data predictions by our 3-layer deep RNN after training on 2000 samples chunked into windows of 13 time steps

This is a big improvement. There's still some over- and under-shooting going on, but generally we have a much better match than before.

More data really does help!

Thanks to our procedural data generator, can crank up the size of our training set by another factor of 10 to see what happens. We'll leave everything else the same, but increase the training set from 2,000 to 20,000 samples.

The loss results are in Figure B3-79.

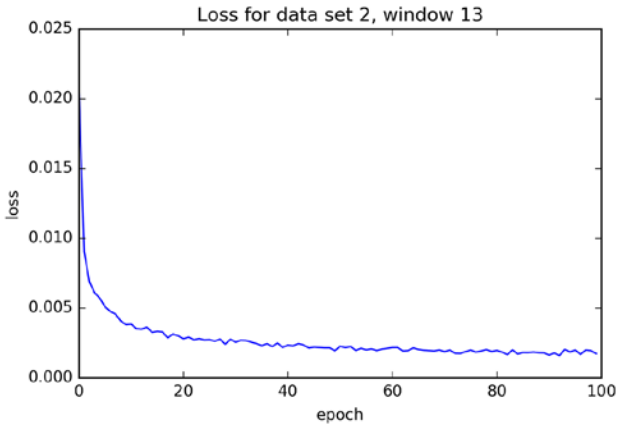


Figure B3-79: Loss from training our 3-layer network with 20,000 samples and a window size of 13 time steps

The loss has dropped to about 0.0015, which is less than half of the roughly 0.004 we had before.

Figure B3-80 shows the predictions on our test data.

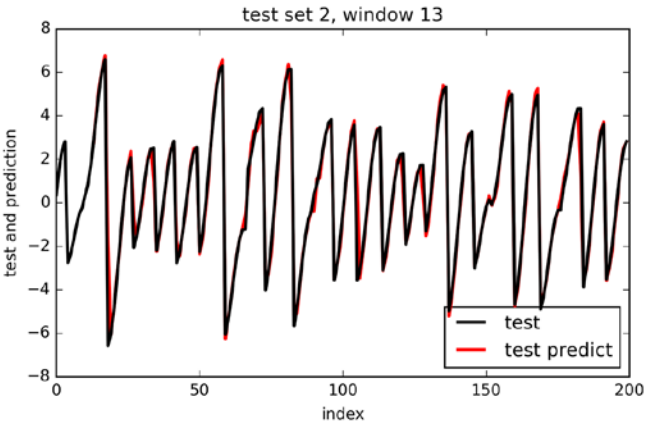


Figure B3-80: Test data predictions by our 3-layer deep RNN after training on 20,000 samples chunked into windows of 13 time steps

This is the best set of predictions of this difficult dataset that we've seen yet. There are still a few obvious misses, but compared to previous results this is pretty close.

Even more data helps even more!

It's worth noting that the time taken to learn these increasingly large training sets is roughly proportional to the number of samples. Each epoch of the 20,000 sample training set took a little over 5 seconds using only the CPU on a late 2014 iMac (that is, there was no GPU acceleration). So Figure B3-80 took a bit over 28 hours to compute. Every one of these epochs took about 10 times longer than each epoch of the 2,000 sample set, which in turn were about 10 times longer than epochs when training the 200 sample set. More data is great, but processing it comes at a price.

The good news is that we pay that price only once, during training. We've been steadily improving our 3-layer RNN without changing the model, so all of these models take the same amount of time to predict new values after training is over. Our up-front training cost is amortized over every use of our model, forever.

As we've seen, RNNs are sensitive to how they're trained. Deep RNNs are even more sensitive. Because we didn't tune these architectures at all as we added new layers, we're probably leaving a lot of performance on the table. By adjusting the window size, the learning rate, the parameters to our Adam optimizer, and our choice of loss function, we might be able to improve our best results in Figure B3-80. In practice, it is usually worth exploring the effects of different modeling and training parameters to see what works best for a given network and data.

Returning Sequences

This section's notebook is Bonus03-Keras-6-RNN-Sequence-Shapes.ipynb.

In our deep networks above, we used the output of one RNN as the input of another RNN. We saw that the earlier RNN layer (the one providing the input to the next) needed a new argument. Its name was `return_sequences` and we set it to `True` (the default is `False`). Now it's time to make good on our promise to discuss what that argument is about.

Let's return to our first, simple network of an RNN that followed by a dense layer, as we saw back in Figure B3-55. Our goal was to hand the network a sequence of time steps, and then have it predict the next value after the sequence.

Our samples contained just one feature, which held a series of values from a 1D curve. These made up the time steps.

When we gave the RNN our sample, it read the first time step and produced an output. This output was the contents of the internal state, after passing through the RNN's internal selection gate. The output could be thought of as the RNN's prediction of the next value of the curve.

But we didn't care about that prediction, because we already knew the second time step. Keras knew we had more time steps to come, so it automatically ignored that output, and didn't even send it to the dense layer. Instead, it gave the RNN the second time step. Again, the RNN produced an output, and again, Keras ignored it. Figure B3-81 shows the idea, for an RNN with 4 elements of internal data. Here we've handed the RNN the third time step, and it's produced the third output, which we're ignoring.

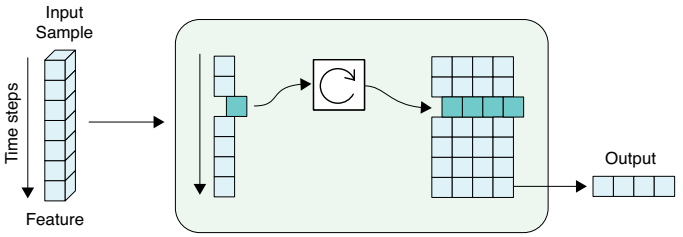


Figure B3-81: Using an RNN with 4 elements of state. The input sample has a single feature, containing a list of time steps. After each time step is evaluated, the 4-cell RNN produces an output 4 elements long, as shown here for the 3rd time step. We've been ignoring all outputs except the last one.

We did this over and over, handing the RNN sequential time steps and ignoring the outputs, until we gave it the last time step in the sample. The output of *that* time step was the prediction for the value of the sequence after the end of our inputs, so that output was the value we were after all along. We fed that to our dense layer, and the output was the prediction.

Suppose our inputs had more than one feature. If our data held weather measurements at the top of a mountain, maybe each sample held temperature, wind speed, and humidity. Let's say what we want from our RNN is a prediction of how good the radio reception would have been on the mountain at that time. So at each time step we give the RNN the values

for all three features at that time step. The output is again the RNN's internal state after the selection gate, so it has as many elements as there are elements in the internal state. As before, we get back one such output for every time step input we provide, and we only pay attention to the last one. Figure B3-82 shows the idea.

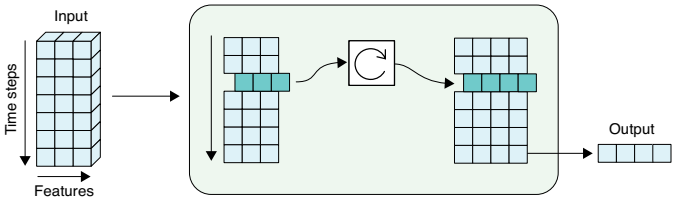


Figure B3-82: When we have multiple features in our sample, we provide all the features for a given time step to the RNN.

There are a couple of things worth noting in Figure B3-82.

First, our input is a 3D tensor. In this example, it has 1 sample, 7 time steps, and 3 features, so it has dimensions 1 by 7 by 3. The number of time steps, and the number of features, don't appear in the output, which is a 2D tensor of shape 1 by 4. The 1 is because we only care about one output (the last one), and the 4 comes from the internal state of the RNN, which we've assuming has 4 elements.

We "lost" the number of features because they are used internally by the RNN to control the forgetting, remembering, and selecting of the internal state. We "lost" the number of time steps because we chose to ignore all but the last one.

Our input could have had 19 features and 37 time steps, and the output would still be 1 by 4.

Let's expand the picture a little to include the unrolled RNN diagram. In Figure B3-83 we have a sample with 5 time steps and 3 features. Once again, the RNN's internal state has 4 elements. We can see in the figure that at each time step, an entire row of features is fed to the RNN, which produces an output. Then the RNN's state changes, which sets up the RNN for the next input, as shown by the open downward-pointing arrow. We only pay attention to the last output. The output is a 2D grid of shape 1 by 4.

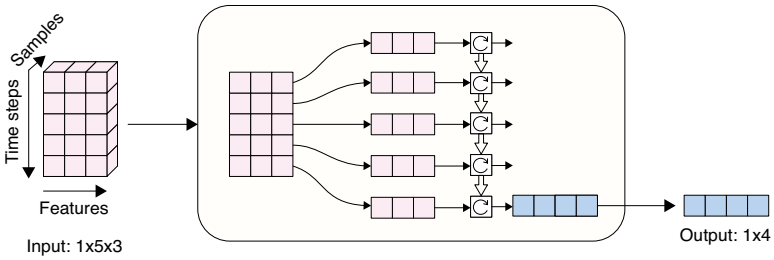


Figure B3-83: Passing a sample to an RNN. The unrolled RNN diagram is shown vertically. Read the contents of the rounded box from top to bottom over time. As before, the outputs are ignored except for the final time step.

If we want to feed this output to a dense layer, we don't have to do a thing.

But suppose we want to take our sequence of outputs and present them as a sequence of inputs to another RNN layer, as we did in some of our deep RNN models above. We know that an RNN needs a 3D input, and the output here is 2D.

We could just give it a depth of 1, producing a shape that's 1 by 1 by 4. While this is now legal for an RNN, it doesn't make any sense. A tensor with this shape would be interpreted as a single sample (the first 1) with 1 time step (the second 1), containing 4 features (the 4 at the end). That's nothing like our single sample of 5 time steps and 3 features.

Losing the time step information is a big problem, because that's the idea at the heart of an RNN. We're giving our first layer 5 time steps, and it's producing 5 outputs. We then want to hand those 5 outputs to the next layer. Each output will have 4 elements (since we're supposing that our RNN has 4 elements of internal state), so those 4 values will be interpreted by the next RNN as 4 features. But we need the 5 time steps.

That's actually easy to do. We just tell Keras to *not* ignore the output after each step. We tell it to take the outputs and stack them up to make a grid. It will be as tall as there are time steps, and as wide as there are elements in the internal state. Now we can give that grid a depth of 1, and it makes sense as an input to an RNN.

Figure B3-84 shows the idea.

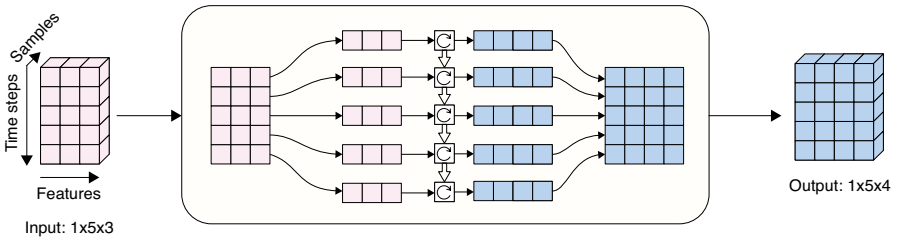


Figure B3-84: To send an RNN's output to another RNN, we just remember the output at every time step. These outputs are stacked together to form a 2D grid, and then we give it a depth of 1 to mean it's all one sample, and we're ready to give this to another RNN.

To tell Keras to remember the output after each time step and build up this grid, we tell it that we want the RNN to return not just a single output, but the whole sequence of outputs corresponding to the sequence of inputs.

By setting the optional argument `return_sequences` to `True`, we're telling Keras to do exactly what Figure B3-84 shows.

Now that we know what `return_sequences` is all about, we can usually invoke it without even thinking about all of this. If our RNN's output is going into another RNN, just set `return_sequences` to `True`. If we want only the output after the last time step, we can set `return_sequences` to `False`, or just leave it off, since that's the default value.

A few input shapes and their outputs with `return_sequences` set to both `False` and `True` are shown in Figure B3-85.

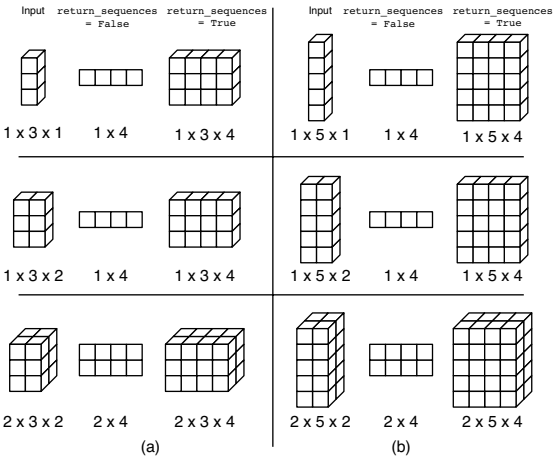


Figure B3-85: The output of a 4-cell RNN for different shapes of input. In each box, the input shape is on the left, the output with `return_sequences=False` is in the middle, and the output with `return_sequences=True` is on the right.

It's useful to see at a glance whether an RNN returns just the final output or the full sequence. We mark the icon for an RNN that returns a sequence with a small box on the output side, suggesting multiple outputs, as in Figure B3-86.

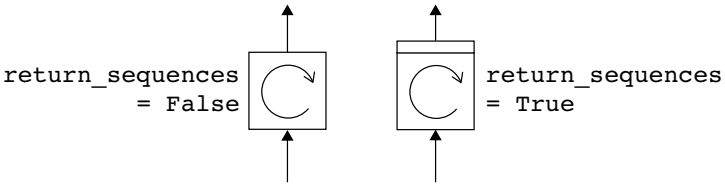


Figure B3-86: Icons for RNN units. Left: When `return_sequences` is `False`, the RNN returns only the final value in the sequence. Right: When `return_sequences` is `True`, the RNN returns its output for each time step. We mark this with a small box on the output side of the icon.

Stateful RNNs

We've been focusing on one sample at a time, but in practice we usually train in mini-batches. This poses an interesting question for RNNs, since their internal memory is always influenced by previous inputs. When should we clear that memory and let the RNN start over?

The usual approach is to clear the internal memory at the start of a new batch, or mini-batch. We don't clear or reset the weights belonging to the neurons inside the RNN, since those tell it how to do its job. We only clear its changing memory that holds the inputs it's recently seen. The thinking is

that when a new batch begins we'll possibly be getting data that isn't a continuation of the most recent samples, so we don't want to remember stuff from back then.

Usually, we shuffle our samples between epochs, so they arrive in an unpredictable order each time. But we can keep their order consistent from epoch to epoch, if we want to. We do this when we call `fit()` to train our model, setting the optional argument `shuffle` to `False` (the default is `True`).

When the data is always arriving in sequence, there's no reason to reset the memory at the start of each batch, because those samples follow the samples in the previous batch. In other words, the batching just breaks up the grouping of the samples, and not their sequence. In that situation, we can tell Keras to *not* reset the memory at the start of each batch. This sometimes can help us train a little faster.

In Keras, when we take over the responsibility for clearing the memory we say that the RNN is in the `stateful` mode. In this mode, Keras only resets the internal state when we tell it to. Usually this is at the start (or end) of each epoch.

Stateful mode can make training go a little faster, but it comes with limitations. The batch size must be determined in advance, and it becomes a part of the model. The dataset must be a multiple of this batch size. For instance, if the batch size is 100, the dataset must be 100, 200, 300, and so on samples long. If it's 130 or 271 samples, we'll get an error.

When we later give new data to the model for it to evaluate, that data also has to come in batches of the same size we used when we trained. If we want only one prediction, but our batch size is 100, then we can either pad out our one request with 99 more copies of itself, or just load up all the unused entries in the batch with 0's. We'll still end up waiting for the network to evaluate all those samples, though.

To make a stateful network, we need to do four things.

First, we need to include the optional argument `stateful` to each RNN (such as an LSTM or GRU) and set it to `True`. This tells Keras that we're taking care of when to reset the cell's state.

Second, we need to include the argument `batch_size` to the first RNN we make, and set it to the batch size that we're going to use during training.

Third, when we call `fit()` we need to set `shuffle` to `False`.

Finally, when we want to reset the state, we need to explicitly call `reset_states()` on our model.

Listing B3-69 shows an example of the first two points while building a small RNN with two `LSTM` layers and a one-neuron `Dense` layer at the end. This is adapted from the *stateful_lstm.py* example in the Keras documentation [Chollet17b]. We assume that the variable `time_steps` holds the number of time steps in our inputs, and `batch_size` is set to the batch size we plan to use.

```
model = Sequential()
model.add(LSTM(50,
               input_shape=(time_steps, 1),
```

```
                return_sequences=True,
                batch_size=batch_size, stateful=True))
model.add(LSTM(50, stateful=True))
model.add(Dense(1))
```

Listing B3-69: Building a stateful RNN. We need to give the first LSTM a value for `batch_size`, and set `stateful=True` in each LSTM. Adapted from [Chollet17b].

To train our model, we need to remember to tell `fit()` not to shuffle our data. Because we want to reset the state after each epoch, we don't want to do the usual thing of telling `fit()` how many epochs to train for, and then walking away, because then the RNN will never be reset.

Instead, we'll tell `fit()` to train for only 1 epoch, and we'll put that call in a loop. The loop will repeat for the number of epochs we want to train for. Doing it this way lets us put in a call to `reset_states()` at the end of each epoch of training.

Listing B3-70 shows this step.

```
for i in range(number_of_epochs):
    model.fit(X_train, y_train, batch_size=batch_size,
              epochs=1, verbose=1, shuffle=False)
    model.reset_states()
```

Listing B3-70: Training a stateful RNN. We need to tell `fit()` not to shuffle the data, and then we need to call `reset_states()` after each epoch. Adapted from [Chollet17b].

Time-Distributed Layers

As we've seen, when we set an RNN unit's `return_sequences` argument to `True`, Keras saves its output after each time step.

We saw in Figure B3-84 that this results in one output for each time step in the sample. The outputs for a sample get gathered together into a grid, and the grids for many samples get gathered together into a volume.

Let's suppose we want to process this output volume in a `Dense`, or fully-connected layer. We'd need to flatten it first into a 1D list, and then feed that list to the layer. Sometimes that's fine.

But other times, we'd like the dense layer to process the individual outputs one by one. So we want to run the dense layer over each of the separate lists coming out of the RNN in Figure B3-84 rather than over the 2D grid they get assembled into.

Once these lists have been assembled into a grid, it would be hard to pull them apart. We could write our own custom layer, or make a custom contraption using the Functional API (discussed below), but we'd like an easier approach that lets us treat these individual outputs one by one.

Keras provides a special-purpose layer for exactly this job. It's called a `TimeDistributed` layer. It's not really a layer, though. Keras calls it a *wrapper* layer. The idea is that it's a container that we put one or more layers into, and then those layers get treated in a special way.

To get a feeling for what the `TimeDistributed` wrapper does for us, let's build a tiny network without one. Figure B3-87 shows an RNN with 4 elements of state, followed by a `Dense` layer with 5 neurons. Since there's no box on top of the RNN icon, we know that it doesn't return a sequence.

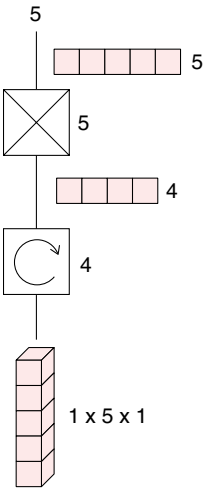


Figure B3-87: A small network to set up our discussion of the `TimeDistributed` layer

As we can see, the input is 1 sample, made of 1 feature, with 5 time steps. After passing through the RNN, we get a single 4-element list. We can then feed that directly into a `Dense` layer of 5 neurons, getting back a list of 5 values at the output.

Let's have the RNN return the individual sequence outputs by setting `return_sequences` to `True`. Now we have Figure B3-88 (a), where we had to insert a `Flatten` layer between the RNN and the dense layer.

In part (a) of Figure B3-88, we see that the RNN's 3D output does not fit the `Dense` layer's need for a 1D list, so we can flatten it first. But then the `Dense` layer is processing all 20 outputs at once. In part (b), we wrap the `Dense` layer in a `TimeDistributed` layer. Now Keras will hand it each sequence of the output in turn, and then combine the results again.

Although flattening the RNN output as in Figure B3-88(a) works, and everything will run, it's not what we want. The problem is that the `Dense` layer will process all 20 values coming out of the RNN at once. What we want is for it to process each time step individually.

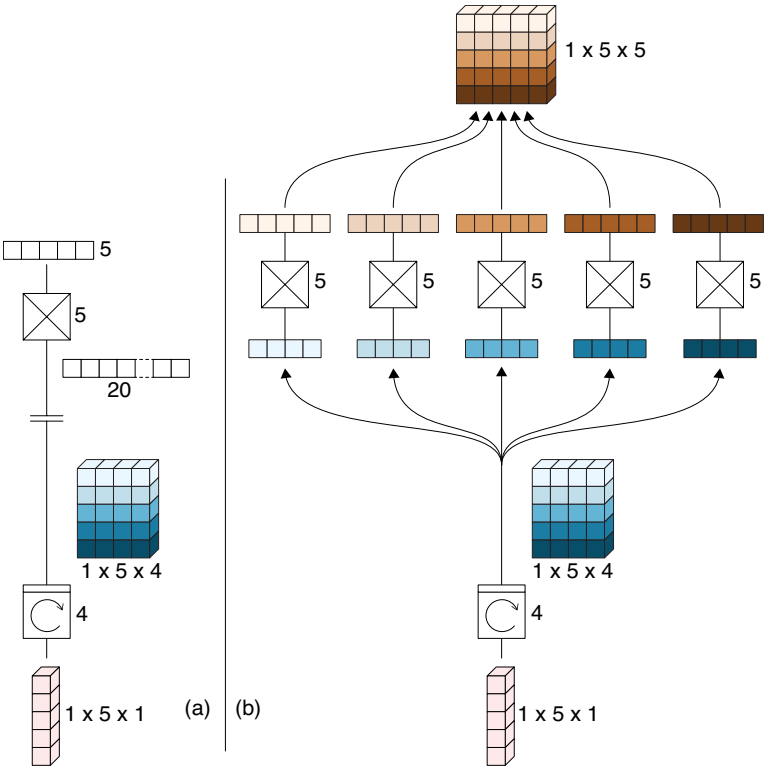


Figure B3-88: When an RNN returns a sequence, we can apply some other layer to each step of the sequence by wrapping it in a `TimeDistributed` layer.

If we wrap the `Dense` layer in a `TimeDistributed` wrapper, we invoke a bunch of machinery inside of Keras that gives us an operation shown in Figure B3-88(b). Each time step is individually handed to the `Dense` layer and then the results are combined. Note that in this figure there is only one `Dense` layer which gets applied to all 5 time steps.

Another version of Figure B3-88(b) is shown in Figure B3-89. On the left we show how we'd draw our network schematically. The five-sided shape around the `Dense` layer is our icon for the `TimeDistributed` layer. The V at the bottom is meant to suggest the branching of the lines in Figure B3-88(b), telling us that the one input is being broadened. On the right is an expanded view of what's happening inside of the `TimeDistributed` layer. Again, there's just one `Dense` layer that is being applied to each time step in this sample.

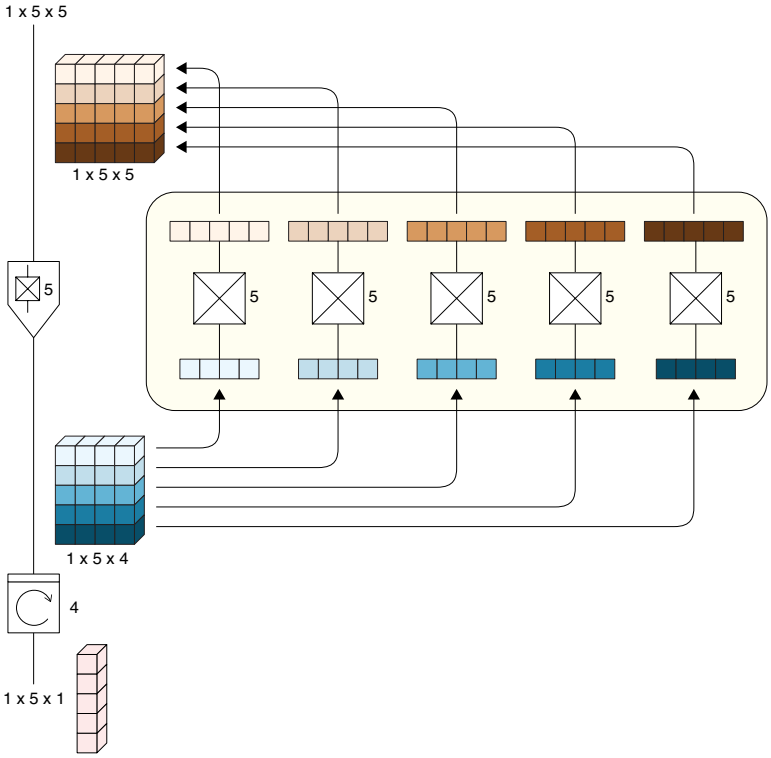


Figure B3-89: By wrapping our `Dense` layer in a `TimeDistributed` layer, the `Dense` layer will individually process each sequential output from the RNN, and then those results will be assembled into a new tensor.

To create a layer wrapped in a `TimeDistributed` layer, we just nest the calls, as in Listing B3-71.

```
model = Sequential()
model.add(LSTM(4, return_sequences=True, input_shape=[window_size, 1]))
model.add(TimeDistributed(Dense(5))
```

Listing B3-71: Feeding the output of an `LSTM` layer to a `Dense` layer, wrapped up in a `TimeDistributed` wrapper, as in Figure B3-89

Figure B3-90 shows our icon for a `TimeDistributed` layer with different contents. With the Functional API (discussed below) we can wrap many layers with just one `TimeDistributed` wrapper. Alternatively, we can wrap each one individually.

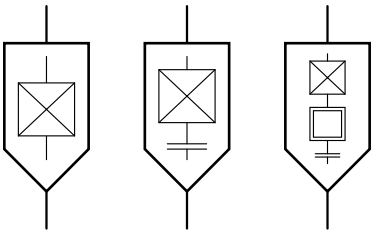


Figure B3-90: Three collections of layers, each in a `TimeDistributed` wrapper

Generating Text

The letter by letter notebook is Bonus03-Keras-7-Generate-Text-By-Letter.ipynb.

The word by word notebook is Bonus03-Keras-8-Generate-Text-By-Word.ipynb.

In Chapter 19 we experimented with generating new text based on the stories of Sherlock Holmes.

This isn't hard to do, but it requires more than just a couple of lines of Python programming. The notebooks for this section contain all the code for making new text, either letter by letter or word by word. Rather than go through all the details, we'll just walk through the big pieces and mention some highlights. Our code is influenced by a popular presentation available online [Karpathy15].

In previous notebooks in this chapter we've presented the code as essentially a single big list of lines to be executed in order. We broke up the lines into conceptual chunks and placed them in cells, but that didn't change how we wrote or used the code. For variation, this time we packaged up each of the steps into its own procedure. Then when we're ready to make text, we just call some of those procedures and let them do their work.

Our first step is to read in the source text. We replaced multiple spaces with single spaces, and removed newline characters since they don't have any semantic meaning.

This code is shown in Listing B3-72, where we've wrapped up the job of reading and processing the file into a routine called `get_text()`.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.layers import LSTM
from tensorflow.keras.optimizers import RMSprop
import numpy as np
import random
import sys

def get_text(input_file):
    # open the input file and do minor processing
    file = open(input_file, 'r')
    text = file.read()
    file.close()
    text = text.lower()
    # replace newlines with blanks, and double blanks with singles
    text = text.replace('\n',' ')
```

```
    text = text.replace('  ', ' ')
    print('corpus length:', len(text))
    return text
```

Listing B3-72: To generate new text, we start by reading in and processing the text file with our source text.

Now we have to chop up the input into overlapping windows. We need to pick the window size and how much they overlap. The routine `build_fragments()` creates these little fragments for us, as in Listing B3-73.

```
def build_fragments(text, window_length):
    # make overlapping fragments of window_length characters
    fragments = []
    targets = []
    for i in range(0, len(text)-window_length, window_step):
        fragments.append(text[i: i + window_length])
        targets.append(text[i + window_length])
    print('number of fragments of length window_length=',
                window_length,':', len(fragments))
    return (fragments, targets)
```

Listing B3-73: We build text fragments by breaking up the input text into overlapping pieces, each with `window_length` characters.

Since our network wants numbers, not letters, we'll assign a unique number to each letter. To make it easy to go back and forth, we'll make two dictionaries. One is keyed on characters and returns their number, and the other is keyed on number and returns their character. We'll call the number an "index." We can get the total number of unique characters by using Python's `set()` operation. Just for general tidiness we'll sort that list before using it. Listing B3-74 shows the routine `build_libraries()` that does the job.

```
def build_dictionaries(text):
    unique_chars = sorted(list(set(text)))
    print('total unique chars:', len(unique_chars))
    char_to_index = dict((ch, index) \
        for index, ch in enumerate(unique_chars))
    index_to_char = dict((index, ch) \
        for index, ch in enumerate(unique_chars))
    return (unique_chars, char_to_index, index_to_char)
```

Listing B3-74: We build a pair of dictionaries to let us turn each letter into a unique number, and vice-versa.

Now we want to turn our samples and targets into one-hot vectors. We're already familiar with using one-hot targets. We'll use one-hot encoding for the samples here as well because we want each letter to be a feature in our data. That feature will have as many time steps as there are unique characters in our data. They'll all be 0 except for a 1 corresponding to the character being represented.

Listing B3-75 shows one way to build the one-hot versions. We'll make a couple of grids full of zeroes, and then set the ones where needed.

```
def encode_training_data(fragments, window_length, targets,
                         char_to_index, index_to_char):
    # Turn inputs and targets into one-hot versions
    X = np.zeros((len(fragments), window_length, len(char_to_index)),
                         dtype=np.bool)
    y = np.zeros((len(fragments), len(char_to_index)), dtype=np.bool)
    for i, fragment in enumerate(fragments):
        for t, char in enumerate(fragment):
            X[i, t, char_to_index[char]] = 1
        y[i, char_to_index[targets[i]]] = 1
    return (X, y)
```

Listing B3-75: Turning our fragments and targets into one-hot versions called `X` and `y`.

Now let's build the model. After a little playing around, we chose the simple deep model of Figure B3-91. It's just two `LSTM` layers and a single `Dense` layer. The first `LSTM` has `return_sequences=True`, because it feed another `LSTM`. The second one produces a single output, which will lead us to the letter the network is predicting. To get that letter, we use a `Dense` layer with one neuron per letter, and a softmax output. This will give us the probability of each character being the next one.

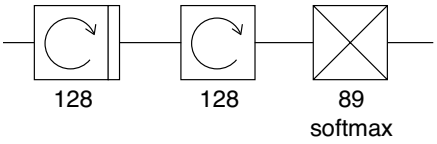


Figure B3-91: Our simple deep network for generating text one letter at a time

Listing B3-76 gives the source for building this model.

```
def build_model(window_length, num_unique_chars):
    # build the model. Two layers of a single LSTM cell with 128
    # elements of memory,then a dense layer with as many outputs
    # as there are characters.
    # We'll train with the RMSprop optimizer. Some experiments suggest that
    # a learning rate of 0.01 is a good place to start.
    model = Sequential()
    model.add(LSTM(128, return_sequences=True,
              input_shape=(window_length, num_unique_chars)))
    model.add(LSTM(128))
    model.add(Dense(num_unique_chars, activation='softmax'))
    optimizer = RMSprop(lr=0.01)
    model.compile(loss='categorical_crossentropy', optimizer=optimizer)
    return model
```

Listing B3-76: Build our little RNN architecture

Now we're ready to generate text. We'll call a new routine called `generate_text()` that will train the model for a single epoch, and then print out some text that it generates. This way we can see how the quality of the text improves over time.

After each call to `fit()` to train the model, we'll pick a random starting point in the original document and extract characters from there. We'll pick as many characters as in the window size we trained on. We'll one-hot encode that sequence of characters and give the result to `predict()`. This will give us back one probability for each unique character in the original text, telling us how likely it is that that character is the one that comes next after the input text.

We could just use the most probable character, but in practice that tends to give us a lot of repeated words. A nice alternative is to juggle around the probabilities a little so that less-likely letters also have a chance of being chosen. A nice algorithm for that adds metaphorical "heat" to the probabilities to change their values [Chollet17c]. We've wrapped that up in a routine called `choose_probability()` that's in the notebook.

Once we've got the prediction for the next character, we append that prediction to a growing output string. Then we append the new character to the end of our input to the model, while also dropping the first character from that string, so the input is always the length of the training windows. Then we train the system for another epoch and do it all again.

The code for `generated_text()` is shown in Listing B3-77. Rather than simply printing strings to the output, we hand them to a routine named `print_string()` that both prints them, and saves them in a file that we've opened.

```
def generate_text(model, X, y, number_of_epochs, temperatures,
                  index_to_char, char_to_index, file_writer):
    # train the model, output generated text after each iteration
    for iteration in range(number_of_epochs):
        print_string('--------------------------------------------------\n',
                     file_writer)
        print_string('Iteration '+str(iteration)+'\n', file_writer)
        history = model.fit(X, y, batch_size=batch_size, epochs=1)
        start_index = random.randint(0, len(text) - window_length - 1)

        for temperature in temperatures:
            print_string('\n----- temperature: '+str(temperature)+'\n',
                         file_writer)
            seed = text[start_index: start_index + window_length]
            generated = seed
            print_string('----- Generating with seed: <'+seed+'>\n',
                         file_writer)

            for i in range(generated_text_length):
                x = np.zeros((1, window_length, len(index_to_char)))
                for t, char in enumerate(seed):
                    x[0, t, char_to_index[char]] = 1.

                preds = model.predict(x, verbose=0)[0]
                next_index = choose_probability(preds, temperature)
                next_char = index_to_char[next_index]

                generated += next_char
                seed = seed[1:] + next_char
```

```
        print_string(generated+'\n\n', file_writer)
        file_writer.flush()
```

Listing B3-77: Generate new text using our trained model.

The majority of the work in this program involves messing about with the data, making the dictionaries and windows and doing the one-hot encoding and so on. The actual neural network code was just a few lines to make the network, and one line each to train it and get predictions.

To train the system, we picked a window length of 40 characters, and a step of 3, so each training string overlapped 37 characters with the one before. We used a batch size of 100, and generated 1000 new characters after each training step, using "temperatures" of 0.5, 1.0, and 1.5. The text with temperature 0.5 tended to produce the same words frequently, and temperature 1.5 produced mostly words, but also lots of strings that weren't words. It's fun to play with the temperature to find the sweet spot where the output is interesting, with the occasional weird almost-word.

As we mentioned in Chapter 19, this can take a long time to run. On a late 2014 iMac, without GPU support, each iteration takes about 1400 seconds, or a little more than 23 minutes. Networks like this often take 800 epochs or so to start producing text that is close to the source. That would be about 13 days of 24/7 crunching. So we ran this network for 100 epochs on Amazon Web Services, watching the loss drop from about 2.6 to 1.1. Here's the start of what it generated after that much training, starting with the seed last time in my life. Certainly a gray m, and with a temperature of 1.0.

> last time in my life. Certainly a gray myself under the great tautoh;
> harm I should be a busy because cameful allo done." "Why dud
> that you dedy hour any one of these chimnes of this pricaption
> is to his, If the tall. Up appeared to very set over with Mr. Trem,
> there, if we confeeliin, I fawny of days if so far"

Clearly we have a long way to go. But remember that this is letter by letter, from a system that has no idea of English or language or any such structure. Given that it started from nothing and had only such a small amount of training, this is pretty great.

An alternative way to generate text that we discussed in Chapter 19 is to focus on sequences of words, rather than letters. This is appealing in many ways, but it's also slower to train. If we have 7000 or 8000 unique words, that's a lot more work to manage than 89 unique characters.

We experimented with this a bit and chose the architecture in Figure B3-92 to generate new text word by word. The full code is provided in the accompanying notebook.

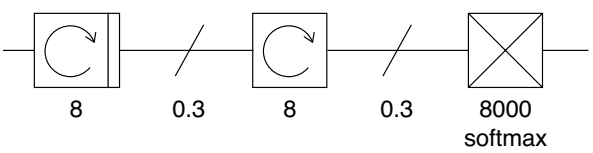


Figure B3-92: A network for word-by-word text generation

We trained with the 8000 most frequently-used words in the text, replacing all others with the marker GLORP. Here's the output after the first epoch, starting with the seed, tell her future husband the whole story and to trust to his generosity . Milverton chuckled . You evidently do not know the Earl , said he . From the baffled look upon Holmes's face , I could. Note that the punctuation marks have been isolated as their own words.

> tell her future husband the whole story and to trust to his GLORP . Milverton chuckled . You evidently do not know the Earl said he . From the baffled look upon Holmes's face , I could each clear at screen At there by put got His you openly is do that were once Your plans from my He greatest life to did mantle it first India , drive as come really It black build my is put hearty Stanley sprang , afraid once quite whom had comes sole snuff Francisco"

Training on an Amazon Web Services GPU-enabled p2.xlarge instance took 15 minutes per epoch. Over the first 10 epochs, the training loss dropped from about 6.8 to about 5.3. But with so many thousands of words to choose from, things didn't get much better.

Here's the output from epoch 10, starting with the seed, it would be a grief to me to be forced to take any extreme measure . You smile , sir , but I assure you that it really would. GLORP is part of my trade , ,

> it would be a grief to me to be forced to take any extreme measure . You smile , sir , but I assure you that it really would . GLORP is part of my trade , ," I door small little who very lamps into dropped imagine the the GLORP , . his that the would nose , tell . Smith said was the The and is . a know to would are none very had there was are It a Mother upon away my for - and the about are not the for to I open one , it far ?

We'd have to spend a lot of time training this model before the results got interesting.

These examples used tiny RNNs. Using larger RNNs, or even better, transformers (as discussed in Chapter 20) could give us better results in less time.

The Functional API

So far in this chapter we've built our models by placing one layer after another. The *Sequential API* that we've been using was designed for just this sort of architecture. Keras offers a second way to build our models, called the *Functional API*.

The reason for a second API is that sometimes we want to build models that are not strictly sequential. For example, in Chapter 18 we build a model called a *variational autoencoder*. It starts with a sequence of layers, but then it splits into two, with two different layers getting input from the same predecessor, as in Figure B3-93. Then we combine those two layers back into one and continue with a sequential model.

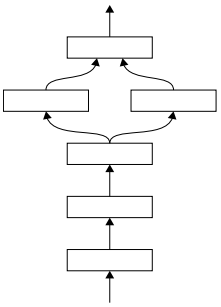


Figure B3-93: This model cannot be made with the Sequential API. We need to use the Functional API to build this.

We can't make a network like Figure B3-93 with the Sequential API, because it assumes that each layer gets input and sends output to no more than one layer.

Using the Functional API, creating layers and connecting them are two separate operations. We can first make whatever layers we need, and then connect them together however we like.

The functional API is powerful, but it can also be complex and subtle. Here we'll stick just the basics that will let us make a model like Figure B3-93. If we just want to make a straight chain of layers, like one the sequential API builds, we can make that with the functional API as well.

The key thing to know about this approach is that each layer is its own object. That is, we create it and assign it to a variable. Once we create a layer, it contains its own weights, parameters, and internal processing. We will then connect that layer to other layers to build the model.

But because each layer is an object, we can use it more than once. Let's look at an imaginary model that we might use for image classification. We've decided that the first two layers will be fully-connected, or `Dense`, layers of 100 and 200 neurons, as in Figure B3-94(a).

We can build both models in Figure B3-94 and use one and then the other. The layers in red are not copied, but shared, so any learning from either network is also used in the other.

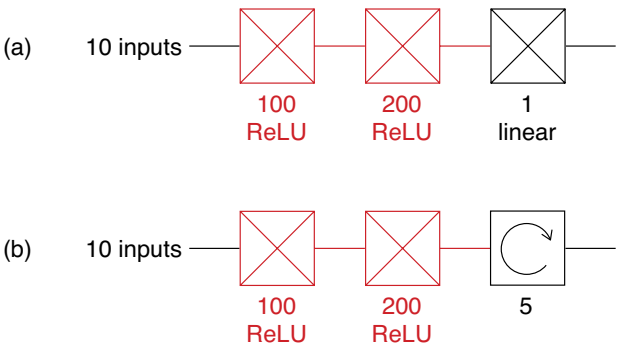


Figure B3-94: Re-using layers in Keras. (a) A small network using two hidden `Dense` layers, shown in red. (b) A different network that re-uses the two red layers from part (a) as the first stages.

Let's suppose that later on we want to build a different model for roughly the same task. But this time we want to put a recurrent layer at the end, as in Figure B3-94(b). We can re-use the first two layers from our first model. These aren't copies, but the same layers, just taking place in this new network. They retain all the weights that they've learned when they were part of the other network. So as we train either model, the other model is trained as well.

It may help to think of the layers, the connections, and the models as three different ideas. We start with a "soup" of layers, all floating around and not connected to anything, as in Figure B3-95(a). Then we decide to build some connections from one layer to the next, as in Figure B3-95(b). That set of connections is a model. Later we might build a second set of connections, making a second model, as in Figure B3-95(c). We're re-using the same layers in each model, just changing where their inputs come from and where the outputs go. When any layer learns, that change will be incorporated into any other model that layer is used in. Any training that happens in any model stays with the layers that learned it, so the other model will benefit from those improved weights.

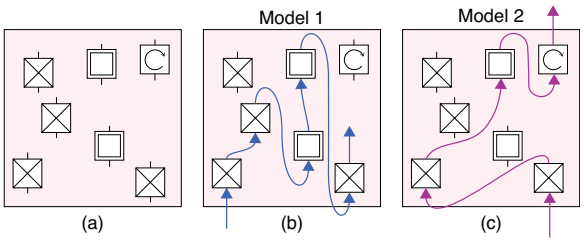


Figure B3-95: In the Functional API, we distinguish the layers from how they're connected. Left: A "soup" of layers. Middle: Connections between layers makes a model. Right: A different set of connections between the layers at the far left that compose another model.

This flexibility in connections and re-use of layers allows us to perform a useful operation called *pre-training*. In this technique, we teach a piece of a network in isolation before we teach the whole thing. The idea is to build a small network with just the layers we want to train, and teach that for a while. This is a useful technique when we anticipate that one piece of the network is going to be take much more time to train than the rest of it. We can teach the difficult part first, in isolation, which will often be much faster than training the whole thing. When those layers have become good at their jobs, we then connect them to the larger model and train up the whole thing.

This idea generalizes even to the level of the model. We can create a model, and then make a second model that includes the first one. We don't even need to explicitly include all the layers. Keras allows us to place one model into another by name, just as if it was a layer.

Input Layers

The Functional API supports all the layers offered by the Sequential API, including those we've seen above. But it requires one additional layer to act as the input layer.

Recall that the input layer is usually implicit, since it's just a place to "park" the incoming data. When we make a sequential model, we tell Keras the size of the input layer with the `input_shape` argument on the first layer, and Keras makes an input layer for us that's the right size to hold one sample of that shape.

In the functional API, it's our job to create that layer explicitly and add it into the model.

The new layer is called an `Input` layer, and it takes one argument called `shape` that tells it the structure of the input. This is identical to how we use `input_shape`, so it's unfortunate that they have different names.

Let's think back to the MNIST data set. To create an input for a piece of flattened MNIST data containing a list of 784 elements, we could write the code in Listing B3-78.

```
input_layer = Input(shape=[784])
```

Listing B3-78: Creating an input layer for 784 elements

A common alternative way to write a one-element list in Python is `(784,)`, where the comma tells the system that this isn't just the number 784 in parentheses, but a list with one element.

Let's suppose our first layer following the `Input` layer was a convolution layer, expecting an input tensor of shape 28 by 28 by 1. Then we could write Listing B3-79.

```
input_layer = Input(shape=[28,28,1])
```

Listing B3-79: Creating an input layer for a tensor of 28 by 28 by 1 elements

Now that we have our input layer, we can move on to making a model.

Making A Functional Model

To build a model in the Functional API, we create each layer as its own object, and then add it to the model by identifying where it gets its input from.

Our first job is to make a layer, which we save in a variable. To use it in a model, we need to specify its inputs. We don't need to specify where the output goes, because the system can figure that out from the other layers. Let's say our current layer is named `layer_1`, and later we create a layer named `layer_2` that says it gets input from `layer_1`. Then it's easy to work out that `layer_2` is one of the outputs of `layer_1`. If we like, multiple layers can take their input from `layer_1`.

Let's imagine a simple model that takes a flattened list of 784 values as input, and returns a single number giving the probability that the image is an MNIST-style digit. We're not categorizing the inputs here, but rather just presenting a single value at the output that tells us if the input is or is not an MNIST digit. Let's propose trying the network of Figure B3-96 for the job. We could easily make this model with the Sequential API, but let's use the Functional API to see how it's done.

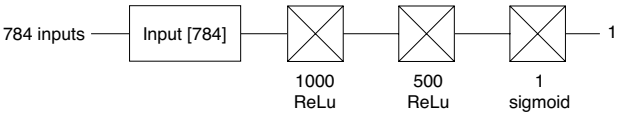


Figure B3-96: A simple network to tell us if an input is an MNIST-style digit image

To create this model, we need an `Input` layer and three `Dense` layers. Listing B3-80 makes the layers and saves each in a variable. We'll call these "unconnected layers," since they're not connected to anything.

```
input = Input(shape=(784,))
dense_1 = Dense(1000, activation='relu')
dense_2 = Dense(500, activation='relu')
output = Dense(1, activation='sigmoid')
```

Listing B3-80: Creating the layers for our network of Figure B3-96

Now we need to connect these layers. We'll create a new object called a "connection layer." Like some of the layers we've seen before, this is really just a wrapper or container. A connection layer points to two objects: an unconnected layer, and another connection layer. This lets us build up a chain of connection layers that define a whole network.

Let's see how this works. On the right of Figure B3-97 we see the four unconnected layers we just made in Listing B3-80. Let's start building connection layers at the bottom, with the input layer. The input layer is a special case that doesn't take input from any other layer. As we said, the connection layer points to two objects. First there's the layer it's referring to, which in this case is the input layer. The other object is the connection layer that provides the input to this layer. Since this layer doesn't take input, we leave that pointer empty. We just built our first connection layer. We'll call this `C1_input`, where `C` refers to this being a connection layer, and the `1` distinguishes it from other such layers we'll make later.

Let's move up now to the first hidden layer, which we called `dense_1`. To build its connection layer, we point its first value at the unconnected layer `dense_1`. Then we point it at the connection layer that provides `dense_1` with input. This is `C1_input`, which we just made. That's it for this connection layer, which we'll call `C1_dense_1`.

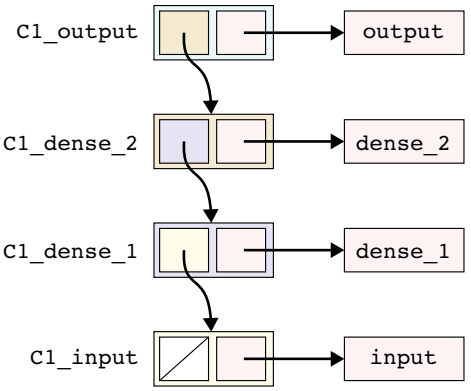


Figure B3-97: Building connection layers. On the right are the original, unconnected layers. On the left are the connection layers. Each connection layer points to an unconnected layer, and to the preceding connection layer. The connection layer for the input layer is a special case.

Moving upwards, we repeat the process for the next two layers.

This chain of layers is everything we need to build a model. When Keras writes the code to route the output of each layer to some other layer, it just follows the chain of connection layers.

The key thing is that the connection layers aren't duplicating the unconnected layers. If we make another connection layer that points to, say, `dense_2`, then we're not modifying `dense_2` in any way.

Listing B3-81 shows the code for Figure B3-97. We make our connection layers implicitly by treating each unconnected layer like a function, and giving it the argument of the connection layer it points to. The syntax can seem a little weird. Keras makes it easy to make the connection layer `C1_input`, which only needs to point to the unconnected `input` layer.

```
# The input layer has no previous connections
C1_input = input
C1_dense_1 = dense_1(C1_input)
C1_dense_2 = dense_2(C1_dense_1)
C1_output = output(C1_dense_2)
```

Listing B3-81: Building our connected layers, combining a layer with its connection to a previous layer. Only `input_layer` does not have an input.

Now we have four new variables, each prefixed with `C1_`. Each one tells us about a layer and where it gets its input from. We can make a model out of these connection layers by calling `Model()` with the input and output connection layers as arguments, as in Listing B3-82.

```
network_1 = Model(C1_input, C1_output)
```

Listing B3-82: Making our model based on the input and output connection layers. The other layers are included implicitly.

Notice that we don't have to specify all the connection layers between the input and the output. Keras can figure out that they're needed because it can follow the chain of connected layers backwards from the output layer to the input layer.

Now that we have our model, we can treat it just like the sequential models we saw above. So we'll call `compile()` to actually build the model and then `fit()` to train it.

Let's say that later we want to play around with this architecture a little. Maybe we'd improve performance by replacing the next-to-final stage of processing in the `dense_2` layer with a convolution layer followed by a flatten layer. We don't want to train from scratch, since our first dense layer and our output layer (also a dense layer) are the same. We'd like to make a second model out of the pieces of the first, but we don't want to disassemble the first model.

This is easy with the Functional API. We start by making a couple of new unconnected layers (our convolution and flatten layers). Now we build up a new set of connection layers, as shown in Figure B3-98. The first two layers, and the last, will re-use the layers from our first model. All of their learned weights come along for the ride.

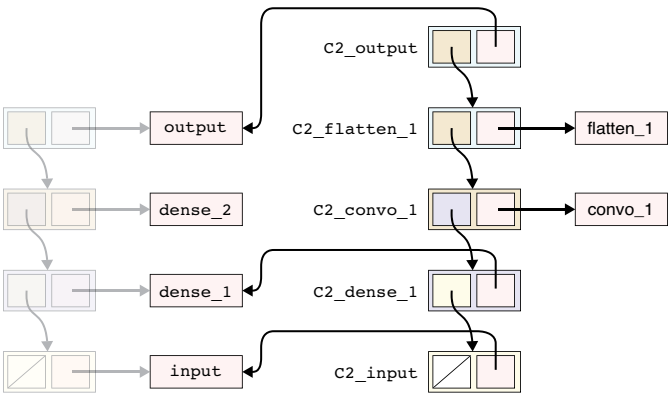


Figure B3-98: Building connections

We can even flip back and forth, training the first model for a little while, then the second, then back to the first.

The code for this new model is shown in Listing B3-83.

```
# define the new layers
convo_1 = Conv2D(32, (5,5))
flotten_1 = Flatten()

# Build the new connection layers
C2_input = input
C2_dense_1 = dense_1(C2_input)
C2_convo_1 = convo_1(C2_dense_1)
C2_flatten_1 = flatten_1(C2_convo_1)
```

```
C2_output = output(C2_flatten_1)

# build the model
model2 = Model(C2_input, C2_output)
```

Listing B3-83: Building our new model using some layers from the first model

Sometimes we don't want to change the shared layers. For example, we might find that `dense_1` changes considerably depending on whether we're training the first or second model. Perhaps we want to keep all the layers in the first model where they are, but just train the two new layers in our second model. In that case we can use the freezing mechanism to prevent any layer from changing. We just set any layer's optional parameter `trainable` to `False` to freeze it, and then set it to `True` if we want to make it trainable again later.

We started this section by considering the difficulty of using the Sequential API to build a branching architecture such as in Figure B3-93. Figure B3-99(a) shows a version of such a model.

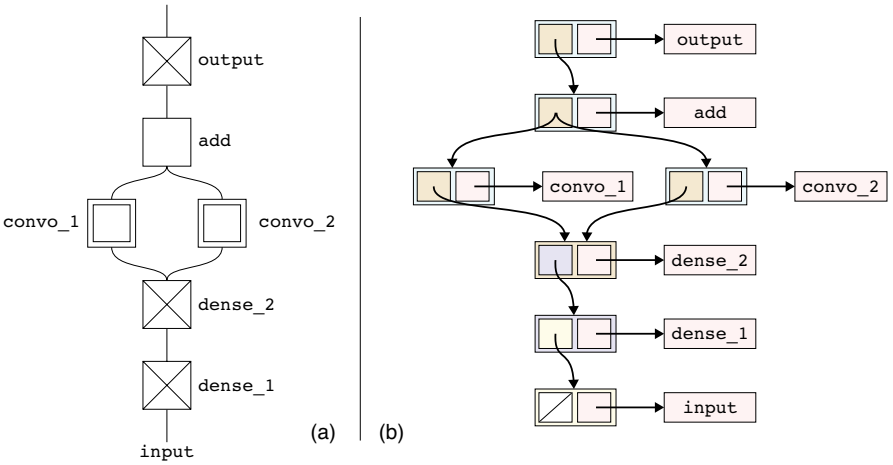


Figure B3-99: A branching architecture using the Functional API. (a) The architecture we'd like to make. (b) A set of connection layers to represent it.

Figure B3-99(b) shows the structure of a set of connection layers that can do the job. Note that the connection layers for both convolution layers get their input from the same connection layer, associated with `dense_2`. This kind of branching doesn't require any special effort when using the Functional API, since we just point our connection layers where we want them to go.

Figure B3-99(b) has a layer called `add` that we haven't discussed. Keras offers us a variety of layers that can combine multiple layers. They're called "Merge Layers," and the Keras documentation lists a half-dozen possibilities for us [Chollet17a]. Each of these layers takes a list of other layers in the model, and combines their outputs. We can also write our own custom

layers if we need something that's not already available. In this case, the layer adds together the output tensors of the two convolution layers. Of course, they must be the same size for this to make sense.

We've seen that the Functional API lets us re-use layers in multiple models, and build models whose connections are not simply a single stack of layers.

Summary

This wraps up our discussion of Keras. There's much more to be found! Guidance for building more complex structures can be found online in various blogs and GitHub repos, as well as the Keras documentation itself.

One of the best ways to learn how to write deep learning code in any language, and for any library, is to look at how other people have done it. The web is filled with blog posts and articles with advice on writing code in Keras, PyTorch, and TensorFlow (as well as other libraries), and there are many, many GitHub repos that contain the complete source code for a tremendous diversity of project applications and complexities.

Any time you're starting a new project, it's often a great idea to look for something that someone has already implemented. You can see how they approached the program, and how they used the library functions to prepare the data, and build and train their learners. It's a great way to learn about useful new routines, and perhaps surprising new ways to use routines you thought you already knew.

In the best case, you can reuse some of their code and save yourself some time and debugging. But even if you can't, you can usually find valuable insight into how to structure your approach to the problem (or, equally valuable, how to *not* structure your approach!).

I often start my projects small, with tiny databases and toy learners. When I've confirmed that the pieces are all working, I can gradually grow a larger system from there, checking my results after each change. Working in a Jupyter notebook makes this really easy, as I can change just one cell at a time and reevaluate the whole notebook, checking that everything still works. If it doesn't, I know exactly where I've introduced the change that broke things. Of course, working in a full development environment such as PyCharm, with breakpoints and debuggers, is also a great way to grow a program. Choose the style (or a different one) that suits you best.

The deep learning software environment is changing and growing rapidly. As you read this, there may be new libraries or frameworks out there that offer tools or conveniences beyond what's provided by the packages I've mentioned here. Don't be afraid of jumping in; you may discover a new favorite way to build your systems!

Deep learning is challenging, rewarding, and fun. It's also dangerous and can deliberately or accidentally cause harm to real people and other living things. There is immense power here. Use it responsibly to make the world a better place.

References

[Cain13]: Cain, Answer to "What Is New Deck Order?" The Magic Cafe, 2013. *https://www.themagiccafe.com/forums/viewtopic.php?topic=501138*

[Chollet17a]: François Chollet, "Keras Documentation," 2017. *https://keras.io/ and https://github.com/fchollet/keras*

[Chollet17b]: François Chollet, "stateful_lstm.py," Keras documentation, 2017. *https://github.com/fchollet/keras/blob/master/examples/stateful_lstm.py*

[Chollet17c]: François Chollet, *Deep Learning with Python*, Manning Publications, 2017.

[Jagtap20]: Rohan Jagtap, "Implementing Custom Data Generators in Keras," 2020. *https://towardsdatascience.com/implementing-custom-data-generators-in-keras-de56f013581c*

[Karpathy15]: Andrej Karpathy, "The Unreasonable Effectiveness of Recurrent Neural Networks," Andrej Karpathy Blog, 2015. *https://karpathy.github.io/2015/05/21/rnn-effectiveness/*

[Karpathy16a]: Andrej Karpathy, "Convolutional Neural Networks (CNNs/ConvNets)," Stanford CS 231n Course Notes, 2016. *http://cs231n.github.io/convolutional-networks/*

[Karpathy16b]: Andrej Karpathy, "Convolutional Neural Networks for Visual Recognition," Stanford CS 231 Course Notes, 2016. *http://cs231n.github.io/neural-networks-2/*

[LeCun13]: Yann LeCun, Corinna Cortes, Christopher J. C. Burges, "The MNIST Database of Handwritten Digits," 2013. *http://yann.lecun.com/exdb/mnist/*

[PythonWiki17]: Python Wiki contributors, "Generators," The Python Wiki, 2017. *https://wiki.python.org/moin/Generators*

[Snoek16a]: Jasper Snoek, "Spearmint," 2016. *https://github.com/JasperSnoek/spearmint*

[Snoek16b]: Jasper Snoek, "Spearmint" (updated), 2016. *https://github.com/HIPS/Spearmint*

[Springenberg15]: Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, Martin Riedmiller, "Striving for Simplicity: The All Convolutional Net," ICLR 2015, 2015. *https://arxiv.org/abs/1412.6806*

[Srivastava14]: Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research*, Number 15, pp. 1929–1958, 2014. *http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf*

Image Credits

Figure B3-41, Eurasian eagle owl, *https://pixabay.com/en/eurasian-eagle-owl-bird-wildlife-1627541*