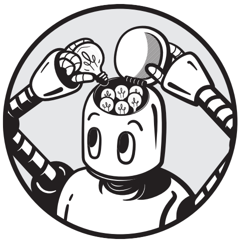
B2

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.

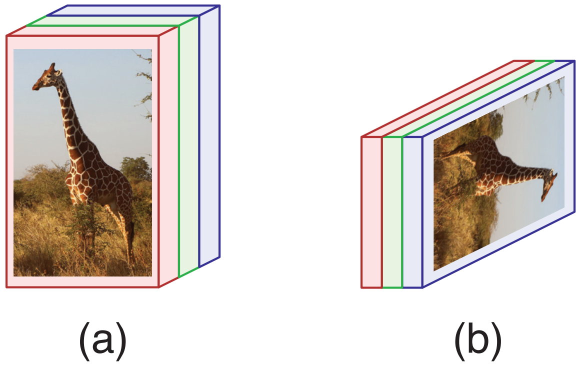


Figure B2-1: Two ways to stack images of size 100 wide by 200 high. We read the sizes of the blocks by the number of layers going away, then down, then right. (a) Stacking images from front to back. This block has dimensions 3 by 200 by 100. This is the `channels_first` organization. (b) Stacking images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

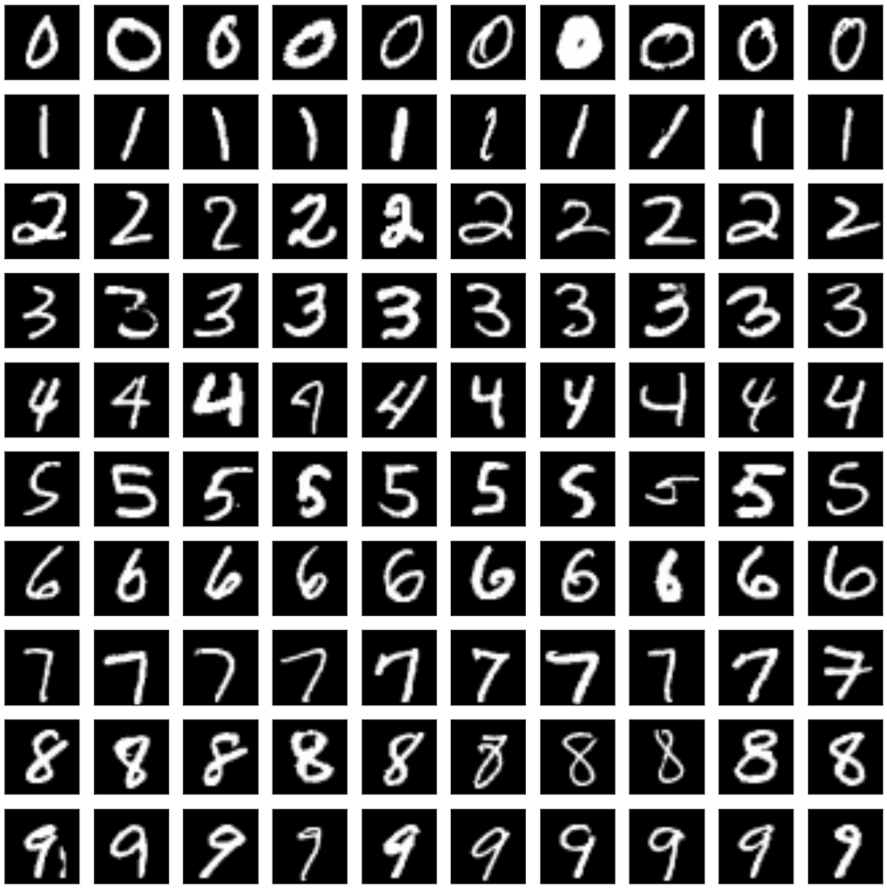


Figure B2-2: A random selection of images in the MNIST training set, organized by label

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.

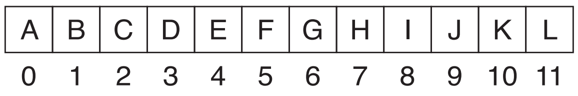


Figure B2-3: We have 12 items arranged in a one-dimensional list. Each element in the list is made up of just a single letter. Each element requires only a single index from 0 to 11 to identify it.

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.

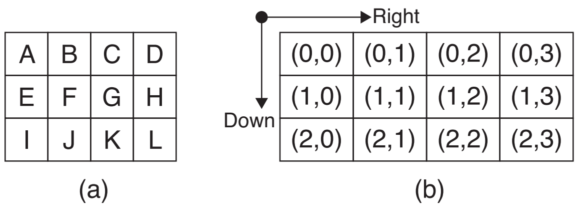


Figure B2-4: Our one-dimensional list of Figure B2-3 re-shaped into a 2D list of 3 rows and 4 columns

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.

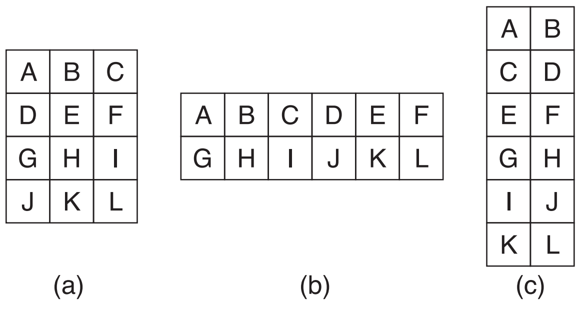


Figure B2-5: Three more ways to arrange our 12 items into a 2D list. From left to right, these grids have dimensions 4 by 3, 2 by 6, and 6 by 2.

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.

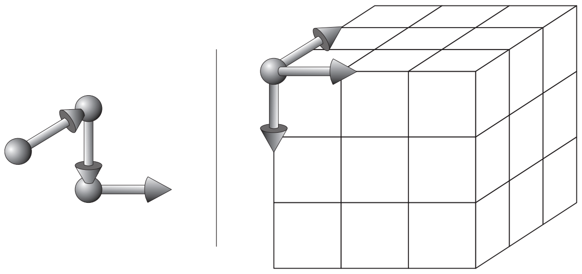


Figure B2-6: Our convention for identifying cells in a 3D block will be to start in the near, upper-left corner. We name cells in the order (away, down, right). (a) The three directions in sequence. (b) Finding a cell in a 3D volume. The first index tells us how far to move away, the second index how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

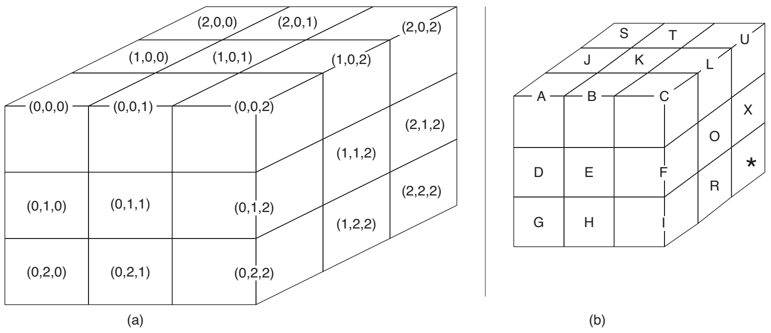


Figure B2-7: Identifying each cell in a 3 by 3 by 3 cube. Each cell requires 3 numbers to identify it. (a) Using our convention of Figure B2-6 we count away, down, and right. The rightmost index changes the fastest, then the middle index, and finally the left-most index. (b) Filling in the letters A-Z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

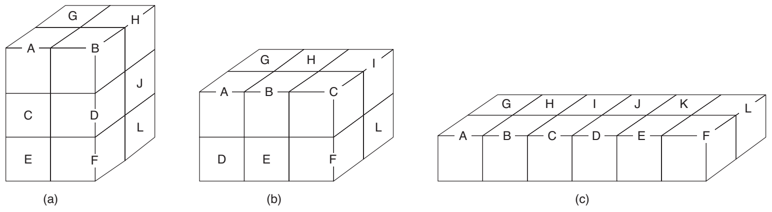


Figure B2-8: Three ways to organize our 12 elements into 3D blocks. From left to right, these have dimensions 2 by 3 by 2, 2 by 2 by 3, and 2 by 1 by 6.

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.

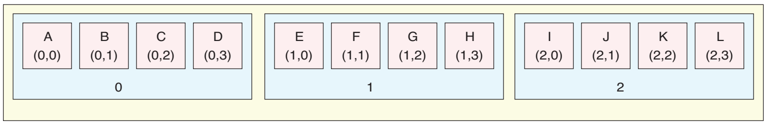


Figure B2-9: The 2D grid of Figure B2-4 has three rows of four elements each. We can show this as a hierarchy of 1D lists. Each set of four elements (that is, a row) is in a list. To identify any element, we first choose the list we want (the row), and then the element we want from that list (the column).

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.

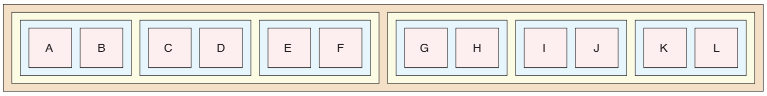


Figure B2-10: The 12 elements of our list arranged in the 3D block of size 2 by 3 by 2, as in Figure B2-8. To identify any cell, we need three numbers, corresponding to the indices in each of the nested lists.

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.

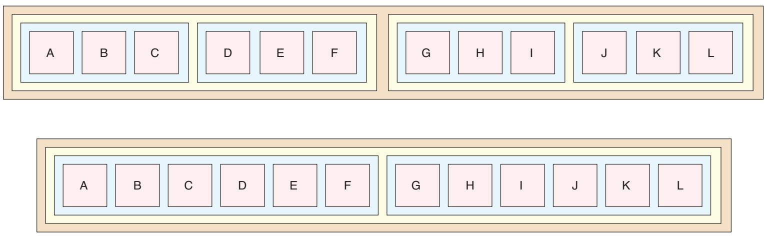


Figure B2-11: Interpreting the middle and right blocks of Figure B2-8 as lists-of-lists. Top: The block is 2 by 2 by 3. Bottom: The block is 2 by 1 by 6.

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.

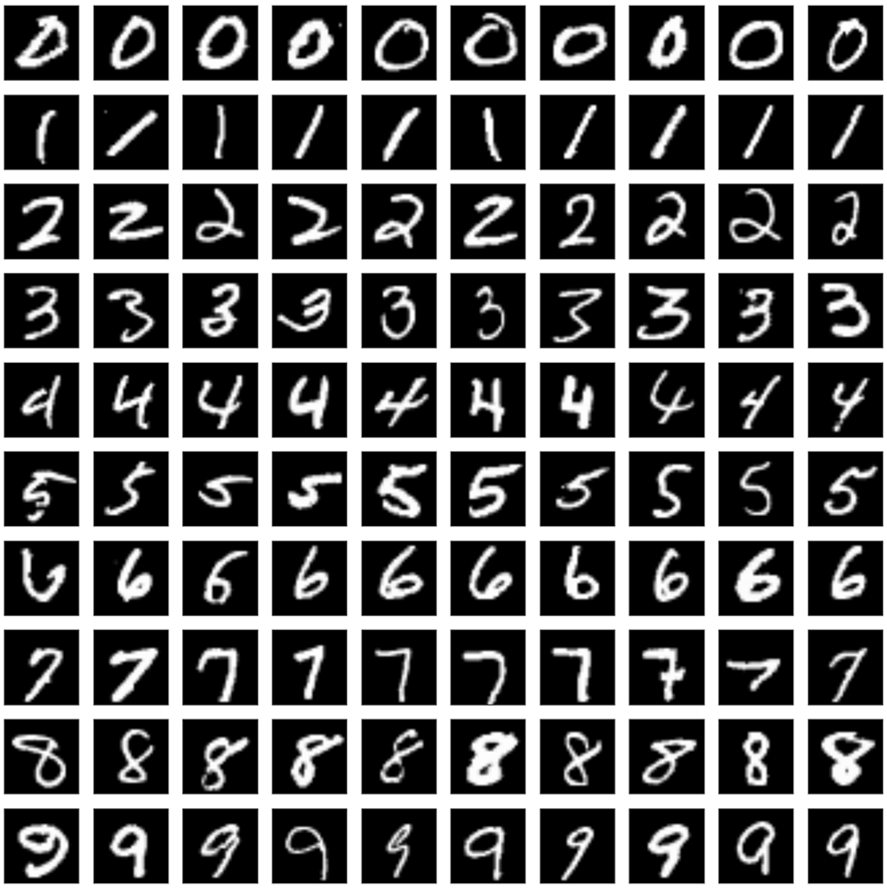


Figure B2-12: A random sampling of the images from the MNIST training set

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.

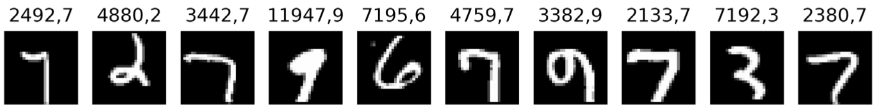


Figure B2-13: Some images from the MNIST training set that demonstrate a bleeding of the image very near, or right up to, the border. The numbers above each example shows its index in the training set, followed by its assigned label.

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

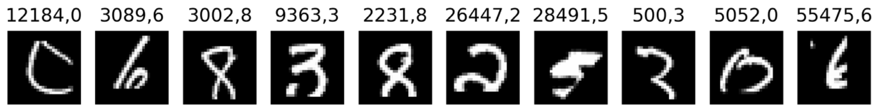


Figure B2-14: Some images from the MNIST training set that have been cropped, chopping away some of what seems very likely to have been drawn, and sometimes creating multiple, disconnected pieces.

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.

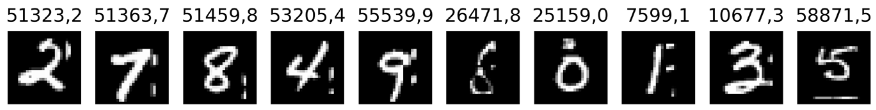


Figure B2-15: Some images from the MNIST training set that demonstrate noise artifacts. Some of these might be due to thresholding or cropping errors.

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.

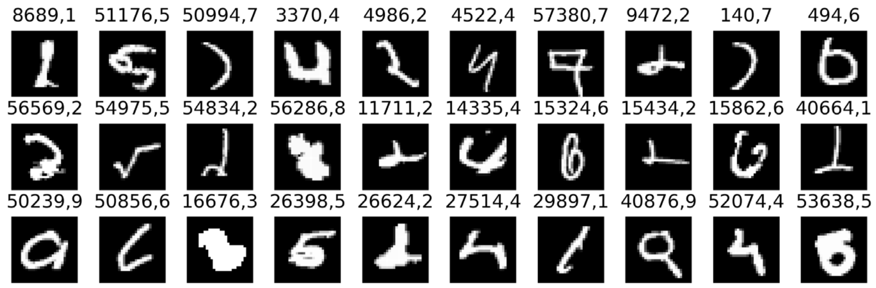


Figure B2-16: Some images from the MNIST training set that appear particularly challenging to categorize

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.

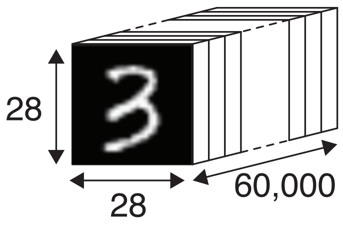


Figure B2-17: Our training data, X_train, has shape 60000 by 28 by 28. That means it's a stack of 60,000 objects, each an image that's 28 by 28 pixels.

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.

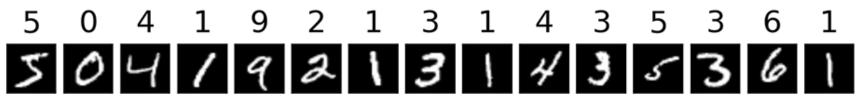


Figure B2-18: first 15 images in X_train. These match the labels in y_train, shown above each sample, so we're good.

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.

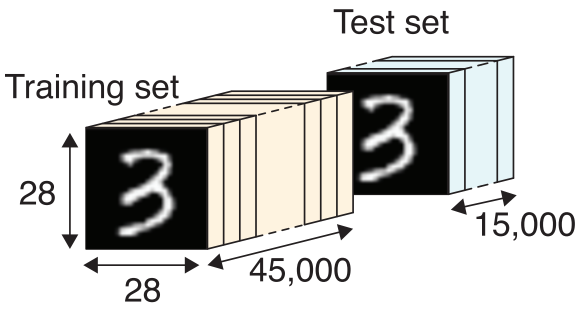


Figure B2-19: Splitting a dataset of 60,000 images. We're using 25% of the data for the test set and the other 75% for the training set.

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.

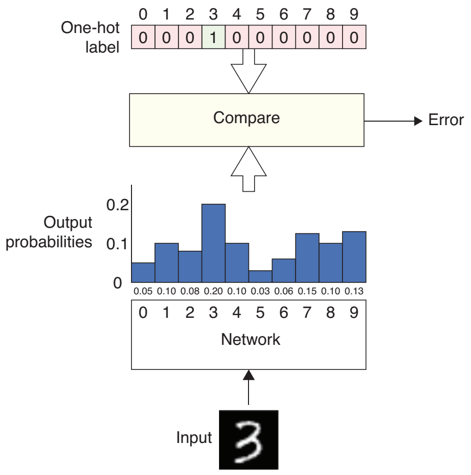


Figure B2-20: Computing the error. We feed an image (here a picture of a 3) to our network, and we get back a probability from 0 to 1 for each possible label from 0 to 9. We compare these 10 numbers with the 10 values in the one-hot label. The more the prediction is like the label, the smaller the error.

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_categorical()` 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.

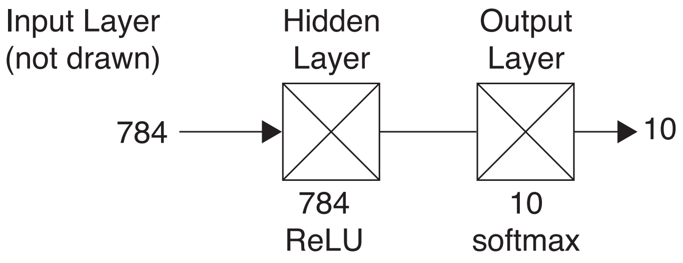


Figure B2-21: Our first, very simple deep-learning model consists of a fully-connected layer of 784 neurons (one for each input pixel) followed by a fully-connected layer of 10 neurons (one for each output class).

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 × 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.

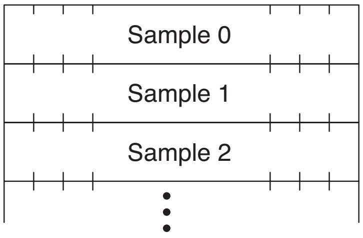


Figure B2-22: Turning our 3D input into a 2D grid, containing one long row of pixels for each image.

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 timesaver 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.

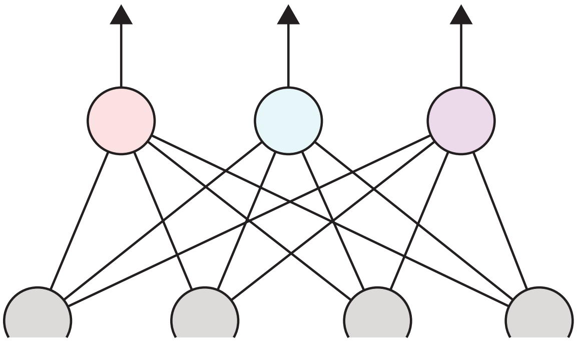


Figure B2-23: A schematic view of a `Dense` layer. The three colored neurons make up the dense layer. Each of them connects to every neuron in the preceding layer (in gray).

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.

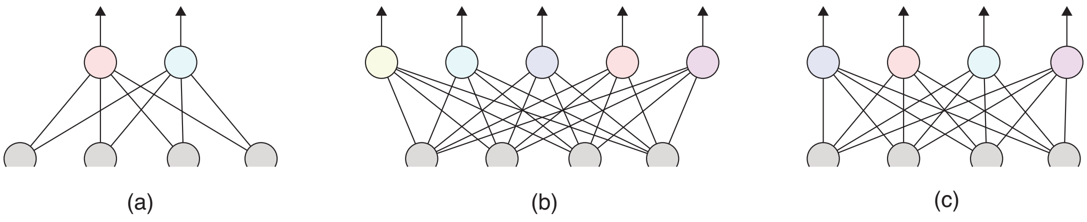


Figure B2-24: Our fully-connected layer is shown with colored neurons, connecting to a previous layer with gray neurons. The number of neurons in the fully-connected layer is independent of the number of neurons in the layer that precedes it.

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 × 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 × 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.

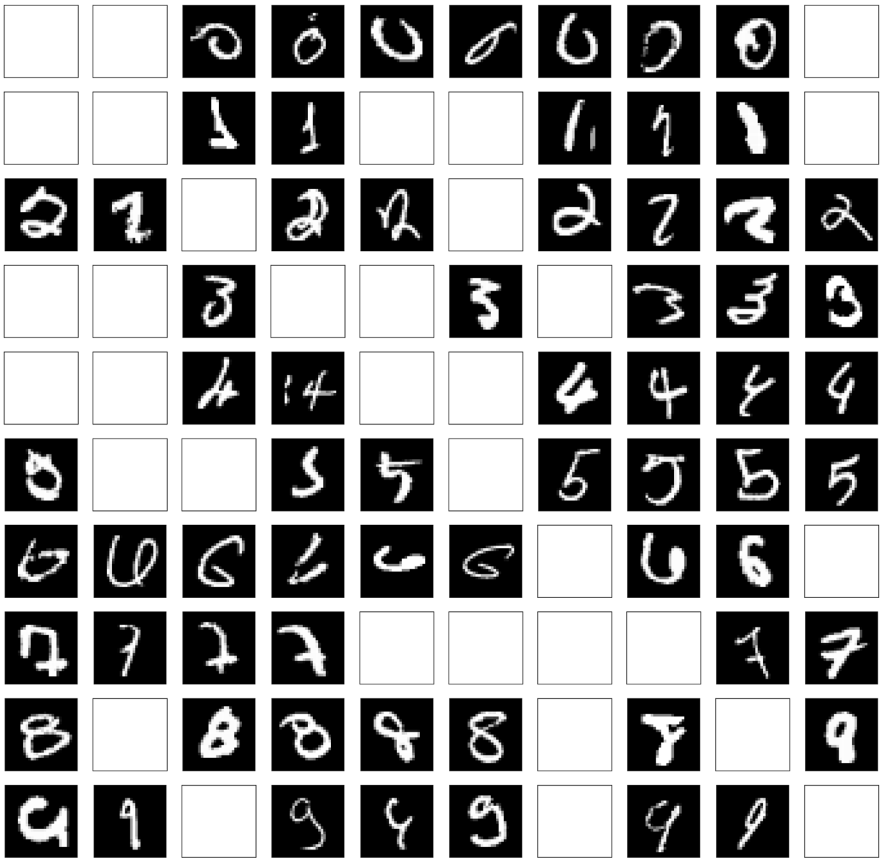


Figure B2-25: A visualization of the test-set images whose predicted value did not match their label

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.

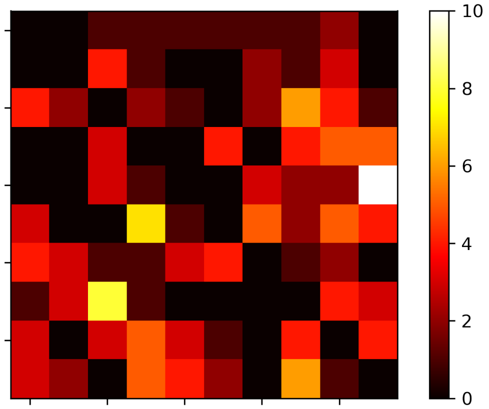


Figure B2-26: A heat map of the population of the error visualization of Figure B2-25, telling us how many images landed in each of the cells. Black indicates an empty list, while brighter reds, then yellows, and finally white represent longer lists.

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.

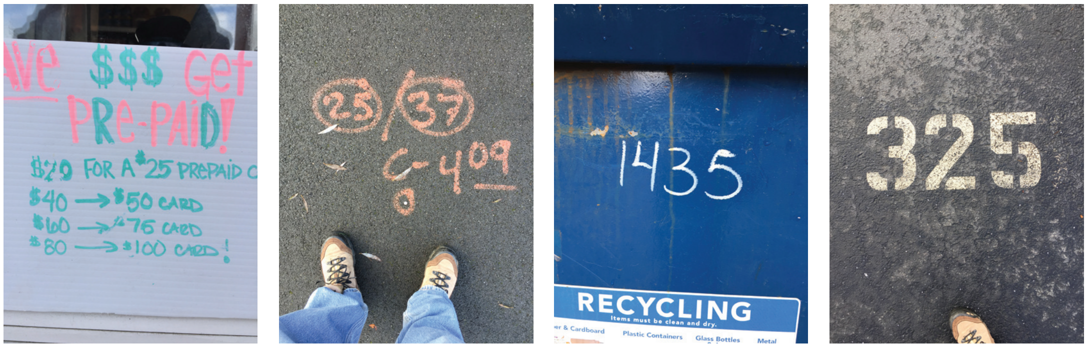


Figure B2-27: Four photos from the Seattle area. Left to right: a sign in a coffee shop window, spray-painted marks on the ground near a construction site, a building number on the side of a dumpster, and a 3 digit parking stall number.

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.

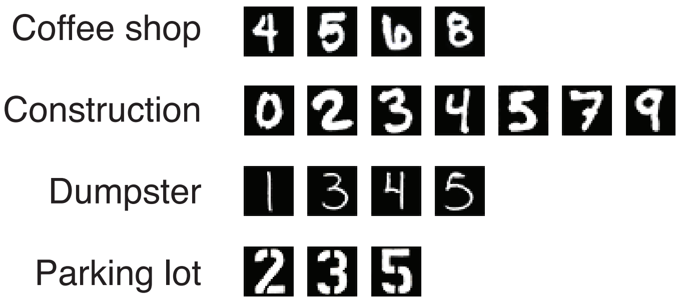


Figure B2-28: Extraction of the digits from Figure B2-27. Each digit has been rotated to be upright and then processed like the original MNIST images.

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.

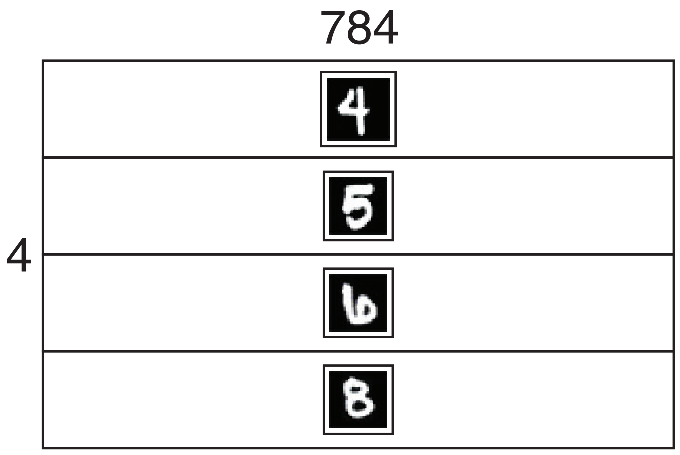


Figure B2-29: Arranging the four images of our coffee shop data into a 2D grid, one row per image.

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.

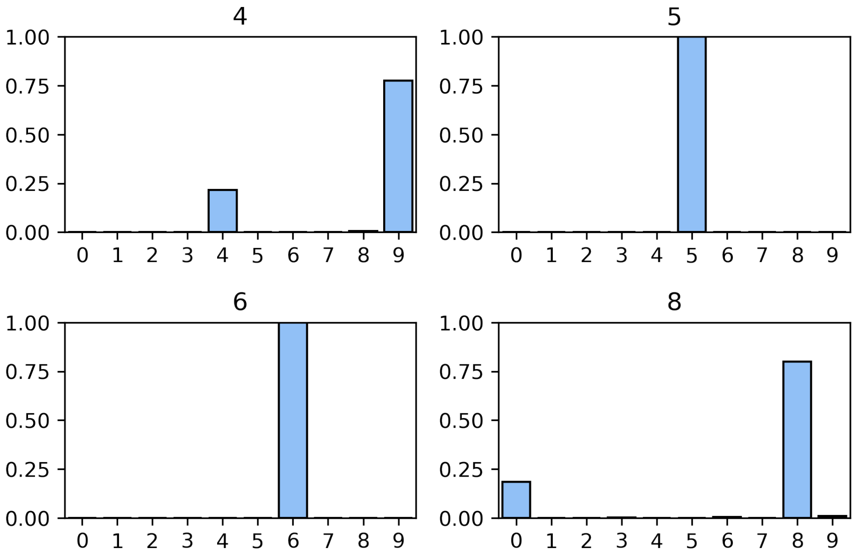


Figure B2-30: Plots of the probabilities from Listing B2-2. The system was pretty sure the first digit was a 9, but thought it might also be a 4. It was very sure of the 5 and 6, but a little less confident about the 8.

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.

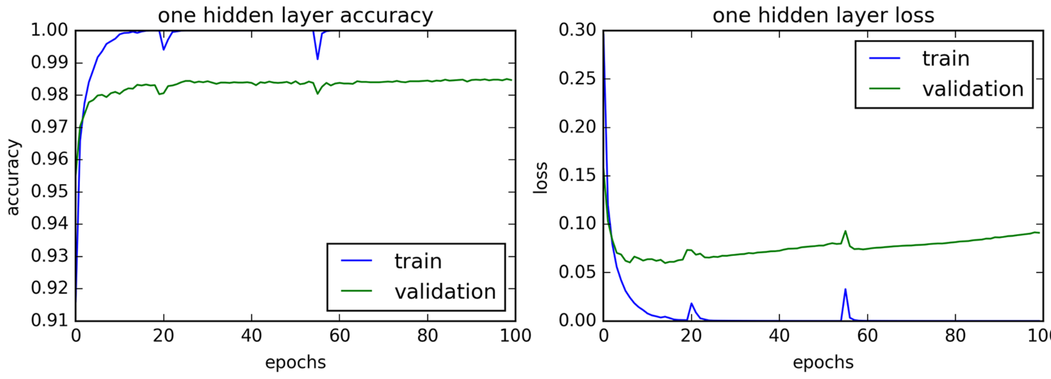


Figure B2-31: The accuracy and loss of our one-layer network plotted against the number of epochs

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.

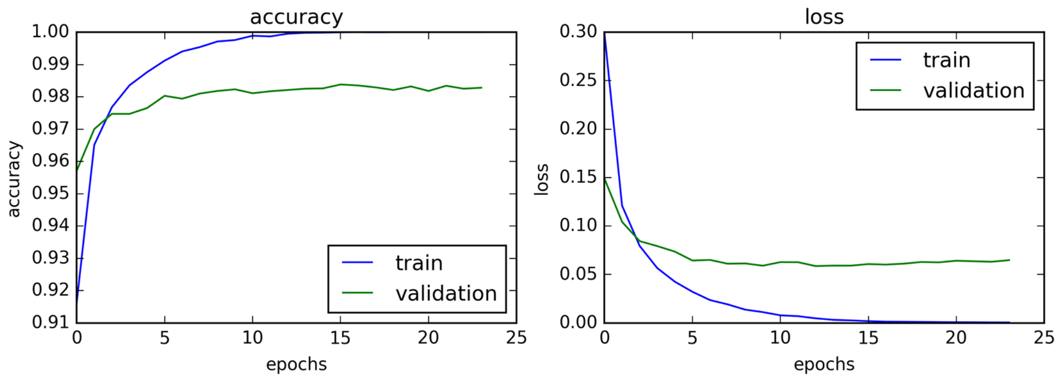


Figure B2-32: Accuracy and loss for our early-stopping run. Note that the validation loss settles down at about epoch 13. The early stopping routine we set up waits another 10 epochs for improvement, and then halts training at epoch 23.

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*