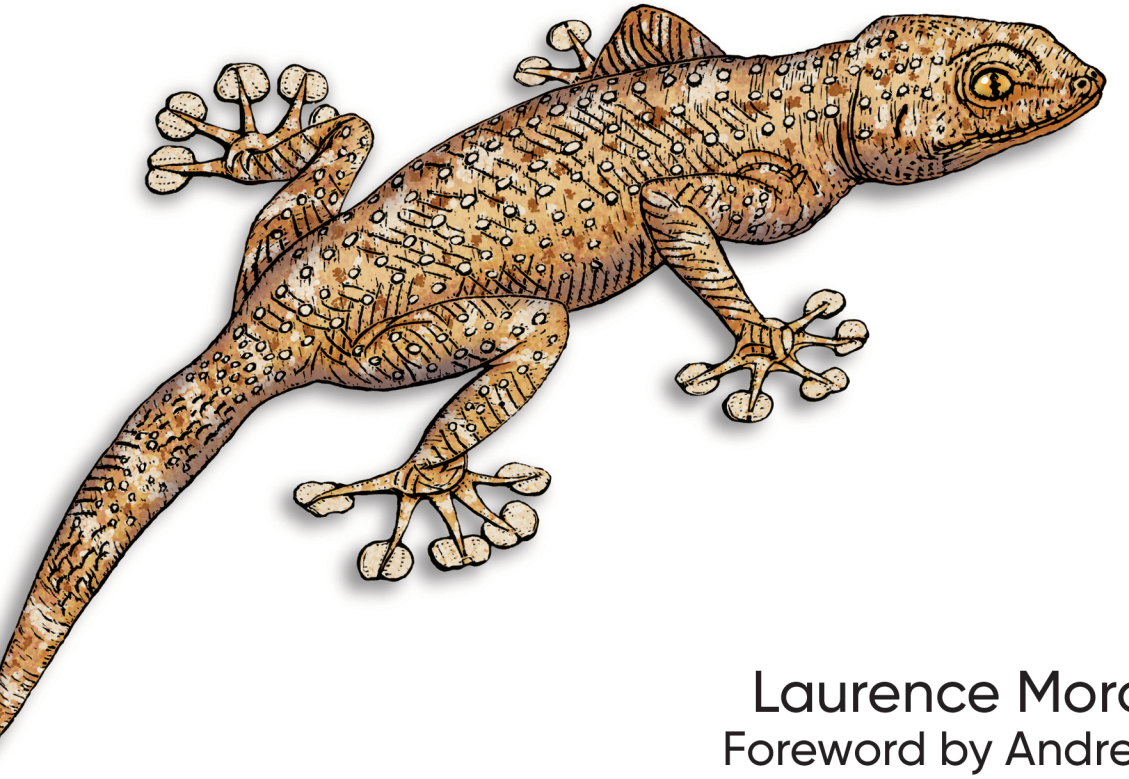
O'REILLY®

AI and Machine Learning for Coders

A Programmer's Guide to Artificial Intelligence



Laurence Moroney
Foreword by Andrew Ng

O'REILLY®

AI and Machine Learning for Coders

If you're looking to make a career move from programmer to AI specialist, this is the ideal place to start. Based on Laurence Moroney's extremely successful AI courses, this introductory book provides a hands-on, code-first approach to help you build confidence while you learn key topics. All you need is experience with Python and its notation for data and array processing.

You'll learn how to implement the most common scenarios in machine learning, including computer vision, natural language processing (NLP), and sequence modeling for web, mobile, cloud, and embedded runtimes. Most books on machine learning begin with a daunting amount of advanced math. This guide provides practical lessons that let you work directly with the code.

- Understand machine learning basics by working with code samples
- Use TensorFlow to build models for a variety of scenarios
- Build a model with a neural network containing only one neuron
- Implement computer vision, including feature detection in images
- Tokenize and sequence words and sentences with NLP
- Embed your models in Android and iOS devices
- Serve models over the web and in the cloud with TensorFlow Serving

Laurence Moroney leads AI advocacy at Google, teaching software developers how to build AI systems with machine learning. He's a frequent contributor to the TensorFlow YouTube channel, a recognized global keynote speaker, and a prolific author.

"The book is a great introduction to understand and practice machine learning and artificial intelligence models by using TensorFlow.

—**Jialin Huang PhD**
Data and Applied Scientist, Microsoft

"Laurence Moroney has been a major force in building TensorFlow into one of the world's leading AI frameworks. I was privileged to support his teaching TensorFlow with deeplearning.ai and Coursera. I wish you the best in your journey learning TensorFlow. With Laurence as a teacher, great adventures await you."

—**Andrew Ng**
Founder, deeplearning.ai

AI

US $59.99 CAN $79.99

ISBN: 978-1-492-07819-7

Twitter: @oreillymedia
facebook.com/oreilly

AI and Machine Learning for Coders

A Programmer's Guide to Artificial Intelligence

Laurence Moroney

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

AI and Machine Learning for Coders
by Laurence Moroney

Printed in the United States of America.

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

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

Acquisitions Editor: Rebecca Novack
Development Editor: Angela Rufino
Production Editor: Katherine Tozer
Copyeditor: Rachel Head
Proofreader: Piper Editorial, LLC

Indexer: Judith McConville
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: O'Reilly Media, Inc.

October 2020: First Edition

Revision History for the First Edition
2020-10-01: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492078197* for release details.

978-1-492-07819-7

[LSI]

CHAPTER 2

Introduction to Computer Vision

The previous chapter introduced the basics of how machine learning works. You saw how to get started with programming using neural networks to match data to labels, and from there how to infer the rules that can be used to distinguish items. A logical next step is to apply these concepts to computer vision, where we will have a model learn how to recognize content in pictures so it can "see" what's in them. In this chapter you'll work with a popular dataset of clothing items and build a model that can differentiate between them, thus "seeing" the difference between different types of clothing.

Recognizing Clothing Items

For our first example, let's consider what it takes to recognize items of clothing in an image. Consider, for example, the items in Figure 2-1.

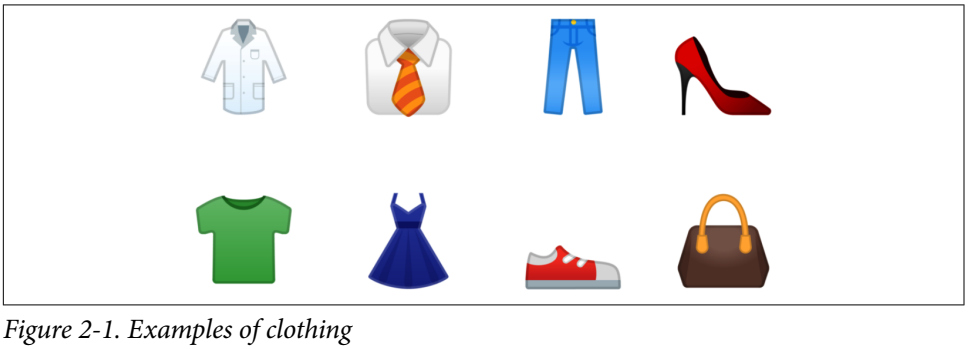


Figure 2-1. Examples of clothing

There are a number of different clothing items here, and you can recognize them. You understand what is a shirt, or a coat, or a dress. But how would you explain this to somebody who has never seen clothing? How about a shoe? There are two shoes in

this image, but how would you describe that to somebody? This is another area where the rules-based programming we spoke about in Chapter 1 can fall down. Sometimes it's just infeasible to describe something with rules.

Of course, computer vision is no exception. But consider how you learned to recognize all these items—by seeing lots of different examples, and gaining experience with how they're used. Can we do the same with a computer? The answer is yes, but with limitations. Let's take a look at a first example of how to teach a computer to recognize items of clothing, using a well-known dataset called Fashion MNIST.

The Data: Fashion MNIST

One of the foundational datasets for learning and benchmarking algorithms is the Modified National Institute of Standards and Technology (MNIST) database, by Yann LeCun, Corinna Cortes, and Christopher Burges. This dataset is comprised of images of 70,000 handwritten digits from 0 to 9. The images are 28 × 28 grayscale.

Fashion MNIST is designed to be a drop-in replacement for MNIST that has the same number of records, the same image dimensions, and the same number of classes —so, instead of images of the digits 0 through 9, Fashion MNIST contains images of 10 different types of clothing. You can see an example of the contents of the dataset in Figure 2-2. Here, three lines are dedicated to each clothing item type.

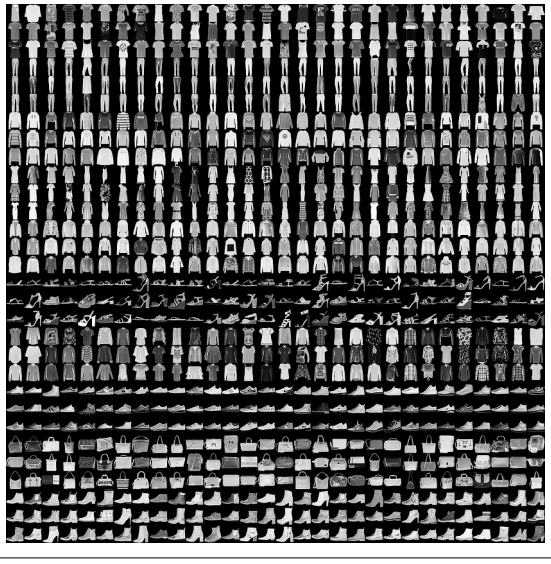


Figure 2-2. Exploring the Fashion MNIST dataset

It has a nice variety of clothing, including shirts, trousers, dresses, and lots of types of shoes. As you may notice, it's monochrome, so each picture consists of a certain number of pixels with values between 0 and 255. This makes the dataset simpler to manage.

You can see a closeup of a particular image from the dataset in Figure 2-3.

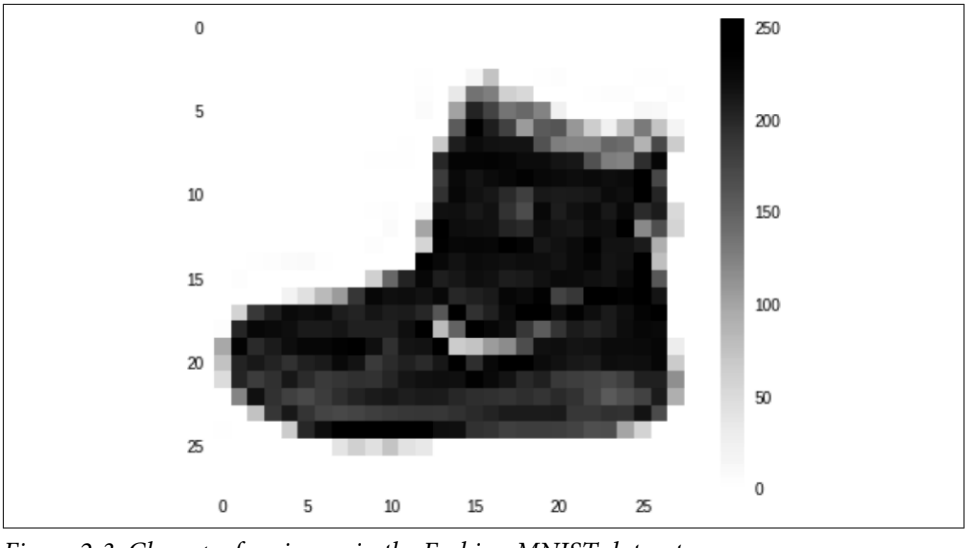


Figure 2-3. Closeup of an image in the Fashion MNIST dataset

Like any image, it's a rectangular grid of pixels. In this case the grid size is 28 × 28, and each pixel is simply a value between 0 and 255, as mentioned previously. Let's now take a look at how you can use these pixel values with the functions we saw previously.

Neurons for Vision

In Chapter 1, you saw a very simple scenario where a machine was given a set of X and Y values, and it learned that the relationship between these was Y = 2X – 1. This was done using a very simple neural network with one layer and one neuron.

If you were to draw that visually, it might look like Figure 2-4.

Each of our images is a set of 784 values (28 × 28) between 0 and 255. They can be our X. We know that we have 10 different types of images in our dataset, so let's consider them to be our Y. Now we want to learn what the function looks like where Y is a function of X.

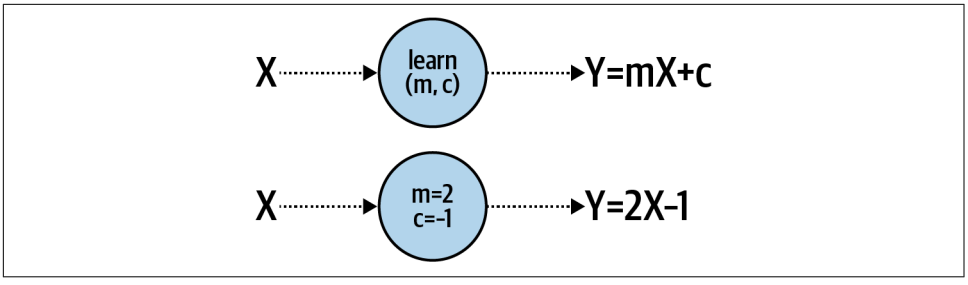


Figure 2-4. A single neuron learning a linear relationship

Given that we have 784 X values per image, and our Y is going to be between 0 and 9, it's pretty clear that we cannot do Y = mX + c as we did earlier.

But what we *can* do is have several neurons working together. Each of these will learn *parameters*, and when we have a combined function of all of these parameters working together, we can see if we can match that pattern to our desired answer (Figure 2-5).

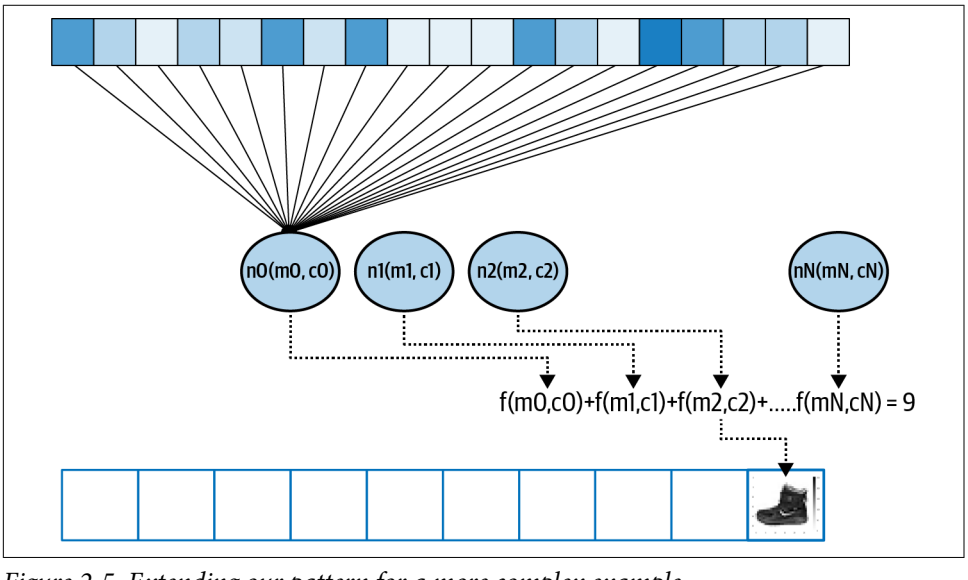


Figure 2-5. Extending our pattern for a more complex example

The boxes at the top of this diagram can be considered the pixels in the image, or our X values. When we train the neural network we load these into a layer of neurons—Figure 2-5 shows them just being loaded into the first neuron, but the values are loaded into each of them. Consider each neuron's weight and bias (m and c) to be randomly initialized. Then, when we sum up the values of the output of each neuron we're going to get a value. This will be done for *every* neuron in the output layer, so

neuron 0 will contain the value of the probability that the pixels add up to label 0, neuron 1 for label 1, etc.

Over time, we want to match that value to the desired output—which for this image we can see is the number 9, the label for the ankle boot shown in Figure 2-3. So, in other words, this neuron should have the largest value of all of the output neurons.

Given that there are 10 labels, a random initialization should get the right answer about 10% of the time. From that, the loss function and optimizer can do their job epoch by epoch to tweak the internal parameters of each neuron to improve that 10%. And thus, over time, the computer will learn to "see" what makes a shoe a shoe or a dress a dress.

Designing the Neural Network

Let's now explore what this looks like in code. First, we'll look at the design of the neural network shown in Figure 2-5:

```
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(10, activation=tf.nn.softmax)
])
```

If you remember, in Chapter 1 we had a `Sequential` model to specify that we had many layers. It only had one layer, but in this case, we have multiple layers.

The first, `Flatten`, isn't a layer of neurons, but an input layer specification. Our inputs are 28 × 28 images, but we want them to be treated as a series of numeric values, like the gray boxes at the top of Figure 2-5. `Flatten` takes that "square" value (a 2D array) and turns it into a line (a 1D array).

The next one, `Dense`, is a layer of neurons, and we're specifying that we want 128 of them. This is the middle layer shown in Figure 2-5. You'll often hear such layers described as *hidden layers*. Layers that are between the inputs and the outputs aren't seen by a caller, so the term "hidden" is used to describe them. We're asking for 128 neurons to have their internal parameters randomly initialized. Often the question I'll get asked at this point is "Why 128?" This is entirely arbitrary—there's no fixed rule for the number of neurons to use. As you design the layers you want to pick the appropriate number of values to enable your model to actually learn. More neurons means it will run more slowly, as it has to learn more parameters. More neurons could also lead to a network that is great at recognizing the training data, but not so good at recognizing data that it hasn't previously seen (this is known as *overfitting*, and we'll discuss it later in this chapter). On the other hand, fewer neurons means that the model might not have sufficient parameters to learn.

It takes some experimentation over time to pick the right values. This process is typically called *hyperparameter tuning*. In machine learning, a hyperparameter is a value that is used to control the training, as opposed to the internal values of the neurons that get trained/learned, which are referred to as parameters.

You might notice that there's also an *activation function* specified in that layer. The activation function is code that will execute on each neuron in the layer. TensorFlow supports a number of them, but a very common one in middle layers is `relu`, which stands for *rectified linear unit*. It's a simple function that just returns a value if it's greater than 0. In this case, we don't want negative values being passed to the next layer to potentially impact the summing function, so instead of writing a lot of `if-then` code, we can simply activate the layer with `relu`.

Finally, there's another `Dense` layer, which is the output layer. This has 10 neurons, because we have 10 classes. Each of these neurons will end up with a probability that the input pixels match that class, so our job is to determine which one has the highest value. We could loop through them to pick that value, but the `softmax` activation function does that for us.

So now when we train our neural network, the goal is that we can feed in a 28 × 28-pixel array and the neurons in the middle layer will have weights and biases (m and c values) that when combined will match those pixels to one of the 10 output values.

The Complete Code

Now that we've explored the architecture of the neural network, let's look at the complete code for training one with the Fashion MNIST data:

```
import tensorflow as tf
data = tf.keras.datasets.fashion_mnist

(training_images, training_labels), (test_images, test_labels) = data.load_data()

training_images  = training_images / 255.0
test_images = test_images / 255.0

model = tf.keras.models.Sequential([
            tf.keras.layers.Flatten(input_shape=(28, 28)),
            tf.keras.layers.Dense(128, activation=tf.nn.relu),
            tf.keras.layers.Dense(10, activation=tf.nn.softmax)
        ])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(training_images, training_labels, epochs=5)
```

Let's walk through this piece by piece. First is a handy shortcut for accessing the data:

```
data = tf.keras.datasets.fashion_mnist
```

Keras has a number of built-in datasets that you can access with a single line of code like this. In this case you don't have to handle downloading the 70,000 images—splitting them into training and test sets, and so on—all it takes is one line of code. This methodology has been improved upon using an API called TensorFlow Datasets, but for the purposes of these early chapters, to reduce the number of new concepts you need to learn, we'll just use `tf.keras.datasets`.

We can call its `load_data` method to return our training and test sets like this:

```
(training_images, training_labels),
(test_images, test_labels) = data.load_data()
```

Fashion MNIST is designed to have 60,000 training images and 10,000 test images. So, the return from `data.load_data` will give you an array of 60,000 28 × 28-pixel arrays called `training_images`, and an array of 60,000 values (0–9) called `training_labels`. Similarly, the `test_images` array will contain 10,000 28 × 28-pixel arrays, and the `test_labels` array will contain 10,000 values between 0 and 9.

Our job will be to fit the training images to the training labels in a similar manner to how we fit Y to X in Chapter 1.

We'll hold back the test images and test labels so that the network does not see them while training. These can be used to test the efficacy of the network with hitherto unseen data.

The next lines of code might look a little unusual:

```
training_images  = training_images / 255.0
test_images = test_images / 255.0
```

Python allows you to do an operation across the entire array with this notation. Recall that all of the pixels in our images are grayscale, with values between 0 and 255. Dividing by 255 thus ensures that every pixel is represented by a number between 0 and 1 instead. This process is called *normalizing* the image.

The math for why normalized data is better for training neural networks is beyond the scope of this book, but bear in mind when training a neural network in TensorFlow that normalization will improve performance. Often your network will not learn and will have massive errors when dealing with non normalized data. The Y = 2X – 1 example from Chapter 1 didn't require the data to be normalized because it was very simple, but for fun try training it with different values of X and Y where X is much larger and you'll see it quickly fail!

Next we define the neural network that makes up our model, as discussed earlier:

```
model = tf.keras.models.Sequential([
            tf.keras.layers.Flatten(input_shape=(28, 28)),
            tf.keras.layers.Dense(128, activation=tf.nn.relu),
            tf.keras.layers.Dense(10, activation=tf.nn.softmax)
        ])
```

When we compile our model we specify the loss function and the optimizer as before:

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

The loss function in this case is called *sparse categorical cross entropy*, and it's one of the arsenal of loss functions that are built into TensorFlow. Again, choosing which loss function to use is an art in itself, and over time you'll learn which ones are best to use in which scenarios. One major difference between this model and the one we created in Chapter 1 is that instead of us trying to predict a single number, here we're picking a *category*. Our item of clothing will belong to 1 of 10 categories of clothing, and thus using a *categorical* loss function is the way to go. Sparse categorical cross entropy is a good choice.

The same applies to choosing an optimizer. The `adam` optimizer is an evolution of the stochastic gradient descent (`sgd`) optimizer we used in Chapter 1 that has been shown to be faster and more efficient. As we're handling 60,000 training images, any performance improvement we can get will be helpful, so that one is chosen here.

You might notice that a new line specifying the metrics we want to report is also present in this code. Here, we want to report back on the accuracy of the network as we're training. The simple example in Chapter 1 just reported on the loss, and we interpreted that the network was learning by looking at how the loss was reduced. In this case, it's more useful to us to see how the network is learning by looking at the accuracy—where it will return how often it correctly matched the input pixels to the output label.

Next, we'll train the network by fitting the training images to the training labels over five epochs:

```
model.fit(training_images, training_labels, epochs=5)
```

Finally, we can do something new—evaluate the model, using a single line of code. We have a set of 10,000 images and labels for testing, and we can pass them to the trained model to have it predict what it thinks each image is, compare that to its actual label, and sum up the results:

```
model.evaluate(test_images, test_labels)
```

Training the Neural Network

Execute the code, and you'll see the network train epoch by epoch. After running the training, you'll see something at the end that looks like this:

```
58016/60000 [=====>.] - ETA: 0s - loss: 0.2941 - accuracy: 0.8907
59552/60000 [=====>.] - ETA: 0s - loss: 0.2943 - accuracy: 0.8906
60000/60000 [] - 2s 34us/sample - loss: 0.2940 - accuracy: 0.8906
```

Note that it's now reporting accuracy. So in this case, using the training data, our model ended up with an accuracy of about 89% after only five epochs.

But what about the test data? The results of `model.evaluate` on our test data will look something like this:

```
10000/1 [====] - 0s 30us/sample - loss: 0.2521 - accuracy: 0.8736
```

In this case the accuracy of the model was 87.36%, which isn't bad considering we only trained it for five epochs.

You're probably wondering why the accuracy is *lower* for the test data than it is for the training data. This is very commonly seen, and when you think about it, it makes sense: the neural network only really knows how to match the inputs it has been trained on with the outputs for those values. Our hope is that, given enough data, it will be able to generalize from the examples it has seen, "learning" what a shoe or a dress looks like. But there will always be examples of items that it hasn't seen that are sufficiently different from what it has to confuse it.

For example, if you grew up only ever seeing sneakers, and that's what a shoe looks like to you, when you first see a high heel you might be a little confused. From your experience, it's probably a shoe, but you don't know for sure. This is a similar concept.

Exploring the Model Output

Now that the model has been trained, and we have a good gage of its accuracy using the test set, let's explore it a little:

```
classifications = model.predict(test_images)
print(classifications[0])
print(test_labels[0])
```

We'll get a set of classifications by passing the test images to `model.predict`. Then let's see what we get if we print out the first of the classifications and compare it to the test label:

```
[1.9177722e-05 1.9856788e-07 6.3756357e-07 7.1702580e-08 5.5287035e-07
 1.2249852e-02 6.0708484e-05 7.3229447e-02 8.3050705e-05 9.1435629e-01]
9
```

You'll notice that the classification gives us back an array of values. These are the values of the 10 output neurons. The label is the actual label for the item of clothing, in this case 9. Take a look through the array—you'll see that some of the values are very small, and the last one (array index 9) is the largest by far. These are the probabilities that the image matches the label at that particular index. So, what the neural network is reporting is that there's a 91.4% chance that the item of clothing at index 0 is label 9. We know that it's label 9, so it got it right.

Try a few different values for yourself, and see if you can find anywhere the model gets it wrong.

Training for Longer—Discovering Overfitting

In this case, we trained for only five epochs. That is, we went through the entire training loop of having the neurons randomly initialized, checked against their labels, having that performance measured by the loss function, and then updated by the optimizer five times. And the results we got were pretty good: 89% accuracy on the training set and 87% on the test set. So what happens if we train for longer?

Try updating it to train for 50 epochs instead of 5. In my case, I got these accuracy figures on the training set:

```
58112/60000 [==>.] - ETA: 0s - loss: 0.0983 - accuracy: 0.9627
59520/60000 [==>.] - ETA: 0s - loss: 0.0987 - accuracy: 0.9627
60000/60000 [====] - 2s 35us/sample - loss: 0.0986 - accuracy: 0.9627
```

This is particularly exciting because we're doing much better: 96.27% accuracy. For the test set we reach 88.6%:

```
[====] - 0s 30us/sample - loss: 0.3870 - accuracy: 0.8860
```

So, we got a big improvement on the training set, and a smaller one on the test set. This might suggest that training our network for much longer would lead to much better results—but that's not always the case. The network is doing much better with the training data, but it's not necessarily a better model. In fact, the divergence in the accuracy numbers shows that it has become overspecialized to the training data, a process often called *overfitting*. As you build more neural networks this is something to watch out for, and as you go through this book you'll learn a number of techniques to avoid it.

Stopping Training

In each of the cases so far, we've hardcoded the number of epochs we're training for. While that works, we might want to train until we reach the desired accuracy instead of constantly trying different numbers of epochs and training and retraining until we get to our desired value. So, for example, if we want to train until the model is at 95%

accuracy on the training set, without knowing how many epochs that will take, how could we do that?

The easiest approach is to use a *callback* on the training. Let's take a look at the updated code that uses callbacks:

```
import tensorflow as tf

class myCallback(tf.keras.callbacks.Callback):
  def on_epoch_end(self, epoch, logs={}):
    if(logs.get('accuracy')>0.95):
      print("\nReached 95% accuracy so cancelling training!")
      self.model.stop_training = True

callbacks = myCallback()
mnist = tf.keras.datasets.fashion_mnist

(training_images, training_labels),
(test_images, test_labels) = mnist.load_data()

training_images=training_images/255.0
test_images=test_images/255.0

model = tf.keras.models.Sequential([
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(128, activation=tf.nn.relu),
            tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

 model.fit(training_images, training_labels, epochs=50,
           callbacks=[callbacks])
```

Let's see what we've changed here. First, we created a new class called `myCallback`. This takes a `tf.keras.callbacks.Callback` as a parameter. In it, we define the `on_epoch_end` function, which will give us details about the logs for this epoch. In these logs is an accuracy value, so all we have to do is see if it is greater than .95 (or 95%); if it is, we can stop training by saying `self.model.stop_training = True`.

Once we've specified this, we create a `callbacks` object to be an instance of the `myCallback` function.

Now check out the `model.fit` statement. You'll see that I've updated it to train for 50 epochs, and then added a `callbacks` parameter. To this, I pass the `callbacks` object.

When training, at the end of every epoch, the callback function will be called. So at the end of each epoch you'll check, and after about 34 epochs you'll see that your training will end, because the training has hit 95% accuracy (your number may be

slightly different because of the initial random initialization, but it will likely be quite close to 34):

```
56896/60000 [====>..] - ETA: 0s - loss: 0.1309 - accuracy: 0.9500
58144/60000 [====>.] - ETA: 0s - loss: 0.1308 - accuracy: 0.9502
59424/60000 [====>.] - ETA: 0s - loss: 0.1308 - accuracy: 0.9502
Reached 95% accuracy so cancelling training!
```

Summary

In Chapter 1 you learned about how machine learning is based on fitting features to labels through sophisticated pattern matching with a neural network. In this chapter you took that to the next level, going beyond a single neuron, and learned how to create your first (very basic) computer vision neural network. It was somewhat limited because of the data. All the images were 28 × 28 grayscale, with the item of clothing centered in the frame. It's a good start, but it is a very controlled scenario. To do better at vision, we might need the computer to learn features of an image instead of merely the raw pixels.

We can do that with a process called *convolutions*. You'll learn how to define convolutional neural networks to understand the contents of images in the next chapter.

About the Author

Laurence Moroney leads AI Advocacy at Google. His goal is to educate software developers in building AI systems with machine learning. He's a frequent contributor to the TensorFlow YouTube channel, *youtube.com/tensorflow*, a recognized global keynote speaker, and author of more books than he can count, including several best-selling science fiction novels, and a produced screenplay. He's based in Sammamish, Washington where he drinks way too much coffee.

Colophon

The animal on the cover of *AI and Machine Learning for Coders* is a lobe-footed or web-footed gecko (*Palmatogecko rangei*). There are over a thousand species of gecko, which are small lizards. This species evolved in the Namib Desert in Namibia and southwestern Africa.

Web-footed geckos are nearly translucent, four- to six-inch long creatures, with oversized eyes that they lick to keep clear. Their webbed feet move like scoops through the desert sand, and they have adhesive pads on their toes that further enhance their agility. Like most geckos, the web-footed variety are nocturnal insectivores.

Unlike other reptiles, most geckos can vocalize. The web-footed geckos have an especially broad range of vocalizations, including clicks, croaks, barks, and squeaks.

The web-footed gecko population has not been evaluated by the International Union for Conservation of Nature. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The color illustration is by Karen Montgomery, based on a black and white engraving from *Lydekker's Royal Natural History*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.