O'REILLY®

AI and Machine Learning for Coders

A Programmer's Guide to Artificial Intelligence

Early Release
RAW & UNEDITED

Laurence Moroney

AI and Machine Learning for Coders

A Programmer’s Guide to Artificial Intelligence

Laurence Moroney

AI and Machine Learning for Coders

by Laurence Moroney

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

Editors: Angela Rufino and Rebecca Novack

Production Editor: Christopher Faucher

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

April 2021: First Edition

Revision History for the Early Release

- 2020-04-23: First release

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

978-1-492-07812-8

Chapter 1. Introduction to TensorFlow

When it comes to creating Artificial Intelligence, Machine Learning and Deep Learning are a great place to start. When trying to get started it's easy to get overwhelmed with options and lots of new terminology. This book aims to demystify that for programmers, taking you through writing code to implement concepts of Machine Learning and Deep Learning, building models that behave more like a human does with scenarios like Computer Vision, Natural Language Processing and more. Thus, they become a form of synthesized, or artificial intelligence.

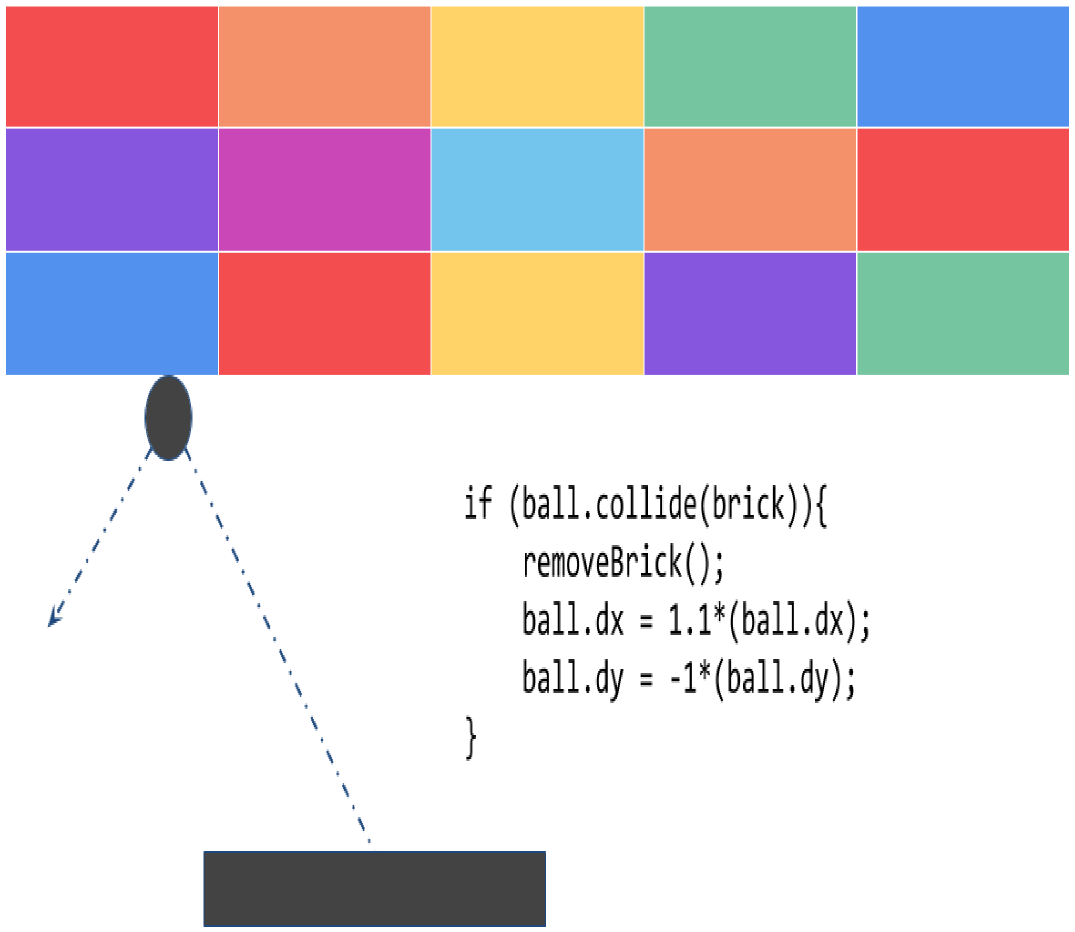
But what *is* it? When we refer to *Machine Learning*, what, in fact is this phenomenon? Let's take a quick look at that, and think about it from a *programmer's* perspective before we go any further.

What is Machine Learning?

Before looking at the ins and outs of ML, let’s just consider how it evolved from traditional programming. We’ll start by capturing the concept of what traditional programming is, then consider cases where it is limited, and show how ML evolved to handle those cases, and as a result has opened up new opportunities to implement new scenarios, unlocking many of the concepts of artificial intelligence.

Traditional programming involves us writing rules, expressed in a programming language, that act on data, and give us answers. This applies just about everywhere something can be programmed with code.

For example, consider a game, like the popular ‘breakout’ one. Code determines the movement of the ball, the score, and the various conditions for winning or losing the game. Think about the scenario when the ball bounces off a brick, like in Figure 1.

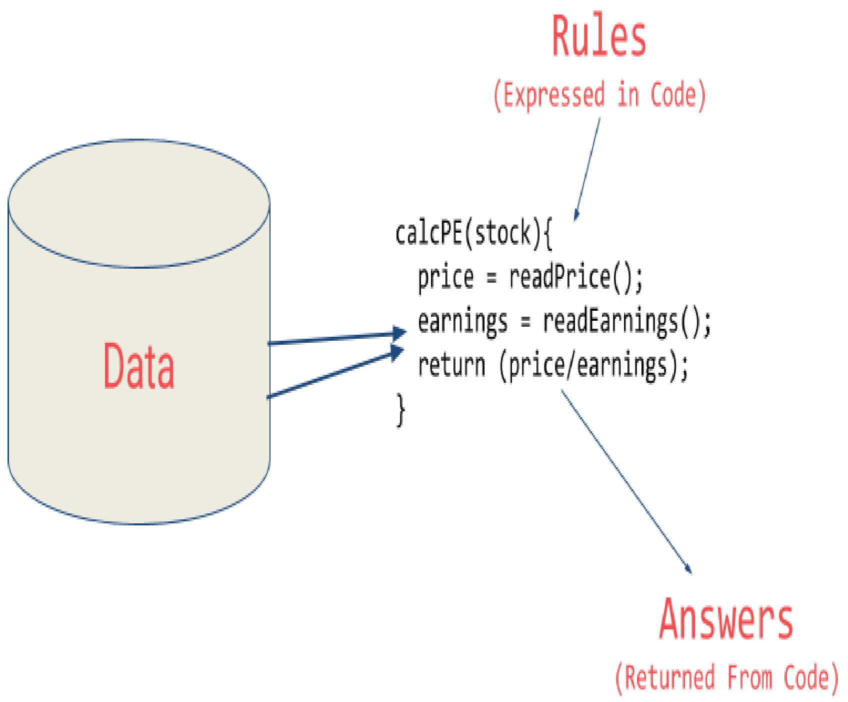


```
if (ball.collide(brick)){
    removeBrick();
    ball.dx = 1.1*(ball.dx);
    ball.dy = -1*(ball.dy);
}
```

Figure 1-1. Code in a Breakout Game

Here, the motion of the ball can be determined by its dx and dy properties. When it hits the brick, the brick is removed, and the velocity of the ball will increase as well as changing direction. Your code acts on data about the game situation.

Or, also consider, perhaps in a Financial Services scenario. You have data about a company's stock, such as its current price, and its current earnings. You'll calculate a valuable ratio called a P/E (for Price divided by Earnings) with a scenario like Figure 2.



```
calcPE(stock){
  price = readPrice();
  earnings = readEarnings();
  return (price/earnings);
}
```

Figure 1-2. Code in a Financial Services scenario.

Here your code might read the price, read the earnings and return a value that is the former divided by the latter.

If I were to try to sum up traditional programming like this into a single diagram, it might look like Figure 3.

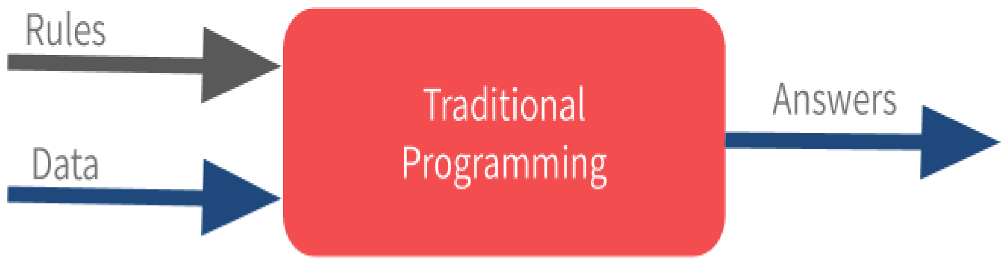


Figure 1-3. High Level Concept of Traditional Programming

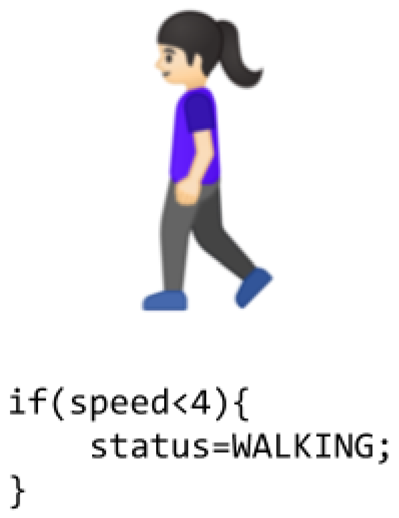
As you can see you have rules, expressed in a programming language. These rules act on data, and the result is answers.

Limitations of Traditional Programming

The model from Figure 3 has been the backbone of development since it inception. But it has an inherent limitation -- namely that only scenarios for which a programmer can derive the rules can be implemented. What about other scenarios? Usually they are infeasible to develop because the code is too complex. It's just not possible to write code to handle it.

Consider, for example activity detection. Fitness monitors that can detect our activity are a *recent* innovation, not just because of the availability of cheap and small hardware, but also because the algorithms to handle detection weren't previously feasible. Let's explore why.

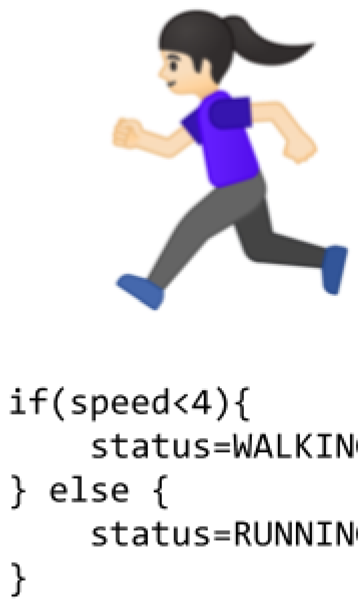
Figure 4 shows a naive activity detection algorithm for walking. It can consider her speed. If it's less than a particular value, we can determine that she's probably walking.



```
if(speed<4){
    status=WALKING;
}
```

Figure 1-4. Algorithm for activity detection.

Given that our data is speed, we could now extend this to detect if she is running like in Figure 5.



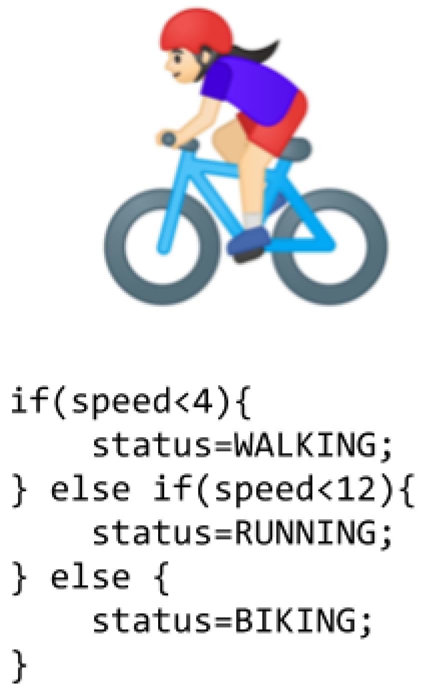
```
if(speed<4){
    status=WALKING;
} else {
    status=RUNNING;
}
```

Figure 1-5. Extending the algorithm for running

As we can see, going by the speed, we might say if it is less than a particular value (say 4 mph) she is walking, otherwise she is running.

It still sort of works.

And again, we now want to extend this because Cycling is a popular fitness activity. The algorithm could look like Figure 6.



```
if(speed<4){
    status=WALKING;
} else if(speed<12){
    status=RUNNING;
} else {
    status=BIKING;
}
```

Figure 1-6. Extending the algorithm for Biking.

It still sort of works. I know it’s naive by just detecting speed -- some people run faster than others, and you might run downhill faster than you cycle uphill, for example. But it still sort of works. However, what happens when you want to implement another scenario -- for example golfing? See Figure 7.

```
// ???
```

Figure 1-7. How do we write a golfing algorithm

We’re now stuck. How do we determine that someone is golfing using this methodology? She might walk for a bit, stop, do some activity, walk for a bit more, stop etc. But how can we tell this is golf.

Our ability to detect this activity using traditional rules has hit a wall. But maybe there’s a better way.

Enter Machine learning.

From Programming to Learning.

Let’s look back at the diagram that we used to demonstrate what traditional programming is. You can see it in Figure 8. Here you have rules that act on data and give you answers. In our activity detection scenario, the data was the speed that our friend was going at, and from that we could write rules to detect her activity, be it walking, biking or running. We hit a wall when it came to golfing, because we couldn’t come up with rules to determine what that activity looks like.

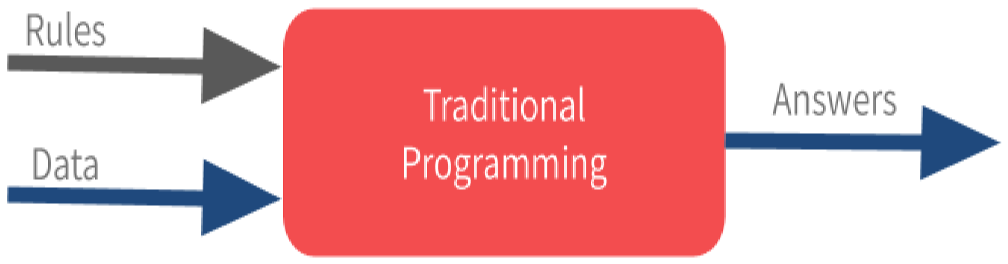


Figure 1-8. The diagram of traditional programming.

But what would happen if we flip the axes around on this diagram. So, instead of us coming up with the *rules*, what if we come up with the *answers*, and along with the data have a way of *figuring out* what the rules might be.

In other words, let’s change the diagram of traditional programming to something new. See Figure 9. We can consider this the high level diagram to define *Machine Learning*.

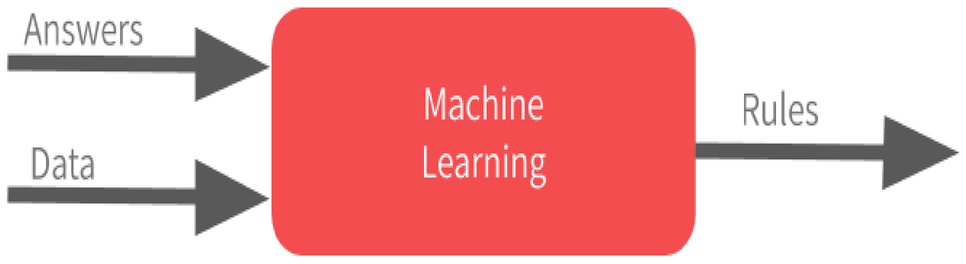


Figure 1-9. Changing the Axes to get Machine Learning.

So what are the implications of this? Well -- now, instead of us trying to figure out what the rules are, we get lots of data about our scenario, we *label* that data, and the computer can figure out what the rules are that make one piece of data match a particular label, and another piece of data match its label.

How would this work for our activity detection scenario? Well, now we can look at all the sensors that give us details about this person. If she has a wearable it could information such as heart rate, location, speed, etc. And if we collect a lot of instances of this data while she's doing activities, we end up with a scenario of having data that says 'this is what walking looks like', 'this is what running looks like' etc. See Figure 10.

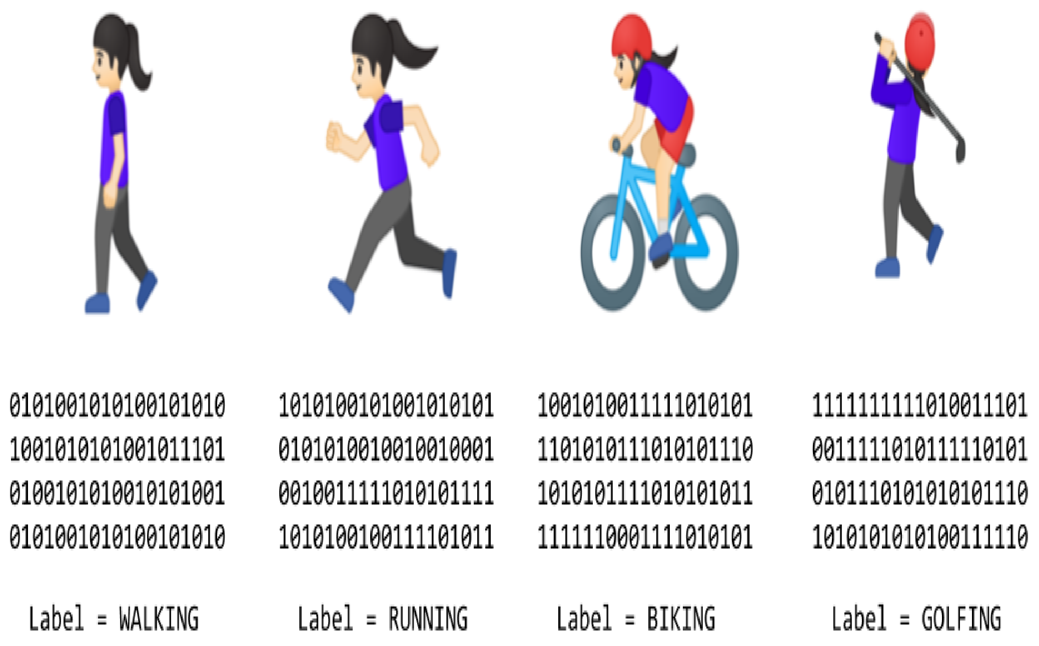


Figure 1-10. From coding to ML. Gathering and Labelling data.

Now, your job as a programmer changes from figuring out the rules to determine the activities to writing the code that matches the data to the label. If you could do this, then you can expand the scenarios that you can implement with code. Machine Learning is a technique that can do this, but in order to code these, you’ll need a framework. And that’s where TensorFlow enters the picture. Let’s first take a look at what it is, and how to install it, and then later in this chapter you’ll write your first code that learns the pattern between two values like the scenario above. It’s a simple ‘hello world’ type scenario, but it has the foundational code pattern that’s used in extremely complex ones.

What is TensorFlow?

TensorFlow is an open source platform for creating and using Machine Learning models. It implements many of the common algorithms and patterns needed for machine learning, saving you from needing to learn all the underlying math and logic, and just to focus on your scenario. It’s aimed at everyone from hobbyists to professional developers to researchers pushing the boundaries of Artificial Intelligence. Importantly, it also supports deployment of models to the web, cloud, mobile and embedded systems. We’ll be covering each of these scenarios in this book.

The high level architecture of TensorFlow can be seen in Figure 11.

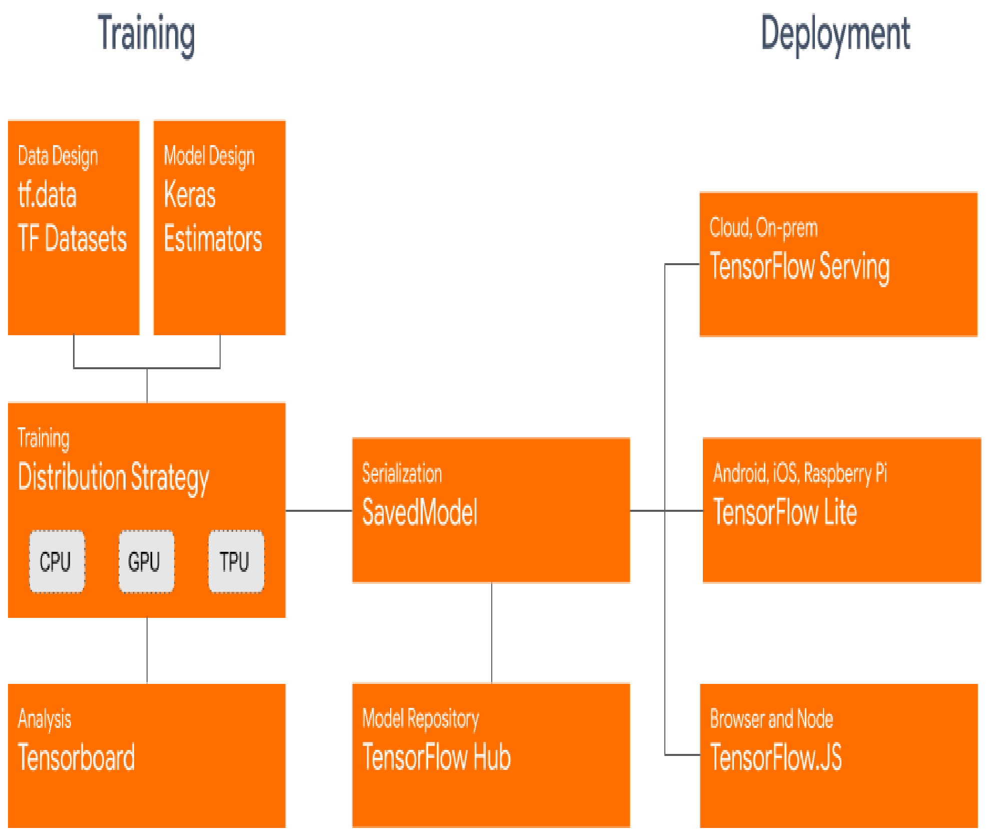


Figure 1-11. TensorFlow High Level architecture

The process of creating machine learning models is called **training**. This is where a computer uses a set of algorithms to learn about inputs and what distinguishes them from each other. So, for example, if you want a computer to recognize cats and dogs, you can use lots of pictures of both, and the computer will try to figure out what makes a cat a cat, and what makes a dog a dog. You’ll learn all about them in this book.

So, for training models, there are a number of things that you need to support. First, is a set of APIs for designing the models themselves. With TensorFlow there are three main ways to do this. You can code everything by hand (not recommended), use built in **estimators**, which are already-implemented neural networks that you can customize, and there's **Keras**, a high level API which allows you to encapsulate common machine learning paradigms into code. This book will primarily focus on using the Keras APIs when creating models.

When *training* a model, there are a number of ways that you can do it. For the most part, you'll probably just use a single chip, be it a CPU, a GPU or something new called a Tensor Processing Unit (TPU), but in many working and research environments, parallel training across multiple chips can be used, and for them a **distribution strategy**, where training is spanned across multiple chips is necessary. TensorFlow supports this.

The life blood of any model is in its data. As discussed earlier, if you want to create a model that can recognize cats and dogs, it needs to be trained with lots of examples of cats and dogs. But how can you manage these examples? You'll see, over time, that this can often involve a lot more coding than the creation of the models themselves. So TensorFlow ships with APIs to try to ease this process, and they're called **TensorFlow Data Services**. For learning, they include lots of pre-processed datasets that you can use with a single line of code. They also give you the tools for processing raw data to make it easier to use.

Beyond creating models, you'll also need to be able to get them into people's hands where they can be used. To this end, TensorFlow includes APIs for **serving**, where you can provide model inference over an HTTP connection for cloud or web users. For models to run on mobile or embedded systems, there's **TensorFlow Lite** which provides tools to provide model inference on Android and iOS as well as Linux-based embedded systems such as a Raspberry Pi. Finally, if you want to provide models to your browser users, **TensorFlow.js** provides the ability to train and execute models in this way.

Next, let's learn how to install TensorFlow so that you can get up and running with creating and using ML models with it!

Using TensorFlow

In this section, we'll look at the 3 main ways that you can install and use TensorFlow. We'll start with how to install it on your developer box. We'll then look at the popular **PyCharm** IDE and how you can install TensorFlow with it. Finally, we'll look at Google Colab and how it can be used to access TensorFlow code with a cloud-based backend in your browser.

Installing TensorFlow in Python

While TensorFlow supports the creation of models using multiple languages, including Python, Swift, Java and more, in this book we'll focus on using **Python.**

When using Python, there are many ways to install frameworks, but the default one supported by the team is **pip**.

So, in your python environment, installing TensorFlow is as easy as using

```
>pip install tensorflow
```

Note that starting with version **2.1,** this will install the **GPU** version of TensorFlow by default. Prior to that it used the **CPU** version. So, before installing, make sure you have a supported GPU, and all the requisite drivers for it. Details on this are available at https://www.tensorflow.org/install/gpu

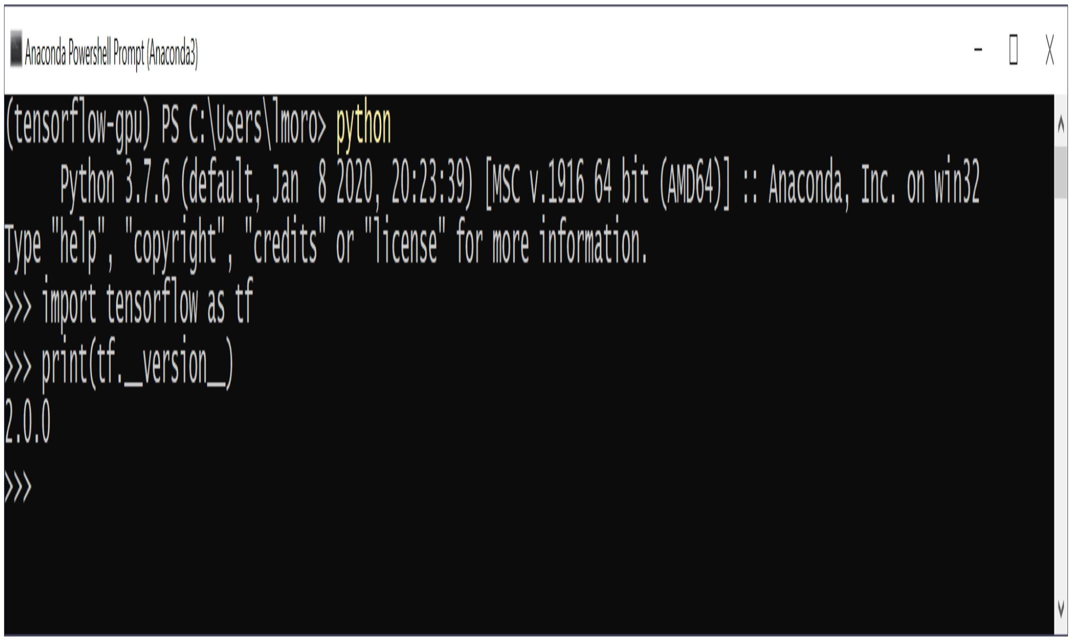
If you don’t have the required GPU or drivers, you can still use the CPU version of tensorflow on any Linux, PC or Mac with

```
>pip install tensorflow-cpu
```

Once you’re up and running, you can test your TensorFlow version with the following code:

```
import tensorflow as tf
print(tf.__version__)
```

And you should see output like that in Figure 12. This will print the currently running version of TensorFlow.



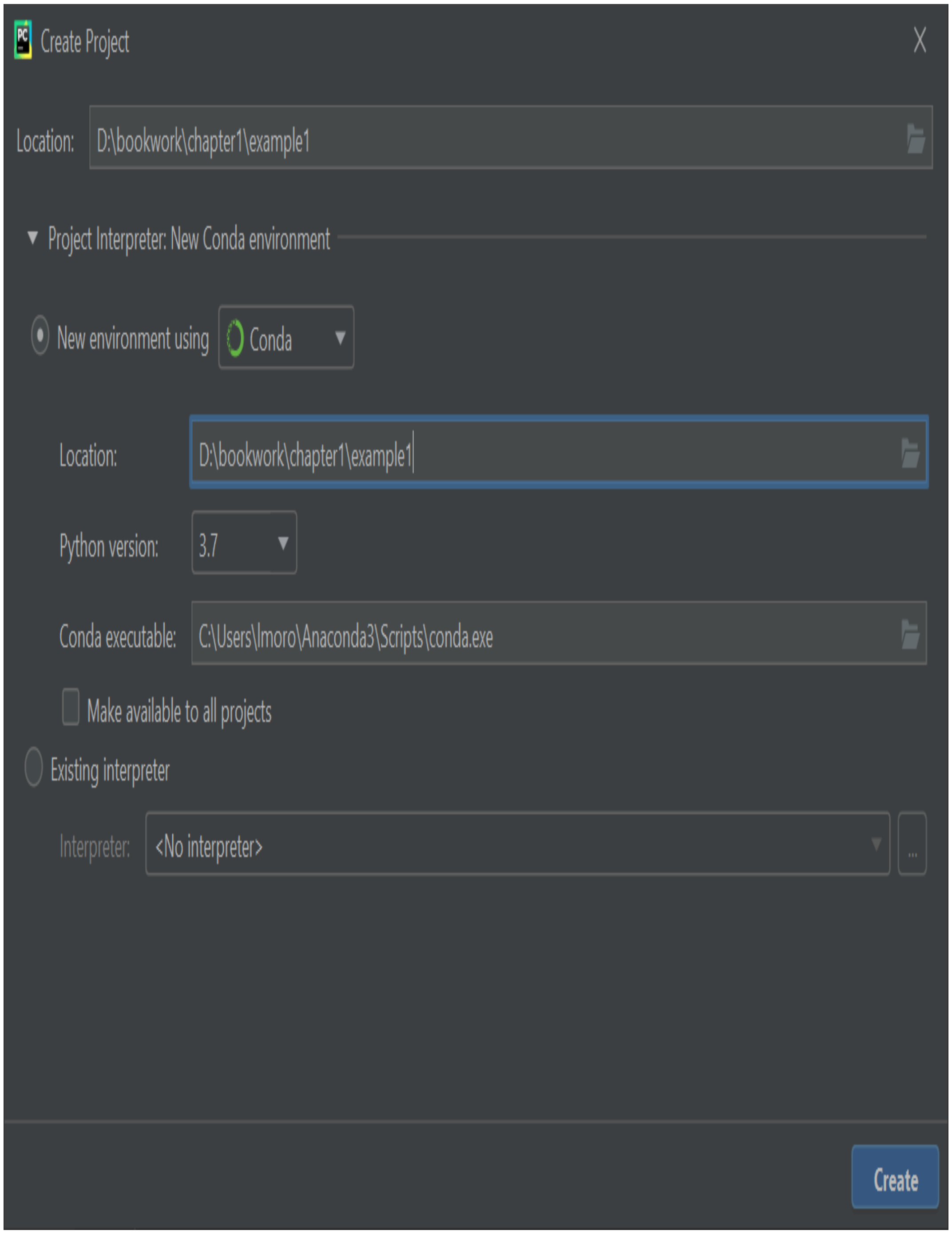
```
(tensorflow-gpu) PS C:\Users\lmoro> python
    Python 3.7.6 (default, Jan  8 2020, 20:23:39) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import tensorflow as tf
>>> print(tf._version_)
2.0.0
>>>
```

Figure 1-12. Running TensorFlow in Python

Using TensorFlow in PyCharm

I’m particularly fond of using the free, community version of PyCharm for building models using TensorFlow. PyCharm is available at https://www.jetbrains.com/pycharm/. PyCharm is really useful for many reasons, but one of my favorites, with regard to TensorFlow is that it makes the management of virtual environments easy -- this means you can have python environments with versions of tools such as TensorFlow that are specific to your particular project very easy. So, for example, if you want to use TensorFlow 2.0 in one project, and TensorFlow 2.1 in another, you can separate with these virtual environments and not deal with installing/uninstalling dependencies when you switch. Additionally, with TensorFlow you can do step-by-step debugging of your Python code -- a must if you’re getting started!

You can do this when you create a new project. So, for example, in Figure 13 I have a new project that is called example 1, and I'm specifying that I am going to create a new environment using Conda. When I create the project I'll have a clean new virtual python environment into which I can install any version of TensorFlow I want.


Create Project

Location: D:\bookwork\chapter1\example1

Project Interpreter: New Conda environment

New environment using Conda

Location: D:\bookwork\chapter1\example1

Python version: 3.7

Conda executable: C:\Users\lmoro\Anaconda3\Scripts\conda.exe

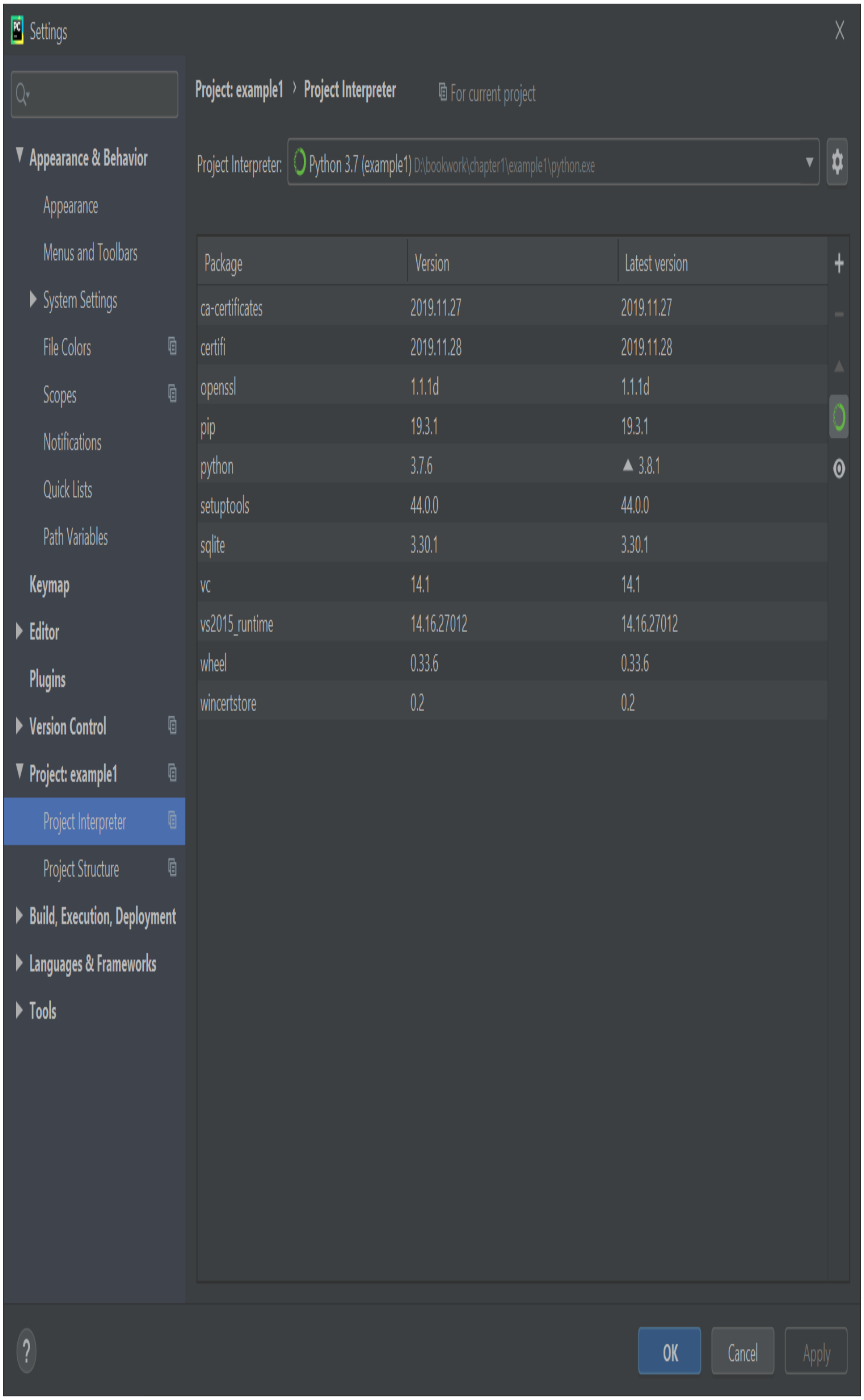
Make available to all projects

Existing interpreter

Interpreter: <No interpreter>

Figure 1-13. Creating a new Virtual Environment using PyCharm

Once you've created your project, you can then use the File->Settings dialog and choose 'Project'. You'll then see choices to change the settings for the Project Interpreter and the Project Structure. Choose the Project Interpreter link, and you'll see the interpreter that you're using, as well as a list of packages that are installed in this virtual environment. See Figure 14.



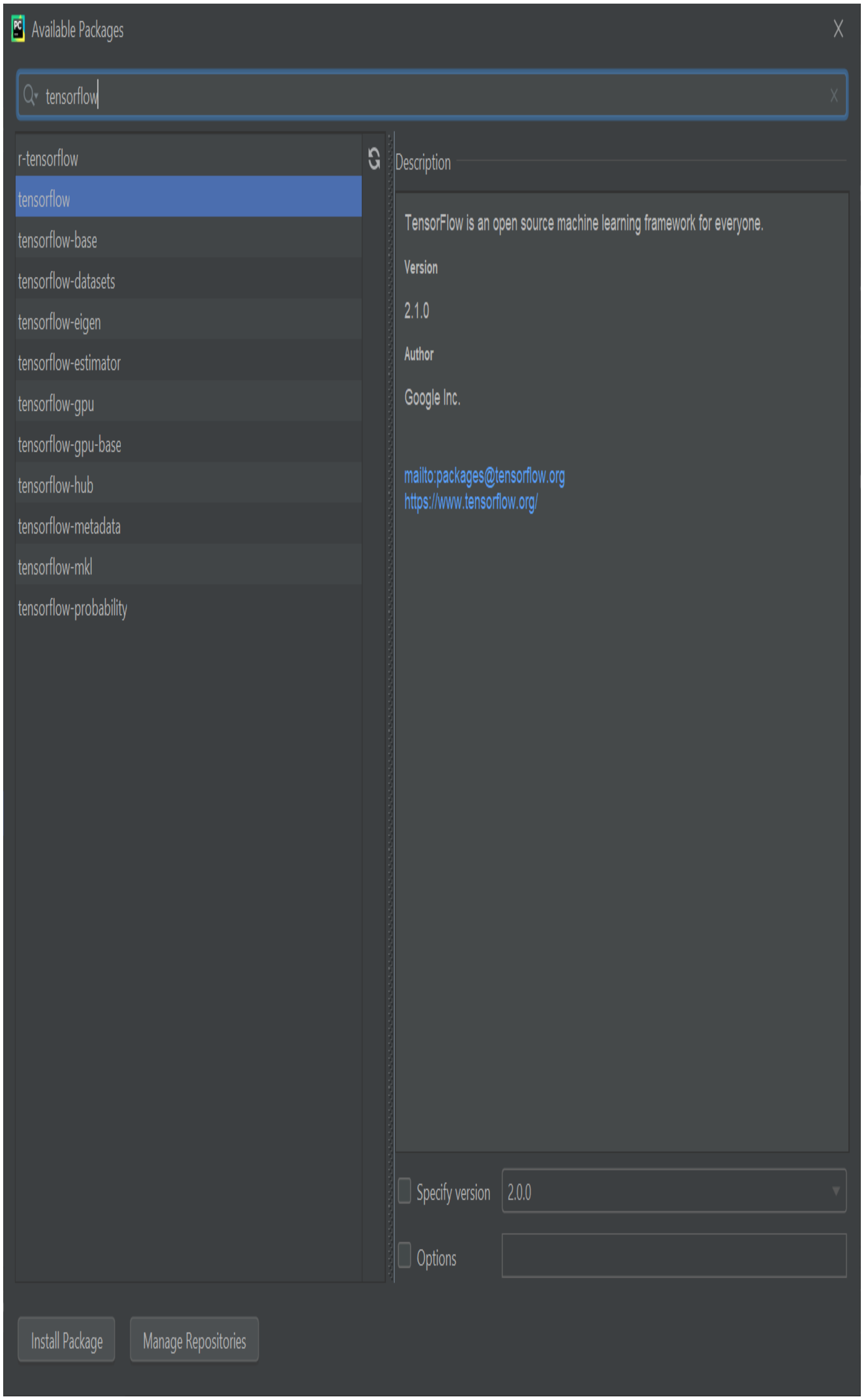
Project: example1 › Project Interpreter   For current project

Project Interpreter: Python 3.7 (example1) D:\bookwork\chapter1\example1\python.exe

Package	Version	Latest version
ca-certificates	2019.11.27	2019.11.27
certifi	2019.11.28	2019.11.28
openssl	1.1.1d	1.1.1d
pip	19.3.1	19.3.1
python	3.7.6	▲ 3.8.1
setuptools	44.0.0	44.0.0
sqlite	3.30.1	3.30.1
vc	14.1	14.1
vs2015_runtime	14.16.27012	14.16.27012
wheel	0.33.6	0.33.6
wincertstore	0.2	0.2

Figure 1-14. Adding packages to a virtual environment

Click the ‘+’ button beside the list, and a dialog will open with the packages that are currently available. Type ‘tensorflow’ into the search box and you’ll see all available packages with tensorflow in the name. See Figure 15.



TensorFlow is an open source machine learning framework for everyone.

Version

2.1.0

Author

Google Inc.

mailto:packages@tensorflow.org
https://www.tensorflow.org/

Specify version 2.0.0

Options

Figure 1-15. Installing TensorFlow with PyCharm

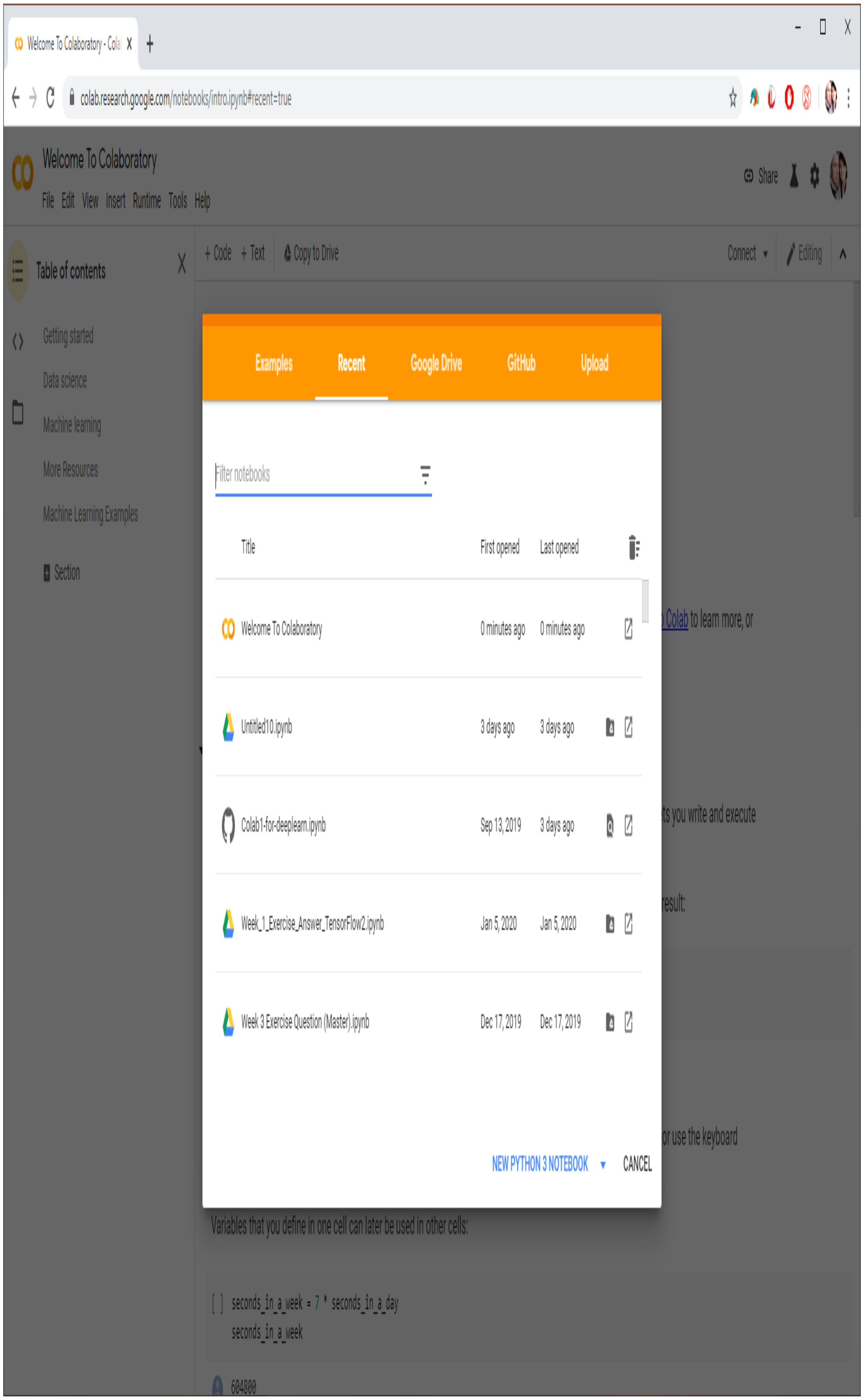
Once you’ve selected TensorFlow, or any other package you want to install, and then click the ‘Install Package’ button, PyCharm will do the rest.

Once TensorFlow is installed, you can now write and debug your TensorFlow code in Python.

Using TensorFlow in Google Colab

Another option, which is perhaps easiest for getting started is to use Google Colab. This is available at **colab.research.google.com** and is a hosted Python environment that you can access via the browser. What’s really neat about colab is that it provides GPU and TPU backends so you can train models using state-of-the-art hardware at no cost.

When you visit the colab site, you’ll be given the option to open previous colabs, or start a new notebook as shown in Figure 16.


Welcome To Colaboratory - Cola

colab.research.google.com/notebooks/intro.ipynb#recent=true

Welcome To Colaboratory

File Edit View Insert Runtime Tools Help

Share

Table of contents

Getting started

Data science

Machine learning

More Resources

Machine Learning Examples

Section

+ Code + Text Copy to Drive

Connect Editing

Examples Recent Google Drive GitHub Upload

Filter notebooks

Title	First opened	Last opened
Welcome To Colaboratory	0 minutes ago	0 minutes ago
Untitled10.ipynb	3 days ago	3 days ago
Colab1-for-deeplearn.ipynb	Sep 13, 2019	3 days ago
Week_1_Exercise_Answer_TensorFlow2.ipynb	Jan 5, 2020	Jan 5, 2020
Week 3 Exercise Question (Master).ipynb	Dec 17, 2019	Dec 17, 2019

NEW PYTHON 3 NOTEBOOK CANCEL

Colab to learn more, or

ts you write and execute

result:

or use the keyboard

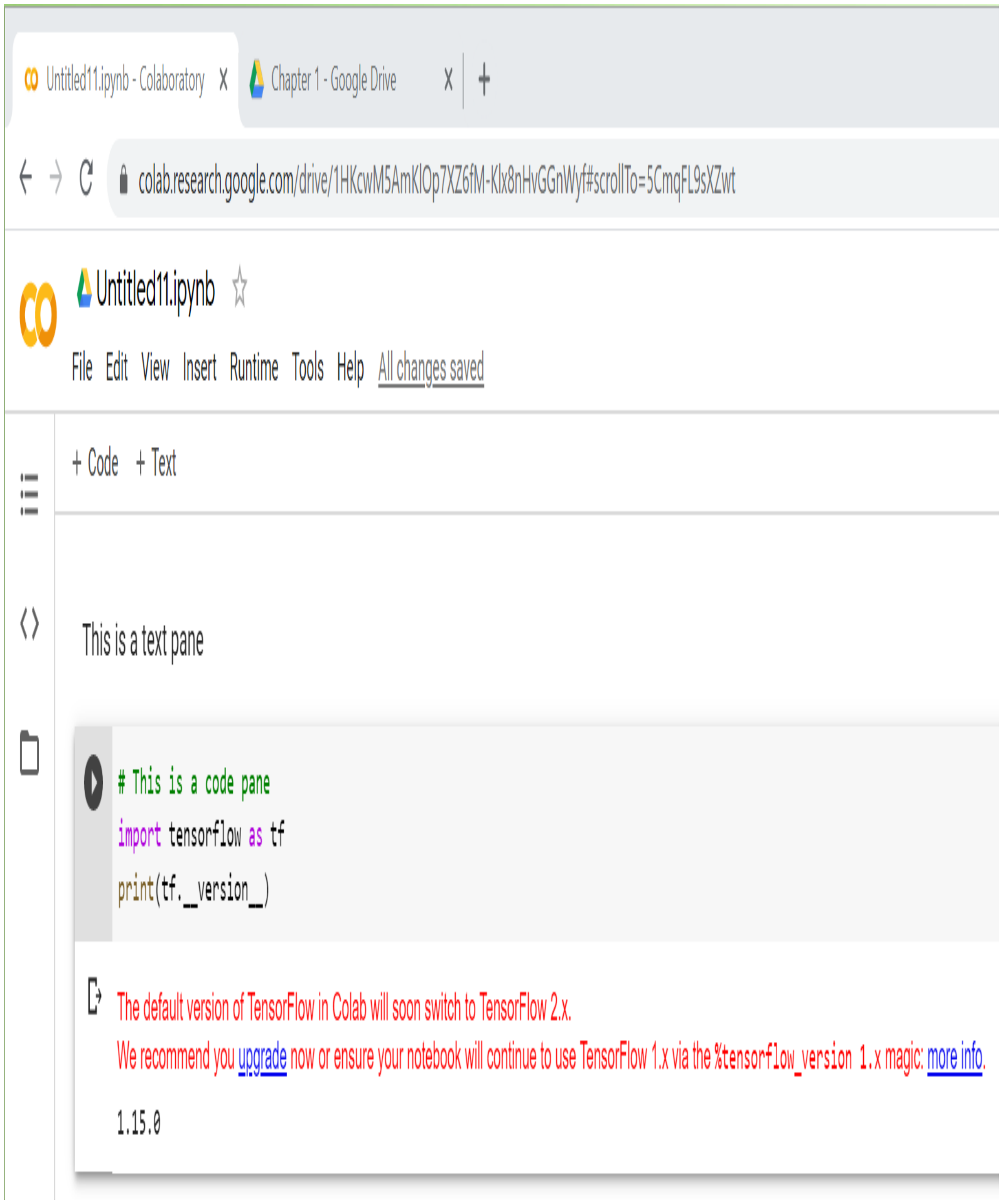
Variables that you define in one cell can later be used in other cells:

```
[ ] seconds_in_a_week = 7 * seconds_in_a_day
    seconds_in_a_week
```

```
604800
```

Figure 1-16. Starting with Google Colab

Once you start a new Python 3 notebook using the link, you’ll see the editor. This allows you to add panes of code or text. The code can be executed by clicking the play button to the left of the pane. See Figure 17.


Untitled11.ipynb - Colaboratory
Chapter 1 - Google Drive
colab.research.google.com/drive/1HKcwM5AmKlOp7XZ6fM-Klx8nHvGGnWyf#scrollTo=5CmqFL9sXZwt
Untitled11.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text

This is a text pane

```
# This is a code pane
import tensorflow as tf
print(tf.__version__)
```

```
The default version of TensorFlow in Colab will soon switch to TensorFlow 2.x.
We recommend you upgrade now or ensure your notebook will continue to use TensorFlow 1.x via the %tensorflow_version 1.x magic: more info.
1.15.0
```

Figure 1-17. Running TensorFlow code in Colab

It’s always a good idea to check the TensorFlow version like I did in Figure 7, to be sure you’re running the correct version. Often colab’s built-in TensorFlow will be a version or two behind. If so, you can still use pip install as earlier, by simply using a block of code like this:

```
!pip install tensorflow==2.0.0
```

Once you run it, your current environment within colab will use the desired version of TensorFlow.

Getting Started with Machine Learning

As we saw earlier in the chapter, the Machine Learning paradigm is one where you have data, that data is labeled and you want to figure out the rule that matches the data to the label. The simplest possible scenario to show this in code is as follows. Consider these two sets of numbers. There’s a relationship between the X and the Y. Can you see it?

```
X = -1, 0, 1, 2, 3, 4
Y = -3, -1, 1, 3, 5, 7
```

So, for example, if X is -1, then Y is -3. If X is 3, then Y is 5 and so on.

After a few seconds (hopefully) you probably saw that the pattern here is that Y = 2X-1. How did you get that? Everybody is different, but I typically hear that people see that X increases by 1 in its sequence, and Y increases by 2. Thus Y = 2X +/- something. They then look at when X = 0, and see that Y = -1, so they figure that the answer could be Y=2X-1. They then look at the other values and see that this hypothesis ‘fits’, and the answer is Y=2X-1.

That’s very similar to the Machine Learning process. Let’s take a look at some TensorFlow code that you’d write to have a neural network do this figuring out for you.

Here’s the full code. Don’t worry if it doesn’t make sense yet, we’ll go through it line by line.

```
import tensorflow as tf
import numpy as np
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
model = Sequential([Dense(units=1, input_shape=[1])])
model.compile(optimizer='sgd', loss='mean_squared_error')
xs = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)
model.fit(xs, ys, epochs=500)
print(model.predict([10.0]))
```

Let’s start with the first line. You’ve probably heard of Neural Networks, and you’ve probably seen diagrams that explain them using layers of interconnected neurons, a little like Figure 18.

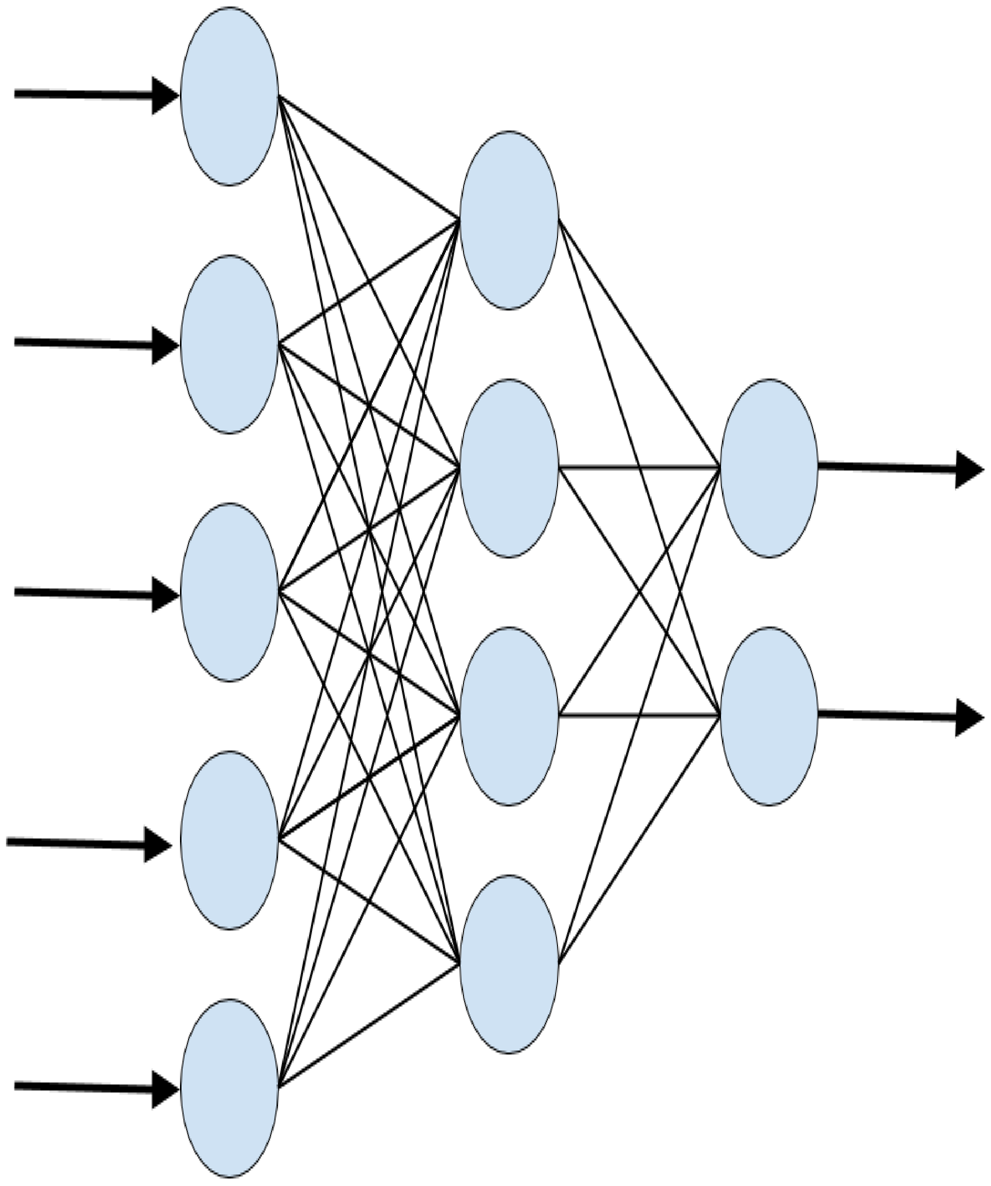


Figure 1-18. A Typical Neural Network

When you see a neural network like this, consider each of the ‘circles’ to be a **neuron**, and each of the columns of circles to be a **layer**.

So if we look back at our code, and look at just the first line, we'll see that we're defining the simplest possible neural network. There's only 1 layer, and it contains only 1 neuron.

```
model = Sequential([Dense(units=1, input_shape=[1])])
```

When using TensorFlow, you define your layers using **Sequential.** Inside the sequential you then specify what each layer looks like. We only have 1 line inside our sequential, thus we have only 1 layer.

You then define what the layer looks like using the `keras.layers API`. There's lots of different layer types, but here we are using a Dense type. Dense means a set of fully (or densely) connected neurons, which is what you can see in Figure 18 where every neuron is connected to every neuron in the next layer. It's the most common form of layer type. Our Dense has only 1 unit specified. Thus we have 1 dense layer with 1 neuron in our entire neural network. Finally, when you specify the *first* layer in a neural network (and in this case it's our only layer), you have to tell it what the shape of the input data is. In this case our input data is our X, which is just a single value, so we'll specify that that's its shape.

The next line is where the fun really begins. Let's look at it again:

```
model.compile(optimizer='sgd', loss='mean_squared_error')
```

If you've looked at Machine Learning before, you've probably seen that it involves a lot of mathematics. If you haven't done calculus in years it might have been a barrier to entry. Here's the part where the math comes in. It's really the core to machine learning. In a scenario such as this one, the computer has *no idea* what the relationship between X and Y is. So it will make a guess.

Say for example it guesses that Y=10X+10. It then needs to measure how good or how bad that guess is. That's the job of the **loss function**.

It already knows the answers when X is -1, 0, 1, 2, 3, and 4. So the loss function can compare these to the answers for the guessed relationship. If it guessed Y=10X+10, then when X is -1, Y will be 0. The correct answer there was -3, so it's a bit off. But when X is 4, the guessed answer is 50, where the correct 1 is 7 -- which is really far off.

Armed with this knowledge, the computer can then make another guess. And that's the job of the **optimizer.** In particular this is where the heavy calculus is used, but with TensorFlow that can be hidden from you. You just pick the appropriate optimizer to use for different scenarios. In this case we picked one called 'sgd', which stands for **Stochastic Gradient Descent**, which is a complex mathematical function, which when given the values, the previous guess, the results of calculating the errors (or loss) on that guess, it can then generate another one. Over time, its job is to minimize the loss, and by so doing, the guessed formula will get closer and closer to the correct answer.

Next up we simply format our numbers into the data format that the layers expect. In Python, there's a library called **numpy** that

TensorFlow can use, and here we put our numbers into a numpy array to make it easy to process them:

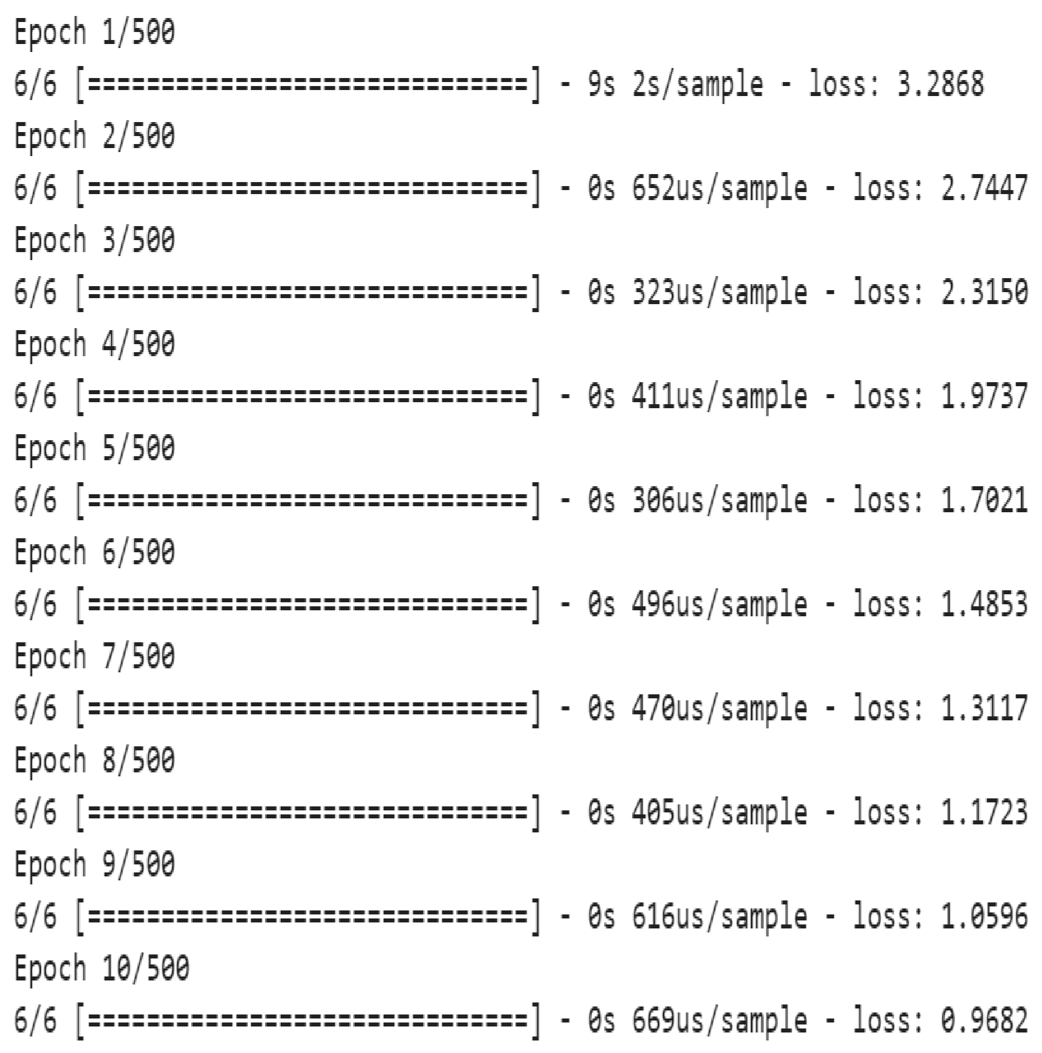
```
xs = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)
```

The *learning* process will then begin with the model.fit command, like this:

```
model.fit(xs, ys, epochs=500)
```

You can read this as ‘fit the xs to the ys, and try it 500 times’. So, on the first try, the computer will guess the relationship (i.e. something like Y=10X+10), and from there, it will measure how good or bad that guess was. It will then feed those results to the optimizer that will generate another guess. This process will then be repeated, with the logic being that the loss (or error) will go down over time, and as a result the ‘guess’ will get better and better.

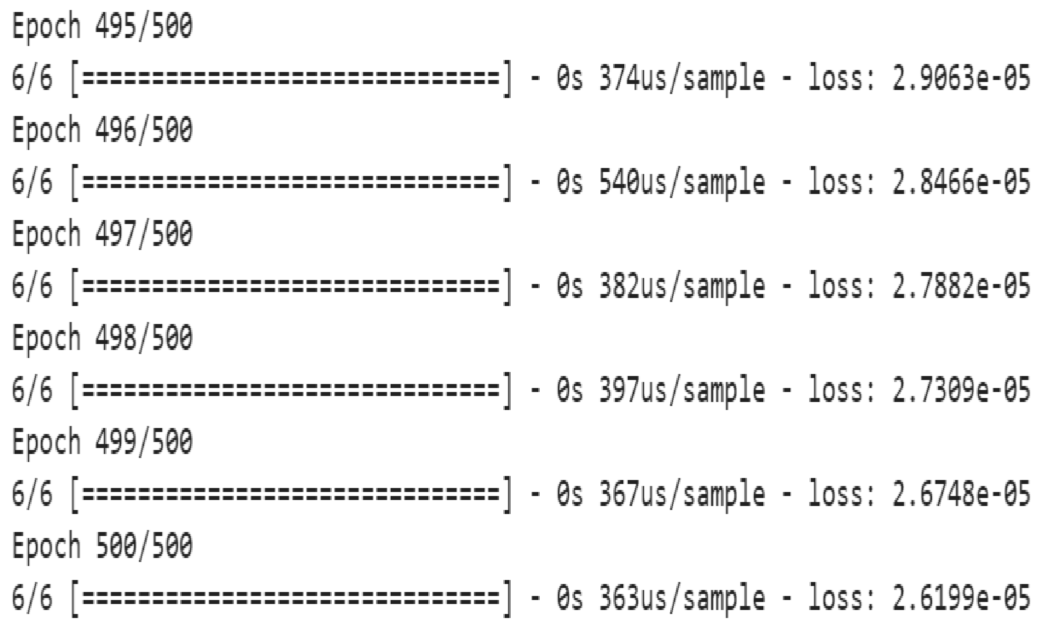
Figure 19 shows a screenshot of this running in a colab. Take a look at the loss values over time.



```
Epoch 1/500
6/6 [==============================] - 9s 2s/sample - loss: 3.2868
Epoch 2/500
6/6 [==============================] - 0s 652us/sample - loss: 2.7447
Epoch 3/500
6/6 [==============================] - 0s 323us/sample - loss: 2.3150
Epoch 4/500
6/6 [==============================] - 0s 411us/sample - loss: 1.9737
Epoch 5/500
6/6 [==============================] - 0s 306us/sample - loss: 1.7021
Epoch 6/500
6/6 [==============================] - 0s 496us/sample - loss: 1.4853
Epoch 7/500
6/6 [==============================] - 0s 470us/sample - loss: 1.3117
Epoch 8/500
6/6 [==============================] - 0s 405us/sample - loss: 1.1723
Epoch 9/500
6/6 [==============================] - 0s 616us/sample - loss: 1.0596
Epoch 10/500
6/6 [==============================] - 0s 669us/sample - loss: 0.9682
```

Figure 1-19. Training the Neural Network

We can see that over the first 10 epochs, the loss went from 3.2868 to 0.9682. After only 10 tries, the network was over 3x better than its initial guess. Then take a look at what happens by the 500th epoch. See Figure 20.



```
Epoch 495/500
6/6 [==============================] - 0s 374us/sample - loss: 2.9063e-05
Epoch 496/500
6/6 [==============================] - 0s 540us/sample - loss: 2.8466e-05
Epoch 497/500
6/6 [==============================] - 0s 382us/sample - loss: 2.7882e-05
Epoch 498/500
6/6 [==============================] - 0s 397us/sample - loss: 2.7309e-05
Epoch 499/500
6/6 [==============================] - 0s 367us/sample - loss: 2.6748e-05
Epoch 500/500
6/6 [==============================] - 0s 363us/sample - loss: 2.6199e-05
```

Figure 1-20. Training the Neural network - last 5 epochs

We can now see the loss is 2.61 by 10-5 .The loss has gotten so small, that it has pretty much figured out that the relationship between the numbers is Y=2X-1. The *machine* has *learned* the pattern between them.

Our last line of code then used the trained model to get a *prediction* like this:

```
print(model.predict([10.0]))
```

The term *prediction* is typically used when dealing with ML models. Don’t think of it as looking into the future! This term is used because we’re dealing with a certain amount of uncertainty. Think back to the activity detection scenario we spoke about earlier. When she was moving at a certain speed she was *probably* walking. Similarly when a model learns about the patterns between two things it will tell us what the answer *probably* is. In other words it is *predicting* the answer. (Later you’ll also see about *inference* where the model is picking one answer amongst many, and *inferring* that it has picked the correct one.)

What do you think the answer will be when we ask the model to predict the Y when X is 10. You might instantly think 19, but that's not correct. It will pick a value *very close* to 19. Why? Well there's several reasons. First of all our loss wasn't 0. It was still a very small amount, so we should expect any predicted answer to be off by a very small amount. Secondly, the neural network is trained on only a small amount of data -- in this case only 6 pairs of (X,Y). From this data it infers that the relationship is *probably* a straight line, but it can't *assume* that it is, thus, the answer will be *very close* to that of a straight line but not necessarily so.

Run the code for yourself to see what you get. I, for example got **18.977888** when I ran it. Your answer may differ slightly -- because when the neural network is first initialized there's a random element, so your initial guess would be slightly different from mine, and so on.

Seeing what the network learned

This is obviously a very simple scenario -- where we are matching Xs to Ys in a linear relationship. Neurons have *weight* and *bias* parameters that they learn, which makes a single neuron fine for learning a relationship like this. Namely when Y = 2X-1, the weight is 2, and the bias is -1. With TensorFlow we can actually take a look at the weights and biases that are learned, with a simple change to our code like this:

```
import tensorflow as tf
import numpy as np
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
l0 = Dense(units=1, input_shape=[1])
model = Sequential([l0])
model.compile(optimizer='sgd', loss='mean_squared_error')

xs = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)

model.fit(xs, ys, epochs=500)

print(model.predict([10.0]))
print("Here is what I learned: {}".format(l0.get_weights()))
```

The difference -- I created a variable called l0 to hold the Dense layer. Then, after the network finished learning, I could print out the values (or weights) that the layer learned.

The output was this:

```
Here is what I learned: [array([[1.9967953]], dtype=float32),
array([-0.9900647], dtype=float32)]
```

Thus, the learned relationship between X and Y was Y=1.9967953X-0.9900647.

This is pretty close to what we'd expect (Y=2X-1), and we could argue that it's even closer, because we are *assuming* that the relationship would hold for other values!

So that's it for your first, Hello World, of Machine Learning. This might seem like massive overkill for something as simple as a linear relationship between two values. And you'd be right. But the cool thing about this is that the pattern of code you created is the same pattern that's used for far more complex scenarios. You'll see these starting in Chapter 2, where you’ll explore some basic Computer Vision, where the machine will learn to ‘see’ patterns in pictures, and identify what’s in them!

Chapter 2. Taking the next step, an introduction to Computer Vision

In the previous chapter you saw the basics of how Machine Learning works, and how to get started with programming using neural networks to match data to labels, and from there to infer the rules that can be used to distinguish items. A logical next step for this would be to apply the 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.

Recognizing Clothing Items

So, for our first example, let’s consider what it takes to recognize items of clothing in an image. Imagine, for example, the items of clothing in Figure 1.

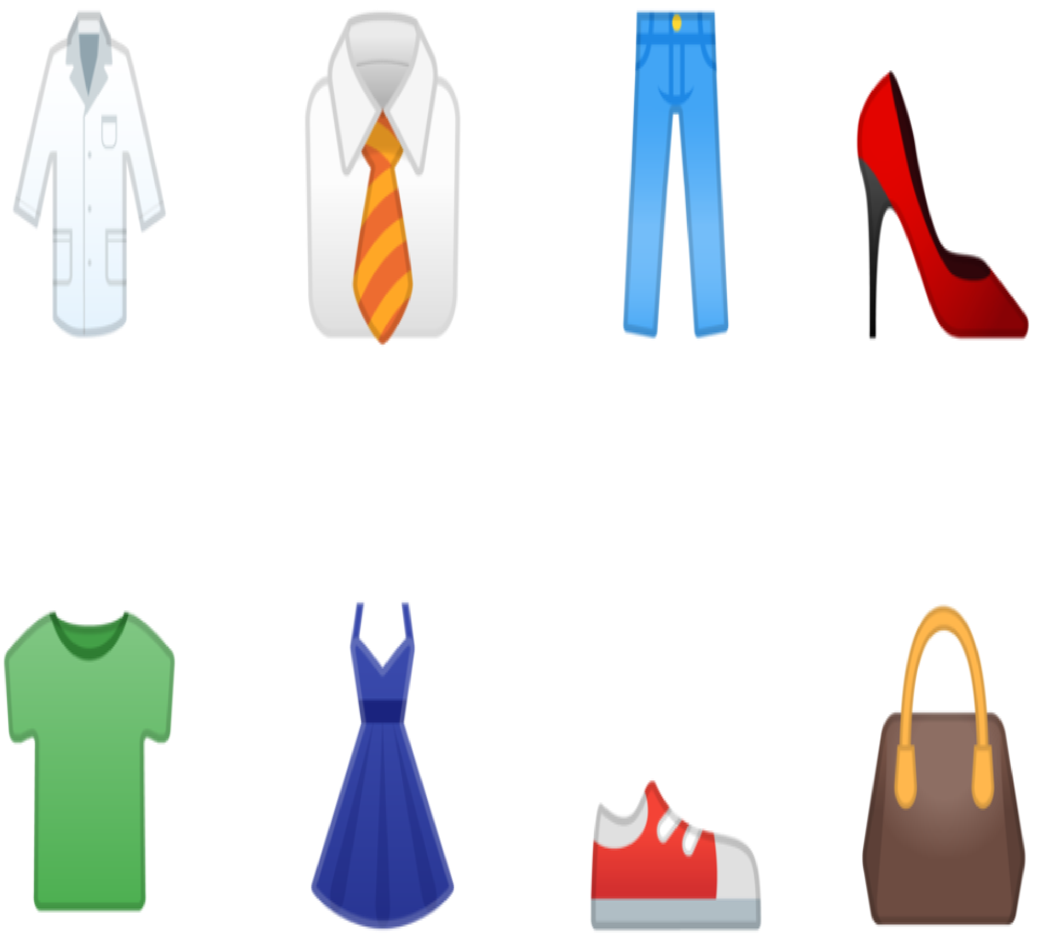


Figure 2-1. Examples of Clothing

There are a number of items of clothing 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 can we have a computer learn how to recognize computing the same way you did --

by having seen lots of items, and having experience with how they're used. Basically you learned by being exposed to lots of data.

So, 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 recognize items of clothing, using a well known set of data called Fashion MNIST.

The Data: Fashion MNIST

One of the foundational datasets for learning and benchmarking algorithms is called MNIST by Yann LeCun, Corinna Cortes and Christopher Burges. This dataset is comprised of 70,000 handwritten digits from 0 to 9. The images are 28x28 greyscale.

Fashion MNIST is designed to be a drop in replacement for MNIST that keeps the same number of records, the same dimension of images, and the same number of classes -- so instead of teh digits 0 through 9, Fashion MNIST has 10 different items of clothing.

Fashion MNIST has its home at zalando research at github.com/zalandoresearch/fashion-mnist.

You can see an example of the contents of Fashion MNIST in Figure 2. Here three lines are dedicated to each clothing item type.

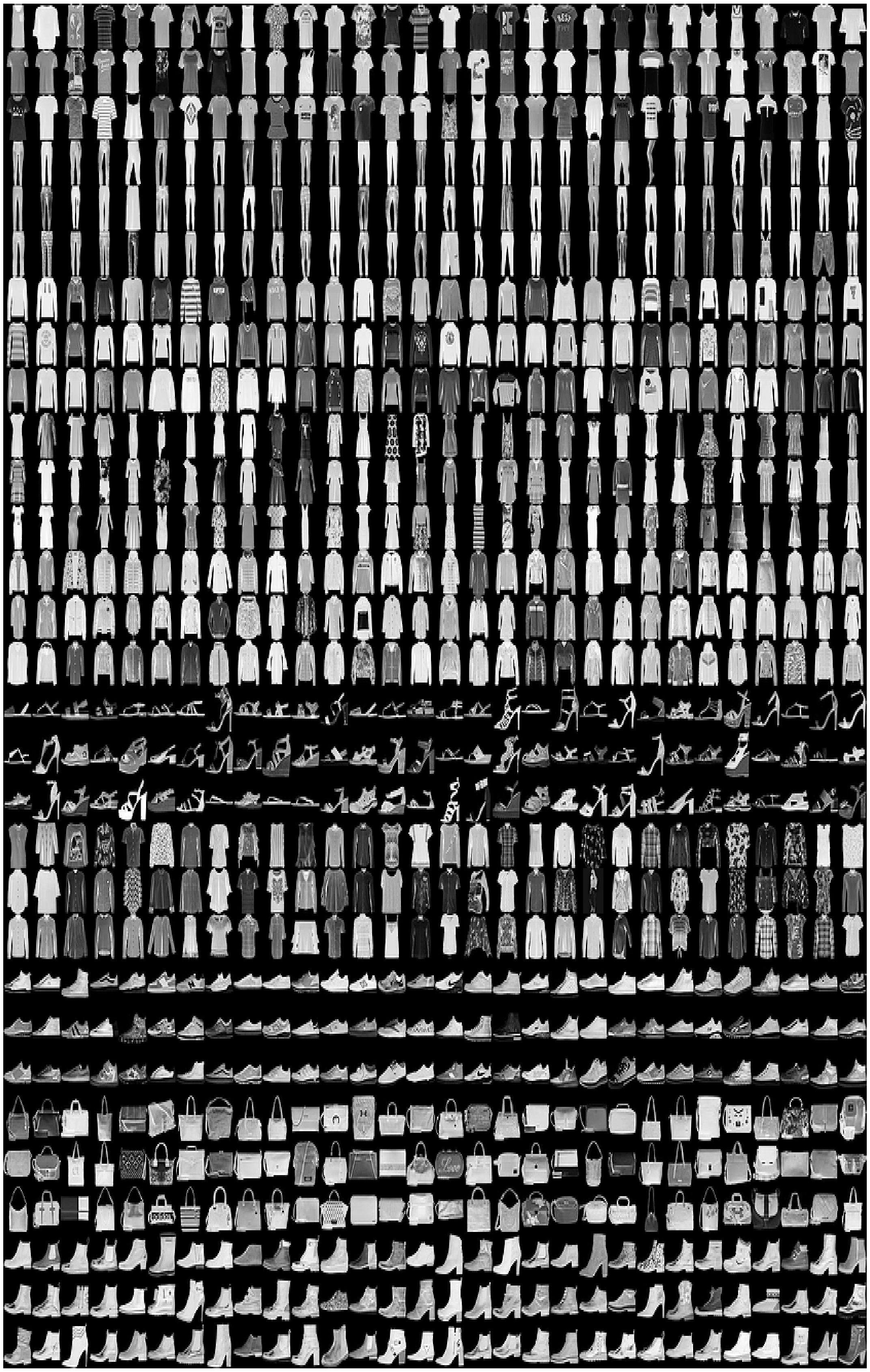


Figure 2-2. Exploring the Fashion MNIST dataset

It has a nice variety of clothing -- including shirts, trousers, dresses and lots of types of shoe! As you may notice, it’s monochrome, so each picture is comprised of a number of pixels valued between 0 and 255, making it a simpler-to-manage dataset.

You can see a close up on a particular image from the dataset in Figure 3.

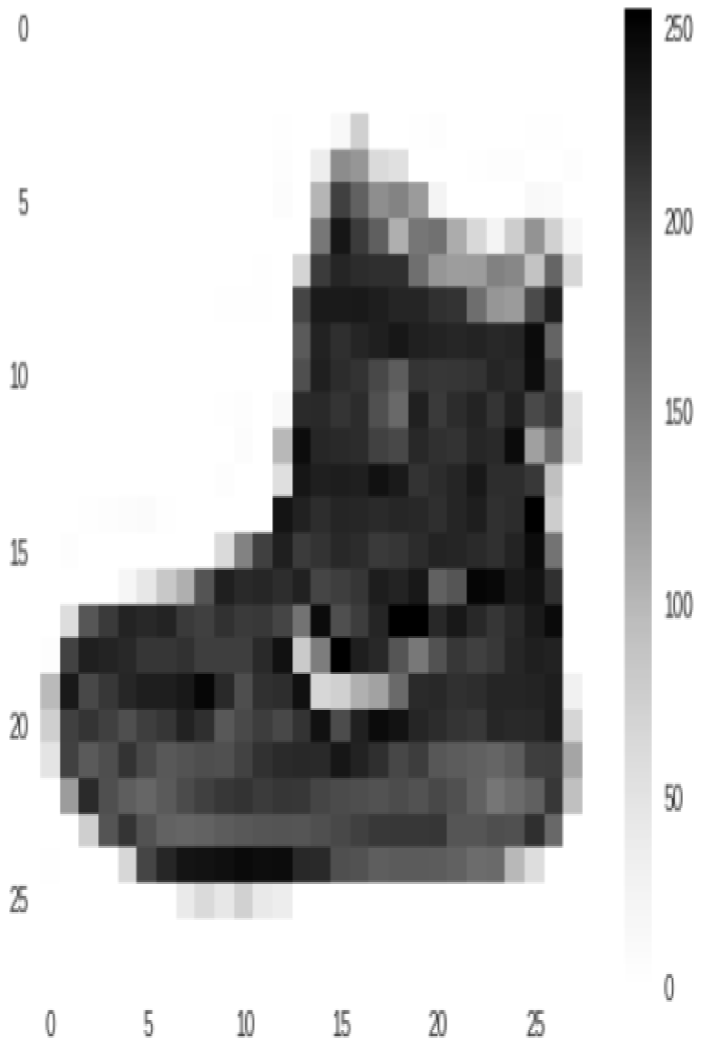


Figure 2-3. Close up on an Image in the Fashion MNIST dataset

Like any image it’s a rectangular grid of pixels. In this case it’s 28x28, and each pixel is simply a value between 0 and 255 as mentioned previously. So let’s now take a look at how you can use these pixel values with the learned 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. It was done using a very simple neural network with 1 layer and 1 neuron.

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

X

Y=mX+c

learn

(m,c)

X

Y=2X-1

m=2

c=-1

Figure 2-4. A Single Neuron learning a linear relationship

Now let's consider our image to be 784 values (28 x 28) between 0 and 255. They can be our X. We know that we have 10 different types of image 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.

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 like we did earlier.

But what we *can* do is have a number of 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. This could look like Figure 5.

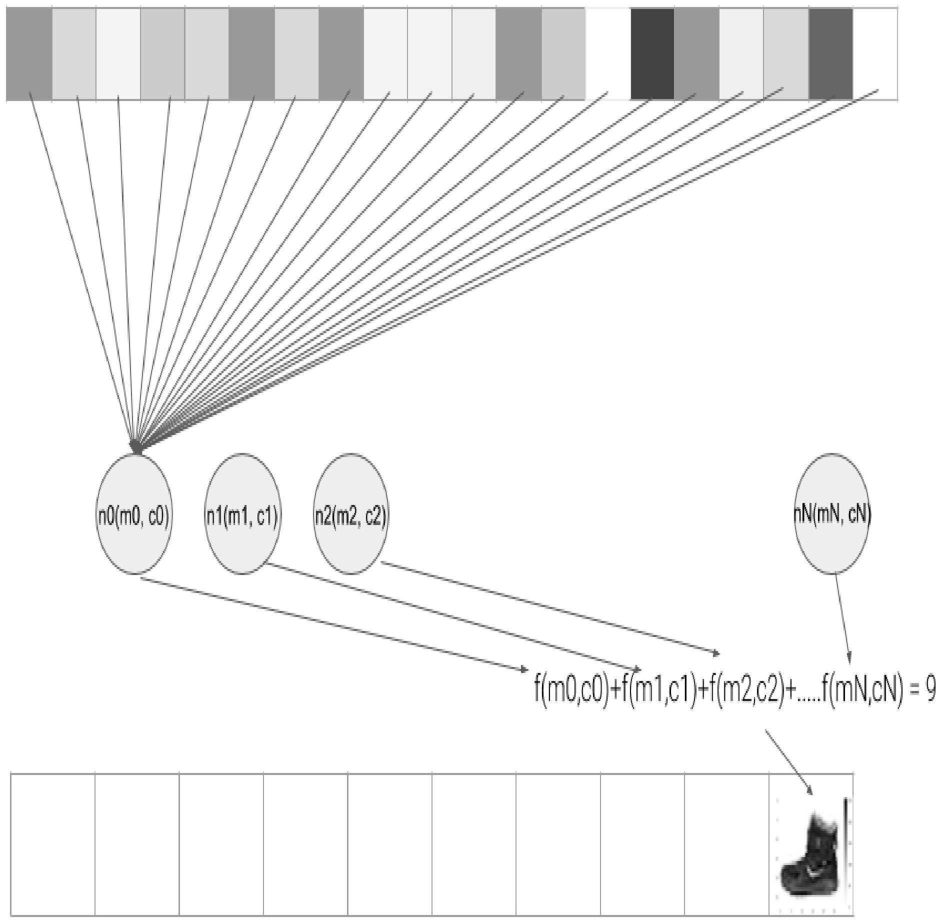


Figure 2-5. Extending our pattern for a more complex example

So, the grey boxes at the top of this diagram could 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, and Figure 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 we showed in Figure 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 look at what this looks like in code. First we’ll look at the design of the neural network shown in Figure 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** to specify that we had a number of layers. It only had 1 layer, but in this case we have multiple.

The first, **Flatten** isn’t a layer of neurons, but an input layer specification. Our images are 28x28 images, but we want them to be treated as a series of numeric values, like the grey boxes at the top of Figure 5. Flatten takes that ‘square’ value and turns it into a line.

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 5. 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, and the number of neurons doesn’t have a fixed rule. As you design the layers you want to pick the appropriate number of values that will work -- i.e. your model will actually learn. More neurons means it will run more slowly, as it has to learn more parameters. Less neurons means that it might not have sufficient functions to learn. It takes some experimentation over time to pick the right values. This process is typically called **hyperparameter tuning.**

You might notice that there’s also a specification called the ‘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 that you’ll see on middle layers is ‘relu’, which stands for **re**ctified **l**inear **u**nit. It’s a simple function that simply 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** 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 28x28 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 looked at the architecture of the neural network, let’s look at the complete code for training one with the Fashion MNIST data.

Here’s the complete code:

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

Let’s look through this piece by piece. First of all we’ll see a handy shortcut for accessing the data:

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

Keras has a number of built-in datasets that you can just 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 etc. It’s just a single line of code. This methodology has been improved upon using an API called **TensorFlow Datasets** (tensorflow.org/datasets), but for the purposes of these early chapters, to reduce the new concepts you need to learn, we’ll just use the 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 ones. So, the return from data.load_data() will give you an array of 60,000 28x28 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 28x28 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. If you recall all of the pixels in our images are grey scale between 0 and 255. Dividing by 255 then 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, normalization will work better. Often your network will not learn, and have massive errors when dealing with non-normalized data. The Y=2X-1 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 of course is where 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 is one of the arsenal of loss functions that’s 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 scenario. One major difference between this model and the one you created in chapter 1, is that instead of us trying to predict a single number, we’re picking a **category**. Our item of clothing will be 1 of 10 categories of clothing, and thus using a **categorical** loss function is the way to go. Hence, sparse categorical cross entropy is a good choice.

Similarly the optimizer to use is also an art in itself! The ‘adam’ optimizer is an evolution of the stochastic gradient descent (‘sgd’) that you 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 would 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. Here we just 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 reduced. In this case, it’s more useful to us to see how the network is learning by seeing the accuracy -- where it will return, from the training data, 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 5 epochs.

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

Finally we can do something new -- a single line of code to evaluate the model. 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, it ended up with an accuracy of about 89% after only 5 epochs.

But that's with the **training** data. The results of the model.evaluate on the test data will look something like this:

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

This shows us that with the test data, the accuracy of the model was 87.36%, which isn’t bad considering we only trained it for 5 epochs.

The question is always asked -- why is it **lower** for the test than it is for the training data? This is very commonly seen, and when you think about it, it’s because the neural network only really knows how to match the inputs it has been trained with to the outputs for those values. Our hope is that, given enough data, it will have enough examples from this to be able to generalize, knowing what a shoe or a dress looks like. But there will always be examples of ones that it hasn’t seen being 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 you have a good gauge at 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, if we print out the first of the classifications, and compare it to the test label, let’s see what we get:

```
[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 classifications 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 most of the values are very small, but the last value (array index 9) is the largest by far. 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 gets it right!

Try a few different values for yourself, and see if you can find any where the model gets it wrong!

Training for Longer - discovering Overfitting

In this case we trained for only 5 epochs and we got 89% accuracy on the training set and 87% on the test set. We went through the 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. We did that only 5 times, and got those pretty good results.

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

```
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 -- at 96.27%. For the test set we reach 88.6%

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

So even though the training results are much better, the test set results are negligibly better. This could lead us to a false sense of security that training our network for much longer can 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. It has become over-specialized 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 we've seen we hard coded the number of epochs we're training for. While that works, we might want to train until we reach a 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 would we approach this?

The easiest is to use a callback on the training.

Let's take a look at the updated code to use 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 take a look at what we’ve changed here.

First of all we create 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%), and if it is, we can stop training by saying self.model.stop_training = True.

Once we’ve specified this, we’ll 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 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 test set has hit 95% accuracy. Your number may change 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!
```

So you’ve now seen how to create your first, very basic, computer vision neural network. It was somewhat limited because of the data. All the images were 28x28, greyscale, 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.

You’ll do that with a process called *convolutions*, and 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 the world of software developers in how to build AI systems with Machine Learning. He’s a frequent contributor to the TensorFlow YouTube channel at 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.