```
def to_curr(anynum, len=0):
    """ Returns a number as a string with $ and commas. Length is optional """
    s = "Invalid amount"
    try:
        x = float(anynum)
    except ValueError:
        x= None
    if isinstance(x,float):
        s = '$' + f"{x:,.2f}"
        if len > 0:
            s=s.rjust(len)
    return s
```

You can create the same file yourself and name it `myfunctions.py` if you want to follow along. Notice that the file contains only functions. So if you run it, it won't do anything on the screen because there is no code in there that calls any of those functions.

To use those functions in any Python app or program you write, first make sure you copy that `myfunc.py` file to the same folder as the rest of the Python code that you're writing. Then, when you create a new page, you can import `myfunc` as a module just as you would any other module created by somebody else. Just use

```
import myfunc
```

You will have to use the module name in front of any of the functions that you call from that module. So if you want to make the code a little more readable, you can use this instead:

```
import myfunc as my
```

With that as your opening line, you can refer to any function in your custom module with `my.` as the prefix. For example, `my.to_date()` to call the `to_date` function. Here is a page that imports the module and then tests out all three functions using that `my` syntax:

```
# Import all the code from myfunc.py as my.
import myfunc as my

# Need dates in this code
from datetime import datetime as dt

# Some simple test data.
```

```
string_date="12/31/2019"
# Convert string date to datetime.date
print(my.to_date(string_date))

today = dt.today()
# Show today's date in mm/dd/yyyy format.
print(my.mdy(today))

dollar_amt=12345.678
# Show this big number in currency format.
print(my.to_curr(dollar_amt))
```

When you run this code, assuming it's all typed correctly with no errors, the output should look like this:

```
2019-12-31
'12/27/2018'
$12,345.68
```

We also mentioned earlier that you can skip using the prefix if you import items by name. In this case, that means you could call `to_date()` and `mdy()` and `to_curr()` without using the `my.` prefix. The first line of code would need to be

```
from myfunc import to_date, mdy, to_curr
```

The rest of the code would be the same as in the previous example, except you can leave off the `my.` prefixes as in the following code:

```
# Import all the code from myfunc.py by name.
from myfunc import to_date, mdy, to_curr

# Need dates in this code
from datetime import datetime as dt

# Some simple test data.

string_date="12/31/2019"
# Convert string date to datetime.date
print(to_date(string_date))

today = dt.today()
# Show today's date in mm/dd/yyyy format.
print(mdy(today))

dollar_amt=12345.678
# Show this big number in currency format.
print(to_curr(dollar_amt))
```

So that's it for Python libraries, packages, and modules. The whole business can be pretty confusing because people tend to use the terms interchangeably. And that's because they all represent code written by others that you're allowed to use in any Python code you write yourself. The only real difference is in size. A library may contain several packages. A package may contain several modules. The modules themselves will usually contain functions, classes, or some other pre-written chunks of code that you're free to use.

In the books and chapters to follow, you'll see lots of modules and classes because it's those things that make Python so modular and so applicable to many different types of work and study. But keep in mind that the core principles of the Python language that you've learned in these first three books apply everywhere, whether you're doing data science or AI or robotics. You'll just be using that core language to work with code that others have already written for that one specialized area.

4 Using Artificial Intelligence in Python

Contents at a Glance

IN THIS CHAPTER

- Understanding the concepts of AI
- Learning not to fear AI
- Understanding AI techniques
- Neural networks are your friends

Chapter 1
Exploring Artificial Intelligence

Artificial intelligence (AI) has been a much-maligned set of words over the past few years. The popular news media tends to take any small advance in AI out of context and proclaims "smart computers are here!" A simple example will suffice to show this point.

In 2017, Facebook engineers programmed two programs to value certain objects more than others (balls, blocks, and such) and then had the two programs, thorough a rules set and a language like English, negotiate with each other to maximize the acquisition of objects that the programs valued. The programs did not have a "language syntax checker" and because of the way the programs learned (a machine learning technique), the communication between the programs soon became syntactically incorrect English (a good example is when a program wanted something, it would say "I want" and the program logic decided that if one "I want" is good, then saying many "I want I want I want" should be better).

Of course, the news media reported this as a new language (it wasn't) and later, when the programs were shut down (because the experiment was finished, a normal thing to do), some pundit decided that the Facebook engineers had shut the programs down right before they had become sentient. Needless to say, this wasn't the case. Interestingly enough, all these programs are available to download for free, so we better hope that everybody shuts them down before they become sentient.

This chapter introduces you to the concept of AI and describes its limitations. We explore different AI techniques and then give a bit of history to put the current AI revolution in context.


REMEMBER

From Dictionary.com, "Artificial intelligence is the theory and development of computer systems able to perform tasks that normally require human intelligence, such as visual perception, speech recognition, decision-making, and translation between languages."

AI Is a Collection of Techniques

The point of this chapter is to demystify AI and to start to understand just how useful and cool these new AI techniques are. Note that we said "AI techniques" and not just "AI." General "AI," in the sense of how intelligent a person is, does not exist, but we do have a bunch of different algorithms, statistical techniques, and tools that can do some pretty impressive "humanlike" things, such as learning and adapting to the environment.

In this chapter, we show you three of these AI techniques. These are

- Neural networks
- Machine learning
- A TensorFlow classifier with Python

After this, you will become sentient. Oh wait, if you have bought this book, you are already over that threshold. Then, on to Python and AI.

Next, we briefly describe each of the major AI techniques and programs that we go through in the next three chapters.

Neural networks

Just by the name *neural networks*, you know we're going to be talking about brain cells. Human brains have billions of neurons (approximately 80 billion by some counts) and have roughly 10 times the amount of glial cells (which help neurons communicate and play a role in neuron placement). A real key to understanding how the brain works is connections. Each neuron has many connections with other neurons (up to 200,000 connections for some neurons in the brain), so it is not just the neurons that make people intelligent, it's how they are connected.

ARTIFICIAL INTELLIGENCE IS THE TECHNOLOGY OF THE FUTURE . . . AND ALWAYS WILL BE . . .

"What magical trick makes us intelligent? The trick is that there is no trick. The power of intelligence stems from our vast diversity, not from any single, perfect principle."

—MARVIN MINSKY, THE SOCIETY OF MIND (1987)

John Shovic: I had the honor of having lunch and spending an afternoon with Dr. Minsky in 1983 in Pocatello, Idaho. I was working as a VLSI (very large scale integrated circuit) design engineer at American Micro Systems, a company located in Pocatello, and I had just started my Ph.D. work part-time at the University of Idaho. My first AI class was under my belt, and I was beside myself that Dr. Marvin Minsky from the AI Lab at MIT was coming to Pocatello. I made a few phone calls and managed to get him to meet with two of us who had been in the AI class, for lunch and then an afternoon meeting. Looking back, I am amazed that he would take this kind of time for us, but that was the kind of guy he was.

It was like we were Rolling Stones fans and Mick Jagger was coming to town to spend time with us. Honestly, much as I would like to spend an afternoon with Mick Jagger, I would rather have Dr. Minsky.

I had just finished his book, Artificial Intelligence, and I was bursting with questions about his work. He was very interested in what we had learned from our AI class at the University of Idaho and he was quite complimentary about Dr. John Dickinson, the professor of the AI course, and his choice of subjects. I had a lot of enthusiasm about the topic and was thinking that I might make a career of it. Of all we talked about, I remember one thing clearly. He said (and this is paraphrased), "I started in AI about 20 years ago and I was convinced then that 20 years later we would have true AI. I now think we are still about 20 years off from having true AI."

Well, I remember thinking about that day 20 years later, in 2003, and realizing we still weren't there. At that point, I started wondering whether AI is the technology of the future and always will be. Here I am in 2019, another 16 years later, and we still don't have general AI. But one thing has changed in those 16 years. We now have a bunch of AI techniques, learning and searching techniques, that can do some pretty impressive things and may someday lead to that general AI that Dr. Minsky was talking about. Who knows? It may even take less than 20 years!

REMEMBER

Elephants have over 270 billion neurons, but they aren't nearly as densely connected as human neurons. Robert Metcalf, the founder of 3Com and an entrepreneur professor at the University of Texas, famously said (referring to Ethernet) that "the value of a telecommunications network is proportional to the square of the number of connected users of the system." With that squared value of connections coming in, we make up for those missing 200 or so billion elephant neurons by having ours much more densely connected.

However, is it any surprise that elephants have good memories and a complex social system?

AI researchers thought (and were right) that if we could reverse engineer neurons and their connections then we could use this information to construct computer models that could be used to make smarter programs. However, the limitations of the first popular model (Preceptrons) quickly led to general disappointment and the first "AI Winter" (see sidebar).

Over the past 30 years, however, much more information has gone into the development of neural networks and now they are quite useful in a number of ways, including convolutional neural networks and deep learning architectures.

THE AI WINTER

The term AI Winter was first discussed at the 1984 meeting of the American Association of Artificial Intelligence. Our friend from another sidebar, Dr. Marvin Minsky, warned the business community that enthusiasm for AI had spiraled out of control in the 1980s and that disappointment would certainly follow. Three years after he said that, the multi-billion dollar AI business collapsed. An *AI Winter* refers to the moment that investment and research dollars dry up. In economic terms, this is called a bubble, much like the dot-com bubble in 1999, the real estate bubble that collapsed in 2007, and the Dutch tulip bulb market bubble in Holland in 1637. Overhype leads to over-expectations, which lead to inevitable disappointment.

This happened in the 2000s again (partially because of the dot-com bubble) and researchers went to great lengths to avoid using the AI term in their proposals and papers. It seems to me that we are again approaching such a bubble, but none of us are anywhere near smart enough to know when winter is coming (pardon this, *Game of Thrones* fans). One of the big differences is that a number of AI techniques have moved into the mainstream (read consumer), such as Amazon Alexa and Google Home. So there's no doubt that some of the current exuberance and investment in AI is irrational, but we have gotten a lot of good tools out of it this time that we can continue to use even when the bubble pops.

Neural networks are good models for how brains learn and classify data. Given that, it is no surprise that neural networks show up in machine learning and other AI techniques.

A neural network consists of the following parts:

- The input layer of neurons
- An arbitrary amount of hidden layers of neurons
- An output layer of neurons connecting to the world
- A set of weights and biases between each neuron level
- A choice of activation function for each hidden layer of neurons

When you set up a network of neurons (and you do define the architecture and layout — creating an arbitrary network of connections has turned into what we call "evolutionary computing" — see sidebar) one of the most important choices you make is the activation function (sometimes known as the *threshold* for the neuron to fire). This activation function is one of the key things you define when you build your Python models of neurons in the next chapter.

Machine learning

Learning is the classic example of what it means to be human. We learn. We adapt.

Today, our AI researchers have brought quasi-human–level learning to computers in specific tasks, such as image recognition or sound processing ("Alexa, find me a new brain"), and there is hope to get to a level of similar learning in other tasks, like driving a car.

DEEP LEARNING

Deep learning has a pretty loose definition. The deep refers to the use of a number of layers of different learning to produce the desired result. It also generally uses neural networks at one level or another, or sometimes even multiple layers of networks.

The general idea is that one layer feeds the output to the next layer and so on. Each layer transforms the data into more abstract information. An example of this might be a picture of blocks being broken down by pixels, analyzed by a neural network and the resulting x, y coordinates of objects are passed down to another layer in the deep learning stack to do more processing for color.

Machine learning isn't fully automated. You can't say, "Computer, read this book" and expect it understand what it is reading. Currently using machine learning techniques require large amounts of human-classified and -selected data, data analysis, and training.

There are many different techniques used for machine learning. Some of these are

- Statistical analysis
- Neural networks
- Decision trees
- Evolutionary algorithms

Neural networks have many uses both in classification and in machine learning. *Machine learning* really refers to the different ways of using these techniques expert systems that ruled AI in the 1980s. In the late 1980s, John wrote his Ph.D. dissertation on how to build a deeply pipelined machine to speed up the evaluation of expert systems. Hardware acceleration of machine learning is an active area of research today.

In the following chapter, we show you how to use Python to build machines that demonstrate all the important tasks of machine learning.

AI IN SMARTPHONES

The current AI revolution is making its way into the handhelds. Both Samsung and Apple are racing to add these features to their phones. As of the writing of this book, the Samsung product requires processing off the phone and in the cloud. The A12 chip in the latest Apple smartphones is pretty impressive. It features a four core CPU (central processing units) and a six core GPU (graphics processing unit — a specialized CPU for AI and graphics techniques). It is a processor chip that is dedicated for AI applications, such as facial ID-recognition software. It also has a specialized neural network for a variety of applications.

The AI is capable of performing over 5 trillion operations per second. Take that, Animojis in your text messages.

Samsung is touting that they will have an even more advanced AI chip in their upcoming phones.

TensorFlow — A framework for deep learning

TensorFlow is an open-source, multi-machine set of API (application programming interfaces) used for building, learning, and doing research on deep learning. It hides a lot of the complexity of deep learning behind the curtain and makes the technology more accessible.

EVOLUTIONARY COMPUTING

Evolutionary computing is a set of computational algorithms and heuristics (a fancy word for rules of thumb) for global optimization that are inspired by biological evolution. It is derived from machine learning techniques but it differs in that evolutionary computing is generally used to solve problems whose solution is not known (*machine learning programs are taught what they are looking for*) by the programmer beforehand.

By using evolutionary computing algorithms, you can find exuberant but guided solutions to problems in a design space. It has been used to create solutions as diverse as better spacecraft antennas and improved bus routing. It uses very computationally intensive algorithms to search a problem space for solutions. It's very computationally intensive because you have a "population" of solutions that you have look at to see what is doing well and then select the new ones to move to the next generation. Survival of the fittest, in a real sense.

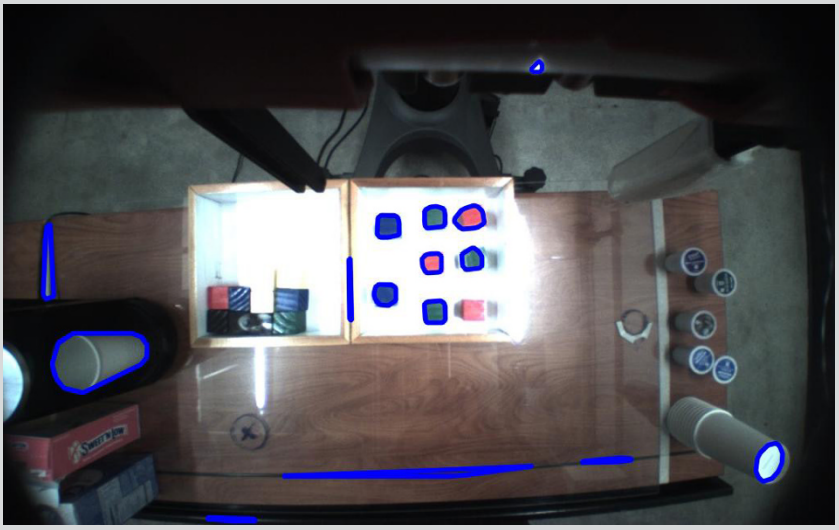
All these population evaluations are difficult for computers to process, which is why so many of the evolution algorithms really require massive parallel computing.

I had a student in my graduate Advanced Robotics class at the University of Idaho use evolutionary computing to allow a robot named Baxter to pick out red or yellow blocks from one box and move them to another box. He used photos from the robot and evolutionary computing to try to find an algorithm to identify and locate these blocks so Baxter could find them. In the picture you can see the two boxes whose blocks Baxter was to move. He did not succeed using evolutionary computing algorithms to find a solution. Why? Using evolutionary computing techniques to find a solution to an ill-defined and unconstrained problem requires exponentially greater computing

(continued)

(continued)

resources, and even these still might not come up with a solution. Smaller, constrained problems are a better match for evolutionary techniques.



TensorFlow started in 2011 as a proprietary AI environment in the Google Brain group. It has been extensively used within Google and, when it was released to open source, it was embraced by the AI community immediately. Of over 1,500 GitHub source repositories using TensorFlow, only five are from Google. Why is this significant? It shows just how well TensorFlow has been welcomed by the user community. GitHub is a popular website where people place their code for applications and projects in a way that other people can use and learn from them.

TensorFlow gets its name from the way the data is processed. A *tensor* is a multidimensional matrix of data, which is transformed by each of the TensorFlow layers it moves through. TensorFlow is extremely Python-friendly and can be used on many different machines and also in the cloud. TensorFlow is an easy-to-understand-and-use language to get your head into AI applications quickly.

TIP

So, TensorFlow is built on matrices. And another name for a matrix is a tensor, which is where the name *TensorFlow* comes from.

Current Limitations of AI

The key word in all of AI is *understanding*. A machine-learning algorithm has been taught to drive a car, for example, but the resulting program does not "understand" how to drive a car. The emphasis of the AI is to perform an analysis of the data, but the human controlling the data still has interpret the data and see if it fits the problem. Interpretation also goes beyond just the data. Humans often can tell whether the data or a conclusion is true or false, even when they can't really describe just how they know that. AI just accepts it as true.

Considering that we don't even understand a lot of human behavior and abilities ourselves, it is unlikely that anyone will be developing mathematical models of such behavior soon. And we need those models to start getting AI programs to achieve anything approaching human thought processes.

But with all its limitations, AI is very useful for exploring large problem spaces and finding "good" solutions. AI is not anywhere near a human . . . yet.

Now let's go and start using AI in Python!

IN THIS CHAPTER

- How machines learn
- Understanding the basics of machine learning
- Teaching a machine to learn something
- Using TensorFlow to do machine learning

Chapter 2

Building a Neural Network in Python

Neural networks and various other models of how the brain operates have been around as long as people have been talking about AI. Marvin Minsky, introduced in our last chapter, started mainline interest in modeling neurons with his seminal work in perceptrons in 1969. At the time, there was widespread "irrational exuberance" about how the perceptron was going to make AI practical very quickly. This attracted a good deal of venture capital to the area, but when many of these ventures failed, investment in neural networks dried up. It is like the concept of fashion in science. What's popular, sells.

Fast forward 30 years. There is now renewed interest in neural networks. Better models have been built, but the really important thing is that we now have real, useful, and economical applications based on neural networks. Will this lead to another bubble? Most assuredly, but this time the interest should continue because of the application areas that have been developed.

This chapter introduces you to the concept of neural networks and how to implement them in Python.

Understanding Neural Networks

These are the six attributes of a neural network:

- The input layer of neurons
- An arbitrary amount of hidden layers of neurons
- An output layer of neurons connecting to the world
- A set of weights and biases between each neuron level
- A choice of activation function for each hidden layer of neurons
- A loss function that will provide "overtraining" of the network

Figure 2-1 shows the architecture of a two-layer neural network. Note the three layers in this "two-layer" neural network: The input layer is typically excluded when you count a neural network's layers. By looking at this diagram, you can see that the neurons on each layer are connected to all the neurons of the next layer. Weights are given for each of the inter-neural connecting lines.

A *neuron*, as the word is used in AI, is a software model of a nervous system cell that behaves more or less like an actual brain neuron. The model uses numbers to make one neuron or another be more important to the results. These numbers are called *weights*.

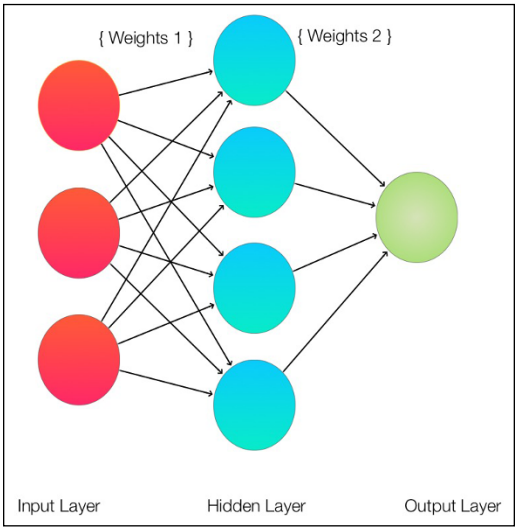


FIGURE 2-1: A two-layer neural network.

Layers of neurons

Figure 2-1 shows the input layer, the hidden layer (so called because it is not directly connected to the outside world), and the output layer. This is a very simple network; real networks can be much more complex with multiple more layers. In fact, deep learning gets its name from the fact that you have multiple hidden layers, in a sense increasing the "depth" of the neural network.

Note that you have the layers filter and process information from left to right in a progressive fashion. This is called a *feed-forward input* because the data feeds in only one direction.

So, now that we have a network, how does it learn? The neuron network receives an example and guesses at the answer (by using whatever default weights and biases that they start with). If the answer is wrong, it backtracks and modifies the weights and biases in the neurons and tries to fix the error by changing some values. This is called *backpropagation* and simulates what people do when performing a task using an iterative approach for trial and error.

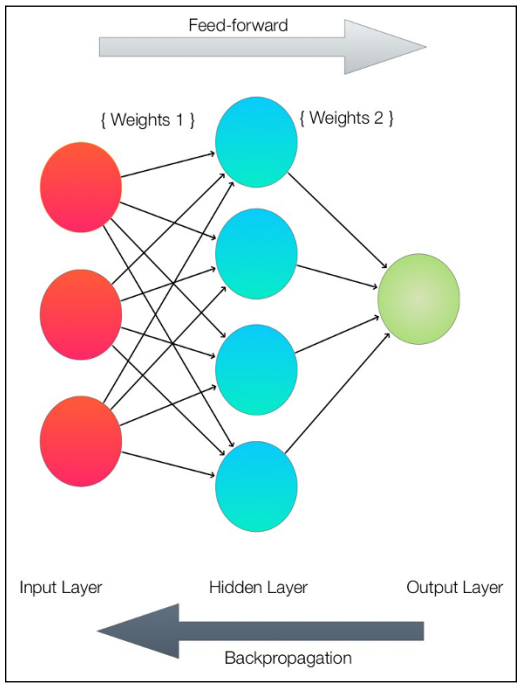


FIGURE 2-2: Feed-forward and backpropagation.

After you do this process many times, eventually the neural network begins to get better (learns) and provides better answers. Each one of these iterations is called an *epoch*. This name fits pretty well because sometimes it can take days or weeks to provide training to learn complex tasks.

An important point to make at this time is that although it may take days or weeks to train a neural net, after it is trained, we can duplicate it with little effort by copying the topology, weights, and biases of the trained network. When you have a trained neural net, you can use it easily again and again, until you need something different. Then, it is back to training.

Neural networks, as such, do model some types of human learning. However, humans have significantly more complex ways available for hierarchically categorizing objects (such as categorizing horses and pigs as animals) with little effort. Neural networks (and the whole deep learning field) are not very good at transferring knowledge and results from one type of situation to another without retraining.

Weights and biases

Looking at the network in Figure 2-1, you can see that the output of this neural networks is only dependent on the weights of the interconnection and also something we call the *biases* of the neurons themselves. Although the weights affect the steepness of the activation function curve (more on that later), the bias will shift the entire curve to the right or left. The choices of the weights and biases determines the strength of predictions of the individual neurons. Training the neural network involves using the input data to fine-tune the weights and biases.


TECHNICAL STUFF

BACKPROPAGATION

In the human brain, learning happens because of the addition of new connections (synapses) and the modification of those connections based on external stimuli.

The methods used to propagate from results to previous layers (also called *feedback*) have changed over the years in AI research, and some experts say that what is behind the latest surge of AI applications and the exit from the last "AI Winter" (see Book 4, Chapter 1) is the change of algorithms and techniques used for backpropagation.

Backpropagation is a pretty mathematically complex topic. For a more detailed description of the math and how it is used, check out *Machine Learning For Dummies*, by John Paul Mueller and Luca Massaron (Wiley).

The activation function

An activation function is an important topic to discuss in building our first neural network. This is a key part of our neuron model. This is the software function that determines whether information passes through or is stopped by the individual neuron. However, you don't just use it as a gate (open or shut), you use it as a function that transforms the input signal to the neuron in some useful way.

There are many types of activation functions available. For our simple neural network, we will use one of the most popular ones — the sigmoid function. A sigmoid function has a characteristic "S" curve, as shown in Figure 2-3.

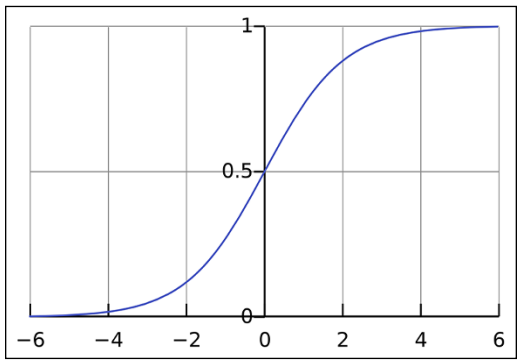


FIGURE 2-3: An example of a sigmoid function.

Remember we talked about neuron bias earlier in this chapter? If you apply a 1.0 value bias to the curve in Figure 2-3, then the whole curve shifts to the right, making the (0,0.5) point move to (1,0.5).

Loss function

The loss function is the last piece of the puzzle that needs to be explained. The loss function compares the result of our neural network to the desired results. Another way of thinking of this is that the loss function tells us how good our current results are.

This is the information that we are looking for to supply it to our backpropagation channel that will improve our neural network.

I am going to use a function that finds the derivative of the loss function towards our result (the slope of the curve is the first derivative calculus fans) to figure out what to do with our neuron weights. This is a major part of the "learning" activity of the network.

I have now talked about all the parts of our neural network, so let's go build an example.

Building a Simple Neural Network in Python

For you to build a neural network, you need to decide what you want it to learn. Here we'll choose a pretty simple goal: implement a three-input XOR gate. (That's an eXclusive OR gate.) Table 2-1 shows the function we want to implement in table form (showing our inputs and desired outputs of our neural network shown in Figure 2-1).

TABLE 2-1: The Truth Table (a Three-Input XOR Gate) for the Neural Network

X1	X2	X3	Y1
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

An Exclusive Or function returns a 1 only if all the inputs are either 0 or 1.

The neural-net Python code

We will be using the Python library called NumPy, which provides a great set of functions to help us organize our neural network and also simplifies the calculations.

USES OF XOR FUNCTIONS

XOR gates are used in a number of different applications, both in software and in hardware. You can use an XOR gate as part of a one-bit adder that adds one bit to another bit (and provides a carry bit to string them together to make big adders), as well as stringing them together to build a pseudo-random number generator.

The coolest application we know of an XOR gate has to do with coding algorithms and the Reed-Solomon error-correction algorithm. Reed-Solomon algorithms kind of mix up your data by using XOR gates and then adding some additional data (redundant data — kind of a mashup of your data), and then you have a more robust data to transmit long distances (like from Pluto, our former ninth planet), which can have all sorts of events that cause noise in the data, corrupting bits and bytes.

When you receive the data, you use XOR gates again to reconstruct the original data, correcting any errors (up to a point) so you have good data. This allows us to transmit data much further with less power, because with the Reed-Solomon code you become error-tolerant.

Why do we know anything about this? Because John worked with a team on chips for Reed-Solomon codes in the 1980s at the University of Idaho for NASA. Our chips and derivatives ended up on projects like the Hubble Space Telescope and on John's personal favorite, the New Horizons space probe, which visited Pluto and has recently visited Ultima Thule in the Oort Cloud. All the incredible pictures are going through all those little XOR gates.

NumPy — NUMERICAL PYTHON

NumPy is a Python library designed to simplify writing code for matrix mathematics (a matrix is also known as a *tensor*) in linear algebra. NumPy also includes a number of higher mathematical functions that are useful in various types of AI. The start of the development of NumPy goes all the back to 1995 and one of the original Python matrix algebra packages.

It is now the preferred library and is also a part of SciPy and MatPlotLib, two common scientific packages for analysis and visualization of data.

Our Python code using NumPy for the two-layer neural network of Figure 2-2 follows. Using nano (or your favorite text editor), open up a file called "2Layer-NeuralNetwork.py" and enter the following code:

```
# 2 Layer Neural Network in NumPy



import numpy as np

# X = input of our 3 input XOR gate
# set up the inputs of the neural network (right from the table)
X = np.array(([0,0,0],[0,0,1],[0,1,0], \
    [0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]), dtype=float)
# y = our output of our neural network
y = np.array(([1], [0],  [0],  [0],  [0], \
     [0],  [0],  [1]), dtype=float)

# what value we want to predict
xPredicted = np.array(([0,0,1]), dtype=float)

X = X/np.amax(X, axis=0) # maximum of X input array
# maximum of xPredicted (our input data for the prediction)
xPredicted = xPredicted/np.amax(xPredicted, axis=0)

# set up our Loss file for graphing

lossFile = open("SumSquaredLossList.csv", "w")

class Neural_Network (object):
  def __init__(self):
    #parameters
    self.inputLayerSize = 3  # X1,X2,X3
    self.outputLayerSize = 1 # Y1
    self.hiddenLayerSize = 4 # Size of the hidden layer

    # build weights of each layer
    # set to random values
    # look at the interconnection diagram to make sense of this
    # 3x4 matrix for input to hidden
    self.W1 = \
            np.random.randn(self.inputLayerSize, self.hiddenLayerSize)
    # 4x1 matrix for hidden layer to output
    self.W2 = \
            np.random.randn(self.hiddenLayerSize, self.outputLayerSize)
```

```
def feedForward(self, X):
  # feedForward propagation through our network
  # dot product of X (input) and first set of 3x4  weights
  self.z = np.dot(X, self.W1)

  # the activationSigmoid activation function - neural magic
  self.z2 = self.activationSigmoid(self.z)

  # dot product of hidden layer (z2) and second set of 4x1 weights
  self.z3 = np.dot(self.z2, self.W2)

  # final activation function - more neural magic
  o = self.activationSigmoid(self.z3)
  return o

 def backwardPropagate(self, X, y, o):
  # backward propagate through the network
  # calculate the error in output
  self.o_error = y - o

  # apply derivative of activationSigmoid to error
  self.o_delta = self.o_error*self.activationSigmoidPrime(o)

  # z2 error: how much our hidden layer weights contributed to output
  # error
  self.z2_error = self.o_delta.dot(self.W2.T)

  # applying derivative of activationSigmoid to z2 error
  self.z2_delta = self.z2_error*self.activationSigmoidPrime(self.z2)

  # adjusting first set (inputLayer --> hiddenLayer) weights
  self.W1 += X.T.dot(self.z2_delta)
  # adjusting second set (hiddenLayer --> outputLayer) weights
  self.W2 += self.z2.T.dot(self.o_delta)

def trainNetwork(self, X, y):
  # feed forward the loop
  o = self.feedForward(X)
  # and then back propagate the values (feedback)
  self.backwardPropagate(X, y, o)


def activationSigmoid(self, s):
  # activation function
  # simple activationSigmoid curve as in the book
  return 1/(1+np.exp(-s))
```

```
  def activationSigmoidPrime(self, s):
    # First derivative of activationSigmoid
    # calculus time!
    return s * (1 - s)


  def saveSumSquaredLossList(self,i,error):
    lossFile.write(str(i)+","+str(error.tolist())+'\n')

  def saveWeights(self):
    # save this in order to reproduce our cool network
    np.savetxt("weightsLayer1.txt", self.W1, fmt="%s")
    np.savetxt("weightsLayer2.txt", self.W2, fmt="%s")

  def predictOutput(self):
    print ("Predicted XOR output data based on trained weights: ")
    print ("Expected (X1-X3): \n" + str(xPredicted))
    print ("Output (Y1): \n" + str(self.feedForward(xPredicted)))

myNeuralNetwork = Neural_Network()
trainingEpochs = 1000
#trainingEpochs = 100000

for i in range(trainingEpochs): # train myNeuralNetwork 1,000 times
  print ("Epoch # " + str(i) + "\n")
  print ("Network Input : \n" + str(X))
  print ("Expected Output of XOR Gate Neural Network: \n" + str(y))
  print ("Actual  Output from XOR Gate Neural Network: \n" + \
         str(myNeuralNetwork.feedForward(X)))
  # mean sum squared loss
  Loss = np.mean(np.square(y - myNeuralNetwork.feedForward(X)))
  myNeuralNetwork.saveSumSquaredLossList(i,Loss)
  print ("Sum Squared Loss: \n" + str(Loss))
  print ("\n")
  myNeuralNetwork.trainNetwork(X, y)

myNeuralNetwork.saveWeights()
myNeuralNetwork.predictOutput()
```

Breaking down the code

Some of the following code is a little obtuse the first time through, so we will give you some explanations.

```
# 2 Layer Neural Network in NumPy



import numpy as np
```

If you get an import error when running the preceding code, install the NumPy Python library. To do so on a Raspberry Pi (or an Ubuntu system), type the following in a terminal window:

```
sudo apt-get install python3-numpy
```

Next, we define all eight possibilities of our X1–X3 inputs and the Y1 output from Table 2-1.

```
# X = input of our 3 input XOR gate
# set up the inputs of the neural network (right from the table)
X = np.array(([0,0,0],[0,0,1],[0,1,0], \
    [0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]), dtype=float)
# y = our output of our neural network
y = np.array(([1], [0],  [0],  [0],  [0], \
     [0],  [0],  [1]), dtype=float)
```

We pick a value to predict (we predict them all, but this is the particular answer we want at the end).

```
# what value we want to predict
xPredicted = np.array(([0,0,1]), dtype=float)

X = X/np.amax(X, axis=0) # maximum of X input array
# maximum of xPredicted (our input data for the prediction)
xPredicted = xPredicted/np.amax(xPredicted, axis=0)
```

Save out our Sum Squared Loss results to a file for use by Excel per epoch.

```
# set up our Loss file for graphing

lossFile = open("SumSquaredLossList.csv", "w")
```

Build the Neural_Network class for our problem. Figure 2-2 shows the network we are building. You can see that each of the layers are represented by a line in the network.

```
class Neural_Network (object):
  def __init__(self):
    #parameters
    self.inputLayerSize = 3  # X1,X2,X3
    self.outputLayerSize = 1 # Y1
    self.hiddenLayerSize = 4 # Size of the hidden layer
```

Set all the network weights to random values to start.

```
    # build weights of each layer
    # set to random values
    # look at the interconnection diagram to make sense of this
    # 3x4 matrix for input to hidden
    self.W1 = \
            np.random.randn(self.inputLayerSize, self.hiddenLayerSize)
    # 4x1 matrix for hidden layer to output
    self.W2 = \
            np.random.randn(self.hiddenLayerSize, self.outputLayerSize)
```

Our feedForward function implements the feed-forward path through the neural network. This basically multiplies the matrices containing the weights from each layer to each layer and then applies the sigmoid activation function.

```
  def feedForward(self, X):
    # feedForward propagation through our network
    # dot product of X (input) and first set of 3x4  weights
    self.z = np.dot(X, self.W1)

    # the activationSigmoid activation function - neural magic
    self.z2 = self.activationSigmoid(self.z)

    # dot product of hidden layer (z2) and second set of 4x1 weights
    self.z3 = np.dot(self.z2, self.W2)

    # final activation function - more neural magic
    o = self.activationSigmoid(self.z3)
    return o
```

And now we add the backwardPropagate function that implements the real trial-and-error learning that our neural network uses.

```
   def backwardPropagate(self, X, y, o):
    # backward propagate through the network
    # calculate the error in output
    self.o_error = y - o

    # apply derivative of activationSigmoid to error
    self.o_delta = self.o_error*self.activationSigmoidPrime(o)

    # z2 error: how much our hidden layer weights contributed to output
    # error
    self.z2_error = self.o_delta.dot(self.W2.T)
```

```
    # applying derivative of activationSigmoid to z2 error
    self.z2_delta = self.z2_error*self.activationSigmoidPrime(self.z2)

    # adjusting first set (inputLayer --> hiddenLayer) weights
    self.W1 += X.T.dot(self.z2_delta)
    # adjusting second set (hiddenLayer --> outputLayer) weights
    self.W2 += self.z2.T.dot(self.o_delta)
```

To train the network for a particular epoch, we call both the `backwardPropagate` and the `feedForward` functions each time we train the network.

```
def trainNetwork(self, X, y):
    # feed forward the loop
    o = self.feedForward(X)
    # and then back propagate the values (feedback)
    self.backwardPropagate(X, y, o)
```

The sigmoid activation function and the first derivative of the sigmoid activation function follows.

```
def activationSigmoid(self, s):
    # activation function
    # simple activationSigmoid curve as in the book
    return 1/(1+np.exp(-s))

def activationSigmoidPrime(self, s):
    # First derivative of activationSigmoid
    # calculus time!
    return s * (1 - s)
```

Next, save the epoch values of the loss function to a file for Excel and the neural weights.

```
def saveSumSquaredLossList(self,i,error):
    lossFile.write(str(i)+","+str(error.tolist())+'\n')

def saveWeights(self):
    # save this in order to reproduce our cool network
    np.savetxt("weightsLayer1.txt", self.W1, fmt="%s")
    np.savetxt("weightsLayer2.txt", self.W2, fmt="%s")
```

Next, we run our neural network to predict the outputs based on the current trained weights.

```
  def predictOutput(self):
    print ("Predicted XOR output data based on trained weights: ")
    print ("Expected (X1-X3): \n" + str(xPredicted))
    print ("Output (Y1): \n" + str(self.feedForward(xPredicted)))

myNeuralNetwork = Neural_Network()
trainingEpochs = 1000
#trainingEpochs = 100000
```

The following is the main training loop that goes through all the requested epochs. Change the variable trainingEpochs above to vary the number of epochs you would like to train your network.

```
for i in range(trainingEpochs): # train myNeuralNetwork 1,000 times
 print ("Epoch # " + str(i) + "\n")
 print ("Network Input : \n" + str(X))
 print ("Expected Output of XOR Gate Neural Network: \n" + str(y))
 print ("Actual Output from XOR Gate Neural Network: \n" + \
     str(myNeuralNetwork.feedForward(X)))
 # mean sum squared loss
 Loss = np.mean(np.square(y - myNeuralNetwork.feedForward(X)))
 myNeuralNetwork.saveSumSquaredLossList(i,Loss)
 print ("Sum Squared Loss: \n" + str(Loss))
 print ("\n")
 myNeuralNetwork.trainNetwork(X, y)
```

Save the results of your training for reuse and predict the output of our requested value.

```
myNeuralNetwork.saveWeights()
myNeuralNetwork.predictOutput()
```

Running the neural-network code

At a command prompt, enter the following command:

```
python3 2LayerNeuralNetworkCode.py
```

You will see the program start stepping through 1,000 epochs of training, printing the results of each epoch, and then finally showing the final input and output. It also creates the following files of interest:

- **weightsLayer1.txt:** This file contains the final trained weights for input-layer-to-hidden-layer connections (a 4x3 matrix).
- **weightsLayer2.txt:** This file contains the final trained weights for hidden-layer-to-output-layer connections (a 1x4 matrix).
- **SumSquaredLossList.csv:** This is a comma-delimited file containing the epoch number and each loss factor at the end of each epoch. We use this to graph the results across all epochs.

Here is the final output of the program for the last epoch (999 since we start at 0).

```
Epoch # 999

Network Input :
[[0. 0. 0.]
 [0. 0. 1.]
 [0. 1. 0.]
 [0. 1. 1.]
 [1. 0. 0.]
 [1. 0. 1.]
 [1. 1. 0.]
 [1. 1. 1.]]
Expected Output of XOR Gate Neural Network:
[[1.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [1.]]
Actual Output from XOR Gate Neural Network:
[[0.93419893]
 [0.04425737]
 [0.01636304]
 [0.03906686]
 [0.04377351]
 [0.01744497]
 [0.0391143 ]
 [0.93197489]]
Sum Squared Loss:
0.0020575319565093496


Predicted XOR output data based on trained weights:
Expected (X1-X3):
[0. 0. 1.]
```

```
Output (Y1):
[0.04422615]
```

At the bottom, you see our expected output is 0. 04422615, which is quite close, but not quite, the expected value of 0. If you compare each of the expected outputs to the actual output from the network, you see they all match pretty closely. And every time you run it the results will be slightly different because you initialize the weights with random numbers at the start of the run.

The goal of a neural-network training is not to get it exactly right — only right within a stated tolerance of the correct result. For example, if we said that any output above 0.9 is a 1 and any output below 0.1 is a 0, then our network would have given perfect results.

The Sum Squared Loss is a measure of all the errors of all the possible inputs.

If we graph the Sum Squared Loss versus the epoch number, we get the graph shown in Figure 2-4. You can see we get better quite quickly and then it tails off. 1,000 epochs are fine for our stated problem.

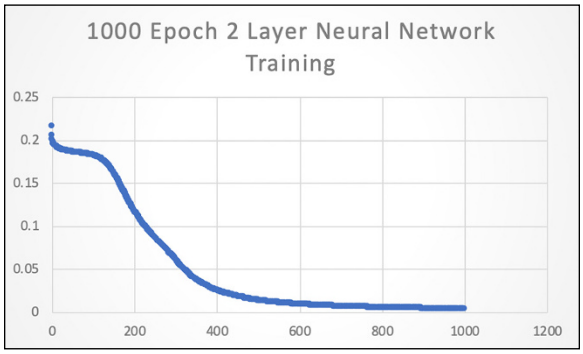


FIGURE 2-4: The Loss function during training.

One more experiment. If you increase the number of epochs to 100,000, then the numbers are better still, but our results, according to our accuracy criteria (> 0.9 = 1 and < 0.1 = 0) were good enough in the 1,000 epoch run.

```
Epoch # 99999

Network Input :
[[0. 0. 0.]
 [0. 0. 1.]
 [0. 1. 0.]
 [0. 1. 1.]
```

```
 [1. 0. 0.]
 [1. 0. 1.]
 [1. 1. 0.]
 [1. 1. 1.]]
Expected Output of XOR Gate Neural Network:
[[1.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [1.]]
Actual Output from XOR Gate Neural Network:
[[9.85225608e-01]
 [1.41750544e-04]
 [1.51985054e-04]
 [1.14829204e-02]
 [1.17578404e-04]
 [1.14814754e-02]
 [1.14821256e-02]
 [9.78014943e-01]]
Sum Squared Loss:
0.00013715041859631841


Predicted XOR output data based on trained weights:
Expected (X1-X3):
[0. 0. 1.]
Output (Y1):
[0.00014175]
```

Using TensorFlow for the same neural network

TensorFlow is a Python package that is also designed to support neural networks based on matrices and flow graphs similar to NumPy. It differs from NumPy in one major respect: TensorFlow is designed for use in machine learning and AI applications and so has libraries and functions designed for those applications.

TensorFlow gets its name from the way it processes data. A *tensor* is a multi-dimensional matrix of data, and this is transformed by each TensorFlow layer it moves through. TensorFlow is extremely Python-friendly and can be used on many different machines and also in the cloud. As you learned in the previous section on neural networks in Python, neural networks are data-flow graphs and are

implemented in terms of performing operations matrix of data and then moving the resulting data to another matrix. Because matrices are tensors and the data flows from one to another, you can see where the TensorFlow name comes from.

TensorFlow is one of the best-supported application frameworks with APIs (application programming interfaces) gpt Python, C++, Haskell, Java, Go, Rust, and there's also a third-party package for R called tensorflow.

Installing the TensorFlow Python library

For Windows, Linux, and Raspberry Pi, check out the official TensorFlow link at `https://www.tensorflow.org/install/pip`.

TensorFlow is a typical Python3 library and API (applications programming interface). TensorFlow has a lot of dependencies that will be also installed by following the tutorial referenced above.


TECHNICAL STUFF

INTRODUCING TENSORS

You already know from Book 4, Chapter 1 that tensors are multidimensional matrices of data. But an additional discussion will be helpful in getting your head wrapped around the vocabulary of tensors. Neural networks are data-flow graphs and are implemented in terms of performing operations matrix of data and then moving the resulting data to another matrix. Tensor is another name for matricies.

- Scalars: A *scalar* can be thought of as a single piece of data. There is one and only one piece of data associated with a scalar. For example, a value of 5 meters or 5 meters/second are examples of a scalar, as is 45 degrees Fahrenheit or 21 degrees Celsius. You can think of a scalar as a point on a plane.
- Vectors: A *vector* differs from a scalar by the fact that it contains at least two pieces of information. An example of a vector is 5 meters east — this describes a distance and a direction. A vector is a one-dimensional matrix (2x1, for example). You can think of a vector as an arrow located on a plane. It looks like a ray on the plane.

  Plane vectors are the simplest form of tensor. If you look at a 3x1 vector, now you have coordinates for 3D space: *x*, *y*, and *z*.
- Tensors: Vectors are special cases of tensors. A *tensor* is a matrix that can be characterized by magnitude and multiple directions. Scalars can be recognized as individual numbers, vectors as ordered sets of numbers, and tensors by a single or multidimensional array of numbers. Here is a great non-mathematical introduction to tensors: `https://www.youtube.com/watch?v=f5liqUk0ZTw`.

Building a Python Neural Network in TensorFlow

For our neural-network example in TensorFlow, we will use the same network that we used to implement an XOR gate with Python. Figure 2-1 shows the two-layer neural network we used; Figure 2-5 shows the new three-layer neural network. Refer to Table 2-1 for the truth table for both networks.

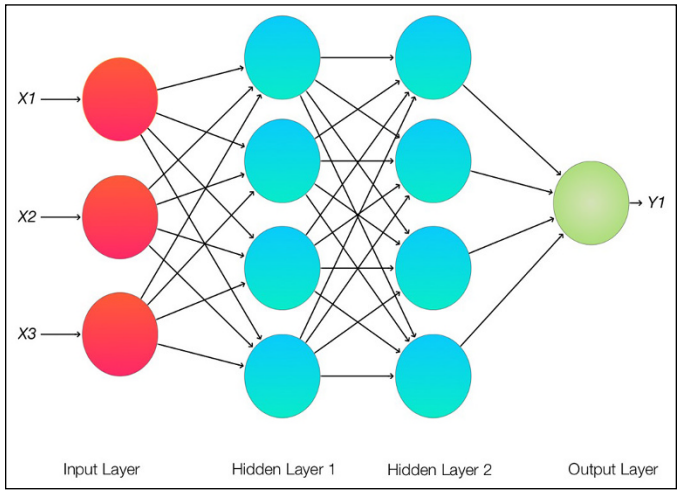


FIGURE 2-5: Our TensorFlow three-layer neural network.

TensorFlow is a Python-friendly application framework and collection of functions designed for AI uses, especially with neural networks and machine learning. It uses Python to provide a user-friendly convenient front-end while executing those applications by high performance C++ code.

Keras is an open source neural-network library that enables fast experimentation with neural networks, deep learning, and machine learning. In 2017, Google decided to natively support Keras as the preferred interface for TensorFlow. Keras provides the excellent and intuitive set of abstractions and functions whereas TensorFlow provides the efficient underlying implementation.

The five steps to implementing a neural network in Keras with TensorFlow are

1. Load and format your data.
2. Define your neural network model and layers.
3. Compile the model.

4. Fit and train your model.
5. Evaluate the model.

Loading your data

This step is pretty trivial in our model but is often the most complex and difficult part of building your entire program. You have to look at your data (whether an XOR gate or a database of factors affecting diabetic heart patients) and figure out how to map your data and the results to get to the information and predictions that you want.

Defining your neural-network model and layers

Defining your network is one of the primary advantages of Keras over other frameworks. You basically just construct a stack of the neural layers you want your data to flow through. Remember TensorFlow is just that. Your matrices of data flowing through a neural network stack. Here you chose the configuration of your neural layer and activation functions.

Compiling your model

Next you compile your model which hooks up your Keras layer model with the efficient underlying (what they call the back-end) to run on your hardware. You also choose what you want to use for a loss function.

Fitting and training your model

This is where the real work of training your network takes place. You will determine how many epochs you want to go through. It also accumulates the history of what is happening through all the epochs, and we will use this to create our graphs.

Our Python code using TensorFlow, NumPy, and Keras for the two-layer neural network of Figure 2-6 follows. Using nano (or your favorite text editor), open up a file called `TensorFlowKeras.py` and enter the following code:

```
import tensorflow as tf

from tensorflow.keras import layers

from tensorflow.keras.layers import Activation, Dense
```

```
import numpy as np

# X = input of our 3 input XOR gate
# set up the inputs of the neural network (right from the table)
X = np.array(([0,0,0],[0,0,1],[0,1,0],
            [0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]), dtype=float)
# y = our output of our neural network
y  = np.array(([1], [0],  [0],  [0],  [0],
             [0],  [0],  [1]), dtype=float)


model = tf.keras.Sequential()

model.add(Dense(4, input_dim=3, activation='relu',
    use_bias=True))
#model.add(Dense(4,  activation='relu', use_bias=True))
model.add(Dense(1, activation='sigmoid', use_bias=True))

model.compile(loss='mean_squared_error',
        optimizer='adam',
        metrics=['binary_accuracy'])

print (model.get_weights())

history = model.fit(X, y, epochs=2000,
        validation_data = (X, y))


model.summary()


# printing out to file
loss_history = history.history["loss"]
numpy_loss_history = np.array(loss_history)
np.savetxt("loss_history.txt", numpy_loss_history,
        delimiter="\n")

binary_accuracy_history = history.history["binary_accuracy"]
numpy_binary_accuracy = np.array(binary_accuracy_history)
np.savetxt("binary_accuracy.txt", numpy_binary_accuracy, delimiter="\n")



print(np.mean(history.history["binary_accuracy"]))

result = model.predict(X ).round()

print (result)
```

After looking at the code, we will run the neural network and then evaluate the model and results.

Breaking down the code

The first thing to notice about our code is that it is much simpler than our two-layer model strictly in Python used earlier in this chapter. That's the magic of TensorFlow/Keras. Go back and compare this code to the code for our two-layer network in pure Python. This is much simpler and easier to understand.

First, we import all the libraries you will need to run our example two-layer model. Note that TensorFlow includes Keras by default. And once again we see our friend NumPy as the preferred way of handling matrices.

```
import tensorflow as tf

from tensorflow.keras import layers

from tensorflow.keras.layers import Activation, Dense

import numpy as np
```

Step 1, load and format your data. In this case, we just set up the truth table for our XOR gate in terms of NumPy arrays. This can get much more complex when you have large, diverse, cross-correlated sources of data.

```
# X = input of our 3 input XOR gate
# set up the inputs of the neural network (right from the table)
X = np.array(([0,0,0],[0,0,1],[0,1,0],
            [0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]), dtype=float)
# y = our output of our neural network
y  = np.array(([1], [0],  [0],  [0],  [0],
              [0],  [0],  [1]), dtype=float)
```

Step 2, define your neural-network model and layers. This is where the real power of Keras shines. It is very simple to add more neural layers, and to change their size and their activation functions. We are also applying a bias to our activation function (`relu`, in this case, with our friend the sigmoid for the final output layer), which we did not do in our pure Python model.

See the commented `model.add` statement below? When we go to our three-layer neural-network example, that is all we have to change by uncommenting it.

```
model = tf.keras.Sequential()

model.add(Dense(4, input_dim=3, activation='relu',
    use_bias=True))
#model.add(Dense(4, activation='relu', use_bias=True))
model.add(Dense(1, activation='sigmoid', use_bias=True))
```

Step 3, compile your model. We are using the same loss function that we used in our pure Python implementation, `mean_squared_error`. New to us is the optimizer, ADAM (a method for stochastic optimization) is a good default optimizer. It provides a method for efficiently descending the gradient applied to the weights of the layers.

One thing to note is what we are asking for in terms of metrics. `binary_accuracy` means we are comparing our outputs of our network to either a 1 or a 0. You will see values of, say, 0.75, which, since we have eight possible outputs, means that six out of eight are correct. It is exactly what you would expect from the name.

```
model.compile(loss='mean_squared_error',
        optimizer='adam',
        metrics=['binary_accuracy'])
```

Here we print out all the starting weights of our model. Note that they are assigned with a default random method, which you can seed (to do the same run with the same starting weights time after time) or you can change the way they are added.

```
print (model.get_weights())
```

Step 4, fit and train your model. We chose the number of epochs so we would converge to a binary accuracy of 1.0 most of the time. Here we load the NumPy arrays for the input to our network (`X`) and our expected output of the network (`y`). The `validation_data` parameter is used to compare the outputs of your trained network in each epoch and generates `val_acc` and `val_loss` for your information in each epoch as stored in the `history` variable.

```
history = model.fit(X, y, epochs=2000,
        validation_data = (X, y))
```

Here we print a summary of your model so you can make sure it was constructed in the way expected.

```
model.summary()
```

Next, we print out the values from the `history` variable that we would like to graph.

```
# printing out to file
loss_history = history.history["loss"]
numpy_loss_history = np.array(loss_history)
np.savetxt("loss_history.txt", numpy_loss_history,
        delimiter="\n")

binary_accuracy_history = history.history["binary_accuracy"]
numpy_binary_accuracy = np.array(binary_accuracy_history)
np.savetxt("binary_accuracy.txt", numpy_binary_accuracy, delimiter="\n")
```

Step 5, evaluate the model. Here we run the model to predict the outputs from all the inputs of `X`, using the `round` function to make them either 0 or 1. Note that this replaces the criteria we used in our pure Python model, which was `<0.1 = "0"` and `>0.9 = "1"`. We also calculated the average of all the `binary_accuracy` values of all the epochs, but the number isn't very useful — except that the closer to 1.0 it is, the faster the model succeeded.

```
print(np.mean(history.history["binary_accuracy"]))

result = model.predict(X ).round()

print (result)
```

Now let's move along to some results.

Evaluating the model

WARNING

When you run TensorFlow programs, you may see something like this:

```
/usr/lib/python3.5/importlib/_bootstrap.py:222: RuntimeWarning: compiletime
   version 3.4 of module 'tensorflow.python.framework.fast_tensor_util' does not
   match runtime version 3.5
  return f(*args, **kwds)
/usr/lib/python3.5/importlib/_bootstrap.py:222: RuntimeWarning: builtins.type
   size changed, may indicate binary incompatibility. Expected 432, got 412
  return f(*args, **kwds)
```

This is because of a problem with the way TensorFlow was built for your machine. These warnings can be safely ignored. The good folks over at TensorFlow.org say this issue will be fixed in the next version.

We run the two-layer model by typing **python3 TensorFlowKeras.py** in our terminal window. After watching the epochs march away (you can change this amount of output by setting the `Verbose` parameter in your `model.fit` command), we are rewarded with the following:

```
...
Epoch 1999/2000
8/8 [==============================] - 0s 2ms/step - loss: 0.0367 - binary_
   accuracy: 1.0000 - val_loss: 0.0367 - val_binary_accuracy: 1.0000
Epoch 2000/2000
8/8 [==============================] - 0s 2ms/step - loss: 0.0367 - binary_
   accuracy: 1.0000 - val_loss: 0.0367 - val_binary_accuracy: 1.0000
_________________________________________________________________
Layer (type)        Output Shape       Param #
=================================================================
dense (Dense)        (None, 4)        16
_________________________________________________________________
dense_1 (Dense)       (None, 1)        5
=================================================================
Total params: 21
Trainable params: 21
Non-trainable params: 0
_________________________________________________________________
0.8436875
[[1.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [1.]]
```

We see that by epoch 2,000 we had achieved the binary accuracy of 1.0, as hoped for, and the results of our `model.predict` function call at the end matches our truth table. Figure 2-6 shows the results of the loss function and binary accuracy values plotted against the epoch number as the training progressed. Figure 2-2 shows what graphically we are implementing.

A couple of things to note. The loss function is a much smoother linear curve when it succeeds. This has to do with the activation choice (`relu`) and the optimizer function (ADAM). Another thing to remember is you will get a different curve (somewhat) each time because of the random number initial values in the weights. Seed your random number generator to make it the same each time you run it. This makes it easier to optimize your performance.

BACKPROPAGATION IN KERAS

With our first neural network in this chapter, we made a big deal about backpropagation and how it was a fundamental part of neural networks. However, now we have moved into Keras/TensorFlow and we haven't said one word about it. The reason for this is that the backpropogation in Keras/TensorFlow is handled automatically. It's done for you. If you want to modify how it is doing it, the easiest way is to modify the optimization parameter in the `module.compile` command (we used ADAM). It is quite a bit of work to dramatically modify the backpropogation algorithm in Keras, but it can be done.

When you run your training for the network, you are using the backpropagation algorithm and optimizing this according to the chosen optimization algorithm and loss function specified when compiling the model.

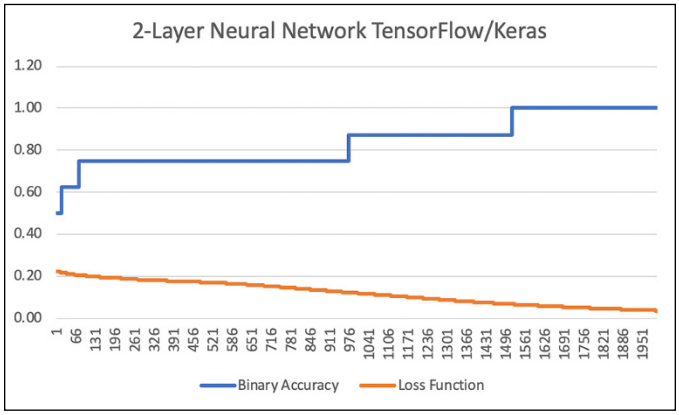


FIGURE 2-6: Results of the two-layer training.

Note when the binary accuracy goes to 1.00 (about epoch 1556). That's when your network is fully trained in this case.

Changing to a three-layer neural network in TensorFlow/Keras

Now let's add another layer to our neural network, as shown in Figure 2-7. Open your TensorFlowKeras.py file in your favorite editor and change the following:

```
model.add(Dense(4, input_dim=3, activation='relu',
    use_bias=True))
#model.add(Dense(4, activation='relu', use_bias=True))
model.add(Dense(1, activation='sigmoid', use_bias=True))
```

Remove the comment character in front of the middle layer, and you now have a three-layer neural network with four neurons per layer. It's that easy. Here is what it should look like now:

```
model.add(Dense(4, input_dim=3, activation='relu',
    use_bias=True))
model.add(Dense(4, activation='relu', use_bias=True))
model.add(Dense(1, activation='sigmoid', use_bias=True))
```

Run the program and you will now have your results from the three-layer neural network, which will look something like this:

```
8/8 [==============================] - 0s 2ms/step - loss: 0.0153 - binary_
   accuracy: 1.0000 - val_loss: 0.0153 - val_binary_accuracy: 1.0000
Epoch 2000/2000
8/8 [==============================] - 0s 2ms/step - loss: 0.0153 - binary_
   accuracy: 1.0000 - val_loss: 0.0153 - val_binary_accuracy: 1.0000
_________________________________________________________________
Layer (type)        Output Shape       Param #
=================================================================
dense (Dense)        (None, 4)        16
_________________________________________________________________
dense_1 (Dense)      (None, 4)        20
_________________________________________________________________
dense_2 (Dense)      (None, 1)        5
=================================================================
Total params: 41
Trainable params: 41
Non-trainable params: 0
_________________________________________________________________
0.930375
[[1.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [1.]]
```

TIP

WHY USE A GUI (GRAPHICAL USER INTERFACE) TO RUN TensorFlow?

As you should know by now, you spend a lot of time coding in text editors to build your models. For simplicity's sake, we exported our data out to Excel to produce the graphs in this chapter. Most of the time, we use our machines in a terminal window, but there is a big advantage to using your computer's full GUI desktop to open these terminal windows for editing. That big advantage is called TensorBoard. *TensorBoard* is a part of TensorFlow and is available to you in a browser such as Chrome or Firefox. You point TensorBoard at your job directory and you suddenly can do all sorts of visual analysis of your neural network experiments.

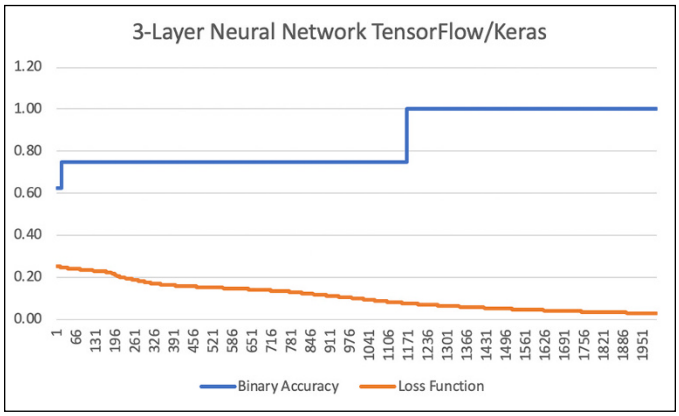


FIGURE 2-7: Results of the three-layer training.

You now can see that you have three layers in your neural network. This is one reason why the TensorFlow/Keras software is so powerful. It's easy to tinker with parameters and make changes.

Notes on our three-layer run: First of all, it converges to a binary accuracy of 1.00 at about epoch 916, much faster than epoch 1556 from our two-layer run. The loss function is less linear than the two-layer run's.

Just for fun and giggles, we changed the number of neurons to 100 per each of the hidden layers. As expected, it converged to a binary accuracy of 1.00 at epoch 78, much faster than the earlier run! Run your own experiments to get a good feel for the way your results will vary with different parameters, layers, and neuron counts.

Believe it or not, you now understand a great deal about how neural networks and machine learning works. Go forth and train those neurons!

IN THIS CHAPTER

- Teaching a machine to learn something
- How machines learn
- Understand the basics of machine learning
- Using TensorFlow to do machine learning

Chapter 3

Doing Machine Learning in Python

What does it mean to learn something? One definition is "the acquisition and mastery of what is already known about something and the extended clarification of meaning of that knowledge." Another definition is that "learning is a relatively permanent change in a person's knowledge or behavior due to experience."

At the current (and most likely for some time in the future) state of machine learning, it is the second definition that best fits with the current state of AI. Our culture has developed algorithms and programs that can learn things about data and about sensory input and apply that knowledge to new situations. However, our machines do not "understand" anything about what they have learned. They have just accumulated data about their inputs and have transformed that input to some kind of output.

However, even if the machine does not "understand" what it has learned, that does not mean that you cannot do some pretty impressive things using these machine-learning techniques that will be discussed in this chapter.

Maybe these techniques that we are developing now may lead the way to something much more impressive in the future.

What does it mean for a machine to learn something? We're going to use the rough idea that if a machine can take inputs and by some process transform those inputs to some useful outputs, then we can say the machine has learned something. This definition has a wide meaning. In writing a simple program to add two numbers you have taught that machine something. It has learned to add two numbers. We're going to focus in this chapter on machine learning in the sense of the use of algorithms and statistical models that progressively improve their performance on a specific task. If this sounds very much like our neural-network experiments in Chapter 2, you are correct. Machine learning is a lot about neural networks, but it's also about other sophisticated techniques.

Learning by Looking for Solutions in All the Wrong Places

One of the real problems with machine learning and AI in general is figuring out how the algorithm can find the best solution. The operative word there is *best*. How do we know a given solution is best? It is really a matter of setting goals and achieving them (the solution may not be best but maybe just good enough). Some people have compared the problem of finding the "best" solution to that of a person wandering around on a foggy day trying to find the highest mountain in the area. You climb up a mountain and get to the top and then proclaim, "I am on the highest mountain." Well, you are on the highest mountain you can see, but you can't see through the fog. However, if you define your goal as being on the top of a mountain more than 1,000 feet high, and you are at 1,250 feet, then you have met your goal. This is called a *local maxima*, and it may or may not be the best maxima available.

In this chapter, most of our goal setting (training the machine) will be done with known solutions to a problem: first training our machine and then applying the training to new, similar examples of the problem.

There are three main types of machine-learning algorithms:

- **Supervised learning:** This type of algorithm builds a model of data that contains both inputs and outputs. The data is known as training data. This is the kind of machine learning we show in this chapter.
- **Unsupervised learning:** For this type of algorithm, the data contains only the inputs, and the algorithms look for the structures and patterns in the data.
- **Reinforcement learning:** This area is concerned with software taking actions based on some kind of cumulative reward. These algorithms do not assume

knowledge of an exact mathematical model and are used when exact models are unavailable. This is the most complex area of machine learning, and the one that may bear the most fruit in the future.

With that being said, let's jump into doing machine learning with Python.

Classifying Clothes with Machine Learning

In this chapter, we use the freely available training Fashion-MNIST (Modified National Institute of Standards and Technology) database that contains 60,000 fashion products from ten categories. It contains data in 28x28 pixel format with 6,000 items in each category. (See Figure 3-1.)

This gives us a really interesting dataset with which to build a TensorFlow/Keras machine-learning application — much more interesting than the standard MNIST machine-learning database that contains only handwritten characters.

FIGURE 3-1: A bit of the Fashion-MNIST database.

Training and Learning with TensorFlow

For this chapter, we are going to once again use TensorFlow/Keras to build some machine-learning examples and look at their results. For more about TensorFlow and Keras, refer to Chapter 2.

Here we use the same five-step approach we used to build layered networks with Keras in Chapter 2. TensorFlow.

1. Load and format your data.
2. Define your neural network model and layers.
3. Compile the model.
4. Fit and train your model.
5. Evaluate the model.

Setting Up the Software Environment for this Chapter

Most of the action in this chapter is, as usual, in the command line, because you still have to type code and run software. However, we are going to display some graphics on the screen and use MatPlotLib to evaluate what your machine-learning program is doing, so please start a GUI (graphics user interface) if you haven't already. If you are running on a headless Raspberry Pi, either add a keyboard, mouse, and monitor or stop now and bring up VNC (virtual network computer). think of using your computer monitor as a display on a second computer — the Raspberry Pi, in this case. Many links on the web describe how to do this and how to bring up the GUI on your main computer. We are using VNC on a headless Raspberry Pi in this chapter. Feel free to connect a mouse, keyboard, and a monitor directly to the Raspberry Pi if you want. Figure 3-2 shows the GUI running on the Raspberry Pi (it is actually running on VNC on our Mac, but you can't tell from this image).

A great source for tutorials on setting up the software and connecting the Raspberry Pi are located at `www.raspberrypi.org`.


TIP

If you are missing some of the libraries that we use in this example, then search the web for how to install them on your specific machine. Every setup is a little different. For example, if you're missing seaborn, then search on "installing seaborn python library on [*name of your machine*]." If you do a search on seaborn for the Raspberry Pi, then you will find "sudo pip3 install seaborn."

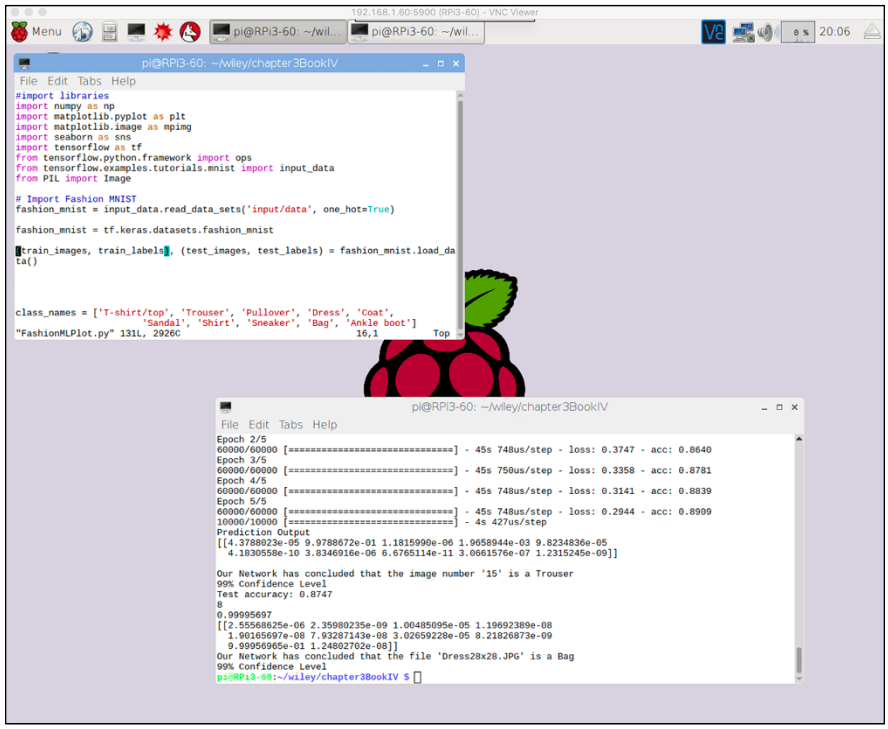


FIGURE 3-2: A full GUI on the Raspberry Pi.

Creating a Machine-Learning Network for Detecting Clothes Types

Our main example of machine learning in Python uses a MNIST-format (*MNIST* means that it is a collection of grayscale images with a resolution of 28x28 pixels) Fashion database of 60,000 images classified into ten types of apparel, as follows:

- 0 T-shirt/top
- 1 Trouser
- 2 Pullover
- 3 Dress
- 4 Coat
- 5 Sandal
- 6 Shirt

- 7 Sneaker
- 8 Bag
- 9 Ankle boot

Getting the data — The Fashion-MNIST dataset

Turns out this is pretty easy, although it will take a while to first load it to your computer. After you run the program for the first time, it will use the Fashion-MNIST data copied to your computer.

Training the network

We will train our machine-learning neural network using all 60,000 images of clothes: 6,000 images in each of the ten categories.

Testing our network

Our trained network will be tested three different ways: 1) a set of 10,000 training photos from the `Fashion_MNIST` data set; 2) a selected image from the `Fashion_MNIST` data set; and 3) a photo of a woman's dress.

This first version of the program will run a test on a 10,000 set of files from the `Fashion_MNIST` database.

Our Python code using TensorFlow, NumPy, and Keras for the `Fashion_MNIST` network follows. Using nano (or your favorite text editor), open up a file called FMTensorFlow.py and enter the following code:

```
#import libraries
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import seaborn as sns
import tensorflow as tf
from tensorflow.python.framework import ops
from tensorflow.examples.tutorials.mnist import input_data
from PIL import Image
```

```
# Import Fashion MNIST
fashion_mnist = input_data.read_data_sets('input/data',
        one_hot=True)

fashion_mnist = tf.keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) \
        = fashion_mnist.load_data()




class_names = ['T-shirt/top', 'Trouser',
        'Pullover', 'Dress', 'Coat',
        'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']


train_images = train_images / 255.0

test_images = test_images / 255.0


model = tf.keras.Sequential()


model.add(tf.keras.layers.Flatten(input_shape=(28,28)))
model.add(tf.keras.layers.Dense(128, activation='relu' ))
model.add(tf.keras.layers.Dense(10, activation='softmax' ))


model.compile(optimizer=tf.train.AdamOptimizer(),
                     loss='sparse_categorical_crossentropy',
                                  metrics=['accuracy'])


model.fit(train_images, train_labels, epochs=5)

# test with 10,000 images
test_loss, test_acc = model.evaluate(test_images, test_labels)

print('10,000 image Test accuracy:', test_acc)
```


Breaking down the code

After you've read the description of the Tensform/Keras program in Chapter 2, this code should look much more familiar. In this section, we break it down into our five-step Keras process.

First, we import all the libraries needed to run our example two-layer model. Note that TensorFlow includes Keras by default. And once again we see our friend NumPy as the preferred way of handling matrices.

```
#import libraries
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import seaborn as sns
import tensorflow as tf
from tensorflow.python.framework import ops
from tensorflow.examples.tutorials.mnist import input_data
from PIL import Image
```

1. **Load and format your data.**

   This time we are using the built-in data-set reading capability. It knows what this data is because of the `import` statement from `tensorflow.examples.tutorials.mnist` in the preceding code.

   ```
   # Import Fashion MNIST
   fashion_mnist = input_data.read_data_sets('input/data',
           one_hot=True)

   fashion_mnist = tf.keras.datasets.fashion_mnist

   (train_images, train_labels), (test_images, test_labels) \
           = fashion_mnist.load_data()
   ```

   Here we give some descriptive names to the ten classes within the `Fashion_MNIST` data.

   ```
   class_names = ['T-shirt/top', 'Trouser',
           'Pullover', 'Dress', 'Coat',
           'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
   ```

   Here we change all the images to be scaled from 0.0–1.0 rather than 0–255.

   ```
   train_images = train_images / 255.0

   test_images = test_images / 255.0
   ```

2. **Define your neural-network model and layers.**

   Again, this is where the real power of Keras shines. It is very simple to add more neural layers, and to change their sizes and their activation functions. We are also applying a bias to our activation function (`relu`), in this case with `softmax`, for the final output layer.

```
model = tf.keras.Sequential()


model.add(tf.keras.layers.Flatten(input_shape=(28,28)))
model.add(tf.keras.layers.Dense(128, activation='relu' ))
model.add(tf.keras.layers.Dense(10, activation='softmax' ))
```

3. **Compile your model.**

We are using the loss function sparse_categorical_crossentropy. This function is new to us in this book. It is used when you have assigned a different integer for each clothes category as we have in this example. ADAM (a method for stochastic optimization) is a good default optimizer. It provides a method well suited for problems that are large in terms of data and/or parameters.

```
model.compile(optimizer=tf.train.AdamOptimizer(),
                      loss='sparse_categorical_
  crossentropy',
                                    metrics=['accuracy'])
```

Sparse categorical crossentropy is a loss function used to measure the error between categories across the data set. *Categorical* refers to the fact that the data has more than two categories (binary) in the data set. *Sparse* refers to using a single integer to refer to classes (0–9, in our example). *Entropy* (a measure of disorder) refers to the mix of data between the categories.

4. **Fit and train your model.**

I chose the number of epochs as only 5 due to the time it takes to run the model for our examples. Feel free to increase! Here we load the NumPy arrays for the input to our network (the database train_images).

```
model.fit(train_images, train_labels, epochs=5)
```

5. **Evaluate the model.**

The model.evaluate function is used to compare the outputs of your trained network in each epoch and generates `test_acc` and `test_loss` for your information in each epoch as stored in the history variable.

```
# test with 10,000 images
test_loss, test_acc = model.evaluate(test_images,
  test_labels)

print('10,000 image Test accuracy:', test_acc)
```

Results of the training and evaluation

I ran my program on the Raspberry Pi 3B+. You can safely ignore the code mismatch warnings and the future deprecation announcements at this point.

Here are the results of the program:

```
Epoch 1/5
60000/60000 [==============================] - 44s 726us/step - loss: 0.5009 -
   acc: 0.8244
Epoch 2/5
60000/60000 [==============================] - 42s 703us/step - loss: 0.3751 -
   acc: 0.8652
Epoch 3/5
60000/60000 [==============================] - 42s 703us/step - loss: 0.3359 -
   acc: 0.8767
Epoch 4/5
60000/60000 [==============================] - 42s 701us/step - loss: 0.3124 -
   acc: 0.8839
Epoch 5/5
60000/60000 [==============================] - 42s 703us/step - loss: 0.2960 -
   acc: 0.8915
10000/10000 [==============================] - 4s 404us/step
10,000 image Test accuracy: 0.873
```

Fundamentally, the test results are saying that with our two-layer neural machine-learning network, we are classifying 87 percent of the 10,000-image test database correctly. We upped the number of epochs to 50 and increased this to only 88.7 percent accuracy. Lots of extra computation with little increase in accuracy.

Testing a single test image

Next is to test a single image (see Figure 3-3) from the `Fashion_MNIST` database. Add this code to the end of your program and rerun the software:

```
#run test image from Fashion_MNIST data

img = test_images[15]
img = (np.expand_dims(img,0))
singlePrediction = model.predict(img,steps=1)
print ("Prediction Output")
print(singlePrediction)
print()
NumberElement = singlePrediction.argmax()
Element = np.amax(singlePrediction)
```

```
print ("Our Network has concluded that the image number '15' is a "
       +class_names[NumberElement])
print (str(int(Element*100)) + "% Confidence Level")
```

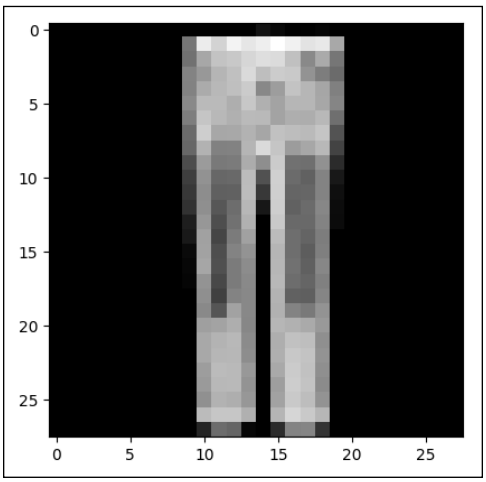


FIGURE 3-3: Image 15 from the Fashion-MNIST test database.

Here are the results from a five-epoch run:

```
Prediction Output
[[1.2835168e-05 9.9964070e-01 6.2637120e-08 3.4126092e-04 4.4297972e-06
  7.8450663e-10 6.2759432e-07 9.8717527e-12 1.2729484e-08 1.1002166e-09]]

Our Network has concluded that the image number '15' is a Trouser
99% Confidence Level
```

Woohoo! It worked. It correctly identified the picture as a trouser. Remember, however, that we only had an overall accuracy level on the test data of 87 percent.

Next, off to our own picture.

Testing on external pictures

To accomplish this test, John took a dress from his wife's closet, hung it up on a wall (see Figure 3-4), and took a picture of it with his iPhone.

Next, using Preview on our Mac, we converted it to a resolution of 28 x 28 pixels down from 3024x3024 pixels straight from the phone. (See Figure 3-5.)

FIGURE 3-4: Unidentified dress hanging on a wall.

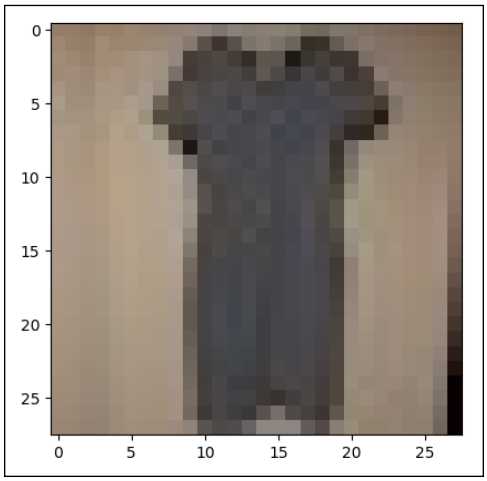


FIGURE 3-5: The dress at 28x28 pixels.

Okay, a few very important comments. First of all, 28x28 pixels does not result in a very clear picture. However, comparing Figure 3-6 to Figure 3-4 from the Fashion-MNIST database, our picture still looks better.

Most of the following code has to do with arranging the data from our JPG picture to fit the format required by TensorFlow. You should be able to use this code to easily add your own pictures for more experiments:

```
# run Our test Image
# read test dress image
imageName = "Dress28x28.JPG"
```

```
testImg = Image.open(imageName)
testImg.load()
data = np.asarray( testImg, dtype="float" )


data = tf.image.rgb_to_grayscale(data)
data = data/255.0


data = tf.transpose(data, perm=[2,0,1])



singlePrediction = model.predict(data,steps=1)
print ("Prediction Output")
print(singlePrediction)
print()
NumberElement = singlePrediction.argmax()
Element = np.amax(singlePrediction)

print ("Our Network has concluded that the file '"
        +imageName+"' is a "+class_names[NumberElement])
print (str(int(Element*100)) + "% Confidence Level")
```


The results, round 1

We should start out by saying these results did not make us very happy, as you will see shortly. We put the Dress28x28.JPG file in the same directory as our program and ran a five- epoch training run. Here are the results:

```
Prediction Output
[[1.2717753e-06 1.3373902e-08 1.0487850e-06 3.3525557e-11 8.8031484e-09
  7.1847245e-10 1.1177938e-04 8.8322977e-12 9.9988592e-01 3.2957085e-12]]

Our Network has concluded that the file 'Dress28x28.JPG' is a Bag
99% Confidence Level
```

So, our neural network machine learning program, after classifying 60,000 pictures and 6,000 dress pictures, concluded at a 99 percent confidence level . . . *wait for it* . . . that John's wife's dress is a bag.

So the first thing we did next was to increase the training epochs to 50 and to rerun the program. Here are the results from that run:

```
Prediction Output
[[3.4407502e-33 0.0000000e+00 2.5598763e-33 0.0000000e+00 0.0000000e+00
```

```
  0.0000000e+00 2.9322060e-17 0.0000000e+00 1.0000000e+00 1.5202169e-39]]

Our Network has concluded that the file 'Dress28x28.JPG' is a Bag
100% Confidence Level
```

The dress is still a bag, but now our program is 100 percent confident that the dress is a bag. Hmmmm.

This illustrates one of the problems with machine learning. Being 100 percent certain that a picture is of a bag when it is a dress, is still 100 percent wrong.

What is the real problem here? Probably the neural-network configuration is just not good enough to distinguish the dress from a bag. We saw that additional training epochs didn't seem to help at all, so the next thing to try is to increase the number of neurons in our hidden level.

What are other things to try to improve this? It turns out there are many. You can use CNN (convolutional neural networks), data augmentation (increasing the training samples by rotating, shifting, and zooming that pictures) and a variety of other techniques that are beyond the scope of this introduction to machine learning.

We did do one more experiment. We changed the model layers in our program to use the following four-level convolutional-layer model. We just love how easy Keras and TensorFlow makes it to dramatically change the neural network.

CNNs work by scanning images and analyzing them chunk by chunk, say at 5x5 window that moves by a stride length of two pixels each time until it spans the entire message. It's like looking at an image using a microscope; you only see a small part of the picture at any one time, but eventually you see the whole picture. Going to a CNN network on my Raspberry Pi increased the single epoch time to 1.5 hours from a 10 seconds epoch previously.

The CNN model code

This code has the same structure as the last program. The only significant change is the addition of the new layers for the CNN network:

```
#import libraries
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import seaborn as sns
import tensorflow as tf
```

```
from tensorflow.python.framework import ops
from tensorflow.examples.tutorials.mnist import input_data
from PIL import Image

# Import Fashion MNIST
fashion_mnist = input_data.read_data_sets('input/data',
        one_hot=True)

fashion_mnist = tf.keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) \
        = fashion_mnist.load_data()




class_names = ['T-shirt/top', 'Trouser',
        'Pullover', 'Dress', 'Coat',
        'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']


train_images = train_images / 255.0

test_images = test_images / 255.0


# Prepare the training images
train_images = train_images.reshape(train_images.shape[0], 28, 28, 1)

# Prepare the test images
test_images = test_images.reshape(test_images.shape[0], 28, 28, 1)


model = tf.keras.Sequential()

input_shape = (28, 28, 1)
model.add(tf.keras.layers.Conv2D(32, kernel_size=(3, 3), activation='relu',
   input_shape=input_shape))
model.add(tf.keras.layers.BatchNormalization())

model.add(tf.keras.layers.Conv2D(32, kernel_size=(3, 3), activation='relu'))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
model.add(tf.keras.layers.Dropout(0.25))

model.add(tf.keras.layers.Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.Dropout(0.25))
```

```
model.add(tf.keras.layers.Conv2D(128, kernel_size=(3, 3), activation='relu'))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
model.add(tf.keras.layers.Dropout(0.25))

model.add(tf.keras.layers.Flatten())

model.add(tf.keras.layers.Dense(512, activation='relu'))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.Dropout(0.5))

model.add(tf.keras.layers.Dense(128, activation='relu'))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.Dropout(0.5))

model.add(tf.keras.layers.Dense(10, activation='softmax'))

model.compile(optimizer=tf.train.AdamOptimizer(),
                      loss='sparse_categorical_crossentropy',
                                  metrics=['accuracy'])


model.fit(train_images, train_labels, epochs=5)

# test with 10,000 images
test_loss, test_acc = model.evaluate(test_images, test_labels)

print('10,000 image Test accuracy:', test_acc)

#run test image from Fashion_MNIST data

img = test_images[15]
img = (np.expand_dims(img,0))
singlePrediction = model.predict(img,steps=1)
print ("Prediction Output")
print(singlePrediction)
print()
NumberElement = singlePrediction.argmax()
Element = np.amax(singlePrediction)

print ("Our Network has concluded that the image number '15' is a "
        +class_names[NumberElement])
print (str(int(Element*100)) + "% Confidence Level")
```

The results, round 2

This run on the Raspberry Pi 3B+ took about seven hours to complete. The results were as follows:

```
10,000 image Test accuracy: 0.8601
Prediction Output
[[5.9128129e-06 9.9997270e-01 1.5681641e-06 8.1393973e-06 1.5611777e-06
  7.0504888e-07 5.5174642e-06 2.2484977e-07 3.0045830e-06 5.6888598e-07]]

Our Network has concluded that the image number '15' is a Trouser
```

The key number here is the 10,000-image test accuracy. At 86 percent, it was actually lower than our previous, simpler machine-learning neural network (87 percent). Why did this happen? This is probably a case related to "overfitting" the training data. A CNN model such as this can use complex internal models to train (many millions of possibilities) and can lead to *overfitting*, which means the trained network recognizes the training set better but loses the ability to recognize new test data.

TIP

Choosing the machine-learning neural network to work with your data is one of the major decisions you will make in your design. However, understanding activation functions, dropout management, and loss functions will also deeply affect the performance of your machine-learning program. Optimizing all these parameters at once is a difficult task that requires research and experience. Some of this is really rocket science!

Visualizing with MatPlotLib

Now that we have moved to a GUI-based development environment, we are going to run our base code again and do some analysis of the run using MatPlotLib. We are using a Raspberry Pi for these experiments, but you can use a Mac, PC, or another Linux system and basically do the same thing. If you can install TensorFlow, MatPlotLib, and Python on your computer system, you can do these experiments.

TIP

To install MatPlotLib on your Raspberry Pi, type `pip3 install matplotlib`.

We add the `history` variable to the output of the `model.fit` to collect data. And then we add MatPlotLib commands to graph the loss and the accuracy from our epochs and then to add figure displays for our two individual image tests. Figure 3-6 shows the results of running this program.

Using nano (or your favorite text editor), open up a file called FMTensorFlowPlot.py, and enter the following code:

```
#import libraries
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import seaborn as sns
import tensorflow as tf
from tensorflow.python.framework import ops
from tensorflow.examples.tutorials.mnist import input_data
from PIL import Image

# Import Fashion MNIST
fashion_mnist = input_data.read_data_sets('input/data', one_hot=True)

fashion_mnist = tf.keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) = fashion_mnist.
   load_data()




class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
                      'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']


train_images = train_images / 255.0

test_images = test_images / 255.0


model = tf.keras.Sequential()


model.add(tf.keras.layers.Flatten(input_shape=(28,28)))
model.add(tf.keras.layers.Dense(128, activation='relu' ))
model.add(tf.keras.layers.Dense(10, activation='softmax' ))


model.compile(optimizer=tf.train.AdamOptimizer(),
                      loss='sparse_categorical_crossentropy',
                                  metrics=['accuracy'])


history = model.fit(train_images, train_labels, epochs=2)
```

```
# Get training and test loss histories
training_loss = history.history['loss']
accuracy = history.history['acc']
# Create count of the number of epochs
epoch_count = range(1, len(training_loss) + 1)

# Visualize loss history
plt.figure(0)
plt.plot(epoch_count, training_loss, 'r--')
plt.plot(epoch_count, accuracy, 'b--')
plt.legend(['Training Loss', 'Accuracy'])
plt.xlabel('Epoch')
plt.ylabel('History')
plt.show(block=False);
plt.pause(0.001)

test_loss, test_acc = model.evaluate(test_images, test_labels)

#run test image from Fashion_MNIST data



img = test_images[15]

plt.figure(1)
plt.imshow(img)
plt.show(block=False)
plt.pause(0.001)

img = (np.expand_dims(img,0))
singlePrediction = model.predict(img,steps=1)

print ("Prediction Output")

print(singlePrediction)

print()

NumberElement = singlePrediction.argmax()

Element = np.amax(singlePrediction)



print ("Our Network has concluded that the image number '15' is a "

                +class_names[NumberElement])
```

```
print (str(int(Element*100)) + "% Confidence Level")


print('Test accuracy:', test_acc)

# read test dress image
imageName = "Dress28x28.JPG"

testImg = Image.open(imageName)

plt.figure(2)
plt.imshow(testImg)
plt.show(block=False)
plt.pause(0.001)
testImg.load()
data = np.asarray( testImg, dtype="float" )


data = tf.image.rgb_to_grayscale(data)
data = data/255.0


data = tf.transpose(data, perm=[2,0,1])



singlePrediction = model.predict(data,steps=1)

NumberElement = singlePrediction.argmax()
Element = np.amax(singlePrediction)
print(NumberElement)
print(Element)
print(singlePrediction)

print ("Our Network has concluded that the file '"+imageName+"' is a
   "+class_names[NumberElement])
print (str(int(Element*100)) + "% Confidence Level")

plt.show()
```

The results of running this program is shown in Figure 3-6. The window labeled Figure 0 shows the accuracy data for each of the five epochs of the machine learning training, and you can see the accuracy slowly increases with each epoch. The window labeled Figure 1 shows the test picture used for the first recognition test (it found a pair of trousers, which is correct), and finally, the window labeled Figure 2 shows the dress picture, which is still incorrectly identified as a bag. Harumph.

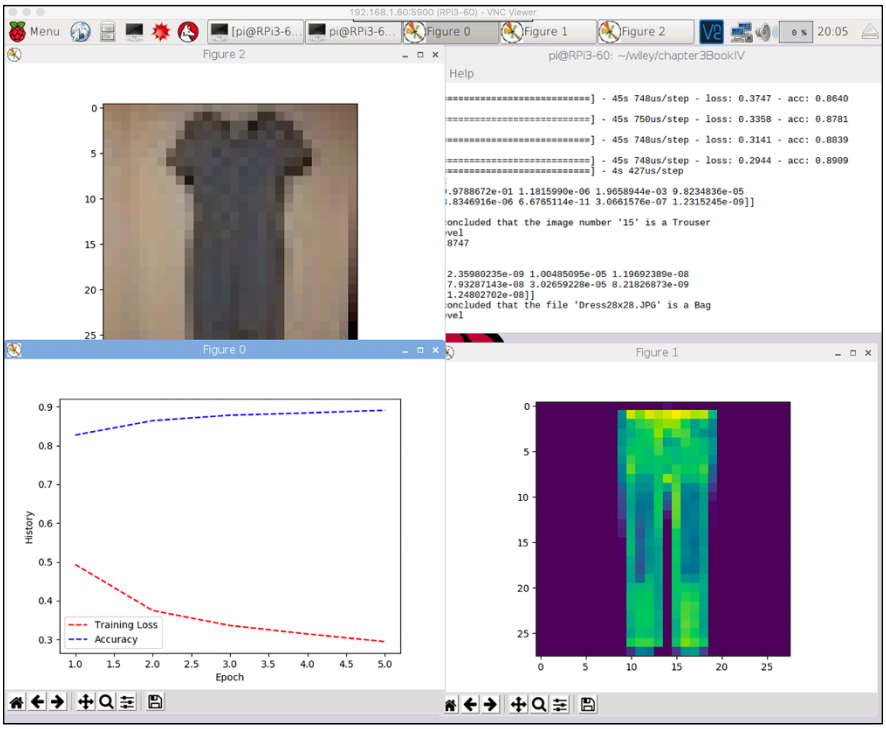


FIGURE 3-6: Our Raspberry Pi GUI with MatPlotLib visualization.

Learning More Machine Learning

After seeing how useful Python is in building and experimenting with machine learning and neural networks, you can see how powerful it is. Even though you are just beginning to understand the theory behind a lot of the models you have use, you should feel that you now have a tremendous amount of ability to build and experiment with making machines learn.

Next step? We recommend the following books:

- *Machine Learning for Dummies,* John Paul Mueller and Luca Massaron
- *Deep Learning with Python,* Francois Chollet
- And a great beginners overview of the whole AI field: *Artificial Intelligence For Dummies,* John Paul Mueller and Luca Massaron

Next, we explore using Python with some other AI applications.

IN THIS CHAPTER

- Limitations on doing AI on your Raspberry Pi
- Using the cloud to do AI
- Using AI on graphics cards

Chapter 4
Exploring More AI in Python

After reading the previous three chapters, you have learned quite a bit about using some of the basics of artificial intelligence, specifically neural networks and machine learning. There is a lot more to AI than just these two things, though. We could look at advanced searching (not Google searching but rather looking at big problem spaces and trying to figure out solutions to the problem using AI). We could also look at the whole problem of autonomous robotics (which we touch upon in Book 7) but this topic is very complicated.

Instead, in this chapter, we talk about other ways of doing AI software beyond the Raspberry Pi. Remember it took us seven hours to run five epochs of training on our large neural network? Sounds like we could use some bigger iron to accomplish more training in less time. That's what this chapter is about.

Limitations of the Raspberry Pi and AI

The Raspberry Pi is an inexpensive full-blown computing device. The Raspberry Pi 3B+, which we have used throughout this book, has the following major specifications:

- CPU: Broadcom quad-core 64bit processor @ 1.4GHz
- GPU: Broadcom Videocore-IV

- RAM: 1GB SDRAM
- Networking: Gigabit Ethernet, 802.11b/g/n/ac WiFi
- Storage: SD card

How does this stack up? For a $35 computer, very well. But for a dedicated AI computer, not so much.

The problems are not enough RAM (1GB isn't very much, especially for a Raspberry Pi to do AI) and not a very sophisticated GPU (graphics processing unit). Figure 4-1 shows the Raspberry Pi 3B+ processing chip.

FIGURE 4-1: The Raspberry Pi processing chip containing the Videocore-IV.

There are two mitigating circumstances that keep the Raspberry Pi in the running when it comes to doing and experimenting with AI. First of all, you can buy an AI Accelerator that can plug into the USB ports of the Raspberry Pi and secondly, you can use the Raspberry Pi to control processors and AI hardware up on the cloud for all the computationally heavy lifting.

TECHNICAL STUFF

THE BROADCOM VIDEOCORE-IV ON THE RASPBERRY PI 3B+

The Videocore-IV is a low-power mobile graphics processor. It is a two-dimensional DSP (digital signal processor) that is set up as basically a four-GPU core unit. These GPU core units are called *slices* and can be very roughly compared to GPU computer units, such as those used by AMD and Nvidia (which can have 256, 512, or more individual GPU units, far outclassing the Videocore 4 units) to power their GPU cards, which are now proving to be very popular with AI researchers and hobbyists.

This processor is really designed to be used in video encoding and decoding applications and not so much for AI use. However, some researchers have made use of the four slices to accelerate neural-network processing on the Raspberry Pi to achieve up to about three times the performance of the four-core main processor used alone.

One of the main barriers to using the Videocore on the Raspberry Pi for AI type of applications is that the specifications, development tools, and product details have only been available under NDA (non-disclosure agreements), which do not go along with open-source development. However, you can now get full documentation and the complete source code for the Raspberry Pi 3B+ graphics stack under a very nonrestrictive BSD license, which should provide a path forward.

TIP

Remember from our previous chapter that the bulk of the computer time in building any kind of machine-learning AI system is for training and that when that training is done, it doesn't take a lot of processing to actually characterize an unknown or new picture. This means you can train on one big machine and then deploy on a simpler computer such as the Raspberry Pi in the application. This doesn't work all the time (especially if you want to keep learning as the program runs) but if it does work, it allows you to deploy sophisticated machine-learning programs on much simpler and less expensive hardware.

Performing AI analysis or training on the small computers that are connected to a network, is called *edge computing* or, phrased a different way, computing on the edge of the network.

Adding Hardware AI to the Raspberry Pi

It turns out that there have been a number of companies starting to build specialized AI compute sticks, many of which can be used on the Raspberry Pi. Typically, you will find that there are Python libraries or wrappers, and often TensorFlow Python libraries, that support using these sticks. Two of the most interesting ones are

- **The Intel Modvidius Neural Compute Stick (NCS):** The Movidius NCS stick plugs into the USB port of the Raspberry Pi or other compute and provides hardware support for deep learning based analysis (refer to Chapters 1–3).

  For example, you can use the Amazon cloud to perform image analysis, processing and classification up in the cloud from your small computer system which moves your computationally expensive task from your Raspberry Pi to the cloud. This does cost money and bandwidth (and latency in your system) to do this. Doing the analysis with your trained deep learning neural network on the edge by using a NCS stick can help and can possibly allow you to disconnect your device running on the edge of the network from the Internet entirely. It runs around 60X faster than doing image analysis on the Raspberry Pi and costs less than $100. Figure 4-2 shows the Movidius Compute Stick.

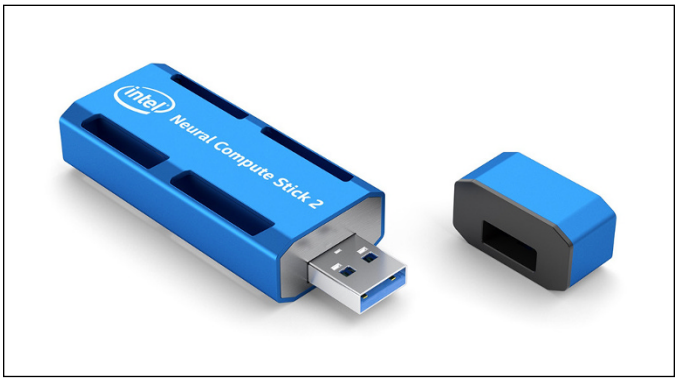


FIGURE 4-2: The Intel Movidius Neural Compute Stick 2.

You can do facial recognition, text analysis, monitoring and maintenance using this NCS stick. Pretty cool!

There is one concept that we need to emphasize here, however. The NCS stick is used to perform analysis and to conduct inferences on data, *but it is not used for training models!* You still need to build and train the models. It has a good interface with Keras and TensorFlow, so this is possible to do in a reasonable fashion.

Think of it as an accelerator for use by your final project when the training is done.

- **The Google Edge TPU accelerator:** The Google Edge TPU (tensor processing unit) has a USB Type-C socket that can be plugged into an Linux-based system to provide accelerated machine learning analysis and inferences. Does the word *tensor* sound familiar? Tensors are matrices like in our neural-network examples in Chapters 2 and 3. Figure 4-3 shows the Google Edge accelerator.

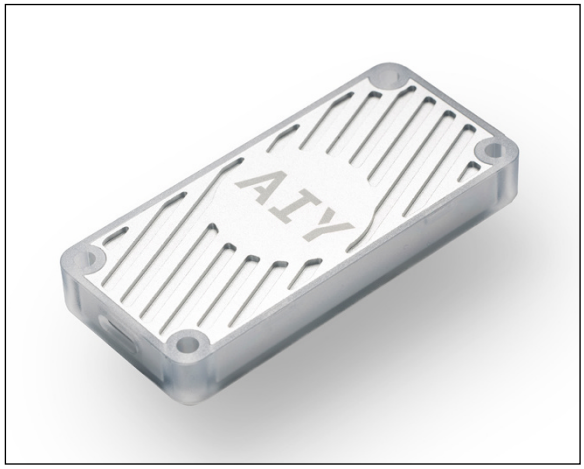


FIGURE 4-3: The Google Edge TPU accelerator.

Well, it turns out, much like the Intel NCS stick above, this device is all about executing trained machine-learning models. We still train the machine-learning networks using other techniques, and then we execute the model on the stick.

A FINAL COMMENT ON MACHINE-LEARNING ACCELERATORS

Oh, boy. In the next four years, this type of specialized hardware for running machine-learning models will explode. You will see multiple different architectures and solutions come from Google, Intel, Nvidia, AMD, Qualcomm, and a number of other smaller companies from around the world. Everybody is starting to climb on the AI accelerator hardware bandwagon.

AI in the Cloud

In the tech industry, everyone loves to use buzzwords such as *the cloud*. Often, the use of such language results in arbitrary and nebulous terms that leave consumers (or even sophisticated technical people) unsure what they

When a company says, "your data is in the cloud" or that "you can work in the cloud," this has nothing to do with being white, fluffy, or aboveground. Your "in the cloud" data is on the ground and is stored somewhere in a data center with a bunch of servers that are more similar to your PC or Mac than you may think.

Some people define the cloud as software or services that run on the Internet rather than on your local machine. This is correct to a degree, but nothing really runs on the Internet; it runs on machines that are *connected* to the Internet. Understanding that in-the-cloud software runs on servers and is not "just out there" tends to really quickly demystify the cloud and its functions.

If you have two computers networked together and use the other computer for a data server, you have your own "cloud."

This goes for basic services like storing your data in the cloud, but there is much more than just storage available on the cloud and that is where it gets really interesting.

The advantage of using the cloud is that you can use services and storage unavailable to you on your local network and (in one of the most important game changers of cloud computing) you can ramp your usage up and down depending on your computing needs on a dynamic basis.

Using the cloud requires Internet access. Not necessarily 100 percent of the time (you can fire off a cloud process and then come back to it later), but you do need connections some of the time. This limits the cloud in applications such as self-driving cars that aren't guaranteed to have good Internet access all the time. Interestingly, this "fire and forget" mode is useful for IOT (Internet of Things) devices where you don't want to stay connected to the net all the time for power considerations.

So, how do you use the cloud? That depends on the service and vendor, but in machine-learning applications, the most common way is to set up the Python on a computer that calls cloud-based functions and applications. All cloud vendors provide examples.

What is a great consumer example of cloud usage? The Amazon Echo and Alexa. It listens to you, compresses the speech data, sends it to the Amazon AWS cloud, translates and interprets your data and then sends back a verbal response or commands to make your lights come on.

A number of cloud providers for storage and services exist and more are arriving all the time. The top four cloud providers for AI at the time of this writing are

- Google cloud
- Amazon Web Services
- IBM cloud
- Microsoft Azure

Google cloud

The Google cloud is probably the most AI-focused cloud provider. You can gain access to TPU (tensor processing units) in the cloud, which, like the Google TPU stick above, can accelerate your AI applications. Much of the Google cloud's functionality reflects the core skill set of the company — that of search.

For example, the Cloud Vision API can detect objects, logos, and landmarks within images. Some excellent students at the University of Idaho are building a Smart City application called ParkMyRide, which uses a Raspberry Pi–based solar-powered camera to take pictures of the street and determines street parking availability by using the Google Cloud Vision API. The software sends a picture of the street to Google and gets back the number of cars found and where they are in the picture. They then supply this information to a smartphone app which displays it graphically. Pretty neat.

Other featured services of the Google cloud are: Video content search applications and speech-to-text/text-to-speech packages (think Google Home — very much like Amazon Alexa). Like Amazon and Microsoft, Google is using its own AI-powered applications to create new services for customers to use.

Amazon Web Services

Amazon Web Services (AWS) is focused on taking their consumer AI expertise and supplying this expertise to businesses. Many of these cloud services are built on the consumer product versions, so as Alexa improves, for example, the cloud services also improve.

Amazon not only has text and natural language offerings, but also machine-learning visualization/creation tools, vision recognition, and analysis.

IBM cloud

The IBM cloud has gotten a bad rap over the past few years for being hard to use. One of the big reasons was that there were so many different options on so many different platforms that it was almost impossible to figure out where to start.

In the past couple of years, it has gotten much better. IBM merged its three big divisions (IBM BlueMix cloud services, SoftLayer data services, and the Watson AI group) into one group under the Watson brand. There are still over 170 services available, so it is still hard to get going, but there is much better control and consistency over the process.

Their machine-learning environment is called the Watson Studio and is used to build and train AI models in one integrated environment. They also provide huge searchable knowledge catalogs and have one of the better IOT (Internet of Things) management platforms available.

One of the cool things they have is a service called Watson Personality Insights that predicts personality characteristics, needs, and values through written text. What would Watson Personality make of the authors of this book? We will run the text of the finished book through Watson and report back to you on the Wiley blog.

Microsoft Azure

Microsoft Azure has an emphasis on developers. They breakdown their AI offerings into three AI categories:

- AI services
- AI tools and frameworks
- AI infrastructures

Similar to Amazon and Google, their AI applications are built on consumer products that Microsoft has produced. Azure also has support for specialized FPGA (field programmable gate arrays — think hardware that can be changed by programming) and has built out the infrastructure to support a wide variety of accelerators. Microsoft is one of the largest, if not the largest, customer of the Intel Movidius chips.

They have products for machine learning, IOT toolkits, and management services, and a full and rich set of data services including databases, support for GPUs and custom silicon AI infrastructure, and a container service that can turn your inside applications into cloud apps.

Microsoft Azure is the one to watch for some pretty spectacular innovations.

AI on a Graphics Card

Graphics cards (see the Nvidia Graphics chip in Figure 4-4) have been an integral part of the PC experience for decades. People often hunt for the latest and greatest graphics card to make their PCs better gaming machines. One thing becomes obvious after a while: Although CPU speed is important, the quality and architecture of the graphics card makes a bigger difference. Why? Because computing high-resolution graphics is computationally expensive, and the way to solve that is to build graphics cards out of computers that were designed to do graphics to share the burden. Thus was born the GPU (graphics processing unit), a specialized computer core that is designed to work with graphics.

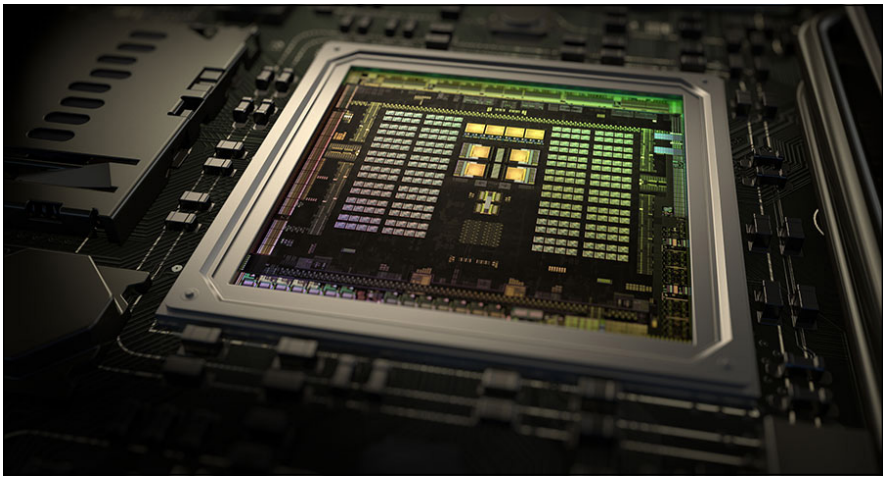


FIGURE 4-4: Nvidia 256 Core GPU chip.

Nvidia and others started building graphics cards that had multiple GPUs on them, which dramatically improved video resolution and frame rates in games. One thing to remember is that graphics algorithms are constructed using data structures called *matrices* (or *tensors*) that are processed in pipelines.

Wait. Tensors? Matrices? This sounds suspiciously like the kind of data structures we use in AI and machine learning. Because of the way machine learning and deep learning are done and implemented, GPUs have proven to be useful and effective.

Deep learning relies on a number of different types of neural networks, (see Chapter 2) and we train and use these networks by using tensors.

Regardless of the type of neural network used, all the techniques rely on performing complex statistical operations. During the training (learning) operations, a multitude of images or data points are fed to the network and then trained

with the correct classification or correct answer. You correlate millions of tensors (matrices) to build a model that will get the right result.

To speed up the training, these operations can be done in parallel, which turns out to be a very good use of the GPUs on a graphics board.

An individual GPU core is much simpler than a CPU core as it is designed for a specific, rather than general, purpose. This makes it cheaper to build multicore GPU chips than to build multicore CPU chips.

The proliferation of graphics cards with many GPU cores has made these computers perfect for machine learning applications. The combination of a powerful multicore CPU and many GPUs can dramatically accelerate machine-learning programs. TensorFlow in particular has versions of the software that is designed to work with GPU boards, removing a lot of the complication of using these boards.

To put it in perspective, our Raspberry Pi 3B+ has 4 processor cores and in some sense, 4 GPU cores. One of the latest GPU boards from Nvidia has 3,584 cores. You can do a lot of fast training and executing machine learning networks using these large core count GPU boards.

The GPU-based boards are not the last step in this evolution of specialized computers and hardware to support AI applications. There are starting to be even more specialized chips. At last count, there are over 50 companies working on chips that will accelerate AI functions.

When we discussed the Microsoft Azure cloud offering earlier, we mentioned that Microsoft has built out infrastructure to support AI acceleration hardware in the cloud. This is one of the big reasons to watch what Microsoft is doing.

The future is in more and more specialized hardware, especially as specialized hardware gets easier and easier to deal with from the user software side.

Where to Go for More AI Fun in Python

If you are interested in furthering your knowledge and abilities in machine learning and AI, check out the following sources for project inspiration. The important thing is to actually build programs and modify other people programs to really learn the technology from experience.

- "Is Santa Claus Real?," Varun Vohra, https://towardsdatascience.com/is-santa-claus-real-9b7b9839776c
- "Keras and deep learning on the Raspberry Pi," Adrian Rosebrock, https://www.pyimagesearch.com/2017/12/18/keras-deep-learning-raspberry-pi/
- "How to easily Detect Objects with Deep Learning on Raspberry Pi," Sarthak Jain, https://medium.com/nanonets/how-to-easily-detect-objects-with-deep-learning-on-raspberrypi-225f29635c74
- "Building a Cat Detector using Convolutional Neural Network," Venelin Valkov, https://medium.com/@curiousily/tensorflow-for-hackers-part-iii-convolutional-neural-networks-c077618e590b
- "Real time Image Classifier on Raspberry Pi Using Inception Framework," Bapi Reddy, https://medium.com/@bapireddy/real-time-image-classifier-on-raspberry-pi-using-inception-framework-faccfa150909

5 Doing Data Science with Python

Contents at a Glance

IN THIS CHAPTER

- What is data science?
- What is big data?
- What are the five steps of data science?

Chapter 1
The Five Areas of Data Science

Data science impacts our modern lives in far more ways than you may think. When you use Google or Bing or DuckDuckGo, you are using a very sophisticated application of data science. The suggestions for other search terms that come up when you are typing? Those come from data science.

Medical diagnoses and interpretations of images and symptoms are examples of data science. Doctors rely on data science interpretations more and more these days.

As with most of the topics in this book, data science looks intimidating to the uninitiated. Inferences, data graphs, and statistics, oh my! However, just as in our previous chapters on artificial intelligence, if you dig in and look at some examples, you can really get a handle on what data science is and what it isn't.

In this chapter we cover just enough statistics and "asking questions of data" to get you going and get some simple results. The purpose is to introduce you to the use of Python in data science and talk about just enough theory to get you started.

If nothing else, we want to leave you with the process of data science and give you a higher level of understanding of what is behind some of the talking heads on television and the various press releases that come from universities. These people are always citing results that come from big data analysis and are often

overstating what they actually mean. An example of this is when one study says coffee is bad for you and the next month a study comes out saying coffee is good for you — and sometimes the studies are based on the same data! Determining what your results mean, beyond simple interpretations, is where the really hard parts of data science and statistics meet and are worthy of a book all their own. At the end of our data science journey, you will know more about the processes involved in answering some of these questions.

There is a mystery to data science, but with just a little knowledge and a little Python, we can penetrate the veil and do some data science.

Python and the myriad tools and libraries available can make data science much more accessible. One thing to remember is that most scientists (including data scientists) are not necessarily experts in computer science. They like to use tools to simplify the coding and to allow them to focus on getting the answers and performing the analysis of the data they want.

Working with Big, Big Data

The media likes to throw around the notion of "big data" and how people can get insights into consumer (and your) behavior from it. *Big data* is a term used to refer to large and complex datasets that are too large for traditional data processing software (read databases, spread sheets, and traditional statistics packages like SPSS) to handle. The industry talks about big data using three different concepts, called the "Three V's": *volume*, *variety*, and *velocity*.

Volume

Volume refers to how big the dataset is that we are considering. It can be really, really big — almost hard-to-believe big. For example, Facebook has more users than the population of China. There are over 250 *billion* images on Facebook and 2.5 *trillion* posts. That is a lot of data. A really big amount of data.

And what about the upcoming world of IOT (Internet of Things)? Gartner, one of the world's leading analysis companies, estimates 22 billion devices by 2022. That is 22 billion devices producing thousands of pieces of data. Imagine that you are sampling the temperature in your kitchen once a minute for a year. That is over ½ million data points. Add the humidity in to the measurements and now you have 1 million data points. Multiply that by five rooms and a garage, all with

temperature and humidity measurements, and your house is producing 6 million pieces of data from just one little IOT device per room. It gets crazy very quickly.

And look at your smartphone. Imagine how many pieces of data it produces in a day. Location, usage, power levels, cellphone connectivity spews out of your phone into databases and your apps and application dashboards like Blynk constantly. Sometimes (as we just recently found out from cellphone companies) location information is being collected and sold even without your consent or opt-in.

Data, data, and more data. Data science is how we make use of this.

Variety

Note that photos are very different data types from temperature and humidity or location information. Sometimes they go together and sometimes they don't. Photos (as we discovered in Book 4, "Artificial Intelligence and Python") are very sophisticated data structures and are hard to interpret and hard to get machines to classify. Throw audio recordings in on that and you have a rather varied set of data types.

Let's talk about voice for a minute. In Book 4, I talked about Alexa being very good at translating voice to text but not so good at assigning meaning to the text. One reason is the lack of context, but another reason is the many different ways that people ask for things, make comments, and so on. Imagine, then, Alexa (and Amazon) keeping track of all the queries and then doing data science on them to find out the sorts of things that people are asking for and the variety of ways they ask for them. That is a lot of data and a lot of information that can be gathered. Not just for nefarious reasons, but to build a system that better services the consumer. It goes both ways.

Data science has a much better chance of identifying patterns if the voice has been translated to text. It is much easier. However, in this translation you do lose a lot of information about tone of voice, emphasis, and so on.

Velocity

Velocity refers to how fast the data is changing and how fast it is being added to the data piles. Facebook users upload about 1 *billion* pictures a day, so in the next couple of years Facebook will have over 1 *trillion* images. Facebook is a high velocity dataset. A low velocity dataset (not changing at all) may be the set of temperature and humidity readings from your house in the last five years. Needless to say, high velocity datasets take different techniques than low velocity datasets.

THE DIFFERENCE BETWEEN DATA SCIENCE AND DATA ANALYTICS

In a real sense, data analytics is a subset of data science — specifically, steps 3-5 in our data science list. (See "Cooking with Gas: The Five-Step Process of Data Science.") There are a number of people that still like to differentiate between these two types of scientists, but the difference becomes less and less noticeable as time goes on. More and more techniques are being develop to do data analysis on big data (not surprisingly named "big data analytics").

Currently, data science generally refers to the process of working out insights from large datasets of unstructured data. This means using predicative analytics, statistics and machine learning to wade through the mass of data.

Data analytics primarily focuses on using and creating statistical analysis for existing sets of data to achieve insights on that data.

With these somewhat vague descriptions, you can see how the two areas are moving closer and closer together. At the risk of ridicule from my fellow academics, I would definitely call data analytics a subset of data science.

Managing volume, variety, and velocity

This is a very complex topic. Data scientists have developed many methods for processing data with variations of the three V's. The three V's describe the dataset and give you an idea of the parameters of your particular set of data. The process of gaining insights in data is called *data analytics*. In the next chapters, we focus on gaining knowledge about analytics and on learning how to ask some data analytics questions using Python. After doing data science for a few years, you will be VVVery good at managing these.

Cooking with Gas: The Five Step Process of Data Science

We generally can break down the process of doing science on data (especially big data) into five steps. I'll finish out this introductory chapter by talking about each of these steps to give us a handle on the flow of the data science process and a feel for the complexity of the tasks. These steps are

1. Capture the data
2. Process the data
3. Analyze the data
4. Communicate the results
5. Maintain the data

Capturing the data

To have something to do analysis on, you have to capture some data. In any real-world situation, you probably have a number of potential sources of data. Inventory them and decide what to include. Knowing what to include requires you to have carefully defined what your business terms are and what your goals are for the upcoming analysis. Sometimes your goals can be vague in that sometimes, "you just want to see what you can get" out of the data.

If you can, integrate your data sources so it is easy to get to the information you need to find insights and build all those nifty reports you just can't wait to show off to the management.

Processing the data

In my humble opinion, this is the part of data science that should be easy, but it almost never is. I've seen data scientists spend months massaging their data so they can process and trust the data. You need to identify anomalies and outliers, eliminate duplicates, remove missing entries, and figure out what data is inconsistent. And all this has to be done appropriately so as not to take out data that is important to your upcoming analysis work. It's not easy to do in many cases. If you have house room temperatures that are 170 degrees C, it is easy to see that this data is wrong and inconsistent. (Well, unless your house is burning down.)

Cleaning and processing your data needs to be done carefully or else you will bias and maybe destroy the ability to do good inferences or get good answers down the line. In the real world, expect to spend a lot of time doing this step.


WARNING

Oh, and one more cleaning thing to worry about, budding data scientist consumers are giving more and more false and misleading data online. According to Marketing Week in 2015, 60 percent of consumers provide intentionally incorrect information when submitting data online.

We humbly admit to doing this all the time to online marketing forms and even to political pollsters, especially when we sense a political agenda in the questions. Bad boys we are.

Understand that it only takes a very small amount of disproportionate information to dramatically devalue a database. More food for thought.

Analyzing the data

By the time you have expended all the energy to get to actually looking at the data to see what you can find, you would think that asking the questions should be relatively simple. It is not. Analyzing big datasets for insights and inferences or even asking complex questions is the hardest challenge, one that requires the most human intuition in all of data science. Some questions, like "What is the average money spent on cereal in 2017?" can be easily defined and calculated, even on huge amounts of data. But then you have the really, really useful questions such as, "How can I get more people to buy Sugar Frosted Flakes?" Now, that is the $64,000 question. In a brazen attempt to be more scientific, we will call Sugar Frosted Flakes by the acronym SFF.

A question such as that has layers and layers of complexity behind it. You want a baseline of how much SFF your customers are currently buying. That should be pretty easy. Then you have to define what you mean by *more people*. Do you really mean *more people*, or do you mean *more revenue*? Change the price to $0.01 per box, and you will have lots more people buying SFF. You really want more revenue or more even more specifically, more margin (margin = price – cost). The question is already more complex.

But the real difficult part of the question is just how are we going to motivate people to buy more SFF? And is the answer in our data that we have collected?

That is the hard part of analysis: Making sure we are asking the right question in the right way of the right kind of data.

Analyzing the data requires skill and experience in statistics techniques like linear and logistic regressions and finding correlations between different data types by using a variety of probability algorithms and formulas such as the incredibly coolly named "Naïve Bayes" formulas and concepts. Although a full discussion of these techniques is out of the scope of this book, we go through some examples later.

Communicating the results

After you have crunched and mangled your data into the format you need and then have analyzed the data to answer your questions, you need to present the results to management or the customer. Most people visualize information better and faster when they see it in a graphical format rather than just in text. There are two major Python packages that data science people us: The language "R" and

MatPlotLib. We use MatPlotLib in displaying our "big data graphics." (If you have read the chapters on AI (Book 4), then you have already experienced MatPlotLib firsthand.)

Maintaining the data

This is the step in data science that everyone ignores. After you have asked your first round of questions, got your first round of answers many professionals will just basically shut down and walk away to the next project. The problem with that way of thinking is that there is a very reasonable chance that you will have to ask more questions of the same data, sometimes quite far in the future. Is important to archive and document the following information so you can restart the project quickly, or even more likely in the future you will run across a similar set of problems and can quickly dust the models off and get to answers faster.

Take time to preserve:

- The data and sources
- The models you used to modify the data (including any exception data and "data throw-out criteria" you used)
- The queries and results you got from the queries

IN THIS CHAPTER

- Using NumPy for data science
- Using pandas for fast data analysis
- Our first data science project
- Visualization with MatPlotLib in Python

Chapter 2
Exploring Big Data with Python

In this chapter we get into some of the tools and processes used by data scientists to format, process, and query their data.

There are a number of Python-based tools and libraries (such as "R") available, but we decided to use NumPy for three reasons. First, it is one of the two most popular tools to use for data science in Python. Second, many AI-oriented projects use NumPy (such as the one in our last chapter). And third, the highly useful Python data science package, Pandas, is built on NumPy.

Pandas is turning out to be a very important package in data science. The way it encapsulates data in a more abstract way makes it easier to manipulate, document, and understand the transformations you make in the base datasets.

Finally, MatPlotLib is a good visualization package for the results of big data. It's very Python-centric, but it suffers from a steep learning curve to get going. However, this has been ameliorated to some degree by new add-on packages, such as "seaborne."

All in all, these are reasonable packages to attack the data science problem and get some results to introduce you to this interesting field.

Introducing NumPy, Pandas, and MatPlotLib

Anytime you look at the scientific computing and data science communities, three key Python packages keep coming up:

- NumPy
- Pandas
- MatPlotLib

These are discussed in the next few sections.

NumPy

NumPy adds big data-manipulation tools to Python such as large-array manipulation and high-level mathematical functions for data science. NumPy is best at handling basic numerical computation such as means, averages, and so on. It also excels at the creation and manipulation of multidimensional arrays known as tensors or matrices. In Book 4, we used NumPy extensively in manipulating data and tensors in neural networks and machine learning. It is an exceptional tool for artificial intelligence applications.

TIP

There are numerous good tutorials for NumPy on the web. A selection of some of good step-by-step ones are:

- **NumPy Tutorial Part 1 – Introduction to Arrays (`https://www.machinelearningplus.com/python/numpy-tutorial-part1-array-python-examples/`):** A good introduction to matrices (also known as tensors) and how they fit into NumPy.
- **NumPy Tutorial (`https://www.tutorialspoint.com/numpy`):** A nice overview of NumPy, where it comes from and how to use it.
- **NumPy Tutorial: Learn with Example (`https://www.guru99.com/numpy-tutorial.html`):** Less theory, but a bunch of great examples to fill in the practical gaps after looking at the first two tutorials.

Here's a simple example of a NumPy program. This program builds a 2x2 matrix, then performs various matrix-oriented operations on the maxtrix:

```
import numpy as np

x = np.array([[1,2],[3,4]])
```

```
print(np.sum(x))  # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

Pandas

Python is great for munging data and preparing data, but not so great for data analysis and modeling. Pandas fills this gap.

Pandas provides fast, flexible, and expressive data structures to make working with relational or labeled data more intuitive. In our opinion, it is the fundamental building block for doing real-world data analysis in Python. It performs well with tabular type of data (such as SQL tables or Excel spreadsheets) and is really good with time-series data (like, say, temperatures taken on a hourly basis).

Remember our discussion on data massaging? Dealing with missing or bad data? This is one of things that Pandas is designed for and does really well. It also allows for complex hierarchical data structures, which can be accessed using Pandas functions in a very intuitive way. You can merge and join datasets as well as convert many types of data into the ubiquitous Pandas data objects, DataFrames.

Pandas is based on NumPy and shares the speed of that Python library, and it can achieve a large increase of speed over straight Python code involving loops.

Pandas DataFrames are a way to store data in rectangular grids that can easily be overviewed. A DataFrame can contain other DataFrames, a one-dimensional series of data, a NumPy tensor (an array — here we go again with similarities to Book 4 on neural networks and machine learning), and dictionaries for tensors and matrices.

Besides data, you can also specify indexes and column names for your DataFrame. This makes for more understandable code for data analysis and manipulation. You can access, delete, and rename your DataFrame components as you bring in more structures and join more related data into your DataFrame structure.

MatPlotLib

MatPlotLib is a library that adds the missing data visualization functions to Python. It is designed to complement the use of NumPy in data analysis and scientific programs. It provides a Python object-oriented API (applications programming interface) for embedded plots into applications using general-purpose GUI interfaces. For those familiar with MatLab, MatPlotLib provides a procedural version called PyLab.

With MatPlotLib, you can make elaborate and professional-looking graphs, and you can even build "live" graphs that update while your application is running. This can be handy in machine-learning applications and data-analysis applications, where it is good to see the system making progress towards some goal.

Doing Your First Data Science Project

Time for us to put NumPy and Pandas to work on a simple data science project.

I am going to choose our dataset from the website Kaggle.com. Kaggle, whose tag line is "Your Home for Data Science," is a Google-owned online community of data scientists and users. Kaggle allows users to find datasets, download the data, and use the data under very open licenses, in most cases. Kaggle also supports a robust set of competitions for solving machine-learning problems, often posted by companies that really need the solution.

For this first problem, I want to choose a pretty simple set of data.

Diamonds are a data scientist's best friend

I chose the "diamonds" database from Kaggle.com because it has a fairly simple structure and only has about 54,000 elements — easy for our Raspberry Pi computer to use. You can download it at `https://www.kaggle.com/shivam2503/diamonds`. Using Kaggle will require you to register and sign in to the community, but does not cost anything to do so.

The metadata (*metadata* is data describing data, hence *meta*data) consists of ten variables, which also can be thought of as column headers. (See Table 2-1.)

TABLE 2-1 Columns in the Diamond Database

Column Header	Type of Data	Description
Index counter	`Numeric`	
carat	`Numeric`	Carat weight of the diamond
cut	`Text`	Describe cut quality of the diamond. Quality in increasing order Fair, Good, Very Good, Premium, Ideal
color	`Text`	Color of the diamond, with D being the best and J the worst

Column Header	Type of Data	Description
clarity	Text	How obvious inclusions are within the diamond: (in order from best to worst, FL = flawless, I3= level 3 inclusions) FL,IF, VVS1, VVS2, VS1, VS2, SI1, SI2, I1, I2, I3
depth	Numeric	Depth %: The height of a diamond, measured from the culet to the table, divided by its average girdle diameter
table	Numeric	Table %: The width of the diamond's table expressed as a percentage of its average diameter
price	Numeric	The price of the diamond
x	Numeric	Length mm
y	Numeric	Width mm
x	Numeric	Depth mm

If you were to use this as a training set for a machine-learning program, you would see a program using NumPy and TensorFlow very similar to the one we show you in Book 4. In this chapter, we are going to show you a set of simple pandas-based data analysis to read our data and ask some questions.

I'm going to use a DataFrame in pandas (a 2D-labeled data structure with columns that can be of different types). The Panels data structure is a 3D container of data. I am sticking with DataFrames in this example because DataFrames makes it easier to visualize 2D data.

TIP

If you are installing NumPy and pandas on the Raspberry Pi, use these commands:

```
sudo apt-get install python3-numpy
sudo apt-get install python3-pandas
```

Now it is time for an example.

Using nano (or your favorite text editor), open up a file called `FirstDiamonds.py` and enter the following code:

```
# Diamonds are a Data Scientist's Best Friend

#import the pandas and numpy library
import numpy as np
import pandas as pd


# read the diamonds CSV file
```

```
# build a DataFrame from the data
df = pd.read_csv('diamonds.csv')


print (df.head(10))
print()

# calculate total value of diamonds
sum = df.price.sum()
print ("Total $ Value of Diamonds: ${:0,.2f}".format( sum))

# calculate mean price of diamonds

mean = df.price.mean()
print ("Mean $ Value of Diamonds: ${:0,.2f}".format(mean))

#  summarize the data
descrip = df.carat.describe()
print()
print (descrip)

descrip = df.describe(include='object')
print()
print (descrip)
```

Making sure you have the `diamonds.csv` file in your directory, run the following command:

```
python3 FirstDiamonds.py
```

And you should see the following results:

```
   Unnamed: 0  carat        cut color clarity  depth  table  price     x
   y     z
0           1   0.23      Ideal     E     SI2   61.5   55.0    326  3.95
   3.98  2.43
1           2   0.21    Premium     E     SI1   59.8   61.0    326  3.89
   3.84  2.31
2           3   0.23       Good     E     VS1   56.9   65.0    327  4.05
   4.07  2.31
3           4   0.29    Premium     I     VS2   62.4   58.0    334  4.20
   4.23  2.63
4           5   0.31       Good     J     SI2   63.3   58.0    335  4.34
   4.35  2.75
5           6   0.24  Very Good     J    VVS2   62.8   57.0    336  3.94
   3.96  2.48
6           7   0.24  Very Good     I    VVS1   62.3   57.0    336  3.95
   3.98  2.47
```

```
7          8   0.26  Very Good     H     SI1   61.9   55.0    337  4.07
   4.11  2.53
8          9   0.22       Fair     E     VS2   65.1   61.0    337  3.87
   3.78  2.49
9         10   0.23  Very Good     H     VS1   59.4   61.0    338  4.00
   4.05  2.39

Total $ Value of Diamonds: $212,135,217.00
Mean $ Value of Diamonds: $3,932.80

count    53940.000000
mean         0.797940
std          0.474011
min          0.200000
25%          0.400000
50%          0.700000
75%          1.040000
max          5.010000
Name: carat, dtype: float64

          cut  color clarity
count   53940  53940   53940
unique      5      7       8
top     Ideal      G     SI1
freq    21551  11292   13065
```

That's a lot of data for a short piece of code!

Breaking down the code

```
# Diamonds are a Data Scientist's Best Friend
```

First, we import all the needed libraries:

```
#import the pandas and numpy library
import numpy as np
import pandas as pd
```

Read the `diamonds` file into a pandas DataFrame. Note: We didn't have to format and manipulate the data in this file. This is not the normal situation in real data science. You will often spend a significant amount of time getting your data where you want it to be — sometimes as much time as the entire rest of the project.

```
# read the diamonds CSV file
# build a DataFrame from the data
df = pd.read_csv('diamonds.csv')
```

Just for a sanity check, let's print out the first ten rows in the DataFrame.

```
print (df.head(10))
print()
```

Here we calculate a couple of values from the column named price. Note that we get to use the column as part of the DataFrame object. It's great that you can do this with Python!

```
# calculate total value of diamonds
sum = df.price.sum()
print ("Total $ Value of Diamonds: ${:0,.2f}".format( sum))

# calculate mean price of diamonds

mean = df.price.mean()
print ("Mean $ Value of Diamonds: ${:0,.2f}".format(mean))
```

Now we run the built-in describe function to first describe and summarize the data about `carat`.

```
#  summarize the data
descrip = df.carat.describe()
print()
print (descrip)
```

This next statement prints out a description for all the nonnumeric columns in our DataFrame: specifically, the cut, color, and clarity columns:

```
descrip = df.describe(include='object')
print()
print (descrip)
```

TIP

To install MatPlotLib on your Raspberry Pi, type **`pip3 install matplotlib`**.

Visualizing the data with MatPlotLib

Now we move to the data visualization of our data with MatPlotLib. In Book 4 we use MatPlotLib to draw some graphs related to the way our machine-learning program improved its accuracy during training. Now we use MatPlotLib to show some interesting things about our dataset.

WARNING

For these programs to work, you need to be running them from a terminal window inside the Raspberry Pi GUI. You can use VNC to get a GUI if you are running your Raspberry Pi headless.

One of the really useful things about pandas and MatPlotLib is that the NumPy and DataFrame types are very compatible with the required graphic formats. They are all based on matrices and NumPy arrays.

Our first plot is a scatter plot showing diamond clarity versus diamond carat size.

Diamond clarity versus carat size

Using nano (or your favorite text editor), open up a file called `Plot_Clarity VSCarat.py` and enter the following code:

```
# Looking at the Shiny Diamonds

#import the pandas and numpy library
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# read the diamonds CSV file
# build a DataFrame from the data
df = pd.read_csv('diamonds.csv')


import matplotlib.pyplot as plt

carat = df.carat
clarity = df.clarity
plt.scatter(clarity, carat)
plt.show()  # or plt.savefig("name.png")
```

Run your program. Now, how is that for ease in plotting? Pandas and MatPlotLib go hand-in-hand.

Remember that diamond clarity is measured by how obvious inclusions (see Figure 2-1) are within the diamond: FL, IF, VVS1, VVS2, VS1, VS2, SI1, SI2, I1, I2, I3 (in order from best to worst: FL = flawless, I3= level 3 inclusions). Note that we had no flawless diamonds in our diamond database.

One would be tempted to make a statement that the largest diamonds are rated as IF. However, remember that you really have no idea how this data was collected and so you really can't draw such general conclusions. All you can say is that "In this dataset, the clarity 'IL' has the largest diamonds."

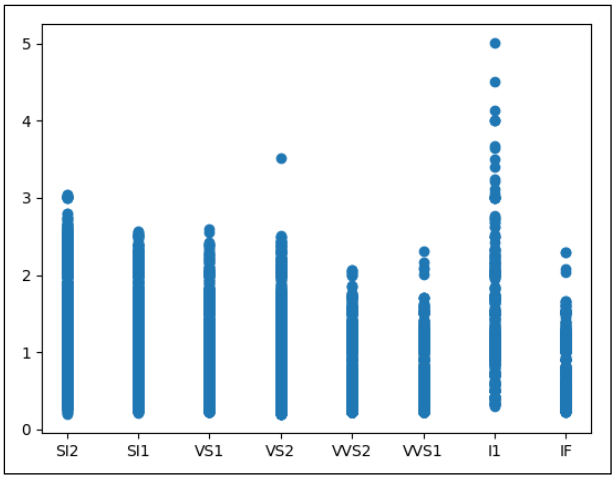


FIGURE 2-1: Diamond clarity (horizontal) versus carat size (vertical).

Number of diamonds in each clarity type

Using nano (or your favorite text editor), open up a file called `Plot_Count Clarity.py` and enter the following code:

```
# Looking at the Shiny Diamonds

#import the pandas and numpy library
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# read the diamonds CSV file
# build a DataFrame from the data
df = pd.read_csv('diamonds.csv')


import matplotlib.pyplot as plt

# count the number of each textual type of clarity

clarityindexes = df['clarity'].value_counts().index.tolist()
claritycount= df['clarity'].value_counts().values.tolist()

print(clarityindexes)
print(claritycount)

plt.bar(clarityindexes, claritycount)
plt.show()  # or plt.savefig("name.png")
```

Run your program. The result is shown in Figure 2-2.

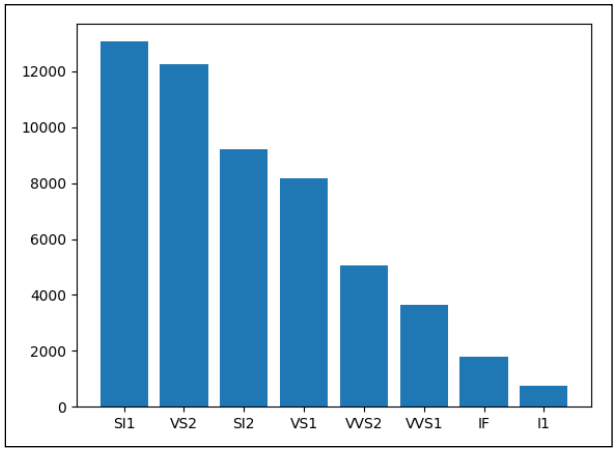


FIGURE 2-2: Diamond clarity count in each type.

Again, remember that diamond clarity is measured by how obvious inclusions are within the diamond: FL,IF, VVS1, VVS2, VS1, VS2, SI1, SI2, I1, I2, I3 (in order from best to worst: FL = flawless, I3= level 3 inclusions). Note that we had no flawless diamonds in our diamond database.

By this graph, we can see that the medium-quality diamonds SI1,VS2, and SI2 are most represented in our diamond dataset.

Number of diamonds in each color type

I looked at clarity, now let's look at color type in our pile of diamonds. Using nano (or your favorite text editor), open up a file called `Plot_CountColor.py` and enter the following code (which generates Figure 2-3):

```
# Looking at the Shiny Diamonds

#import the pandas and numpy library
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# read the diamonds CSV file
# build a DataFrame from the data
df = pd.read_csv('diamonds.csv')


import matplotlib.pyplot as plt
```

```
# count the number of each textual type of color

colorindexes = df['color'].value_counts().index.tolist()
colorcount= df['color'].value_counts().values.tolist()

print(colorindexes)
print(colorcount)

plt.bar(colorindexes, colorcount)
plt.show()  # or plt.savefig("name.png")
```

Run your program.

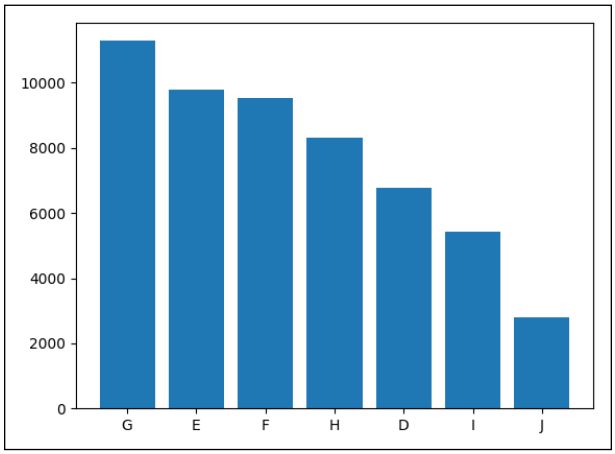


FIGURE 2-3: Diamond color count in each type.

The color "G" represents about 25 percent of our sample size. That "G" is almost colorless. The general rule is less color, higher price. The exceptions to this are the pinks and blues, which are outside of this color mapping and sample.

Using Pandas for finding correlations: Heat plots

The last plot I am going to show you is called a *heat plot*. It is used to graphically show correlations between numeric values inside our database. In this plot we take all the numerical values and create a correlation matrix that shows how closely they correlate with each other. To quickly and easily generate this graph, we use another library for Python and MatPlotLib called seaborn. Seaborn provides an API built on top of MatPlotLib that integrates with pandas DataFrames, which makes it ideal for data science.

TIP

If you don't already have seaborn on your Raspberry Pi (and if you have installed MatPlotLib, you probably already do). Run the example Python program `Plot_Heat.py` to find out whether you do. If not, then run the following command:

```
sudo apt-get install python3-seaborn
```

Using nano (or your favorite text editor), open up a file called `Plot_Heat.py` and enter the following code:

```
# Looking at the Shiny Diamonds

#import the pandas and numpy library
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
import seaborn as sns

# read the diamonds CSV file
# build a DataFrame from the data
df = pd.read_csv('diamonds.csv')

# drop the index column
df = df.drop('Unnamed: 0', axis=1)

f, ax = plt.subplots(figsize=(10, 8))
corr = df.corr()
print (corr)
sns.heatmap(corr, mask=np.zeros_like(corr, dtype=np.bool),
        cmap=sns.diverging_palette(220, 10, as_cmap=True),
                    square=True, ax=ax)

plt.show()
```

Run the program and feast on some real data visualization in Figure 2-4.

The first thing to notice about Figure 2-4 is that the more red the color, the higher the correlation between the two variables. The diagonal stripe from top left to top bottom shows that, for example, carat correlates 100 percent with carat. No surprise there. The *x*, *y*, and *z* variables quite correlate with each other, which says that as the diamonds in our database increase in one dimension, they increase in the other two dimensions as well.

How about price? As carat and size increases, so does price. This makes sense. Interestingly, depth (The height of a diamond, measured from the culet to the table, divided by its average girdle diameter) does not correlated very strongly at all with price and in fact is somewhat negatively correlated.

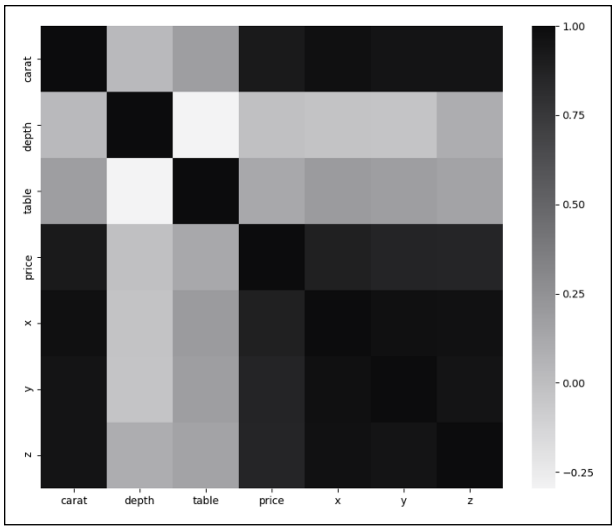


FIGURE 2-4: Correlation heat chart.

It is amazing the amount of inferences you can draw from this kind of a map. Heat maps are fabulous for spotting general cross-correlations in your data.

It would be interesting to see the correlation between color/clarity and price. Why isn't it on this chart? This is because those columns are textual, and you can do only correlations on numerical values. How could you fix that? By substituting a numerical code (1–8, for example) for each color letter and then re-generating the heat chart again. The same technique can be used on diamond clarity.

IN THIS CHAPTER

- Learning how to get access to really big data
- Learning how to use the Google Cloud BigQuery
- Building your first queries
- Visualizing some results with MatPlotLib

Chapter 3
Using Big Data from the Google Cloud

Up to this point, we have been dealing with some relatively small sets of data. Now we are going to use some big data — in some cases, very big data that change every hour!

Sometimes we are working on a powerful enough computer that we can download a large dataset like these, but not every big dataset can be downloaded. For that matter, not all datasets legally can be downloaded. And in the case of the air quality database, you would have download a new version every hour. In cases like those, it's better to leave the data where it is and use the cloud to do the work.

One interesting ramification of doing the heavy lifting in the cloud is that your computer doesn't have to be very big or fast. Just let the cloud do the database and analysis work.

What Is Big Data?

Big data refers to datasets that are too large or complex to be dealt with using traditional data-processing techniques. Data with many cases and many rows offer greater accessibility to sophisticated statistical techniques, and they generally

lead to a smaller false discovery rate. As we discuss in Chapter 1 of this minibook, big data is becoming more and more prevalent in our society as the number of computers and sensors are proliferating and creating more and more data at an ever-increasing rate.

In this chapter, we talk about using the cloud to access these large databases using Python and Pandas and then visualizing the results on a Raspberry Pi.

Understanding the Google Cloud and BigQuery

Well, sometimes to access Big Data, you need to use a BigQuery. It is important to understand that you aren't just storing the data up in the cloud, you are also using the data analysis tools in the cloud. Basically, you are using your computer to command what these computers up in the cloud do with the data.

The Google Cloud Platform

The Google Cloud Platform is a suite of cloud computing services that run on the same infrastructure as Google end-user products such as Google Search and YouTube. This is a cloud strategy that has been successfully used at Amazon and Microsoft. Using your own data services and products to build a cloud offering really seems to produce a good environment for both the user and the company to benefit from advances and improvements to both products and clouds.

The Google Cloud Platform has over 100 different APIs (application programming interfaces) and data service products available for data science and artificial intelligence uses. The primary service we use in this chapter is the Google API called BigQuery.

BigQuery from Google

A REST (Representational State Transfer) software system is a set of code that defines a set of communication structures to be used for creating web services, typically using http and https requests to communicate. This provides a large set of interoperability for different computers with different operating systems trying to access the same web service.

BigQuery is based on a RESTful web service (think of contacting web pages with URL addresses that ask specific questions in a standard format and then getting a response back just like a browser gets a webpage) and a number of libraries for Python and other languages hide the complexity of the queries going back and forth.

Abstraction in software systems is key to making big systems work and reasonable to program. For example, although a web browser uses HTML to display web pages, there are layers and layers of software under that, doing things like transmitting IP packets or manipulating bits. These lower layers are different if you are using a wired network or a WiFi network. The cool thing about abstraction here is up at the webpage level, we don't care. We just use it.

BigQuery is a serverless model. This means that BigQuery has about the highest level of abstraction in the cloud community, removing the user's responsibility for worrying about spinning up VMs (bringing a new virtual machines online in the cloud), RAM, numbers of CPUs, and so on. You can scale from one to thousands of CPUs in a matter of seconds, paying only for the resources you actually use. Understand that in this book, Google will let you use the cloud for free, so you won't even have to pay at all during your trial.

BigQuery has a large number of public big-data datasets, such as those from Medicare and NOAA (National Oceanic and Atmospheric Agency). We make use of these datasets in our examples below.

One of the most interesting features of BigQuery is the ability to stream data into BigQuery on the order of millions of rows (data samples) per second, data you can start to analyze almost immediately.

We will be using BigQuery with the Python library pandas. The Python library google.cloud provides a Python library that maps the BigQuery data into our friendly pandas DataFrames familiar from Chapter 2.

Computer security on the cloud

We would be remiss if we didn't talk just a little bit about maintaining good computer security when using the cloud. Google accomplishes this by using the IAM (identity and access management) paradigm throughout its cloud offerings. This lets administrators authorize who can take what kind of action on specific resources, giving you full control and visibility for simple projects as well as finely grained access extending across an entire enterprise.

We show you how to set up the IAM authentication in the sections that follow.

THE MEDICARE PUBLIC DATABASE

Medicare is the national health insurance program (single payer) in the United States administered by the Centers for Medicare and Medicade Services (CMS). It provides health insurance for Americans aged 65 and over. It also provides health insurance to younger people with certain disabilities and conditions. In 2017 it provided health insurance to over 58 million individuals.

With 58 million individuals in the system, Medicare is generating a huge amount of big data every year. Google and CMS teamed up to put a large amount of this data on the BigQuery public database so you can take a peek at this data and do some analytics without trying to load it all on your local machine. A home computer, PC or Raspberry Pi, won't hold all the data available.

Signing up on Google for BigQuery

Go to cloud.google.com and sign up for your free trial. Although Google requires a credit card to prove you are not a robot, they will not charge you even when your trial is over without you manually switching over to a paid account. If you exceed $300 during your trial (which you shouldn't), Google will notify you but will not charge you.

The $300 limit to the trial should be plenty enough to allow you to do a bunch of queries and learning on the BigQuery cloud platform.

Reading the Medicare Big Data

Now we show you how to set up a project and get your authentication `.json` file to start using BigQuery on your own Python programs.

Setting up your project and authentication

To access the Google cloud you will need to set up a project and then receive your authentication credentials from Google to be able to use their systems. The following steps will show you how to do this:

1. **Go to `https://console.developers.google.com/` and sign in using your account name and password generated earlier.**

2. **Next, click the My First Project button up in the upper-left corner of the screen.**

   It shows you a screen like the one in Figure 3-1.

3. **Click the New Project button on the pop-up screen.**

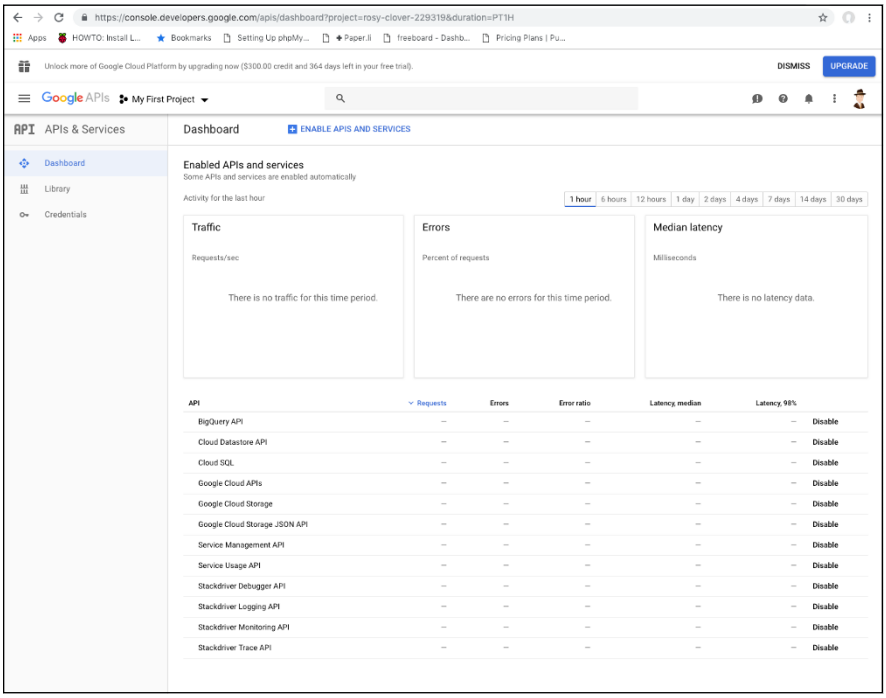


FIGURE 3-1: The Select a Project page on the Google Cloud.

4. **Fill out your project name as** MedicareProject **and click Create.**

5. **Next, select your project,** `MedicareProject`**, from the upper-left menu button.**

REMEMBER

Make sure you don't leave this on the default "My Project" selection. Make sure you change it to `MedicareProject` — otherwise you will be setting up the APIs and authentication for the wrong project. This is an easy mistake to make.

6. **After you have selected** `MedicareProject`**, click on the "+" button near the top to enable the BigQuery API.**

7. **When the API selection screen comes up, search for *BigQuery* and select the BigQuery API. Then click Enable.**

8. **Now to get our authentication credentials. In the left-hand menu, choose Credentials.**

   A screen like the one in Figure 3-2 comes up.

9. **Select the BigQuery API and then click the No, I'm Not Using Them option in the Are You Planning to Use This API with the App Engine or Compute Engine? section.**

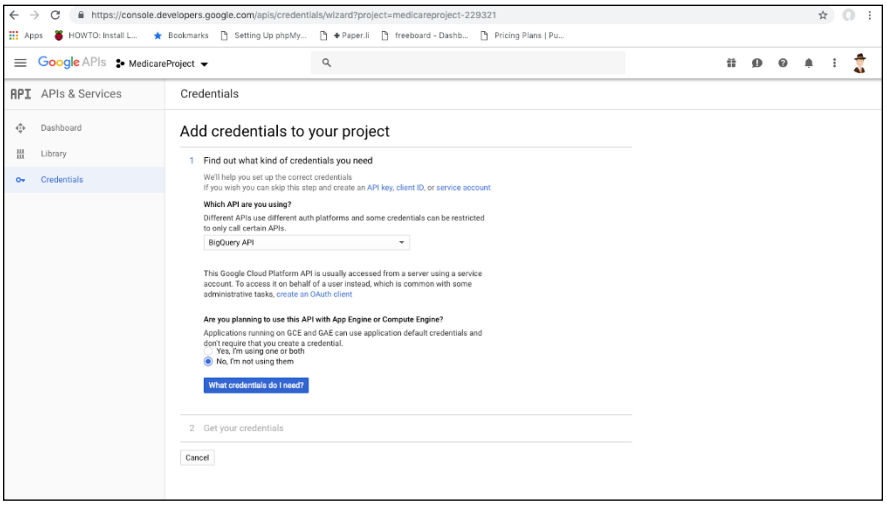


FIGURE 3-2: First credential screen.

10. **Click the What Credentials Do I Need? button to get to our last screen, as shown in Figure 3-3.**

11. **Type** MedicareProject **into the Service Account Name textbox and then select Project➪Owner in the Role menu.**

12. **Leave the JSON radio button selected and click Continue.**

    A message appears saying that the service account and key has been created. A file called something similar to "`MedicareProject-1223xxxxx413.json`" is downloaded to your computer.

13. **Copy that downloaded file into the directory that you will be building your Python program file in.**

Now let's move on to our first example.

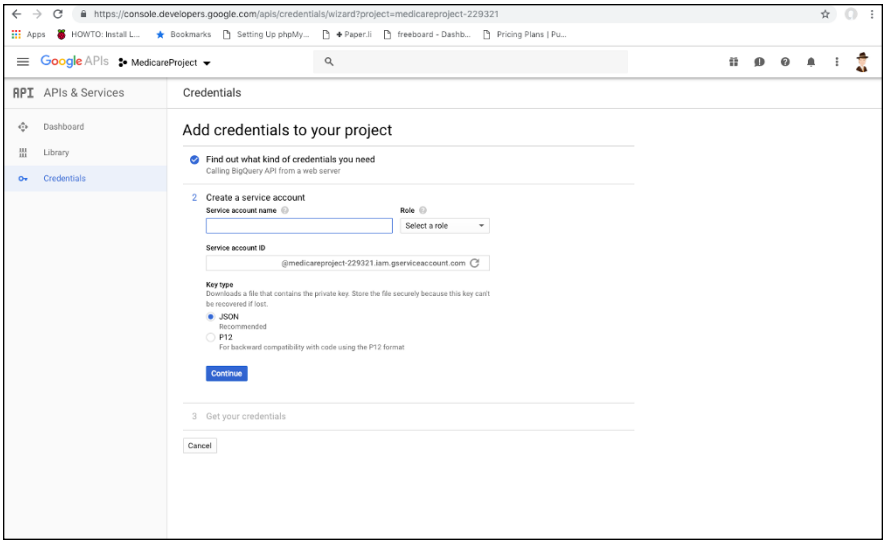


FIGURE 3-3: Second credential screen.

The first big-data code

This program reads one of the public data Medicare datasets and grabs some data for analysis. There are several dozen datasets available now and there will be more and more available with time. We start by using the inpatient_charges_2015 dataset. We use a SQL query to select the information from the dataset that we want to look at and eventually analyze. (Check out the nearby sidebar, "Learning SQL," to see where to learn about SQL if you are not already familiar with this ubiquitous query language.)

Table 3-1 shows all the columns in the inpatient_charges_2015 dataset.

TABLE 3-1 Columns, Types, and Descriptions of the inpatient_charges_2015 Dataset

Column	Type	Description
provider_id	STRING	The CMS certification number (CCN) of the provider billing for outpatient hospital services.
provider_name	STRING	The name of the provider.
provider_street_address	STRING	The street address in which the provider is physically located.
provider_city	STRING	The city in which the provider is physically located.

(continued)

TABLE 3-1 *(continued)*

Column	Type	Description
provider_state	STRING	The state in which the provider is physically located.
provider_zipcode	INTEGER	The zip code in which the provider is physically located.
drg_definition	STRING	The code and description identifying the MS-DRG. MS-DRGs are a classification system that groups similar clinical conditions (diagnoses) and the procedures furnished by the hospital during the stay.
hospital_referral_region_description	STRING	The hospital referral region (HRR) in which the provider is physically located.
total_discharges	INTEGER	The number of discharges billed by the provider for inpatient hospital services.
average_covered_charges	FLOAT	The provider's average charge for services covered by Medicare for all discharges in the MS-DRG. These will vary from hospital to hospital because of differences in hospital charge structures.
average_total_payments	FLOAT	The average total payments to all providers for the MS-DRG including the MSDRG amount, teaching, disproportionate share, capital, and outlier payments for all cases. Also included 5 in average total payments are co-payment and deductible amounts that the patient is responsible for and any additional payments by third parties for coordination of benefits.
average_medicare_payments	FLOAT	The average amount that Medicare pays to the provider for Medicare's share of the MS-DRG. Average Medicare payment amounts include the MS-DRG amount, teaching, disproportionate share, capital, and outlier payments for all cases. Medicare payments *do not* include beneficiary co-payments and deductible amounts nor any additional payments from third parties for coordination of benefits.

Using nano (or any other text editor) enter the following code into your editor and then save it as `MedicareQuery1.py`:

```
import pandas as pd
from google.cloud import bigquery


# set up the query

QUERY = """
        SELECT provider_city, provider_state, drg_definition,
        average_total_payments, average_medicare_payments
        FROM `bigquery-public-data.cms_medicare.inpatient_charges_2015`
        WHERE provider_city = "GREAT FALLS" AND provider_state = "MT"
```

```
        ORDER BY provider_city ASC
        LIMIT 1000
        """

client = bigquery.Client.from_service_account_json(
            'MedicareProject2-122xxxxxf413.json')


query_job = client.query(QUERY)
df = query_job.to_dataframe()

print ("Records Returned: ", df.shape )
print ()
print ("First 3 Records")
print (df.head(3))
```

As soon as you have built this file, replace the `MedicareProject2-122xxxxxf413.json` file with your own authentication filename (which you copied into the program directory earlier).

TIP

If you don't have the google.cloud library installed, type this into your terminal window on the Raspberry Pi:

```
pip3 install google-cloud-bigquery
```

LEARNING SQL

SQL (Structured Query Language) is a query-oriented language used to interface with databases and to extract information from those databases. Although it was designed for relational database access and management, it has been extended to many other types of databases, including the data being accessed by BigQuery and the Google Cloud.

Here are some excellent tutorials to get your head around how to access data using SQL:

- `https://www.w3schools.com/sql/`
- `http://www.sql-tutorial.net/`
- *SQL For Dummies*, Allen G. Taylor
- *SQL All In One For Dummies 3rd Edition,* Allen G. Tayor
- *SQL in 10 Minutes*, Ben Forta

Breaking down the code

First, we import our libraries. Note the `google.cloud` library and the `bigquery` import:

```
import pandas as pd
from google.cloud import bigquery
```

Next we set up the SQL query used to fetch the data we are looking for into a pandas DataFrame for us to analyze:

```
# set up the query

QUERY = """
        SELECT provider_city, provider_state, drg_definition,
        average_total_payments, average_medicare_payments
        FROM `bigquery-public-data.cms_medicare.inpatient_charges_2015`
        WHERE provider_city = "GREAT FALLS" AND provider_state = "MT"
        ORDER BY provider_city ASC
        LIMIT 1000
        """
```

See the structure of the SQL query? We `SELECT` the columns that we want that are given in Table 3-1 `FROM` the database `bigquery-public-data.cms_medicare.inpatient_charges_2015` only `WHERE` the `provider_city` is `GREAT FALLS` and the `provider_state` is `MT`. Finally we tell the system to order the results by ascending alphanumeric order by the `provider_city`. Which, since we only selected one city, is somewhat redundant.

Remember to replace the `json` filename below with your authentication file. This one won't work.

```
client = bigquery.Client.from_service_account_json(
            'MedicareProject2-122xxxxxef413.json')
```

Now we fire the query off to the BigQuery cloud:

```
query_job = client.query(QUERY)
```

And we translate the results to our good friend the Pandas DataFrame:

```
df = query_job.to_dataframe()
```

Now just a few results to see what we got back:

```
print ("Records Returned: ", df.shape )
print ()
print ("First 3 Records")
print (df.head(3))
```

Run your program using `python3 MedicareQuery1.py` and you should see results as below. Note: If you get an authentication error, then go back and make sure you put the correct authentication file into your directory. And if necessary, repeat the whole generate-an-authentication-file routine again, paying special attention to the project name selection.

```
Records Returned:  (112, 5)

First 3 Records
  provider_city provider_state
   drg_ definition  average_total_payments  average_medicare_payments
0   GREAT FALLS            MT  064 - INTRACRANIAL HEMORRHAGE OR CEREBRAL
   INFA...               11997.11                  11080.32
1   GREAT FALLS            MT           039 - EXTRACRANIAL PROCEDURES W/O
   CC/MCC                7082.85                   5954.81
2   GREAT FALLS            MT  065 - INTRACRANIAL HEMORRHAGE OR CEREBRAL
   INFA...                7140.80                   6145.38
Visualizing your Data
```

We found 112 records from Great Falls. You can go back and change the query in your program to select your own city and state.

A bit of analysis next

Okay, now we have established connection with a pretty big-data type of database. Now let's set up another query. We would like to look for patients with "bone diseases and arthropathies without major complication or comorbidity." This is MS_DRG code 554. This is done through one of the most arcane and complicated coding systems in the world, called ICD-10, which maps virtually any diagnostic condition to a single code.

We are going to search the entire `inpatient_charges_2015` dataset looking for the MS_DRG code 554, which is "Bone Diseases And Arthropathies Without Major Complication Or Comorbidity," or, in other words, people who have issues with their bones, but with no serious issues currently manifesting externally.

ICD CODES

ICD10 is the well-established method for coding medical professional diagnoses for billing and analysis purposes. The latest version of ICD-10 was finally made mandatory in 2015 with great angst throughout the medical community. It consists of, at its largest expanse, over 155,000 codes, from M79.603 - Pain in Arm, unspecified to S92.4–Fracture of Greater Toe. These codes are somewhat merged into the MS_DRG codes that are used in the Medicare databases we examine here as they are used for hospital admissions. John Shovic had a medical software startup that used ICD 10 codes for ten years, and he got to have a love/hate relationship with these codes.

His favorite ICD-10 codes:

- V97.33XD: Sucked into jet engine, subsequent encounter.
- Z63.1: Problems in relationship with in-laws.
- V95.43XS: Spacecraft collision injuring occupant, sequela.
- R46.1: Bizarre personal appearance.
- Y93.D1 Activity, knitting and crocheting.

The code for this is as follows:

```
import pandas as pd
from google.cloud import bigquery


# set up the query

QUERY = """
        SELECT provider_city, provider_state, drg_definition,
        average_total_payments, average_medicare_payments
        FROM `bigquery-public-data.cms_medicare.inpatient_charges_2015`
        WHERE drg_definition LIKE  '554 %'
        ORDER BY provider_city ASC
        LIMIT 1000
        """

client = bigquery.Client.from_service_account_json(
            'MedicareProject2-1223283ef413.json')


query_job = client.query(QUERY)
df = query_job.to_dataframe()
```

```
print ("Records Returned: ", df.shape )
print ()
print ("First 3 Records")
print (df.head(3))
```

The only thing different in this program from our previous one is that we added `LIKE '554 %'`, which will match on any DRG that starts with "554."

Running the program gets these results:

```
Records Returned:  (286, 5)

First 3 Records
  provider_city provider_state                                drg_definition
   average_total_payments  average_medicare_payments
0      ABINGTON             PA  554 - BONE DISEASES & ARTHROPATHIES W/O MCC
   5443.67                    3992.93
1         AKRON             OH  554 - BONE DISEASES & ARTHROPATHIES W/O MCC
   5581.00                    4292.47
2        ALBANY             NY  554 - BONE DISEASES & ARTHROPATHIES W/O MCC
   7628.94                    5137.31
```


Now we have some interesting data. Let's do a little analysis. What percent of the total payments for this condition is paid for by Medicare (the remainder paid by the patient)? The code for that will be (let's call it `MedicareQuery3.py`):

```
import pandas as pd
from google.cloud import bigquery


# set up the query

QUERY = """
        SELECT provider_city, provider_state, drg_definition,
        average_total_payments, average_medicare_payments
        FROM `bigquery-public-data.cms_medicare.inpatient_charges_2015`
        WHERE drg_definition LIKE  '554 %'
        ORDER BY provider_city ASC
        LIMIT 1000
        """

client = bigquery.Client.from_service_account_json(
            'MedicareProject2-1223283ef413.json')


query_job = client.query(QUERY)
df = query_job.to_dataframe()
```

```
print ("Records Returned: ", df.shape )
print ()

total_payment = df.average_total_payments.sum()
medicare_payment = df.average_medicare_payments.sum()

percent_paid = ((medicare_payment/total_payment))*100
print ("Medicare pays {:4.2f}% of Total for 554 DRG".format(percent_paid))
print ("Patient pays {:4.2f}% of Total for 554 DRG".format(100-percent_paid))
```

And the results:

```
Records Returned:  (286, 5)

Medicare pays 77.06% of Total for 554 DRG
Patient pays 22.94% of Total for 554 DRG
```

Payment percent by state

Now in this program we select the unique states in our database (not all states are represented) and iterate over the states to calculate the percent paid by Medicare by state for 554. Let's call this one `MedicareQuery4.py`:

```
import pandas as pd
from google.cloud import bigquery


# set up the query

QUERY = """
        SELECT provider_city, provider_state, drg_definition,
        average_total_payments, average_medicare_payments
        FROM `bigquery-public-data.cms_medicare.inpatient_charges_2015`
        WHERE drg_definition LIKE  '554 %'
        ORDER BY provider_city ASC
        LIMIT 1000
        """

client = bigquery.Client.from_service_account_json(
            'MedicareProject2-1223283ef413.json')


query_job = client.query(QUERY)
df = query_job.to_dataframe()
```

```
print ("Records Returned: ", df.shape )
print ()

# find the unique values of State

states = df.provider_state.unique()
states.sort()

total_payment = df.average_total_payments.sum()
medicare_payment = df.average_medicare_payments.sum()

percent_paid = ((medicare_payment/total_payment))*100
print("Overall:")
print ("Medicare pays {:4.2f}% of Total for 554 DRG".format(percent_paid))
print ("Patient pays {:4.2f}% of Total for 554 DRG".format(100-percent_paid))

print ("Per State:")

# now iterate over states

print(df.head(5))
state_percent = []
for current_state in states:
    state_df = df[df.provider_state == current_state]

    state_total_payment = state_df.average_total_payments.sum()

    state_medicare_payment = state_df.average_medicare_payments.sum()

    state_percent_paid = ((state_medicare_payment/state_total_payment))*100
    state_percent.append(state_percent_paid)

    print ("{:s} Medicare pays {:4.2f}% of Total for 554 DRG".format
    (current_state,state_percent_paid))
```


And now some visualization

For our last experiment, let's visualize the state-by-state data over a graph using MatPlotLib. (See Figure 3-4.)Moving over to our VNC program to have a GUI on our Raspberry Pi, we add the following code to the end of the preceding `Medicare Query4.py` code:

```
# we could graph this using MatPlotLib with the two lists
# but we want to use DataFrames for this example

data_array = {'State': states, 'Percent': state_percent}
```

```
df_states = pd.DataFrame.from_dict(data_array)

# Now back in dataframe land
import matplotlib.pyplot as plt
import seaborn as sb

print (df_states)

df_states.plot(kind='bar', x='State', y= 'Percent')
plt.show()
```

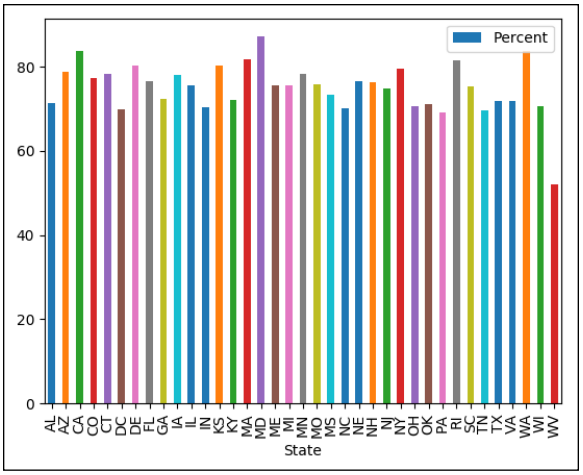


FIGURE 3-4: Bar chart of Medicare % paid per state for 554.

TIP

Do you already have seaborn on your Raspberry Pi (and if you have installed MatPlotLib, you probably already do)? To find out, run the `MedicareQuery4.py` example Python program. If you don't have seaborne installed, then run the following command:

```
sudo apt-get install python3-seaborn
```

Looking for the Most Polluted City in the World on an Hourly Basis

Just one more quick example. There is another public database on BigQuery, OpenAQ, which contains air-quality measurements from 47 countries around the world. And this database is updated *hourly*, believe it or not.

Here is some code that picks up the top three worst polluted cities in the world measured by air quality:

```
import pandas as pd
from google.cloud import bigquery

# sample query from:
QUERY = """
        SELECT location, city, country, value, timestamp
        FROM `bigquery-public-data.openaq.global_air_quality`
        WHERE pollutant = "pm10" AND timestamp > "2017-04-01"
        ORDER BY value DESC
        LIMIT 1000
        """

client = bigquery.Client.from_service_account_json(
                    'MedicareProject2-1223283ef413.json')
query_job = client.query(QUERY)
df = query_job.to_dataframe()

print (df.head(3))
```


Copy this code into a file called PollutedCity.py and run the program.

The current result of running the code (as of the writing of this book) was:

```
        location          city country  value    timestamp
0       Dilovası        Kocaeli   TR  5243.00 2018-01-25 12:00:00+00:00
1  Bukhiin urguu    Ulaanbaatar   MN  1428.00 2019-01-21 17:00:00+00:00
2  Chaiten Norte  Chaiten Norte   CL   999.83 2018-04-24 11:00:00+00:00
```

It looks like Dilovasi, Kocaeli, Turkey is not a healthy place to be right now. Doing a quick Google search of Dilovasi finds that cancer rates are three times higher than the worldwide average. This striking difference apparently stems from the environmental heavy metal pollution persisting in the area for about 40 years, mainly due to intense industrialization.

I'll definitely be checking this on a daily basis.

6 Talking to Hardware with Python

Contents at a Glance

IN THIS CHAPTER

- Discovering how to use a Raspberry Pi
- Understanding how to use small computers
- Using a Raspberry Pi to sense the environment around you
- Making your computer do physical things

Chapter 1

Introduction to Physical Computing

We have been talking about how to program in Python for the last several hundred pages in this book. It is now time to use our newly acquired Python skills to start doing things in the real world. We call this *physical computing* — making a computer interact with the world around you!

In our opinion, it is more difficult to learn about the software (Python) than it is about the hardware. That's why this book is mostly focused on learning how to program computers with Python. But now it is time to learn how to make your computers *do* something with Python.

WARNING

In this chapter, we hook up various sensors and motors to a Raspberry Pi computer. Although the voltages (3.3V and 5V) used with these computers are not dangerous to people, hooking up things incorrectly can burn out your computer or your sensors. For this reason, follow these two rules assiduously:

- **Rule 1:** Turn *all* the power off before you hook up or change any wires.
- **Rule 2:** Double-check your connections, especially the power connections, power and ground. These are the most important wires to check. See the next chapter on why these are so important!

Physical Computing Is Fun

One reason that we want to you to learn about physical computing is that little computers doing physical things (typically called *embedded systems*) are everywhere around you. And we mean everywhere. Go up to your kitchen. Look around. Your refrigerator has a computer, maybe two or three if it has a display. Your blender has a computer. Your oven has a computer. Your microwave has a computer. If you use Phillips Hue lights in your house, your light bulbs have a computer. Your car will have upwards of 20 computers in the vehicle.

One more example. How about the lowly toaster? If you have a "Bagel Button" or a display on your toaster, you have a computer in there. Why? Why are there so many computers in your house? Because it is significantly less expensive to build all your gadgets using a computer than it is to design special hardware. Do you know you can buy computers (in bulk) for about $0.15? Computers are everywhere.

Most of these computers are much simpler, slower, and carrying much less RAM (a form of storage) than your average PC. A PC may have about 4–16 or more GB (that's gigabytes, and 1GB equals approximately 1 billion bytes), but the computer running your toaster probably only has about 100 bytes. This is a difference of over 10,000,000 times the amount of RAM. By the way, you can think of one byte as equal to one character in English. In most Asian countries, one character equals two bytes.

So computers are everywhere. But the interesting thing is that all these little computers are doing physical computing. They are sensing and interacting with the environment. The refrigerator computer is checking to see whether it is at the right temperature, and if it is not, it turns on the cooling machinery, paying attention to what it is doing to minimize the amount of electricity it uses. The stove is updating your display on the front panel, monitoring the buttons and dials and controlling the temperature so you get a good lasagna for dinner.

All of these interactions and controllers are called *physical computing*.

What Is a Raspberry Pi?

In this book, we could use one of these very small computers but the functionality is limited compared to your PC, so we will compromise and use the Raspberry Pi, a $35 computer that has an immense amount of hardware and software available

for use (especially in Python). It's more complex than a toaster, but it's much simpler than even the computer used in your TV.

A Raspberry Pi is a popular SBC (single board computer) that has been around since about 2012. It was created by the Raspberry Pi Foundation to teach basic science and engineering in schools around the world. It turned out to be wildly popular and has sold more than 19 million computers around the world. There a are a bunch of other Raspberry Pi models available (from the $5 Raspberry Pi Zero to the new Raspberry Pi 3B+ we will be using in this book).

To demystify some of the technology that we deal with every day, let's talk about the major blocks of hardware on this computer. Remember, your smartphone has computers inside that are very similar in terms of structure to the Raspberry Pi.

Figure 1-1 shows you the major blocks of the computer:

- **GPIO connector:** This is the general purpose input-output pin connector. We will be using this connector a great deal through the rest of this minibook.
- **CPU/GPU:** Central processing unit/graphics (for the screen) processing unit. This block is the brains of the gear and tells everything else what to do. Your Python programs will be run by this block.
- **USB:** These are standard USB (universal serial bus) ports, the same interfaces you find on big computers. There are many devices you can connect to a USB port, just as on your PC. You will plug in your mouse and keyboard into these ports.
- **Ethernet:** Just like the Ethernet interface on your computer. Connects to a network via wires.
- **WiFi:** This block isn't shown on the diagram, but it is very handy to have. With WiFI, you don't have to trail wires all over to talk with the Internet.
- **HDMI Out:** You plug in your monitor or TV into this port.
- **Audio jack:** Sound and composite video (old standard).
- **Other ports:** Three more interesting ports on the board are:
  - *Micro USB:* This is your 5V power supply.
  - *Camera CSI:* This is for a ribbon cable connection to a Raspberry Pi camera.
  - *Display DSI:* This is for high-speed connections to a variety of custom displays — but this is well beyond the scope of our book.

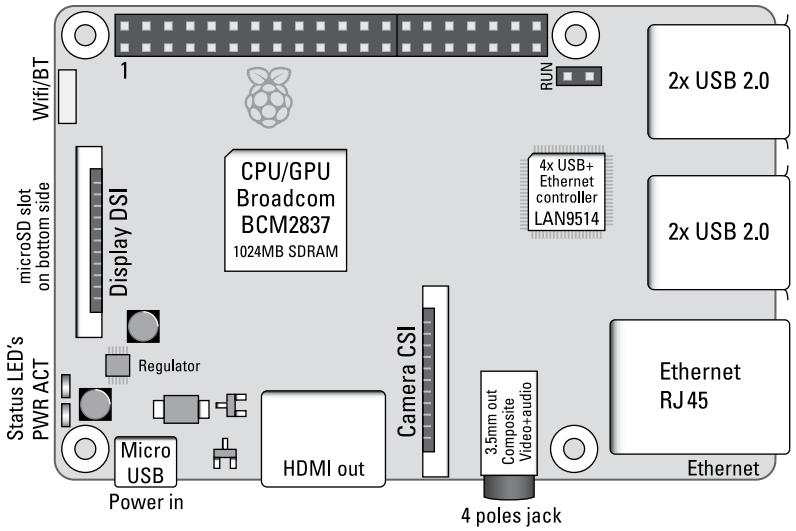


FIGURE 1-1: The main components of the Raspberry Pi 3B+.

Making Your Computer Do Things

In order to get our computer to do and sense things apart from the computer screen and keyboard, we need a computer and one of two other things — a sensor or an actuator. A *sensor* is a small piece of electronics that can detect something about the environment (such as temperature or humidity) and an *actuator* is a fancy word for a motor or cable that does things in the real world.

In the remainder of this chapter, we are going to learn about the necessary ingredients of our first physical computing project, turning an LED on and off. This is the physical computing version of "Hello World" that we all do when we are learning software. Blinking LED, here we come!

TIP

Go out now and buy your Raspberry Pi Starter Kit (comes with the power supply, operating system, and a case) and get it set up before continuing. We recommend grabbing a mouse, keyboard, and monitor to do the set-up for beginners, but more advanced users may want to use SSH (Secure SHell) to do a headless setup. Again, the best place to start is with `www.raspberrypi.org`.

Using Small Computers to Build Projects That Do and Sense Things

Earlier in this chapter, we talked about computers in the kitchen. All those computers sense the environment (the oven temperature, for example) and most are

actually doing something to affect the environment (your blender chopping up ice for a nice Margarita, for example). That pulse, pulse, pulse of your blender is controlled by a computer.

So that we can build projects and learn how to design our own (and believe me, after you get acquainted with the hardware, you are going to be able to design magical things) we need to just jump in and do our first project.

Then, in further chapters, we build more complex things that will be the launching point to your own projects, all programmed in Python!

One last remark before we move on. The software you will be writing in Python is the key to getting all these projects and computers working. Is the hardware or software more important? This is a holy war among engineers, but we really think the software is the more important part, and the easier part to learn for beginners. Now before that statement unleashes a hundred nasty emails, let me humbly acknowledge that none, and we mean none, of this is possible if it wasn't for the hardware and the fine engineers that produce these little miracles. But this is a book on Python!

WHAT ARE SOME OF THE OTHER SMALL COMPUTERS AVAILABLE?

There are literally hundreds of different types of small computers and boards out there for project building. We chose the Raspberry Pi for this book because of the ease of use for Python and the hundreds of Python libraries available for the Raspberry Pi. It is also the best-supported small computer out there with hundreds of websites (including the fabulous `www.raspberrypi.org`) to teach you how to set up and use this computer.

In a real sense, there are two general categories of small computer systems out there that are accessible to the beginning user. There are computers that are based on the Linux operating system (Raspbian, the software on the Raspberry Pi, is a form of Linux) and computers that have a much smaller operating system or even no operating system.

Understand that both versions of computers and operating systems are very useful in different applications.

The Raspberry Pi uses Linux. Linux is a multitasking, complex operating system that can run with multiple CPU cores. (Did we mention the Raspberry PI 3B+ has four CPUs on the chip? And for $35 — amazing!) However, don't confuse the complexity of the operating system with the ability to use the computer. The Raspberry Pi operating

(continued)

(continued)

system supports a whole Windows-like GUI (Graphical User Interface), just like your PC or Mac. The power of the operating system makes this possible.

Arduinos are small computers that only have a small computer and a limited amount of RAM on board. Interestingly enough, even though they are much smaller and simpler than the Raspberry Pi, the development boards are about the same price. In volume however, the Arduino type of computer is much, much cheaper than a Raspberry Pi. An Arduino has much more input-output pins than a Raspberry Pi and has an onboard ADC (analog digital converter), something that the Raspberry Pi lacks. In a later chapter, we show you how to create a project with an external ADC and the Raspberry Pi. And it involves a flame. You know that will be fun.

Another class of small computers similar to Arduinos (and can be programmed by the same IDE (integrated development environment) that Arduino devices use are the ESP8266 and ESP32 boards from Espressif in China. The small computers have much less RAM than the Raspberry Pi, but they come with built-in WiFi (and sometimes Bluetooth) interfaces that make them useful in building projects you want to connect to the Internet, such as IOT (Internet Of Things) projects.

Both types of computers are fun to play with, but the Raspberry Pi has a much better environment for Python development and learning.

The Raspberry Pi: A Perfect Platform for Physical Computing in Python

By now you have your Raspberry Pi computer set up and running on your monitor, keyboard, and mouse. If not, go do that now (remember our friend, `www.raspberrypi.org`) The next few paragraphs are going to be lots more fun with a computer to work with!

The Raspberry Pi is the perfect platform to do physical computing with Python because it has a multiscreen environment, lots of RAM and storage to play with and all the tools to build the projects we want.

We have been talking a lot about the computers in this chapter and not much about Python. Time to change that.

A huge and powerful feature of the Raspberry Pi is the row of GPIO (general purpose input-output) pins along the top of the Raspberry Pi. It is a 40-pin header into which we can plug a large number of sensors and controllers to do amazing things to expand your Raspberry Pi.

GPIO pins

GPIO pins can be designated (using Python software) as input or output pins and used for many different purposes. There are two 5V power pins, two 3.3V power pins and a number of ground pins that have fixed uses (see the description of what voltages (V) are in the next chapter and the differences between 3.3V and 5V). (See Figure 1-2.)

An GPIO pin output pin "outputs" a 1 or a 0 from the computer to the pin. See next chapter for more on how this is done and what it means. Basically, A "1" is 3.3V and a "0" is 0V. We can think of them just as 1s and 0s. (See Figure 1-2.)

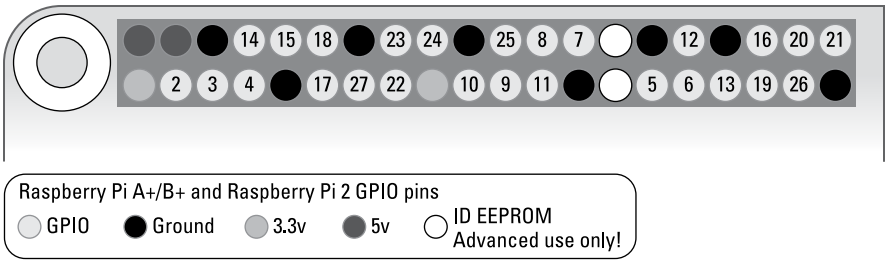


FIGURE 1-2: The functions of the Raspberry Pi GPIO pins.

GPIO libraries

There are a number of GPIO Python libraries that are usable for building projects. The one we use throughout the rest of this book is the `gpiozero` library that is installed on all Raspberry Pi desktop software releases. The library documentation and installation instructions (if needed) are located on `https://gpiozero.readthedocs.io/en/stable/`.

Now we are going to jump into the "Hello World" physical computing project with our Raspberry Pi.

The hardware for "Hello World"

To do this project, we need some hardware. Because we are using Grove connectors (see next chapter) in the rest of the book, let's get the two pieces of Grove hardware that we need for this project:

- **Pi2Grover:** This converts the Raspberry Pi GPIO pin header to Grove connectors (ease of use and can't reverse the power pins!). You can buy this either at shop.switchdoc.com or at Amazon.com. You can get $5.00 off the Pi2Grover board at shop.switchdoc.com by using the discount code PI2DUMMIES at checkout. Lots more on this board in the next chapter. (See Figure 1-3.)

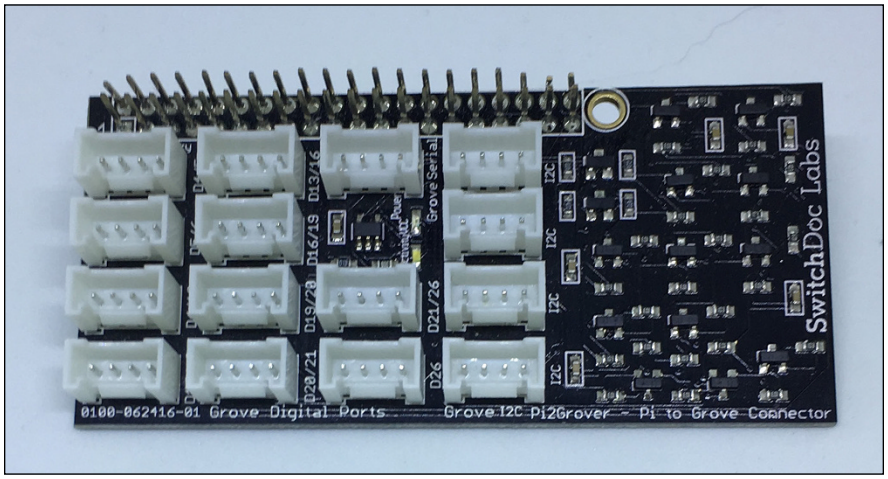


FIGURE 1-3: The Pi2Grover board.

- **Grove blue LED:** A Grove blue LED module including Grove cable. You can buy this on shop.switchdoc.com or on amazon.com. (See Figures 1-4 and 1-5.)

Assembling the hardware

For a number of you readers, this will be the first time you have ever assembled a physical computer based product. because of this, we'll give you the step-by-step process:

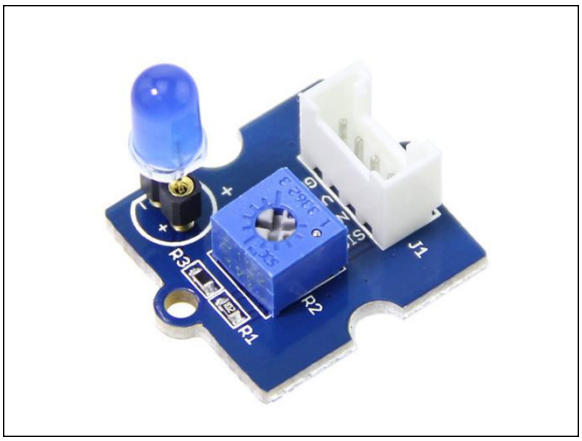


FIGURE 1-4: The Grove blue LED.

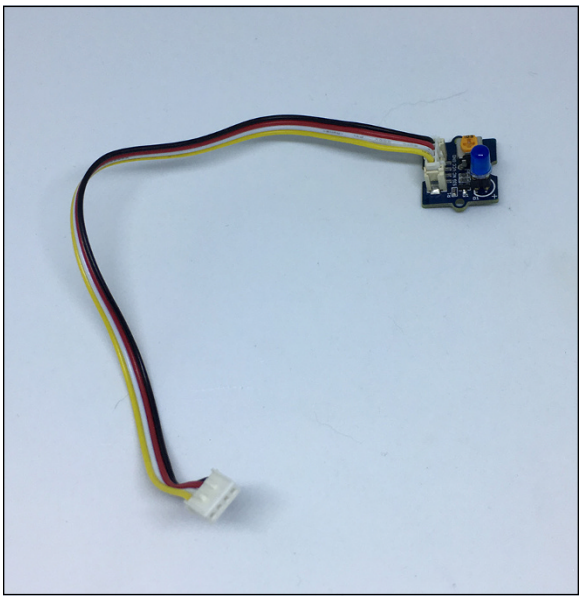


FIGURE 1-5: The Grove cable (included with the LED).

1. **Identify the Pi2Grover board from Figure 1-3 above.**
2. **Making sure you align the pins correctly gently press the Pi2Grover Board (Part A) onto the 40 pin GPIO connector on the Raspberry Pi. (See Figure 1-6.)**

FIGURE 1-6: Aligning the Pi2Grover board on the Raspberry Pi.

3. **Gently finish pushing the Pi2Grover (Part A) onto the Raspberry Pi GPIO pins, making sure the pins are aligned. There will be no pins showing on either end and make sure no pins on the Raspberry Pi are bent. (See Figure 1-7.)**

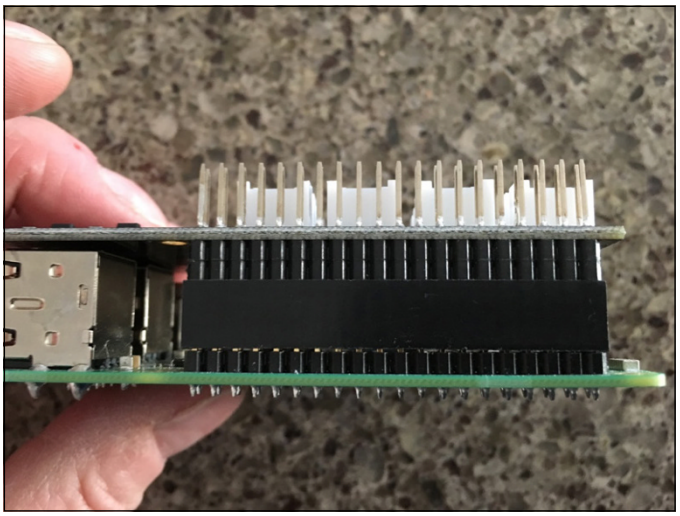


FIGURE 1-7: The installed Pi2Grover board.

4. **Plug one end of the Grove cable into the Grove blue LED board. (See Figure 1-8.)**

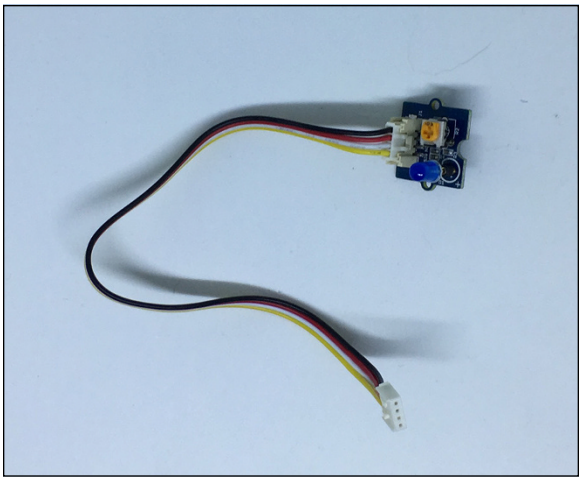


FIGURE 1-8: A Grove cable plugged into the Grove blue LED board.

5. **If your blue LED is not plugged into the Grove blue LED board, then plug in the LED with the flat side aligned with the flat side of the outline on the board as in Figure 1-9.**

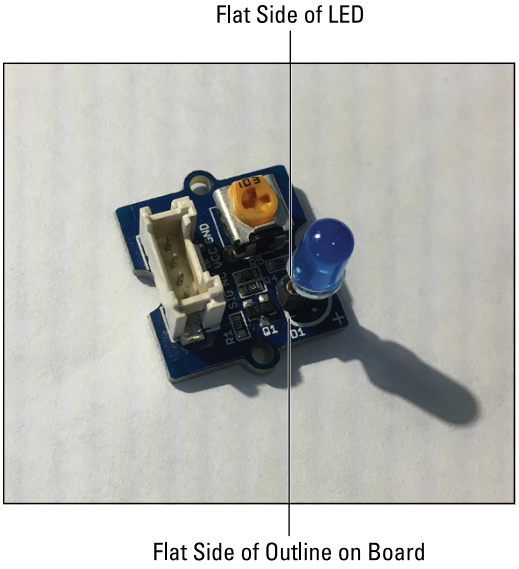


FIGURE 1-9: The LED aligned with the outline on the board.

6. **Plug the other end of the Grove cable into the slot marked D12/D13 on the Pi2Grover board. (See Figure 1-10.)**

   You are now finished assembling the hardware. Now it's time for the Python software.

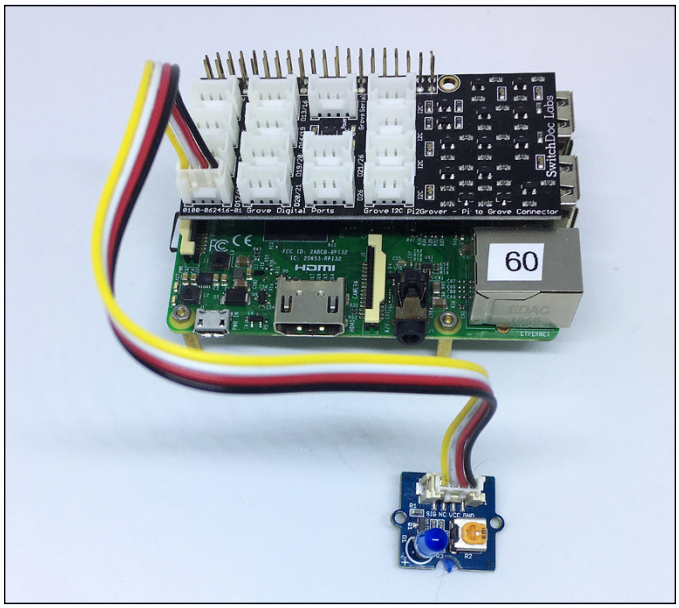


FIGURE 1-10: The completed "Hello World" project.

Controlling the LED with Python on the Raspberry Pi

Now that we have the hardware all connected, we can apply the power to the Raspberry Pi. If all is well, then you will see your Grove blue LED light up, a blue power LED on the Pi2Grover board, a flashing yellow LED (for a while during bootup) on the Raspberry Pi, and a steady red LED light also on the Raspberry Pi.

TECHNICAL STUFF

The Grove blue LED lights up when we turn the Raspberry Pi power on because the GPIO pins on the Raspberry Pi power up as inputs. Because it is an input and nothing is driving the GPIO pin (the Grove LED wants an output, not an input to control the LED), the GPIO pin just floats (called being in tri-state technically). Because of the circuitry in the Pi2Grover board, the input will float towards a "1" and so the LED will turn on. When you turn your GPIO pin to an output in the code below, the LED will turn off.

To get started, follow these steps:

1. **Go to your keyboard and open up a terminal window.**

TIP

If you don't know how to open and use a terminal window and the command line on the Raspberry Pi, go to `https://www.raspberrypi.org/documentation/usage/terminal/` for an excellent tutorial.

2. **Enter the following Python code into a file using the nano text editor or an editor of your choice. Save it to the file `HelloWorld.py`.**

```
from gpiozero import LED
from time import sleep

blue = LED(12)

while True:
    blue.on()
    print( "LED On")
    sleep(1)
    blue.off()
    print( "LED Off")
    sleep(1)
```

TIP

For an excellent tutorial on using nano, go to `https://www.raspberrypi.org/magpi/edit-text/`

3. **Now the big moment. Start your program by running this on the command line your terminal window:**

```
sudo python3 HelloWorld.py
```

You will see the LED blink on and off once per second and the following will appear on the screen in the terminal window:

```
LED On
LED Off
LED On
LED Off
LED On
LED Off
LED On
LED Off
LED On
LED Off
LED On
```

TECHNICAL STUFF

The keyword `sudo` stands for *super user do*. We use `sudo` in front of the `python3` command in this type of code because some versions of the Raspberry Pi operating system restricts access to certain pins and functions from a regular user. By using `sudo`, we are running this as a super user. This means it will run no matter how the particular version of the operating system is set up. In the newer versions of the Raspberry Pi OS, you can just type `python3 HelloWorld.py` and it will work. If it doesn't, go back to `sudo python3 HelloWorld.py`.

You can stop this program using Ctrl+C (^C, in geek terms).

In the code, the following statement imports the function `LED` from the Python `gpiozero` library:

```
from gpiozero import LED
```

This statement imports the function `sleep` from the Python `time` library:

```
from time import sleep
```

This assigns an LED on GPIO 12 (remember D12/D13 on the Pi2Grover Board?):

```
blue = LED(12)
```

Now you start the loop that will go on forever:

```
while True:
```

Turn the LED on:

```
    blue.on()
    print( "LED On")
```

Wait for one (1) second to go by:

```
    sleep(1)
```

Turn the LED off:

```
    blue.off()
    print( "LED Off")
    sleep(1)
```

Rinse and repeat.

Wow, you have now entered the world of physical computing. Just wait until you have finished this book. You will be amazed what you can do!

But Wait, There Is More . . .

Because we have all this hardware set up, how about we do one more interesting project? Let's make this a variable brightness LED by using PWM (pulse width modulation) to vary the brightness of the LED.

Pulse-width modulation (PWM) is a technique by which you vary the amount of time a signal is at a 1 versus the amount of time the signal is at a 0. Because our LED turns on when it is at a 1 and turns off at a 0, if we vary the time it is at a 1 versus a 0 then we can control the brightness to the human eye. This ratio is called the *duty cycle*. (See Figure 1-11.) 100 percent duty cycle means it is on 100 percent of the time, whereas a duty cycle of 0 percent means it is off all the time. Varying the time the signal is on will change the brightness of the LED.

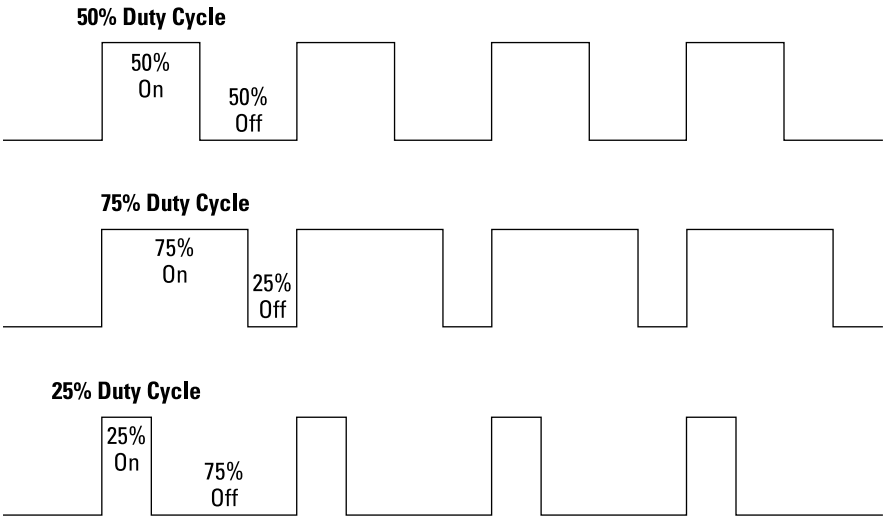


FIGURE 1-11: Duty cycles.

Enter this Python code into nano and save it as `HelloWorld2.py`:

```
from gpiozero import PWMLED
from time import sleep

led = PWMLED(12)

while True:
    led.value = 0  # off
    sleep(1)
    led.value = 0.5  # half brightness
```

```
    sleep(1)
    led.value = 1  # full brightness
    sleep(1)
```

Now run the code:

```
sudo python3 HelloWorld2.py
```

You will see the brightness change every second.

And one more thing, here is how to change your brightness in a continuous fashion:

```
from gpiozero import PWMLED
from signal import pause

led = PWMLED(12)

led.pulse()

pause()
```

With this code, we see a smooth continuous brightening and darkening of the LED.

Boy, you accomplished a lot in this chapter. You have now started to see the possibilities of physical computing. And you have a blue LED!

THE LED CHANGING IS NOT TOTALLY SMOOTH

Turns out that the way the Raspberry Pi Linux operating system works, your program is not the only thing running at the same time. If you want to see everything that is running on your Raspberry Pi, type **`ps xaf`** at your command-line prompt on your terminal. You will be amazed at what is running on your Rapsberry Pi. Because the operating system on the Raspberry Pi is multitasking, meaning more than one task runs at a time, sometimes your PWM task (as it is being run in software) does not get the CPU quite when it wants and that is why there is just a little bit of jitter in the LED. The Raspberry Pi does have two hardware PWM GPIO pins, which can be used if you aren't using the audio output on the Raspberry Pi. On a Raspberry Pi 3B+ you will barely notice it, but you will on slower Pi versions.

IN THIS CHAPTER

- Discovering how to plug hardware together
- Avoiding the Box of Death!
- Working with the four types of sensors
- Understanding using Patch cables

Chapter 2

No Soldering! Grove Connectors for Building Things

Okay, okay. We all have been talking about Python for the past several hundred pages. Time to build something! But before we get to that, we need to talk about how to plug things together.

Grove is a modular, standardized connecter prototyping system. Grove takes a building-block approach to assembling electronics. Compared to the jumper or solder-based system, it is easier to connect, experiment, and build, and it simplifies the learning system, but not to the point where it becomes dumbed down. Some of the other prototype systems out there take the level down to building blocks. There is good stuff to be learned that way, but the Grove system allows you to build real systems. However, it requires some learning and expertise to hook things up.

The Grove system consists of a base unit and various modules (with *standardized connectors*).

The base unit, generally a microprocessor, allows for easy connection of any input or output from the Grove modules, and every Grove module typically addresses a single function, from a simple button to a more complex heart-rate sensor.

You don't need a base unit to connect up to Grove modules. You can use a cable (Grove-to-pin-header converter) to run from the pins on the Raspberry Pi or Arduino to the Grove connectors. See some examples of how to do this later in this chapter.

So What Is a Grove Connector?

Normally, when you're wiring up a board, you have to pay attention. If you plug things in backwards or connect boards incorrectly, you can damage or destroy boards. All it takes is an incorrectly attached wire, and your board is gone forever.

The Grove system, however, works differently. It allows you to connect up boards while taking no chances on hooking up power and ground incorrectly.

A Grove connector is a four-pin standardized size connector used to plug into base units and Grove modules. Figure 2-1 shows the male Grove connector. These standardized connectors (common to all types of Grove connectors) are the key to making this system work. They are keyed so that you cannot plug them in backwards, and the four types of connectors (see "The Four Types of Grove Connectors," later in this chapter) are all designed so that if you plug the wrong type of device into the wrong type of base unit, there is no problem. They aren't destroyed; they just won't work. This is a good thing, a very good thing.

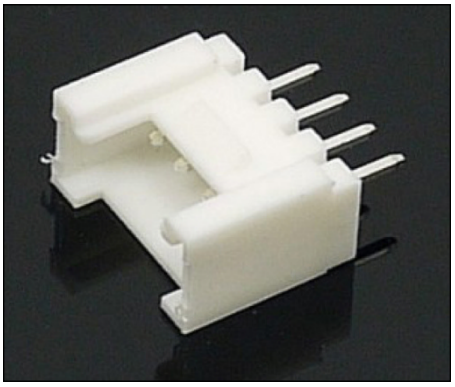


FIGURE 2-1: A Grove connector.

WARNING

The one exception would be if you plugged in a 3.3V I2C Grove module that is non-5V tolerant into a 5V I2C Grove connector you could fry the device. In this book, we avoid such situations by making sure everything we do is 5V!

Selecting Grove Base Units

A Grove base unit is a controller or shield to which you attach the Grove modules. The base unit provides the processing power, and the modules offer the input sensors and output actuators of your system.

For the Arduino

We most talk about the Raspberry Pi in this book, but there are other computers out there too! Arduinos are one of the more popular ones. There are a number of good base unit shields available for the Arduino that provide a lot of Grove connectors. Figure 2-2 shows the base unit designed to plug into an Arduino Uno. They are also available for the Arduino Mega, Due, and others.

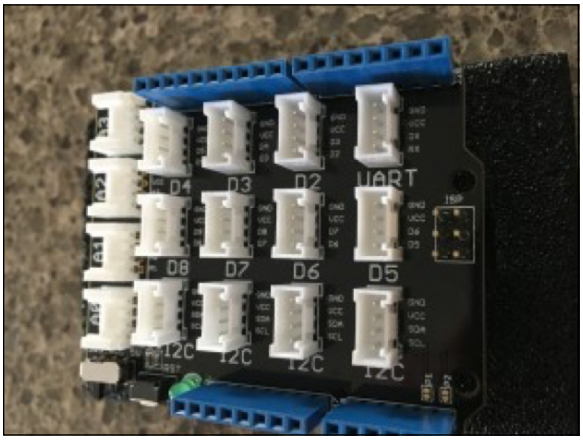


FIGURE 2-2: The Arduino Uno Grove base board.

Some Arduino boards, such as the Mini Pro LP (see Figure 2-3), have Grove connectors built right into the board so you don't even need a base unit.

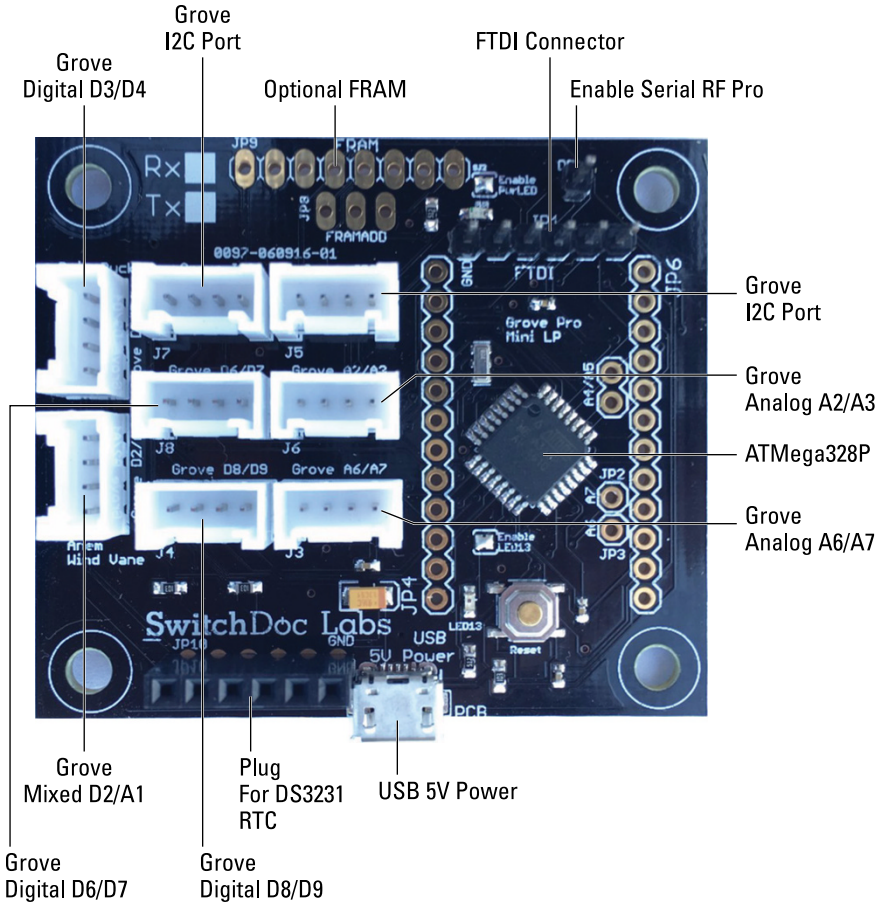


FIGURE 2-3: The Arduino Mini Pro LB board with Grove.

Raspberry Pi Base Unit — the Pi2Grover

On the Raspberry Pi side, the pickings are much slimmer. The base unit devices available tend to be "too smart" and isolate you from the Raspberry Pi hardware and software. This is a huge problem when you want to connect to hardware using Python. We prefer a solution that is closer to the hardware for learning and flexibility. You can still mask the complexity with software drivers.

The base unit we will be using is the Pi2Grover base unit. It basically is just a level shifter (from the Raspberry Pi 3.3V to 5V for all the Grove sensors), and it does not get in the way of writing drivers in Python. (See Figure 2-4.)

A TOAST OF WATER TO VOLTAGES

Hmmm. What is the difference between 3.3V and 5V? *V* refers to *voltage*, which is similar to the water pressure in a pipe. The higher the pressure, the more water comes out. With voltage, the higher the voltage, the more current (like water) will come out of the pipe. If the water pressure is too high, it can break the pipe. Similarly, if the voltage is too high, you can break the computer input. Raspberry Pi's are pretty particular about liking only 3.3V on their input lines and can be damaged if you apply higher voltages (like 5V). This is another reason we like to use the Pi2Grover, because it converts and buffers all the lines from the Raspberry Pi back and forth from 3.3V to 5V with no problem.

One more thing about voltages. Voltages are always measured with reference to something, usually called *ground*. This is why grounds are so important to connect and to have a common ground so your voltages running around always know to what they are referenced.

Not having a common ground in a system (thus confusing the voltages!) leads to very flaky behavior. This leads to the Second Law of Shovic, "*You can always trust your mother, but you can never trust your ground.*" For those who may be interested in what the First Law of Shovic is, that one is a bit easier to understand. The First Law is *"It works better if you plug it in!"*

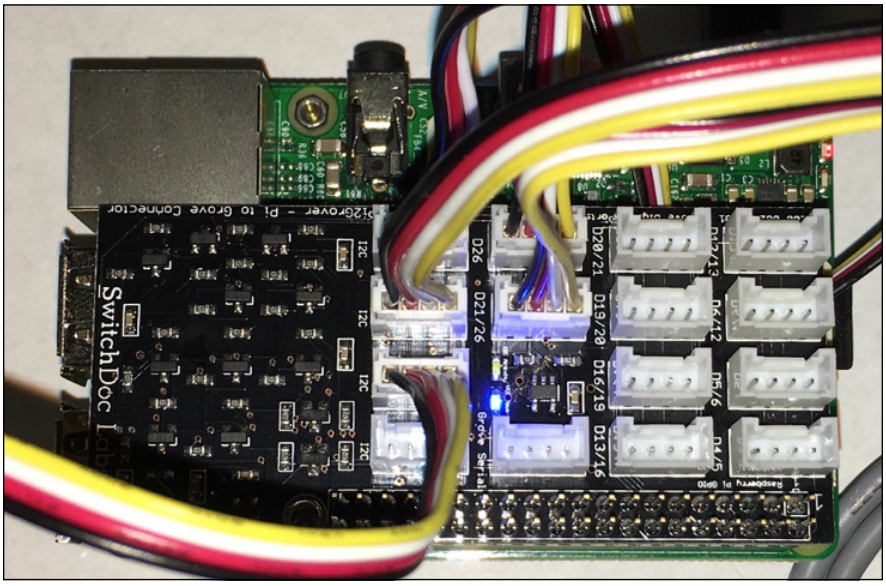


FIGURE 2-4: The Pi2Grover board at work on the Raspberry Pi.

I DON'T WANT TO USE A BASE UNIT!

You do not have to have a hat or shield to use Grove with your Raspberry Pi or Arduino. All you need is to connect the I2C, digital, or analog inputs to the Grove devices by using a Grove-to-pin-header converter.

The Four Types of Grove Connectors

Now, let's talk about some of the specifics of each of the four types of connectors. First of all, all Grove cables are physically identical and can be interchanged. The differences are in the signal types they provide. Now, note! You will never short out power and ground by plugging in one type of Grove connector in the other. Although you do need to be careful and think about what you are doing, it is a lot less risky than soldering or using jumpers to wire up devices to your Pi or Arduino.

Generically, all the Grove connectors are wired the same: Signal 1, signal 2, power, ground.

Wire colors on standard Grove cables are always the same. (See Figure 2-5.)

- **Pin 1:** Yellow (for example, SCL on I2C Grove connectors)
- **Pin 2:** White (for example, SDA on I2C Grove connectors)
- **Pin 3:** Red (VCC on all Grove connectors)
- **Pin 4:** Black (GND on all Grove connectors)

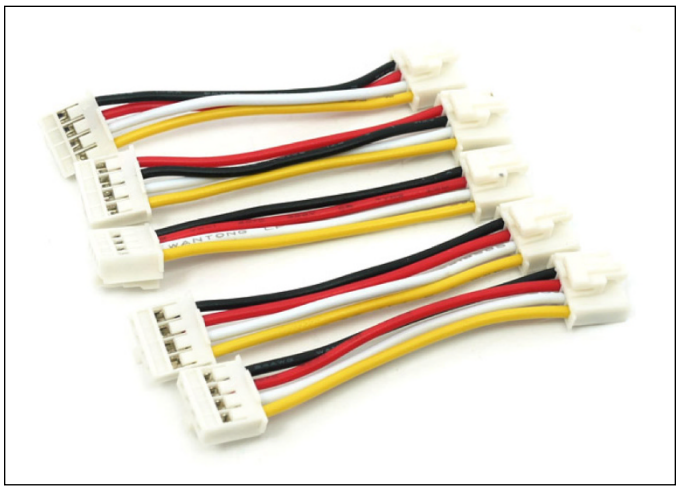


FIGURE 2-5: 5cm-long Grove cables.

The Four Types of Grove Signals

Now it is time to wax poetic about the different types of signals we use to talk to sensors and devices. It's not hard, but pay attention. By using the wrong connector, you may not fry your board, but your project still may not work correctly!

Grove digital — All about those 1's and 0's

Many sensors only need one or two bits. A *bit* is the basis of all digital computer hardware. It can either be a "1" or a "0". There isn't anything in between. Yes, the bits are represented by voltage levels (see the discussion on Voltage earlier in this chapter), but fundamentally we will treat these bits as only having a "1" or "0" value.

Computers often communicate with each other and with external devices by using digital bits. It turns out that there are two ways of getting information from bits. One is the value ("1" or "0") and the other is timing. Such as how long the bit has a value of "1." The thought of this leads us to the Serial Grove ports we talk about later.

A digital Grove connector consists of the standard four lines coming into the Grove plug. The two signal lines are generically called D0 and D1. Most modules only use D0, but some (like the LED Bar Grove display) use both. Often base units will have the first connector called D0 and the second called D1 and they will be wired D0/D1 and then D1/D2, and so on. See Table 2-1 for a description of each pin of the digital Grove connector.

Examples of Grove digital modules are: Switch modules, the Fan module, and the LED module. In Figure 2-6, you can see what the Grove connector looks like on the schematic for the LED Grove module. They range from the simple to the very complex.

TABLE 2-1 The Grove Digital Connector

Pin	Name	Description
Pin 1 - Yellow	D0	Primary digital input/output
Pin 2 - White	D1	Secondary digital input/output
Pin 3- Red	VCC	Power for Grove module (5V or 3.3V)
Pin 4 - Black	GND	Ground

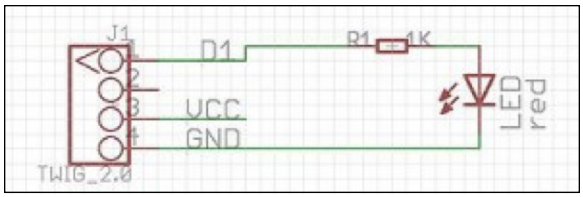


FIGURE 2-6: A simple digital Grove module with LED.

Grove analog: When 1's and 0's aren't enough

A Grove analog connector consists of the standard four lines coming into the Grove plug. The two signal lines are generically called A0 and A1. Most modules only use A0. Often base units will have the first connector called A0 and the second called A1 and they will be wired A0/A1 and then A1/A2, and so on. This simple voltage divider will give you a different analog voltage reading depending on the position of the switch and of course, the voltage present across the green connector on the left side. (See Figure 2-7.) See Table 2-2 for the descriptions of each pin.

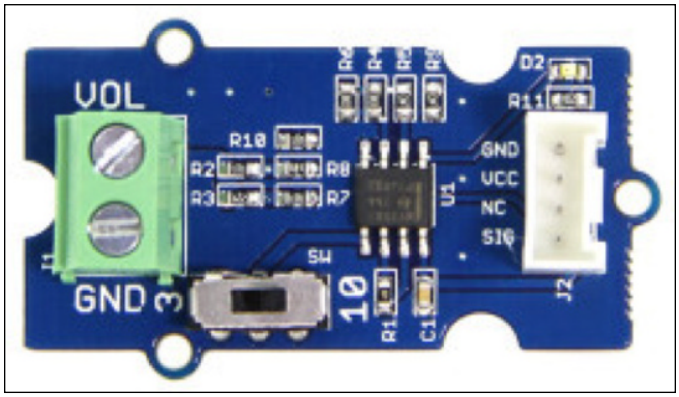


FIGURE 2-7: A Grove analog simple voltage divider.

TABLE 2-2 The Grove Analog Connector

Pin	Name	Description
Pin 1 - Yellow	A0	Primary analog input
Pin 2 - White	A1	Secondary analog input
Pin 3 - Red	VCC	Power for Grove module (5V or 3.3V)
Pin 4 - Black	GND	Ground

Examples of Grove analog modules are: Potentiometer, voltage divider and a Grove air quality sensor.

Grove UART (or serial) — Bit by bit transmission

Remember when we talked about digital signals? How you can convey information not only in the level of the signal ("1" or "0") but also in how long in terms of time it stays at a "1" or "0". That is the basis of sending a serial signal. For example, 8 single bits sent at a specific speed, such as 0100001, can represent the letter *A*. The speed at which the bit is sent is called a *baud rate*. (*Baud* comes from Emile Baudot, who was an inventor and scientist making great progress in the late 1800s with the telegraph).

The Grove UART module is a specialized version of a Grove digital module that uses the digital level and the timing of the signal to receive and transmit data. It uses both Pin 1 and Pin 2 for the serial input and transmit. The Grove UART (also called a *serial interface*) plug is labeled from the base unit's point of view. In other words, Pin 1 is the RX line (which the base unit uses to receive data, so it is an input) where Pin 2 is the TX line (which the base unit uses to transmit data to the Grove module). See the description of each pin on the UART Grove connector in Table 2-3.

Examples of Grove UART modules are: XBee wireless sockets, 125KHz RFID reader. (See Figure 2-8.)

TABLE 2-3 The Grove UART Serial Connector

Pin	Name	Description
Pin 1 - Yellow	RX	Serial receive (from the base unit's point of view — not the Grove board's)
Pin 2 - White	TX	Serial transmit (from the base unit's point of view — not the Grove board's
Pin 3 - Red	VCC	Power for Grove module (5V or 3.3V)
Pin 4 - Black	GND	Ground

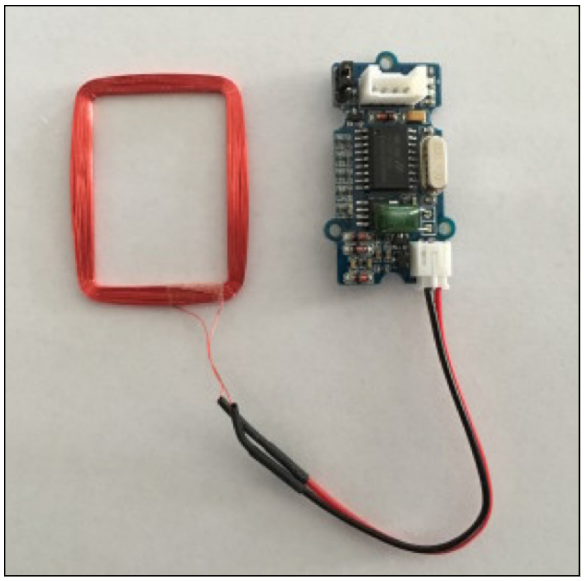


FIGURE 2-8: A Grove UART RFID reader.

ANALOG VERSUS DIGITAL: THE DEBATE CONTINUES

The differences between analog and digital signals are both simple and confusing at the same time. A digital signal has a value of "1" or "0." That's it. Analog voltages are able to take any voltage value, such as 1.2V, or 3.14198V, or any other floating-point value. So with analog, you can have many, many different voltages. Now for the confusing part. We represent a "1" on the devices we are talking about here as a 5V signal and a 0V signal as a "0". And it is even more complicated than that. Typically, any signal above about 2.5V can be considered a "1," and anything less than 0.7V can be considered a "0" if read by a digital port. Okay, okay. Enough about that. Let's just treat signals for this book as digital or analog and leave it at that. Whew!

An analog signal is used when it is important to know what voltage is present at the signal input or output. For example, a value of 1.420V coming from a moisture sensor can indicate a dry plant, whereas a voltage of 3.342V could indicate that the plant has plenty of water. Because the values between 1.420V and 3.342V can indicate how dry the plant is, it is important for us to know what the actual voltage number is. Later, we discuss how to read an analog voltage into a digital computer by converting the analog voltage into a digital number by the use of an ADC (analog-to-digital converter). Then our computer can tell whether the plant is dry or not!

Grove I2C — Using I2C to make sense of the world

Our favorite devices to plug into little computers are I2C sensors. There are hundreds of types of I2C sensors on the market, and they are generally very inexpensive. There are many types of I2C Grove sensors available just ready to plug and go!

The sensor shown in Figure 2-9 is a SI1145 sunlight I2C sensor. But what, there's more! It not only calculates the visible sunlight strength, it also measures the infrared (IR) and even the ultraviolet (UV) components. This inexpensive sensor can tell you whether you are going to get sunburned as well as if your plants are happy!

You just have to love the things you can do these days with computers.

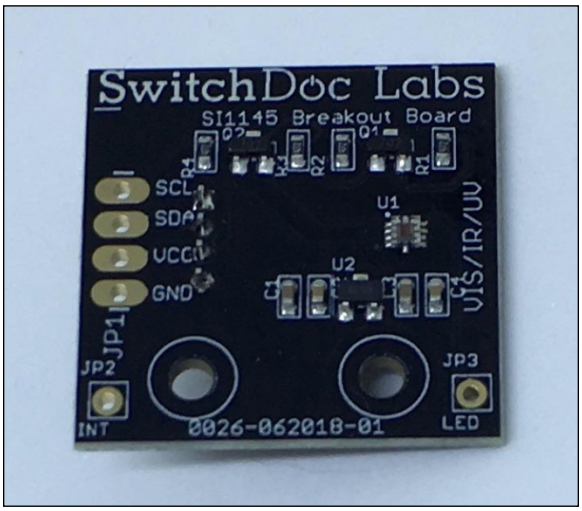


FIGURE 2-9: The Grove I2C sunlight sensor.

The actual sensor on the board is the little colored chip marked "U1." It's got a clear top to let the light through to measure.

WARNING

Most I2C sensors can be used with both 3.3V and 5V base units, but there are a few that are only 3.3V or 5.0V. You need to check the specifications. It is almost always very obvious as to which voltage they will run at. If you connect a 3.3V I2C sensor to your 5V Grove connector, you will probably destroy the device. See Table 2-4 for the pin descriptions of the Grove connector.

I2C — THE DANCE OF CLOCK AND DATA

An I2C bus is often used to communicate with chips or sensors that are on the same board or located physically close to the CPU. It stands for standard Inter-IC device bus. I2C was first developed by Phillips (now NXP Semiconductors). To get around hardware licensing issues, sometimes the bus will be called TWI (two wire interface). SMBus, developed by Intel, is a subset of I2C that defines the protocols more strictly.

I2C provides good support for slow, close peripheral devices that only need be addressed occasionally. For example, a temperate measuring device will generally only change very slowly and so is a good candidate for the use of I2C, whereas a camera will generate lots of data quickly and potentially changes often.

I2C uses only two bidirectional open-drain lines, SCL (serial clock) and SDA (serial data). Kind of like two serial data lines next to each other. *Open-drain* means the I2C device can pull a level down to ground ("0"), but cannot pull the line up to VDD ("1"). Hence the name open-drain. You put a resistor on the line to pull it up to a "1" between "0" serial pulses, very much like a dance between SDA and SCL.

I2C devices are addressed by using a 7-bit address (0-127 in decimal) so you can have many devices on the same I2C bus, which is a very cool feature.

The Grove I2C connector has the standard layout. Pin 1 is the SCL signal and Pin 2 is the SDA signal. Power and ground are the same as the other connectors. This is another special version of the Grove digital connector. In fact, often the I2C bus on a controller (such as the ESP8266, Raspberry Pi, and the Arduino) just uses digital I/O pins to implement the I2C bus. The pins on the Raspberry Pi and Arduino are special with hardware support for the I2C bus. The ESP8266 has a purely software I2C interface, which is called "bit banging" for those children of the 90s.

TABLE 2-4 The Grove I2C Connector

Pin	Name	Description
Pin 1 - Yellow	SCL	I2C clock
Pin 2 - White	SDA	I2C data
Pin 3 - Red	VCC	Power for Grove module (5V or 3.3V)
Pin 4 - Black	GND	Ground

Using Grove Cables to Get Connected

There are many different lengths of Grove cables available, from 5cm all the way up to 50cm long cables. (See Figure 2-10.) You use these to plug your sensors into the Raspberry Pi. These are easy. They come with a Grove connector on each end and are interchangeable.

Grove cables also come as patch cables (between Grove and pins) and we talk about them next.

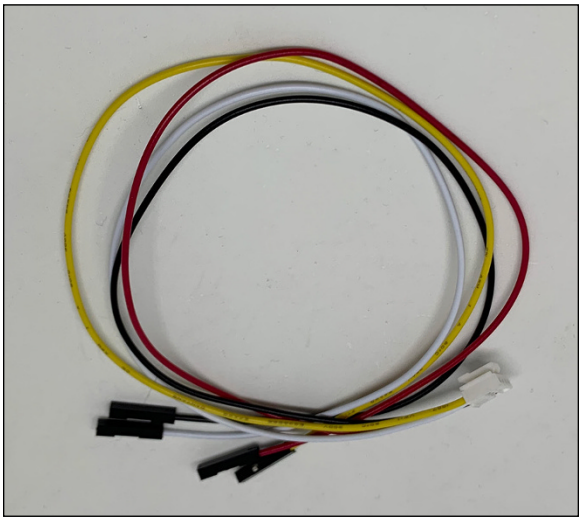


FIGURE 2-10: 20cm Grove cables.

Grove Patch Cables

There always seems to be some kind of device or sensor that does not have Grove connectors and yet you want to use it in your system. The solution to this is to use a patch cable!

It turns out there are easy ways of converting pin headers to Grove connectors using Grove adaptor cables. There are two types of Grove adaptor cables. One converts the `Grove connector to female header pins`, as in Figures 2-11 and 2-12.

The second type of Grove adaptor cables are Grove-connector-to-male-header-pins, as shown in Figure 2-13.

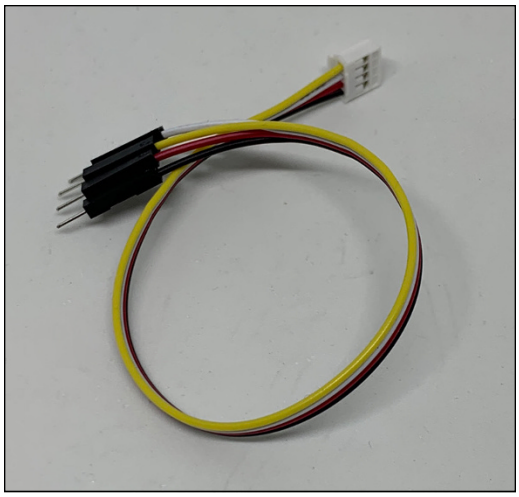


FIGURE 2-11: Grove female header cables.

FIGURE 2-12: A close-up of a Grove female header cables.

The power of the patch cable is that you can connect to non-Grove sensors.

WARNING

Basically, you map the Grove connector to your pin headers. Be careful and make sure you check twice before applying power!

How you map depends on what kind of a sensor you have and what the interface is. Grove connectors support four kinds of interfaces as we talk about earlier in this chapter.

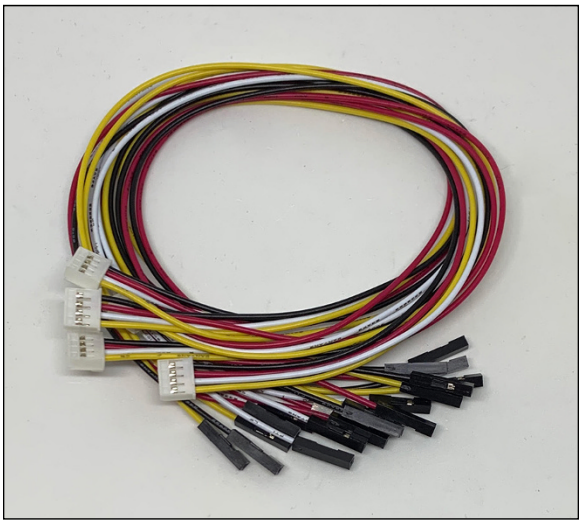


FIGURE 2-13: Grove male header cables.

An example of the power of the patch!

`SunAirPlus`, a solar power controller and data collector, is an example of converting a pin header sensor to use Grove connectors. SunAirPlus has an I2C interface on the pin header that we often want to convert to Grove connectors. We connect the cable in the following way (see Figure 2-14):

Pin 1 – Yellow (SCL)

Pin 2 – White (SDA)

Pin 3 – Red (VDD)

Pin 4 – Black (GND)

Figure 2-15 shows the other end of the adaptor cable plugged into the Pi2Grover adaptor board on the Raspberry Pi.

Second example: The Adafruit Ultimate GPS

The Adafruit Ultimate GPS connects to a Raspberry Pi/Arduino through a serial interface (UART). To use Grove connectors, we connect the cable in the following way:

Pin 1 – Yellow (TX)

Pin 2 – White (RX)

Pin 3 – Red (VIN)

Pin 4 – Black (GND)

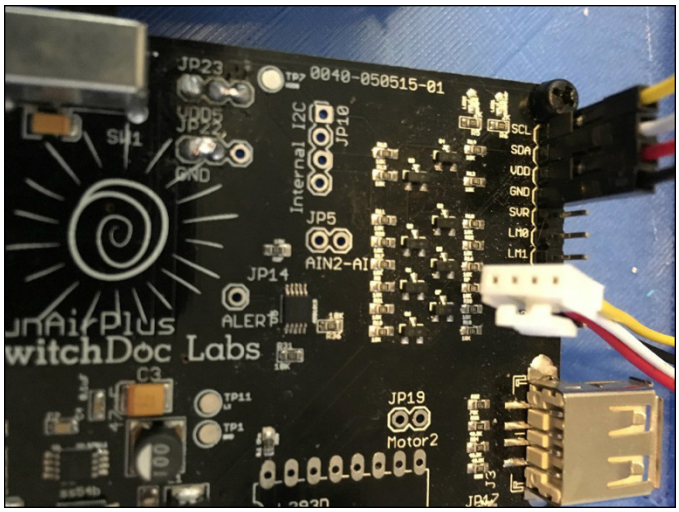


FIGURE 2-14: The SunAirPlus board with the Grove female header patch cable.

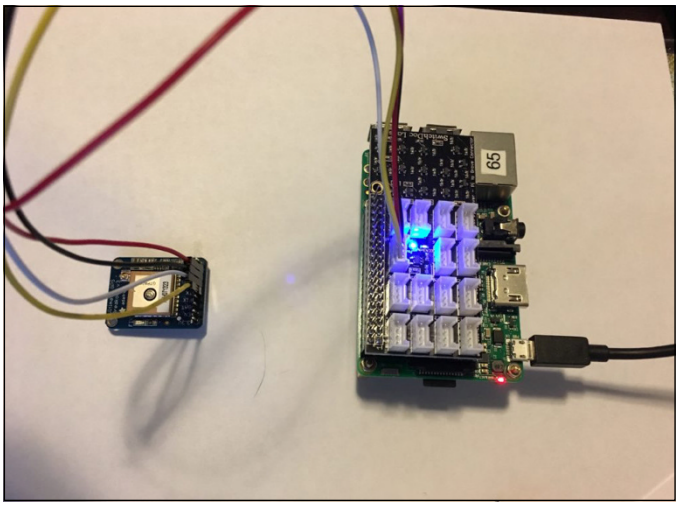


FIGURE 2-15: A Grove adaptor cable attached to Pi2Grover.

Note that serial connectors are a bit odd in that you need to connect the RX on the Grove connector to the TX on the sensor and the TX on the Grove connector to the RX on the sensor. (See Figure 2-16.)

It's time to start building!

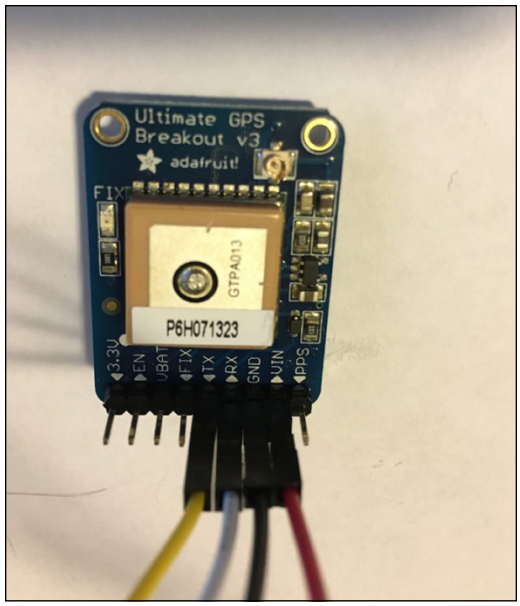


FIGURE 2-16: A close-up of the Adafruit GPS with a Grove patch cable.

IN THIS CHAPTER

- Discovering how to use I2C sensors
- Sensing your environment with a Raspberry Pi
- Collecting and saving data
- Connecting Python to your smartphone

Chapter 3

Sensing the World with Python: The World of I2C

Before we get into how to sense the world in Python, let's go through a few of the hardware issues. You can skip all this and still use Python to talk to these devices, of course, but some background is a good thing to have You can always go back over it later when you have some experience with these devices.

The available sensors for the Raspberry Pi and other small computers number in the thousands. From detecting people in front of your computer (PIR) to detecting a myriad of environmental conditions (temperature/humidity/air quality/and so on), there are many inexpensive ways to have your computer monitor the physical world. As always, the major thing you have to know about these sensors is how you can talk to them with a computer, which is commonly through the *interface*. The interface consists of two things: The *hardware interface*, which contains pins, types, and voltage levels, and the *software interface*, which is usually called a *driver* or an *API (application programming interface)*.

There are four major ways of getting data to your computer from your outside sensors:

- Digital input — GPIO pins programmed to be input lines.
- Digital analog input — Analog values that need to go through an analog-to-digital converter (ADC) to be read by a computer.

- Digital I2C (pronounced I-squared-C) (Inter-Integrated Circuit) bus
- Digital SPI (serial peripheral interface)

In this book, we deal with sensors using digital inputs, analog inputs and I2C interfaces. Why not SPI? Just for simplicity. Most SPI parts also have an I2C interface on the chip, and most small computer boards have an I2C interface built into the board.

Understanding I2C

The first thing to know about I2C is that every device on the I2C bus has an address. For example, the address of the HDC1080 temperature and humidity sensor we use in this chapter has an address of 0x40. What does the "0x" mean in this address? It means that the number that follows is in hexadecimal notation, base 16 instead of base 10 (our normal numbering system).

To understand this interface, let's look at what an I2C bus is. An I2C bus is often used to communicate with chips or sensors that are on the same board or located physically close to the CPU. I2C was first developed by Phillips (now NXP Semiconductors). To get around licensing issues (that have largely gone away), often the bus will be called TWI (Two Wire Interface). SMBus, developed by Intel, is a subset of I2C that defines the protocols more strictly. Modern I2C systems take policies and rules from SMBus, sometimes supporting both with minimal reconfiguration needed. Both the Arduino and the Raspberry Pi support the I2C bus.

I2C provides good support for slow, close peripheral devices that need be addressed only occasionally. For example, a temperature-measuring device will generally only change very slowly and so is a good candidate for the use of I2C, whereas a camera will generate lots of data quickly and potentially changes often.

I2C uses only two bidirectional open-drain lines (*open-drain* means the device can pull a level down to ground, but cannot pull the line up to Vdd. Hence the name *open-drain*. Thus a requirement of I2C bus is that both lines are pulled up to Vdd. This is an important area and not properly pulling up the lines is the first and most common mistake you make when you first use an I2C bus. The Pi2Grover board we use in this book contains 10K Ohm pullup resistors so you should not have to worry about this. The two lines are SDA (serial data line) and the SCL (serial clock line). There are two types of devices you can connect to an I2C bus: Master devices and Slave devices. Typically, you have one Master device (The Raspberry Pi, in our case) and multiple Slave devices, each with their individual 7-bit address. (See Figure 3-1.)

When used on the Raspberry Pi, the Raspberry Pi acts as the Master and all other devices are connected as Slaves.

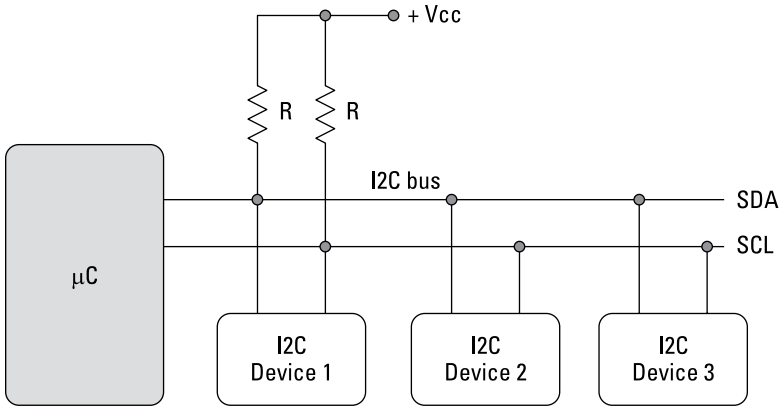


FIGURE 3-1: The I2C bus.

The I2C protocol uses three types of messages:

- Digital single message where a master writes data to a slave
- Digital single message where a master reads data from a slave
- Digital combined messages, where a master issues at least two reads and/or writes to one or more slaves

Lucky for us, most of the complexity of dealing with the I2C bus is hidden by Python drivers and libraries.

Exploring I2C on the Raspberry Pi

TIP

The first thing you want to do on your Raspberry Pi is to learn a bit about the terminal window, command line, and text editors. If you haven't done that yet, refer to Chapter 1 of this minibook.

To use the I2C bus on the Raspberry Pi, you need to make sure that it is enabled in the operating system. Here is a good tutorial from Adafrui9t on how to do just that: `https://learn.adafruit.com/adafruits-raspberry-pi-lesson-4-gpio-setup/configuring-i2c`.

Did you do it right? The easy way to check for this is to type the following command in your terminal window:

```
I2cdetect -y 1
```

If it returns:

```
-bash: i2cdetect: command not found
```

Then you have not enabled your I2C bus. Repeat the tutorial to fix this.

On the other hand, if it returns:

```
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

Then you have been successful! Note that all dashes mean there are no sensors on the I2C bus. In our next section, we are going to add a simple one.

Now, let's talk about how to communicate with I2C devices in Python.

Talking to I2C devices with Python

In order to talk to an I2C device, you should have one on the bus. A good one to start with is the HDC1080 temperature and humidity sensor. (See Figure 3-2.) You can get one of these inexpensive sensors on `store.switchdoc.com` or on `amazon.com`.

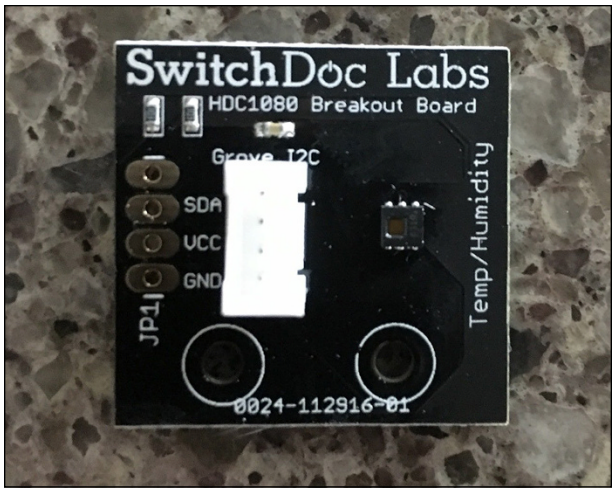


FIGURE 3-2: HDC1080 temperature and humidity sensor.

THE TEXAS INSTRUMENTS HDC1080 TEMPERATURE AND HUMIDITY SENSOR

TECHNICAL STUFF

This is a pretty amazing device considering how inexpensive it is. The HDC1080 is a HDC1000 compatible temperature and humidity sensor. It is located at I2C address 0x40.

The Grove temperature and humidity sensor (HDC1080) utilizes the HDC1080 sensor from Texas Instruments. It is a digital humidity sensor with integrated temperature sensor that provides excellent measurement accuracy at very low power. The device measures humidity based on a novel capacitive sensor. The humidity and temperature sensors are factory calibrated. The innovative WLCSP (wafer level chip scale package) simplifies board design with the use of an ultra-compact package. The HDC1080 is functional within the full –40°C to +125°C temperature range, and 0–100 percent RH range. The accuracy of the chip is +/– 3 percent relative humidity and +/– 0.2C for the temperature.

Note: If you buy one on Amazon, you will need a female-to-Grove patch cable, as discussed in Chapter 2 of this minibook. The SwitchDoc Labs HDC1080 already comes with a Grove connector. You will also need the Pi2Grover Raspberry Pi-to-Grove converter that was described in Chapters 1 and 2 of this minbook, which is also available on store.switchdoc.com or on amazon.com.

Now let's install the HDC1080 I2C sensor on our Raspberry Pi. Follow these steps:

1. **Shut down your Raspberry Pi. When the yellow LED has stopped blinking, unplug the power from your Raspberry Pi.**

REMEMBER

Never plug anything into or pull anything out a Raspberry Pi without shutting the computer down. Exceptions to this are USB ports, audio cables, and Ethernet cables, which are designed to support "hot-plugging." The rest of the Raspberry Pi is not.

2. **Plug a Grove cable into the HDC1080. (See Figure 3-3.)**

REMEMBER

We are using the SwitchDoc Labs HDC1080; if you are using an Amazon device, refer to Chapter 2 of this minibook for the use of a Grove patch cable.

TIP

Always shut down your Raspberry Pi by first typing `sudo halt` on the command line (or by selecting Shutdown from the GUI menu). Wait until the yellow LED on the Raspberry Pi stops blinking before removing the power cord. This ensures that the SDCard on the Raspberry Pi has been prepared for shutdown and you won't corrupt it. Just unplugging your Raspberry Pi may not corrupt the card, but unplugging it without shutting it down increases the likelihood of corruption. Corrupting your SDCard may not be fatal, but repairing it is a long, technical, irritating process.

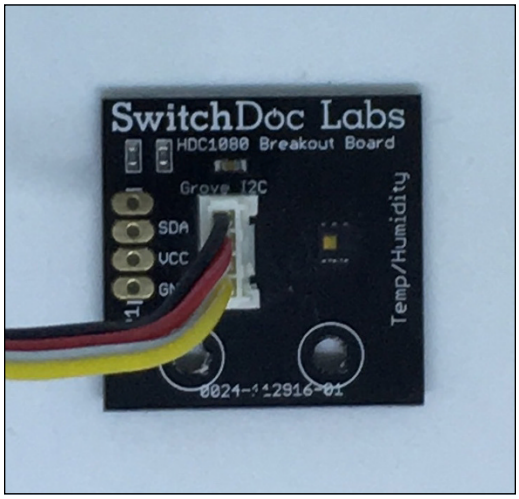


FIGURE 3-3: HDC1080 with the Grove cable plugged in.

3. **Plug the other end of the Grove cable into one of the Grove connectors marked I2C on the Pi2Grover that plugged on top of your Raspberry Pi. (See Figure 3-4.)**

   Note: The I2C is a bus, which means you can use any of the four I2C connectors.

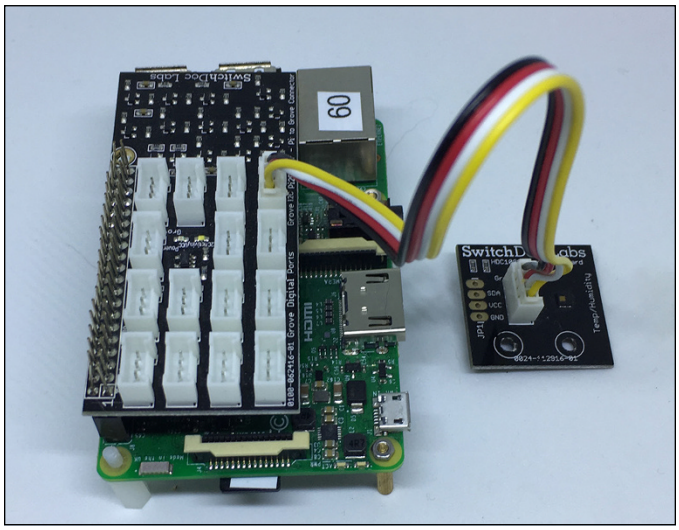


FIGURE 3-4: The HDC1080 hooked up to the Raspberry Pi.

4. **Power up the Raspberry Pi and open a terminal window.**

5. **Type into the terminal** `sudo i2cdetect -y 1` **and you will be rewarded with this:**

```
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: 40 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

Remember the 0x40 address of the HDC1080? There it is in the output above.

Now we are ready to proceed to use Python to read the temperature and humidity from this sensor.

Reading temperature and humidity from an I2C device using Python

The use of Python libraries are key to being productive in writing Python applications. We will be using the SDL_Pi_HDC1080_Python3, available on github.com.

To read the temperature and humidity, follow these steps:

1. **First, create a directory in your main directory:**

```
cd
mkdir I2CTemperature
cd I2CTemperature
```

Now you are in the `I2CTemperature` directory.

2. **Before looking at the Python code for reading your temperature, install the library on our Raspberry Pi. You do this by "cloning" the library located at github.com by using the following command in your terminal window:**

```
git clone https://github.com/switchdoclabs/SDL_Pi_HDC1080_Python3.git
```

Here `git clone` clones the git repository located at the address and copies it to your Raspberry Pi. If you enter `ls` in the terminal window, you will see the following output:

```
pi@RPi3-60:~/I2CTemperature $ ls
SDL_Pi_HDC1080_Python3
pi@RPi3-60:~/I2CTemperature $
```

3. **Using nano (or your favorite text editor), open up a file called `temperatureTest.py` and enter the following code:**

```
import sys

sys.path.append('./SDL_Pi_HDC1080_Python3')

import time
import SDL_Pi_HDC1080



# Main Program
print
print ("")
print ("Read Temperature and Humidity from HDC1080 using I2C bus ")
print ("")

hdc1080 = SDL_Pi_HDC1080.SDL_Pi_HDC1080()


while True:

        print ("------------------")
        print ("Temperature = %3.1f C" % hdc1080.readTemperature())
        print ("Humidity = %3.1f %%" % hdc1080.readHumidity())
        print ("------------------")

        time.sleep(3.0)
```

GITHUB, A REPOSITORY FOR GOOD THINGS

Github.com is a web-based hosting service for version control using Git, a well-known system for providing source control for software. It is mostly used for computer code. It provides access control and collaboration features such as bug tracking, feature requests, task management, and other services for every project.

As of June 2018, GitHub reports having over 28 million users and 57 million repositories making it the largest host of source code in the world.

In 2018, GitHub was acquired by Microsoft, Inc., which pledged to allow github.com to operate as an independent division. So far, so good.

4. **Run the code by typing:**

```
sudo python3 temperatureTest.py
```

You should see the following output, with new temperature and humidity readings every three seconds:

```
Read Temperature and Humidity from HDC1080 using I2C bus

------------------
Temperature = 24.2 C
Humidity = 32.9 %
------------------
------------------
Temperature = 24.2 C
Humidity = 32.9 %
------------------
------------------
Temperature = 24.2 C
Humidity = 32.9 %
------------------
```

You are now reading environmental data from an I2C device. Your Raspberry Pi is connected to the real world.

TIP

Try this experiment. Blow on the HDC1080 sensor board and watch the humidity go up! You will see something like this:

```
-----------------
Temperature = 24.2 C
Humidity = 32.9 %
-----------------
-----------------
Temperature = 24.1 C
Humidity = 33.6 %
-----------------
-----------------
Temperature = 24.1 C
Humidity = 33.9 %
-----------------
-----------------
Temperature = 24.1 C
Humidity = 36.3 %
-----------------
-----------------
Temperature = 24.1 C
Humidity = 36.5 %
-----------------
-----------------
```

Breaking down the program

The first line imports the Python `sys` library:

```
import sys
```

The next line tells Python to search the `SDL_Pi_HDC1080_Python3` directory below our current directory so it can find our library:

```
sys.path.append('./SDL_Pi_HDC1080_Python3')
```

More imports:

```
import time
import SDL_Pi_HDC1080
```

This statement instantiates the hdc1080 object and initializes it:

```
# Main Program
print
print ("")
```

```
print ("Read Temperature and Humidity from HDC1080 using I2C bus ")
print ("")

hdc1080 = SDL_Pi_HDC1080.SDL_Pi_HDC1080()
```

These statements read the temperature and humidity and print them out to the terminal window. Note: You see that all the complexity of using an I2C device is hidden by use of the HDC1080 library:

```
while True:
        print ("------------------")
        print ("Temperature = %3.1f C" % hdc1080.readTemperature())
        print ("Humidity = %3.1f %%" % hdc1080.readHumidity())
```

Sleep for three seconds and then repeat:

```
        print ("------------------")

        time.sleep(3.0)
```

Now that you have this program, you could add all sorts of things to it, such as turning on a red LED if it gets too hot, or turning on a blue LED if it gets to cold.

TIP

You could even tweet your temperature and humidity by using the `https://python-twitter.readthedocs.io` Python library.

TECHNICAL STUFF

LOOKING AT AN I2C DRIVER

I2C devices have an address (like 0x40 the address of our HDC1080) and they also have registers. You can think of these as numbered pointers for which to write commands and read data. The HDC1080 has eight different registers with different hex addresses, as shown in the figure below.

Pointer	Name	Reset value	Description
0x00	Temperature	0x0000	Temperature measurement output
0x01	Humidity	0x0000	Relative Humidity measurement output
0x02	Configuration	0x1000	HDC1080 configuration and status
0xFB	Serial ID	device dependent	First 2 bytes of the serial ID of the part
0xFC	Serial ID	device dependent	Mid 2 bytes of the serial ID of the part
0xFD	Serial ID	device dependent	Last byte bit of the serial ID of the part
0xFE	Manufacturer ID	0x5449	ID of Texas Instruments
0xFF	Device ID	0x1050	ID of the device

(continued)

(continued)

An I2C driver basically reads and writes from these addresses to control the HDC1080 and to read the temperature and humidity data. The figure below shows the format of the temperature register located at pointer address 0x00.

Name	Bits		Description
TEMPERATURE	[15:02]	Temperature	Temperature measurement (read only)
	[01:00]	Reserved	Reserved, always 0 (read only)

From the SDL_Pi_HDC1080 Python library, let's take a look at the Python code to actually read that I2C register:

```
        def readTemperature(self):

                s = [HDC1080_TEMPERATURE_REGISTER] # temp
                s2 = bytearray( s )
                HDC1080_fw.write( s2 )
                time.sleep(0.0625)  # From the data sheet

    #read 2 byte temperature data
                data = HDC1080_fr.read(2)

                buf = array.array('B', data)


                # Convert the data
                temp = (buf[0] * 256) + buf[1]
                cTemp = (temp / 65536.0) * 165.0 - 40
                return cTemp
```

This looks a lot more intimidating than it actually is. Breaking it down, we first define the function:

```
        def readTemperature(self):
```

The we format the pointer address (0x00 in this case) into a byte array:

```
                s = [HDC1080_TEMPERATURE_REGISTER] # temp
                s2 = bytearray( s )
```

We set up the read from the pointer register:

```
                HDC1080_fw.write( s2 )
```

We delays 6.24ms, as required by the data sheet:

```
time.sleep(0.0625)  # From the data sheet
```

We read two bytes:

```
#read 2 byte temperature data
data = HDC1080_fr.read(2)
```

And place the two bytes into a byte array:

```
buf = array.array('B', data)
```

Then we convert the data bytes using the formula in the data sheet:

```
# Convert the data
temp = (buf[0] * 256) + buf[1]
cTemp = (temp / 65536.0) * 165.0 - 40
```

And send the temperature back to the calling program:

```
return cTemp
```

There are many different low-level drivers and programs out there to read and write from I2C devices on the Raspberry Pi. This is an example of one of the most common methods. Other methods include Adafruit_i2c, SMBUS, PyComms, Quick2Wire, and others. We typically use the SMBUS library, but once in a while you will run into a device that requires some non-SMBUS functionality to get it to work.

A Fun Experiment for Measuring Oxygen and a Flame

In this more complex experiment, we take an Grove oxygen sensor and place it under a more or less sealed glass jar with a lit candle. The idea is to measure the oxygen in the glass jar and watch the level go down as the candle consumes the oxygen. After a quick search on the Internet, we expect it to drop about 30 percent before the flame is extinguished. That would be from 21 percent oxygen to about 14.7 percent oxygen.

We are storing the information in a CSV file (comma delimited file) to graph later by using MatPlotLib. You could also easily read this data into an Excel spreadsheet and graph it using Excel.

MatPlotLib is a Python library for making publication quality plots using methods similar to MATLIB. You can output formats such as PDF, Postscript, SVG, and PNG.

Then we lit the candle and watched the data on the browser window connected to the Raspberry Pi.

What we need to do this experiment:

- **Analog-to-digital converter:** This converts the analog output of the oxygen sensor to digital data for the Raspberry Pi.
- **Grove oxygen sensor:** This sensor measures the percentage of oxygen in the air and converts it to an analog value (0 – 5V).
- **A candle:** The candle will consume the oxygen in the bowl. You can use any candle as long as it fits under the bowl.
- **A large glass bowl:** The bowl will cover and seal the candle to measure the oxygen.

Analog-to-digital converters (ADC)

An analog-to-digital converter takes an analog signal (see the difference between an analog and digital signal in Chapter 2 of this minibook) and converts it to a digital signal (16 bits, in this case) for a computer to read.

When you have the digital number in the computer, you can scale it back to volts by multiplying it by (5.0/65535.0) to produce a floating point number representing volts.

No question about it. The lack of an analog-to-digital converter is a real knock on the Raspberry Pi.

The Grove analog-to-digital converter we use in this experiment is a Grove four-channel, 16-bit analog-to-digital converter available on store.switchdoc.com and also on Amazon.com. (See Figure 3-5.)

There are other Grove ADC modules available from Seeedstudio.com, but we wanted to use a 16-bit ADC converter for greater accuracy and the fact it has four channels instead of just one channel.

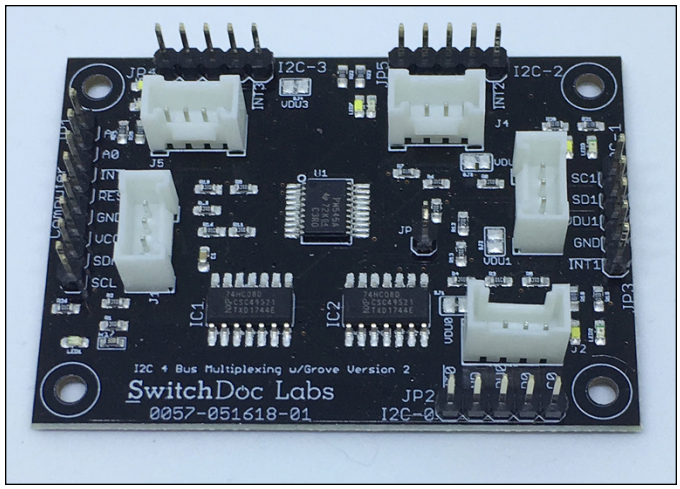


FIGURE 3-5: The Grove four-channel, 16-bit ADC.

The Grove oxygen sensor

The Grove gas sensor (O_2) is a sensor to test the oxygen concentration in the air. (See Figure 3-6.) It detects the current oxygen concentration and outputs voltage values proportional to the concentration of oxygen. You can interpret these numbers by referring to the oxygen concentration linear characteristic graph.

This sensor value only reflects the approximate trend of oxygen gas concentration in a permissible error range, it does not represent the exact oxygen gas concentration. The detection of certain components in the air usually requires a more precise and costly instrument, which cannot be done with a single gas sensor. This sensor also requires about a 30-minute warm-up time.

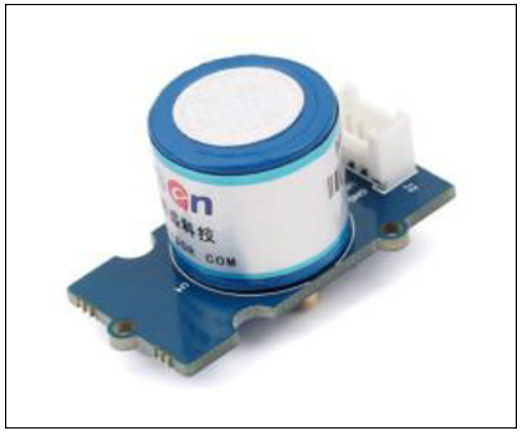


FIGURE 3-6: The Grove oxygen sensor.

Hooking up the oxygen experiment

By now, you have quite a bit of experience hooking up Grove devices to the Raspberry Pi. Follow these steps to set up the oxygen sensor:

1. **Disconnect the power from the Raspberry Pi.**
2. **Plug a Grove cable into the Grove oxygen sensor and then into the Grove connector marked A1 on the Grove four-channel, 16-bit ADC board.**
3. **Plug another Grove cable into the Grove connector marked I2C on the Grove four-channel, 16-bit ADC board. Plug the other end of that Grove cable into one of the connectors marked I2C on the Pi2Grover board plugged into the Raspberry Pi. (See Figure 3-7.)**
4. **Apply the power to the Raspberry Pi.**

FIGURE 3-7: The complete Raspberry Pi/ADC/oxygen sensor hookup.

5. **Run the command `i2cdetect -y 1` inside a terminal window. You should see this output:**

```
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

```
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- 48 -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

Address 0x48 is the Grove four-channel, 16-bit ADC board. If you don't see this, go back and check your wiring.

Now you will test the setup by running a simple Python program.

First make a new directory for the program:

```
cd
mkdir oxygenProject
cd oxygenProject
git clone https://github.com/switchdoclabs/SDL_Pi_Grove4Ch16BitADC
```

Then enter the following code into a file called `senseOxygen.py` in your terminal window using nano:

```
import time,  sys

sys.path.append('./SDL_Pi_Grove4Ch16BitADC/SDL_Adafruit_ADS1x15')

import SDL_Adafruit_ADS1x15

ADS1115 = 0x01    # 16-bit ADC

# Select the gain
gain = 6144  # +/- 6.144V

# Select the sample rate
sps = 250  # 250 samples per second

# Initialize the ADC using the default mode (use default I2C address)
adc = SDL_Adafruit_ADS1x15.ADS1x15(ic=ADS1115)
dataFile = open("oxygenData.csv",'w')

totalSeconds = 0
while (1):

    # Read oxygen channel  in single-ended mode using the settings above
```

```
        print ("--------------------")
        voltsCh1 = adc.readADCSingleEnded(1, gain, sps) / 1000
        rawCh1 = adc.readRaw(1, gain, sps)

        # O2 Sensor
        sensorVoltage = voltsCh1 *(5.0/6.144)
        AMP  = 121
        K_O2  = 7.43
        sensorVoltage = sensorVoltage/AMP*10000.0
        Value_O2 = sensorVoltage/K_O2 - 1.05

        print ("Channel 1 =%.6fV raw=0x%4X O2 Percent=%.2f" % (voltsCh1, rawCh1,
    Value_O2 ))
        print ("--------------------")

        dataFile.write("%d,%.2f\n" % (totalSeconds, Value_O2))
        totalSeconds = totalSeconds + 1
        dataFile.flush()
        time.sleep(1.0)
```

WARNING

When you are done using the oxygen sensor, make sure you put it back in the included capped container and seal the top. Humidity will destroy the sensor over time. (We have destroyed these sensors in the past.)

When you run the program, here are the results:

```
--------------------
Channel 1 =2.436375V raw=0x32C2 O2 Percent= 22.05
--------------------
--------------------
Channel 1 =2.436375V raw=0x32C2 O2 Percent= 22.05
--------------------
--------------------
Channel 1 =2.436375V raw=0x32C1 O2 Percent= 22.05
--------------------
--------------------
Channel 1 =2.436375V raw=0x32C2 O2 Percent= 22.05
--------------------
--------------------
Channel 1 =2.436187V raw=0x32C1 O2 Percent= 22.05
--------------------
```

Breaking down the code

In these statements, we set the parameters for the ADC module:

```
import time,  sys

sys.path.append('./SDL_Pi_Grove4Ch16BitADC/SDL_Adafruit_ADS1x15')

import SDL_Adafruit_ADS1x15

Normal Imports.  Notice the path goes to the subdirectory in the your directory.

ADS1115 = 0x01       # 16-bit ADC

# Select the gain
gain = 6144  # +/- 6.144V

# Select the sample rate
sps = 250  # 250 samples per second
```

Then we open the text file to store our data that you can graph later with Excel or other method:

```
# Initialize the ADC using the default mode (use default I2C address)
adc = SDL_Adafruit_ADS1x15.ADS1x15(ic=ADS1115)

dataFile = open("oxygenData.csv",'w')
```

Read the data from the ADC. Volts and Raw data (Raw data just for information):

```
totalSeconds = 0
while (1):

    # Read oxygen channel  in single-ended mode using the settings above

        print ("--------------------")
        voltsCh1 = adc.readADCSingleEnded(1, gain, sps) / 1000
        rawCh1 = adc.readRaw(1, gain, sps)
```

This is from the specification of the O2 sensor on how to calculate O2 percentage from the voltage from the ADC:

```
        # O2 Sensor
        sensorVoltage = voltsCh1 *(5.0/6.144)
        AMP  = 121
        K_O2  = 7.43
        sensorVoltage = sensorVoltage/AMP*10000.0
        Value_O2 = sensorVoltage/K_O2 - 1.05
```

Here you write the data out to the file:

```
        print ("Channel 1 =%.6fV raw=0x%4X O2 Percent=%.2f" % (voltsCh1, rawCh1,
    Value_O2 ))
        print ("----------------------")

        dataFile.write("%d,%.2f\n" % (totalSeconds, Value_O2))
```

We flush the file to make sure the last value is written to the file. You will eventually terminate this program with a Ctrl-C:

```
        totalSeconds = totalSeconds + 1
        dataFile.flush()
```

Now that you have all the software built, take your candle put it under the bowl with the oxygen sensor, start your program, and light the candle. (See Figure 3-8.) After a while, the flame will go out. Stop your program with a Ctrl-C and look at and graph your data. (See Figure 3-9.)

```
        time.sleep(1.0)
```

FIGURE 3-8: The Start of Our O2 Experiment.

Looking at the numbers, we determined that we started with about 21 percent oxygen and the candle went out at about 15.8 percent oxygen, a reduction of about 25 percent. This is lower than the expected 30 percent reduction of oxygen levels.

The difference? We would guess a combination of sensor accuracy and candle type. One more thing to note: Look at the graph right after the candle went out. You can see that the seal wasn't perfect as the oxygen started to creep up.

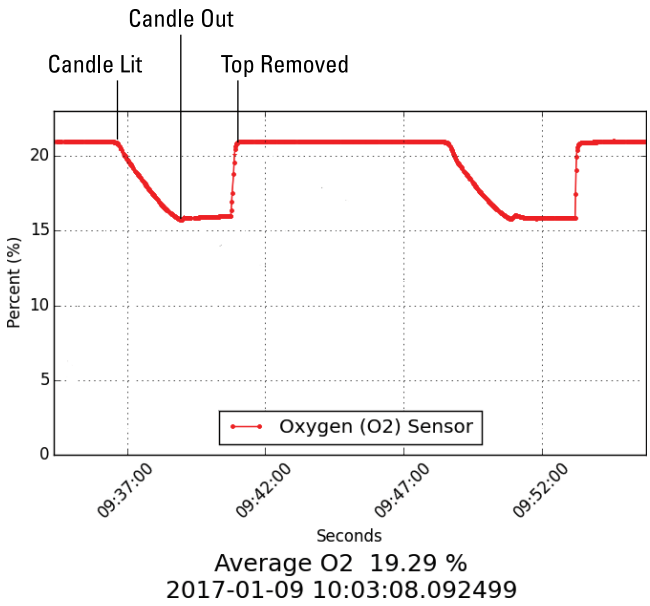


FIGURE 3-9: The graph of the data from our O2 experiment.

Building a Dashboard on Your Phone Using Blynk and Python

When you drive a car, all the information about how fast you are going, how much fuel remains, and other car information is on your dashboard. We're going to show you how construct a simple dashboard so you can view your project data on your smartphone. To illustrate how to do this, we'll use the free app Blynk (free for small dashboards; they charge you a bit for more energy to build more controls). This app is available on the various app stores for both Android and iPhones. We'll use the iPhone to show the usage, but it is pretty much identical for Android phones.

HDC1080 temperature and humidity sensor redux

Earlier in this chapter, you built a temperature and humidity sensing project using a Raspberry Pi. Grab that project now and let's write some more software for it to

connect it up to Blynk and display our values on the unit. Here Figure 3-10 shows us what our dashboard will look like.

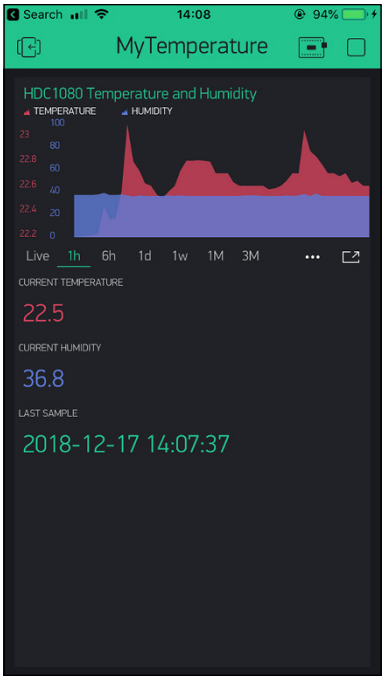


FIGURE 3-10: The MyTemperature dashboard.

OTHER DASHBOARDS

Blynk is hardly the only Internet dashboard out there. You can also check out:

- Freeboard
- XOBXOB
- Adafruit IO
- ThinkSpeak
- IBM Cloud
- Initialstate

All of them have different strengths and weaknesses. All of them have some sort of free option that varies on a regular basis.

How to add the Blynk dashboard

First, we show you how to set up the Blynk app. This is done on an iPhone, but it is very similar to using it on an Android phone. And the Python is identical in both cases!

1. **Install the Blynk app on your mobile phone. (See Figure 3-11.)**

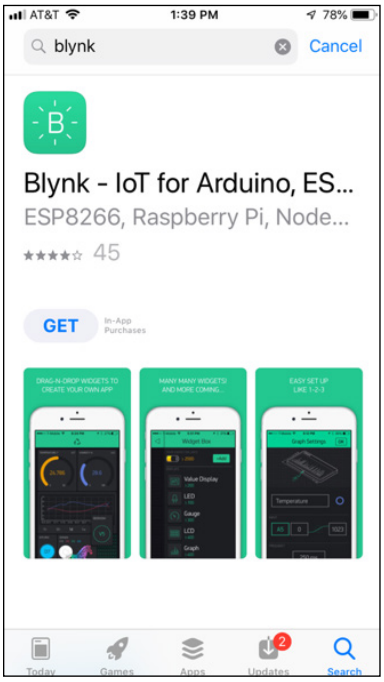


FIGURE 3-11: Blynk in the Appstore.

2. **Open the Blynk app and create an account. (See Figure 3-12.)**

   You need to supply an account and an email address, but they won't charge you anything for this.

3. **Click the button to scan a QR (see Figure 3-13).**
4. **Scan the QR code shown in Figure 3-14.**
5. **You will now see the MyTemperature app on your screen. (See Figure 3-15.)**

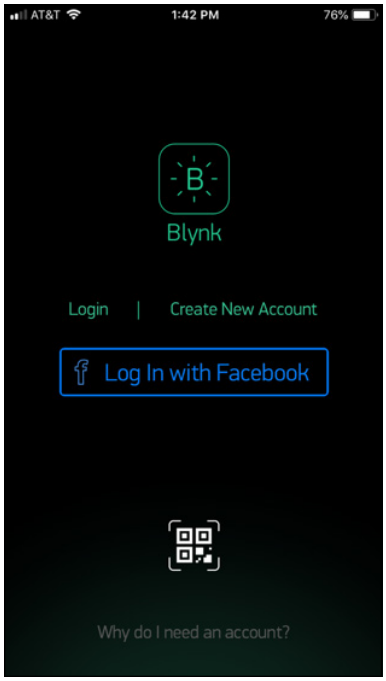


FIGURE 3-12: Creating a Blynk account.

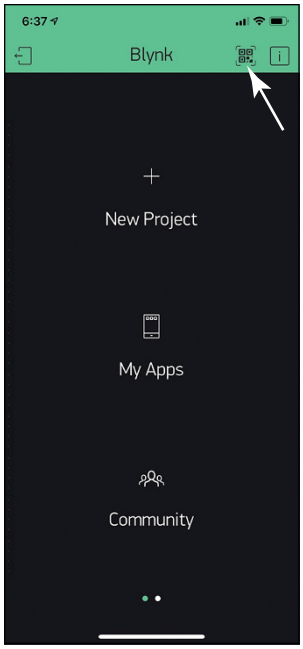


FIGURE 3-13: Click for QR.

FIGURE 3-14: The QR for generating your myTemperature app in Blynk.

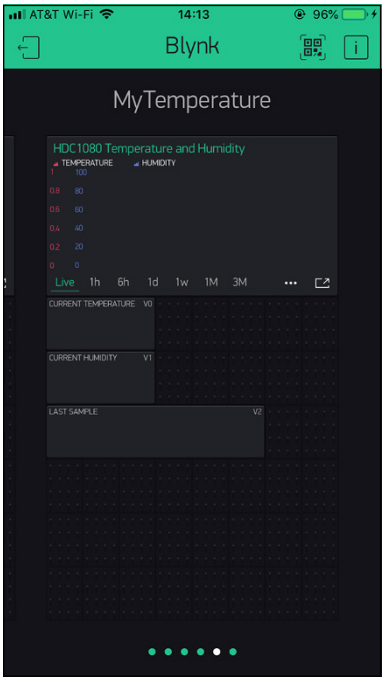


FIGURE 3-15: The MyTemperature app.

6. **Click the middle of the project to select the project. Then click the indicated button to go to project settings.**

   Note: Now copy and paste the authentication token (`AUTH TOKEN`) into an email to yourself or into some other secure document, as we will be putting this in the Python `temperatureTest.py` program file in the next section. Figure 3-16 shows the initial screen of the Blynk app. Figure 3-17 shows the authentication code. You now have your myTemperature app loaded.

You have completed the Blynk myTemperature app installation. Now let's modify the software to support the Blynk app.

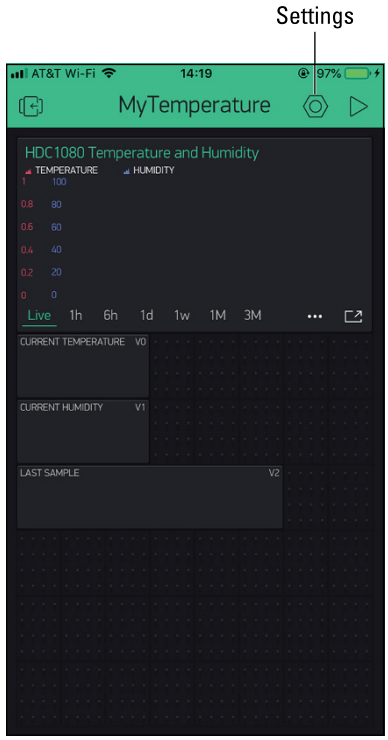


FIGURE 3-16: The initial screen of the Blynk app.

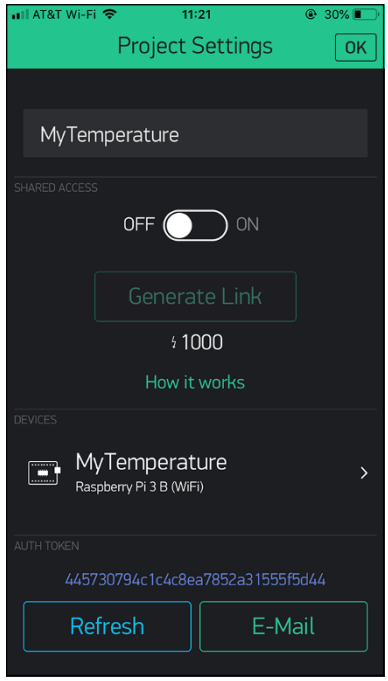


FIGURE 3-17: The authentication token in the MyTemperature app project settings.

The modified temperatureTest.py software for the Blynk app

To modify the software to support the Blynk app, follow these steps:

1. **Create a directory in your main directory by entering the following:**

```
cd
mkdir myTemperature
cd myTemperature
```

Now you are in the `myTemperature` directory.

2. **Before looking at the Python code for reading and then "Blynking" your temperature, install the library on the Raspberry Pi. You do this by "cloning" the library located up at github.com by using the following command in your terminal window:**

```
git clone https://github.com/switchdoclabs/SDL_Pi_HDC1080_Python3.git
```

3. **Enter the code below into a file named `myTemperature.py` using nano or your favorite editor.**

```
#!/usr/bin/env python3

#imports

import sys

sys.path.append('./SDL_Pi_HDC1080_Python3')

import time
import SDL_Pi_HDC1080

import requests
import json


BLYNK_URL = 'http://blynk-cloud.com/'
BLYNK_AUTH = 'xxxx'

# Main Program
print
print ("")
```

```
print ("Read Temperature and Humidity from HDC1080 using I2C bus and send
   to Blynk ")
print ("")

hdc1080 = SDL_Pi_HDC1080.SDL_Pi_HDC1080()

def blynkUpdate(temperature, humidity):
    print ("Updating Blynk")

    try:

        put_header={"Content-Type": "application/json"}
        val = temperature
        put_body = json.dumps(["{0:0.1f}".format(val)])
        r = requests.put(BLYNK_URL+BLYNK_AUTH+'/update/V0', data=put_body,
   headers=put_header)


        put_header={"Content-Type": "application/json"}
        val = humidity
        put_body = json.dumps(["{0:0.1f}".format(val)])
        r = requests.put(BLYNK_URL+BLYNK_AUTH+'/update/V1', data=put_body,
   headers=put_header)


        put_header={"Content-Type": "application/json"}
        val = time.strftime("%Y-%m-%d %H:%M:%S")
        put_body = json.dumps([val])
        r = requests.put(BLYNK_URL+BLYNK_AUTH+'/update/V2', data=put_body,
   headers=put_header)

        return 1

    except Exception as e:
                print ("exception in updateBlynk")
                print (e)
                return 0

while True:

        temperature =  hdc1080.readTemperature()
        humidity = hdc1080.readHumidity()

        print ("------------------")
        print ("Temperature = %3.1f C" % hdc1080.readTemperature())
```