Chapter 2. TensorFlow essentials

This chapter covers

- Understanding the TensorFlow workflow
- Creating interactive notebooks with Jupyter
- Visualizing algorithms by using TensorBoard

Before implementing machine-learning algorithms, let's first get familiarized with how to use TensorFlow. You're going to get your hands dirty writing simple code right away! This chapter covers some essential advantages of TensorFlow to convince you it's the machine-learning library of choice.

As a thought experiment, let's see what happens when we use Python code without a handy computing library. It'll be like using a new smartphone without installing any additional apps. The functionality will be there, but you'd be so much more productive if you had the right tools.

Suppose you're a private business owner tracking the flow of sales for your products. Your inventory consists of 100 items, and you represent each item's price in a vector called `prices`. Another 100-dimensional vector called `amounts` represents the inventory count of each item. You can write the chunk of Python code shown in the following listing to calculate the revenue of selling all products. Keep in mind that this code doesn't import any libraries.

Listing 2.1. Computing the inner product of two vectors without using a library

```
revenue = 0
for price, amount in zip(prices, amounts):
    revenue += price * amount
```

That's a lot of code just to calculate the inner product of two vectors (also known as the *dot product*). Imagine how much code would be required for something more complicated, such as solving linear equations or computing the distance between two vectors.

When installing the TensorFlow library, you also install a well-known and robust Python library called NumPy, which facilitates mathematical manipulation in Python. Using Python without its libraries (NumPy and TensorFlow) is like using a camera without auto mode: you gain more flexibility, but you can easily make careless mistakes (for the record, we have nothing against photographers who micromanage aperture, shutter, and ISO). It's easy to make mistakes in machine learning, so let's keep our camera on autofocus and use TensorFlow to help automate tedious software development.

The following listing shows how to concisely write the same inner product using NumPy.

Listing 2.2. Computing the inner product using NumPy

```
import numpy as np
revenue = np.dot(prices, amounts)
```

Python is a succinct language. Fortunately for you, that means this book doesn't have pages and pages of cryptic code. On the other hand, the brevity of the Python language also implies that a lot is happening behind each line of code, which you should familiarize yourself with carefully as you follow along in this chapter.

Machine-learning algorithms require many mathematical operations. Often, an algorithm boils down to a composition of simple functions iterated until convergence. Sure, you may use any standard programming language to perform these computations, but the secret to both manageable and high-performing code is the use of a well-written library, such as TensorFlow (which officially supports Python and C++).

Tip

Detailed documentation about various functions for the Python and C++ APIs are available at www.tensorflow.org/api_docs/ (http://www.tensorflow.org/api_docs/).

The skills you learn in this chapter are geared toward using TensorFlow for computations, because machine learning relies on mathematical formulations. After going through the examples and code listings, you'll be able to use TensorFlow for arbitrary tasks, such as computing statistics on big data. The focus here is entirely on how to use TensorFlow, as opposed to machine learning. That sounds like a gentle start, right?

Later in this chapter, you'll use TensorFlow's flagship features that are essential for machine learning. These include representation of computation as a dataflow graph, separation of design and execution, partial subgraph computation, and autodifferentiation. Without further ado, let's write your first TensorFlow code!

2.1. ENSURING THAT TENSORFLOW WORKS

First, you should ensure that everything is working correctly. Check the oil level in your car, repair the blown fuse in your basement, and ensure that your credit balance is zero. Just kidding; we're talking about TensorFlow.

Before you begin, follow the procedures in the appendix for step-by-step installation instructions. Create a new file called test.py for your first piece of code. Import TensorFlow by running the following script:

```
import tensorflow as tf
```

Having technical difficulty?

An error commonly occurs at this step if you installed the GPU version and the library fails to search for CUDA drivers. Remember, if you compiled the library with CUDA, you need to update your environment variables with the path to CUDA. Check the CUDA instructions on TensorFlow. (See http://mng.bz/QUMh (http://mng.bz/QUMh) for further information.)

This single import prepares TensorFlow to do your bidding. If the Python interpreter doesn't complain, you're ready to start using TensorFlow!

Sticking with TensorFlow conventions

The TensorFlow library is usually imported with the `tf` alias. Generally, qualifying TensorFlow with `tf` is a good idea to remain consistent with other developers and open source TensorFlow projects. Of course, you may use another alias (or no alias at all), but then successfully reusing other people's snippets of TensorFlow code in your own projects will be an involved process.

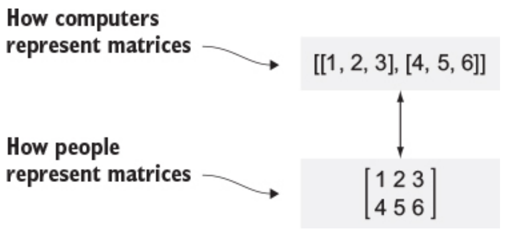
2.2. REPRESENTING TENSORS

Now that you know how to import TensorFlow into a Python source file, let's start using it! As covered in the previous chapter, a convenient way to describe an object in the real world is through listing its properties, or features. For example, you can describe a car by its color, model, engine type, mileage, and so on. An ordered list of features is called a *feature vector,* and that's exactly what you'll represent in TensorFlow code.

Feature vectors are one of the most useful devices in machine learning because of their simplicity (they're just a list of numbers). Each data item typically consists of a feature vector, and a good dataset has hundreds, if not thousands, of these feature vectors. No doubt, you'll often be dealing with more than one vector at a time. A matrix concisely represents a list of vectors, where each column of a matrix is a feature vector.

The syntax to represent matrices in TensorFlow is a vector of vectors, each of the same length. Figure 2.1 is an example of a matrix with two rows and three columns, such as [[1, 2, 3], [4, 5, 6]]. Notice that this is a vector containing two elements, and each element corresponds to a row of the matrix.

Figure 2.1. The matrix in the lower half of the diagram is a visualization from its compact code notation in the upper half of the diagram. This form of notation is a common paradigm in most scientific computing libraries.


How computers represent matrices
[[1, 2, 3], [4, 5, 6]]
How people represent matrices
1 2 3
4 5 6

We access an element in a matrix by specifying its row and column indices. For example, the first row and first column indicate the very first top-left element. Sometimes it's convenient to use more than two indices, such as when referencing a pixel in a color image not only by its row and column, but also by its red/green/blue channel. A *tensor* is a generalization of a matrix that specifies an element by an arbitrary number of indices.

Example of a tensor

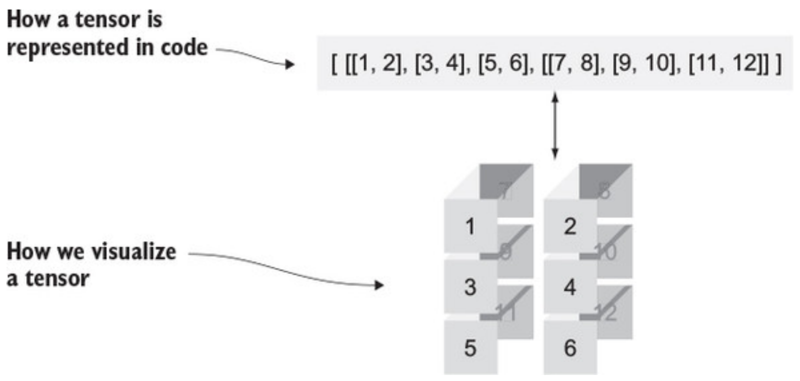
Suppose an elementary school enforces assigned seating for all its students. You're the principal, and you're terrible with names. Luckily, each classroom has a grid of seats, and you can easily nickname a student by their row and column index.

The school has multiple classrooms, so you can't simply say, "Good morning 4,10! Keep up the good work." You need to also specify the classroom: "Hi 4,10 from classroom 2." Unlike a matrix, which needs only two indices to specify an element, the students in this school need three numbers. They're all a part of a rank-3 tensor!

The syntax for tensors is even more nested vectors. For example, as shown in figure 2.2, a 2 × 3 × 2 tensor is [[[1,2], [3,4], [5,6]], [[7,8], [9,10], [11,12]]], which can be thought of as two matrices, each of size 3 × 2. Consequently, we say this tensor has a *rank* of 3.

In general, the rank of a tensor is the number of indices required to specify an element. Machine-learning algorithms in TensorFlow act on tensors, so it's important to understand how to use them.

Figure 2.2. This tensor can be thought of as multiple matrices stacked on top of each other. To specify an element, you must indicate the row and column, as well as which matrix is being accessed. Therefore, the rank of this tensor is 3.



It's easy to get lost in the many ways to represent a tensor. Intuitively, three lines of code in listing 2.3 are trying to represent the same 2 × 2 matrix. This matrix represents two feature vectors of two dimensions each. It could, for example, represent two people's ratings of two movies. Each person, indexed by the row of the matrix, assigns a number to describe their review of the movie, indexed by the column. Run the code to see how to generate a matrix in TensorFlow.

Listing 2.3. Different ways to represent tensors

```
import tensorflow as tf
import numpy as np                                   1

m1 = [[1.0, 2.0],
      [3.0, 4.0]]                                    2

m2 = np.array([[1.0, 2.0],
               [3.0, 4.0]], dtype=np.float32)        2

m3 = tf.constant([[1.0, 2.0],
                  [3.0, 4.0]])                       2

print(type(m1))                                      3
print(type(m2))                                      3
print(type(m3))                                      3

t1 = tf.convert_to_tensor(m1, dtype=tf.float32)      4
t2 = tf.convert_to_tensor(m2, dtype=tf.float32)      4
t3 = tf.convert_to_tensor(m3, dtype=tf.float32)      4

print(type(t1))                                      5
print(type(t2))                                      5
print(type(t3))                                      5
```

- ***1*** **You'll use NumPy matrices in TensorFlow.**
- ***2*** **Defines a 2 × 2 matrix in three ways**
- ***3*** **Prints the type for each matrix**
- ***4*** **Creates tensor objects out of the various types**
- ***5*** **Notice that the types will be the same now.**

The first variable (`m1`) is a list, the second variable (`m2`) is an `ndarray` from the NumPy library, and the last variable (`m3`) is TensorFlow's constant `Tensor` object that you initialize using `tf.constant`.

All operators in TensorFlow, such as `negative`, are designed to operate on tensor objects. A convenient function you can sprinkle anywhere just to make sure you're dealing with tensors as opposed to the other types is `tf.convert_to_tensor( ... )`. Most functions in the TensorFlow library already perform this function (redundantly) even if you forget to do so. Using `tf.convert_to_tensor( ... )` is optional, but we show it here because it helps demystify the implicit type system being handled across the library. Listing 2.3 outputs the following three times:

```
<class 'tensorflow.python.framework.ops.Tensor'>
```

Tip

You can find these code listings on the book's website, to make copying and pasting easier: www.manning.com/books/machine-learning-with-tensorflow (http://www.manning.com/books/machine-learning-with-tensorflow).

Let's take another look at defining tensors in code. After importing the TensorFlow library, you can use the `tf.constant` operator as follows. Here are a couple of tensors of various dimensions.

Listing 2.4. Creating tensors

```
import tensorflow as tf

m1 = tf.constant([[1., 2.]])        1

m2 = tf.constant([[1],
                  [2]])             2

m3 = tf.constant([ [[1,2],
                    [3,4],
                    [5,6]],
                    [[7,8],
                    [9,10],
                    [11,12]] ])     3

print(m1)                           4
print(m2)                           4
print(m3)                           4
```

- ***1*** **Defines a 2 × 1 matrix**
- ***2*** **Defines a 1 × 2 matrix**
- ***3*** **Defines a rank-3 tensor**
- ***4*** **Try printing the tensors.**

Running listing 2.4 produces the following output:

```
Tensor( "Const:0",
        shape=TensorShape([Dimension(1), Dimension(2)]),
        dtype=float32 )
Tensor( "Const_1:0",
        shape=TensorShape([Dimension(2), Dimension(1)]),
        dtype=int32 )
Tensor( "Const_2:0",
        shape=TensorShape([Dimension(2), Dimension(3), Dimension(2)]),
        dtype=int32 )
```

As you can see from the output, each tensor is represented by the aptly named `Tensor` object. Each `Tensor` object has a unique label (`name`), a dimension (`shape`) to define its structure, and a data type (`dtype`) to specify the kind of values you'll manipulate. Because you didn't explicitly provide a name, the library automatically generated the names: `Const:0`, `Const_1:0`, and `Const_2:0`.

Tensor types

Notice that each element of `m1` ends with a decimal point. The decimal point tells Python that the data type of the elements isn't an integer, but instead a float. You can pass in explicit `dtype` values. Much like NumPy arrays, tensors take on a data type that specifies the kind of values you'll manipulate in that tensor.

TensorFlow also comes with a few convenient constructors for some simple tensors. For example, `tf.zeros(shape)` creates a tensor with all values initialized at zero of a specific shape. Similarly, `tf.ones(shape)` creates a tensor of a specific shape with all values initialized at once. The `shape` argument is a one-dimensional (1D) tensor of type `int32` (a list of integers) describing the dimensions of the tensor.

Exercise 2.1

Initialize a 500 × 500 tensor with all elements equaling 0.5.

Answer

```
tf.ones([500,500]) * 0.5
```

2.3. CREATING OPERATORS

Now that you have a few starting tensors ready to be used, you can apply more-interesting operators such as addition or multiplication. Consider each row of a matrix representing the transaction of money to (positive value) and from (negative value) another person. Negating the matrix is a way to represent the transaction history of the other person's flow of money. Let's start simple and run a negation op (short for *operation*) on the `m1` tensor from listing 2.4. Negating a matrix turns the positive numbers into negative numbers of the same magnitude, and vice versa.

Negation is one of the simplest operations. As shown in listing 2.5, negation takes only one tensor as input, and produces a tensor with every element negated. Try running the code. If you master how to define negation, it'll provide a stepping stone to generalize that skill to all other TensorFlow operations.

Note

Defining an operation, such as negation, is different from *running* it. So far, you've *defined* how operations should behave. In section 2.4, you'll *evaluate* (or *run*) them to compute their value.

Listing 2.5. Using the negation operator

```
import tensorflow as tf

x = tf.constant([[1, 2]])          1
negMatrix = tf.negative(x)         2
print(negMatrix)                   3
```

- ***1*** **Defines an arbitrary tensor**
- ***2*** **Negates the tensor**
- ***3*** **Prints the object**

Listing 2.5 generates the following output:

```
Tensor("Neg:0", shape=TensorShape([Dimension(1), Dimension(2)]), dtype=int32)
```

Notice that the output isn't `[[-1, -2]]`. That's because you're printing out the definition of the negation op, not the actual evaluation of the op. The printed output shows that the negation op is a `Tensor` class with a name, shape, and data type. The name was automatically assigned, but you could've provided it explicitly as well when using the `tf.negative` op in listing 2.5. Similarly, the shape and data type were inferred from the `[[1, 2]]` that you passed in.

Useful TensorFlow operators

The official documentation carefully lays out all available math ops: www.tensorflow.org/api_guides/python/math_ops (http://www.tensorflow.org/api_guides/python/math_ops). Specific examples of commonly used operators include the following:

`tf.add(x, y)`—Adds two tensors of the same type, *x* + *y*

`tf.subtract(x, y)`—Subtracts tensors of the same type, *x* – *y*

`tf.multiply(x, y)`—Multiplies two tensors element-wise

`tf.pow(x, y)`—Takes the element-wise *x* to the power of *y* `tf.exp(x)`—Equivalent to *pow*(*e*, *x*), where *e* is Euler's number (2.718 ...)

`tf.sqrt(x)`—Equivalent to *pow*(*x*, 0.5)

`tf.div(x, y)`—Takes the element-wise division of *x* and *y*

`tf.truediv(x, y)`—Same as `tf.div`, except casts the arguments as a float

`tf.floordiv(x, y)`—Same as `truediv`, except rounds down the final answer into an integer

`tf.mod(x, y)`—Takes the element-wise remainder from division

Exercise 2.2

Use the TensorFlow operators you've learned so far to produce the Gaussian distribution (also known as the normal distribution). See figure 2.3 for a hint. For reference, you can find the probability density of the normal distribution online: https://en.wikipedia.org/wiki/Normal_distribution.

Answer

Most mathematical expressions such as ×, −, +, and so on are just shortcuts for their TensorFlow equivalent, for brevity. The Gaussian function includes many operations, so it's cleaner to use shorthand notations as follows:

```
from math import pi
mean = 0.0
sigma = 1.0
(tf.exp(tf.negative(tf.pow(x – mean, 2.0) /
               (2.0 * tf.pow(sigma, 2.0) ))) *
 (1.0 / (sigma * tf.sqrt(2.0 * pi) )))
```

2.4. EXECUTING OPERATORS WITH SESSIONS

A *session* is an environment of a software system that describes how the lines of code should run. In TensorFlow, a session sets up how the hardware devices (such as CPU and GPU) talk to each other. That way, you can design your machine-learning algorithm without worrying about micromanaging the hardware it runs on. You can later configure the session to change its behavior without changing a line of the machine-learning code.

To execute an operation and retrieve its calculated value, TensorFlow requires a session. Only a registered session may fill the values of a `Tensor` object. To do so, you must create a session class by using `tf.Session()` and tell it to run an operator, as shown in the following listing. The result will be a value you can later use for further computations.

Listing 2.6. Using a session

```
import tensorflow as tf

x = tf.constant([[1., 2.]])           1
neg_op = tf.negative(x)               2

with tf.Session() as sess:            3
    result = sess.run(negMatrix)      4
print(result)                         5
```

- *1* **Defines an arbitrary matrix**
- *2* **Runs the negation operator on it**
- *3* **Starts a session to be able to run operations**
- *4* **Tells the session to evaluate negMatrix**
- *5* **Prints the resulting matrix**

Congratulations! You've just written your first full TensorFlow code. Although all it does is negate a matrix to produce `[[-1, -2]]`, the core overhead and framework are just the same as everything else in TensorFlow. A session not only configures *where* your code will be computed on your machine, but also crafts *how* the computation will be laid out in order to parallelize computation.

Code performance seems a bit slow

You may have noticed that running your code took a few more seconds than expected. It may appear unnatural that TensorFlow takes seconds to negate a small matrix. But substantial preprocessing occurs to optimize the library for larger and more complicated computations.

Every `Tensor` object has an `eval()` function to evaluate the mathematical operations that define its value. But the `eval()` function requires defining a session object for the library to understand how to best use the underlying hardware. In listing 2.6, we used `sess.run(...)`, which is equivalent to invoking the `Tensor's eval()` function in the context of the session.

When you're running TensorFlow code through an interactive environment (for debugging or presentation purposes), it's often easier to create the session in interactive mode, where the session is implicitly part of any call to `eval()`. That way, the session variable doesn't need to be passed around throughout the code, making it easier to focus on the relevant parts of the algorithm, as seen in the following listing.

Listing 2.7. Using the interactive session mode

```
import tensorflow as tf
sess = tf.InteractiveSession()          1

x = tf.constant([[1., 2.]])             2
negMatrix = tf.negative(x)              2
```

```
result = negMatrix.eval()             3
print(result)                         4

sess.close()                          5
```

- *1* **Starts an interactive session so the sess variable no longer needs to be passed around**
- *2* **Defines an arbitrary matrix and negates it**
- *3* **You can now evaluate negMatrix without explicitly specifying a session.**
- *4* **Prints the negated matrix**
- *5* **Remember to close the session to free up resources.**

2.4.1. Understanding code as a graph

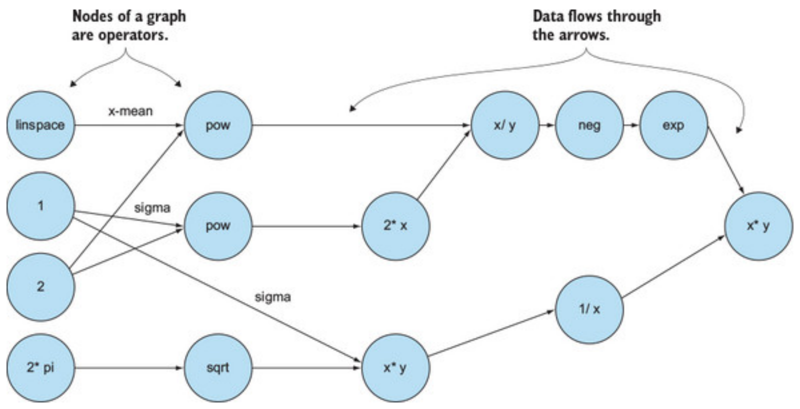
Consider a doctor who predicts the expected weight of a newborn to be 7.5 pounds. You'd like to figure out how that differs from the actual measured weight. Being an overly analytical engineer, you design a function to describe the likelihood of all possible weights of the newborn. For example, 8 pounds is more likely than 10 pounds.

You can choose to use the Gaussian (otherwise known as normal) probability distribution function. It takes as input a number, and outputs a non-negative number describing the probability of observing the input. This function shows up all the time in machine learning and is easy to define in TensorFlow. It uses multiplication, division, negation, and a couple of other fundamental operators.

Think of every operator as a node in a graph. Whenever you see a plus symbol (+) or any mathematical concept, just picture it as one of many nodes. The edges between these nodes represent the composition of mathematical functions. Specifically, the `negative` operator we've been studying is a node, and the incoming/outgoing edges of this node are how the `Tensor` transforms. A tensor flows through the graph, which is why this library is called TensorFlow!

Here's a thought: every operator is a strongly typed function that takes input tensors of a dimension and produces output of the same dimension. Figure 2.3 is an example of how the Gaussian function can be designed using TensorFlow. The function is represented as a graph in which operators are nodes and edges represent interactions between nodes. This graph, as a whole, represents a complicated mathematical function (specifically, the Gaussian function). Small segments of the graph represent simple mathematical concepts, such as negation or doubling.

Figure 2.3. The graph represents the operations needed to produce a Gaussian distribution. The links between the nodes represent how data flows from one operation to the next. The operations themselves are simple, but the complexity arises from the way they intertwine.



TensorFlow algorithms are easy to visualize. They can be simply described by flowcharts. The technical (and more correct) term for such a flowchart is a *dataflow graph*. Every arrow in a dataflow graph is called an *edge*. In addition, every state of the dataflow graph is called a *node*. The purpose of the session is to interpret your Python code into a dataflow graph, and then associate the computation of each node of the graph to the CPU or GPU.

2.4.2. Setting session configurations

You can also pass options to `tf.Session`. For example, TensorFlow automatically determines the best way to assign a GPU or CPU device to an operation, depending on what's available. You can pass an additional option, `log_device_placements=True`, when creating a session, as shown in the following listing, which will show you exactly where on your hardware the computations are evoked.

Listing 2.8. Logging a session

```
import tensorflow as tf

x = tf.constant([[1., 2.]])                                                  1
negMatrix = tf.negative(x)                                                   1

with tf.Session(config=tf.ConfigProto(log_device_placement=True)) as sess: 2
    result = sess.run(negMatrix)                                             3

print(result)                                                                4
```

- ***1* Defines a matrix and negates it**
- ***2* Starts the session with a special config passed into the constructor to enable logging**
- ***3* Evaluates negMatrix**
- ***4* Prints the resulting value**

This outputs info about which CPU/GPU devices are used in the session for each operation. For example, running listing 2.8 results in traces of output like the following to show which device was used to run the negation op:

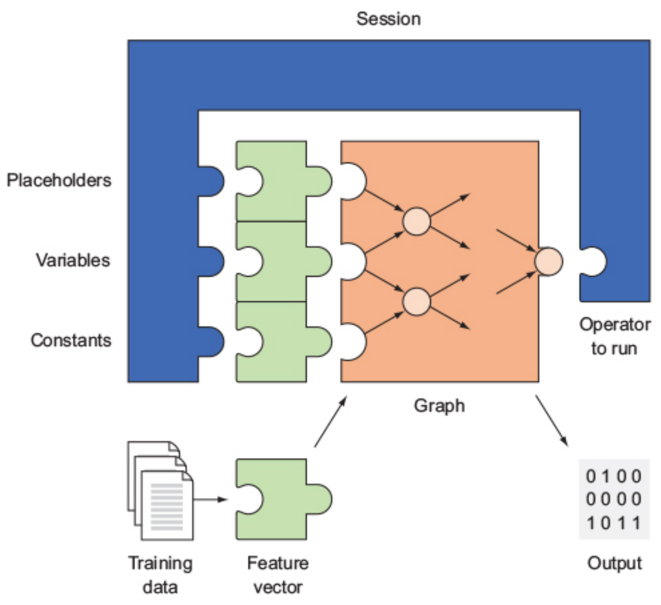
```
Neg: /job:localhost/replica:0/task:0/cpu:0
```

Sessions are essential in TensorFlow code. You need to call a session to "run" the math. Figure 2.4 maps out how the components on TensorFlow interact with the machine-learning pipeline. A session not only runs a graph operation, but also can take placeholders, variables, and constants as input. We've used constants so far, but in later sections we'll start using variables and placeholders. Here's a quick overview of these three types of values:

- *Placeholder*—A value that's unassigned but will be initialized by the session wherever it's run. Typically, placeholders are the input and output of your model.
- *Variable*—A value that can change, such as parameters of a machine-learning model. Variables must be initialized by the session before they're used.
- *Constant*—A value that doesn't change, such as hyperparameters or settings.

The entire pipeline for machine learning with TensorFlow follows the flow of figure 2.4. Most of the code in TensorFlow consists of setting up the graph and session. After you design a graph and hook up the session to execute it, your code is ready to use!

Figure 2.4. The session dictates how the hardware will be used to process the graph most efficiently. When the session starts, it assigns the CPU and GPU devices to each of the nodes. After processing, the session outputs data in a usable format, such as a NumPy array. A session optionally may be fed placeholders, variables, and constants.



2.5. WRITING CODE IN JUPYTER

Because TensorFlow is primarily a Python library, you should make full use of Python's interpreter. *Jupyter* is a mature environment for exercising the interactive nature of the language. It's a web application that displays computation elegantly so that you can share annotated interactive algorithms with others to teach a technique or demonstrate code.

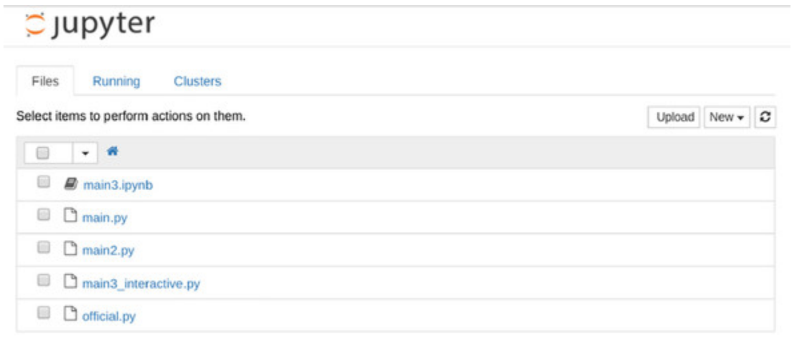
You can share your Jupyter notebooks with others to exchange ideas and download others' to learn about their code. See the appendix to get started with installing the Jupyter Notebook.

From a new terminal, change the directory to the location where you want to practice TensorFlow code, and start a notebook server:

```
$ cd ~/MyTensorFlowStuff
$ jupyter notebook
```

Running this command should launch a new browser window with the Jupyter Notebook dashboard. If no window automatically opens, you can manually navigate to http://localhost:8888 from any browser. You'll see a web page similar to the one in figure 2.5.

Figure 2.5. Running the Jupyter Notebook will launch an interactive notebook on http://localhost:8888.



Select items to perform actions on them.

main3.ipynb
main.py
main2.py
main3_interactive.py
official.py

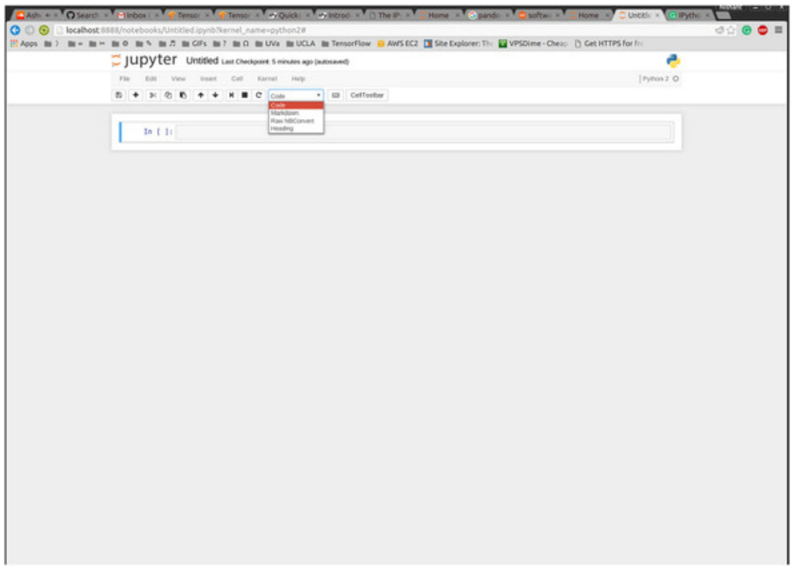
Create a new notebook by clicking the New drop-down menu at upper right; then choose Notebooks > Python 3. This creates a new file called Untitled.ipynb, which you can immediately start editing through the browser interface. You can change the name of the notebook by clicking the current Untitled name and typing in something more memorable, such as TensorFlow Example Notebook.

Everything in the Jupyter Notebook is an independent chunk of code or text called a *cell*. Cells help divide a long block of code into manageable pieces of code snippets and documentation. You can run cells individually, or choose to run everything at once, in order. There are three common ways to evaluate cells:

- Pressing Shift-Enter on a cell executes the cell and highlights the next cell below.
- Pressing Ctrl-Enter maintains the cursor on the current cell after executing it.
- Pressing Alt-Enter executes the cell and then inserts a new empty cell directly below.

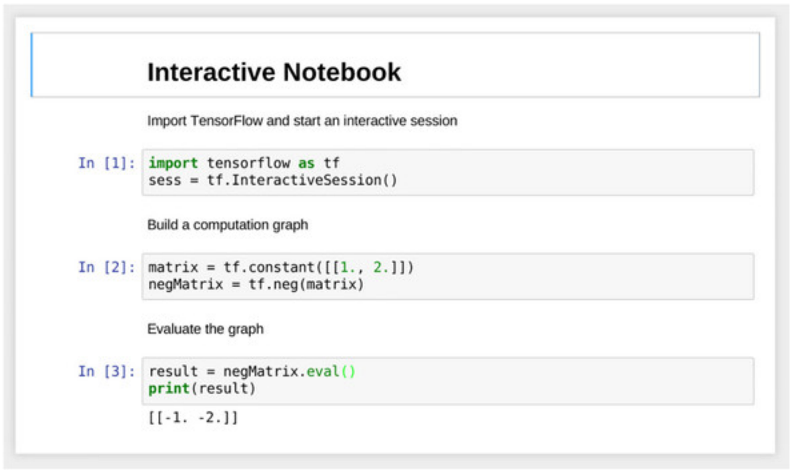
You can change the cell type by clicking the drop-down in the toolbar, as shown in figure 2.6. Alternatively, you can press Esc to leave edit mode, use the arrow keys to highlight a cell, and press Y to change to code mode or M for markdown mode.

Figure 2.6. The drop-down menu changes the type of cell in the notebook. The Code cell is for Python code, whereas the Markdown code is for text descriptions.



Finally, you can create a Jupyter notebook that elegantly demonstrates TensorFlow code by interlacing code and text cells as shown in figure 2.7.

Figure 2.7. An interactive Python notebook presents both code and comments grouped for readability.



Interactive Notebook

Import TensorFlow and start an interactive session

```
In [1]: import tensorflow as tf
        sess = tf.InteractiveSession()
```

Build a computation graph

```
In [2]: matrix = tf.constant([[1., 2.]])
        negMatrix = tf.neg(matrix)
```

Evaluate the graph

```
In [3]: result = negMatrix.eval()
        print(result)
        [[-1. -2.]]
```

Exercise 2.3

If you look closely at figure 2.7, you'll notice that it uses `tf.neg` instead of `tf.negative`. That's strange. Could you explain why we might have done that?

Answer

You should be aware that the TensorFlow library changed naming conventions, and you may run into these artifacts when following old TensorFlow tutorials online.

2.6. USING VARIABLES

Using TensorFlow constants is a good start, but most interesting applications require data to change. For example, a neuroscientist may be interested in detecting neural activity from sensor measurements. A spike in neural activity could be a Boolean variable that changes over time. To capture this in TensorFlow, you can use the `Variable` class to represent a node whose value changes over time.

Example of using a Variable object in machine learning

Finding the equation of a line that best fits many points is a classic machine-learning problem that's discussed in greater detail in the next chapter. The algorithm starts with an initial guess, which is an equation characterized by a few numbers (such as the slope or y-intercept). Over time, the algorithm generates increasingly better guesses for these numbers, which are also called *parameters*.

So far, we've been manipulating only constants. Programs with only constants aren't that interesting for real-world applications, so TensorFlow allows richer tools such as variables, which are containers for values that may change over time. A machine-learning algorithm updates the parameters of a model until it finds the optimal value for each variable. In the world of machine learning, it's common for parameters to fluctuate until eventually settling down, making variables an excellent data structure for them.

The code in listing 2.9 is a simple TensorFlow program that demonstrates how to use variables. It updates a variable whenever sequential data abruptly increases in value. Think about recording measurements of a neuron's activity over time. This piece of code can detect when the neuron's activity suddenly spikes. Of course, the algorithm is an oversimplification for didactic purposes.

Start with importing TensorFlow. TensorFlow allows you to declare a session by using `tf.InteractiveSession()`. When you've declared an interactive session, TensorFlow functions don't require the session attribute they would otherwise, which makes coding in Jupyter Notebooks easier.

Listing 2.9. Using a variable

```
import tensorflow as tf
sess = tf.InteractiveSession()                        1

raw_data = [1., 2., 8., -1., 0., 5.5, 6., 13]         2
spike = tf.Variable(False)                            3
spike.initializer.run()                               4

for i in range(1, len(raw_data)):                     5
    if raw_data[i] - raw_data[i-1] > 5:
        updater = tf.assign(spike, True)              6
        updater.eval()                                6
    else:
        tf.assign(spike, False).eval()
    print("Spike", spike.eval())

sess.close()                                          7
```

- ***1* Starts the session in interactive mode so you won't need to pass around sess**
- ***2* Let's say you have some raw data like this.**

- ***3*** **Creates a Boolean variable called spike to detect a sudden increase in a series of numbers**
- ***4*** **Because all variables must be initialized, initialize the variable by calling run() on its initializer.**
- ***5*** **Loops through the data (skipping the first element) and updates the spike variable when there's a significant increase**
- ***6*** **To update a variable, assign it a new value using tf.assign(<var name>, <new value>). Evaluate it to see the change.**
- ***7*** **Remember to close the session after it'll no longer be used.**

The expected output of listing 2.9 is a list of spike values over time:

```
('Spike', False)
('Spike', True)
('Spike', False)
('Spike', False)
('Spike', True)
('Spike', False)
('Spike', True)
```

2.7. SAVING AND LOADING VARIABLES

Imagine writing a monolithic block of code, of which you'd like to individually test a tiny segment. In complicated machine-learning situations, saving and loading data at known checkpoints makes it much easier to debug code. TensorFlow provides an elegant interface to save and load variable values to disk; let's see how to use it for that purpose.

Let's revamp the code that you created in listing 2.9 to save the spike data to disk so you can load it elsewhere. You'll change the spike variable from a simple Boolean to a vector of Booleans that captures the history of spikes (listing 2.10). Notice that you'll explicitly name the variables so they can be loaded later with the same name. Naming a variable is optional but highly encouraged to organize your code.

Try running this code to see the results.

Listing 2.10. Saving variables

```
import tensorflow as tf
sess = tf.InteractiveSession()

raw_data = [1., 2., 8., -1., 0., 5.5, 6., 13]
spikes = tf.Variable([False] * len(raw_data), name='spikes')
spikes.initializer.run()

saver = tf.train.Saver()

for i in range(1, len(raw_data)):
    if raw_data[i] - raw_data[i-1] > 5:
        spikes_val = spikes.eval()
        spikes_val[i] = True
        updater = tf.assign(spikes, spikes_val)
        updater.eval()

save_path = saver.save(sess, "spikes.ckpt")
```

```
print("spikes data saved in file: %s" % save_path)

sess.close()
```

- ***1*** **Imports TensorFlow and enables interactive sessions**
- ***2*** **Let's say you have a series of data like this.**
- ***3*** **Defines a Boolean vector called spikes to locate a sudden spike in raw data**
- ***4*** **Don't forget to initialize the variable.**
- ***5*** **The saver op will enable saving and restoring variables. If no dictionary is passed into the constructor, then it saves all variables in the current program.**
- ***6*** **Loop through the data and update the spikes variable when there's a significant increase.**
- ***7*** **Updates the value of spikes by using the tf.assign function**
- ***8*** **Don't forget to evaluate the updater; otherwise, spikes won't be updated.**
- ***9*** **Saves the variable to disk**
- ***10*** **Prints out the relative file path of the saved variables**

You'll notice a couple of files generated, one of them being spikes.ckpt, in the same directory as your source code. It's a compactly stored binary file, so you can't easily modify it with a text editor. To retrieve this data, you can use the `restore` function from the `saver` op, as demonstrated in the following listing.

Listing 2.11. Loading variables

```
import tensorflow as tf
sess = tf.InteractiveSession()

spikes = tf.Variable([False]*8, name='spikes')        1
# spikes.initializer.run()                            2
saver = tf.train.Saver()                              3

saver.restore(sess, "./spikes.ckpt")                  4
print(spikes.eval())                                  5

sess.close()
```

- ***1*** **Creates a variable of the same size and name as the saved data**
- ***2*** **You no longer need to initialize this variable because it'll be directly loaded.**
- ***3*** **Creates the saver op to restore saved data**
- ***4*** **Restores data from the spikes.ckpt file**
- ***5*** **Prints the loaded data**

2.8. VISUALIZING DATA USING TENSORBOARD

In machine learning, the most time-consuming part isn't programming, but it's waiting for code to finish running. For example, a famous dataset called ImageNet contains over 14 million images prepared to be used in a machine-learning context. Sometimes it can take up to days or weeks to finish training an algorithm using a large dataset. TensorFlow's handy dashboard, TensorBoard, affords you a quick peek into the way values are changing in each node of the graph, giving you some idea of how your code is performing.

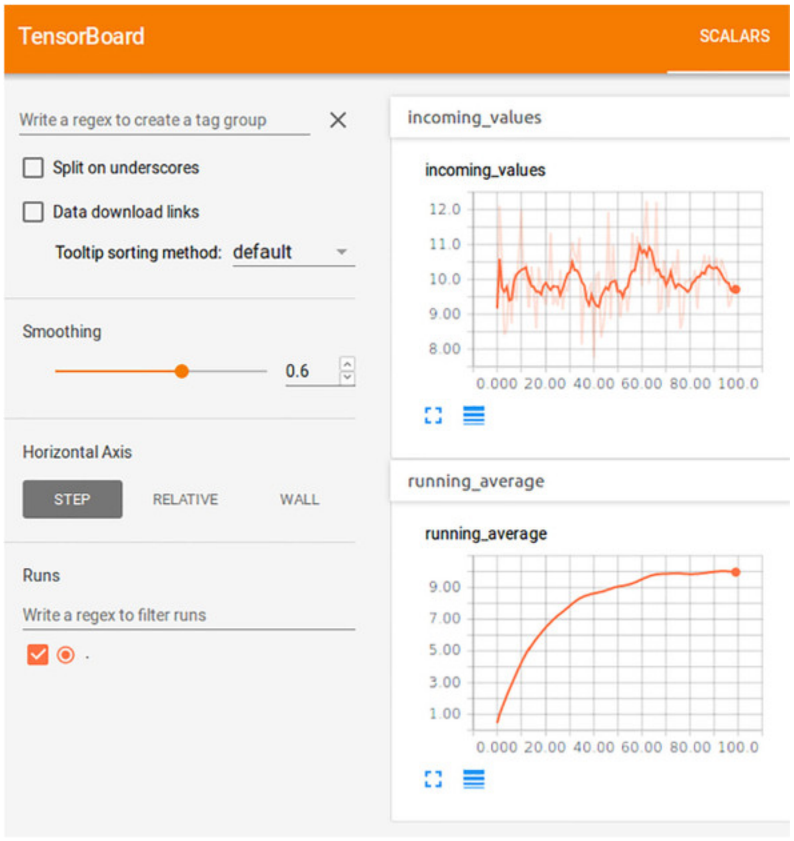
Let's see how to visualize variable trends over time in a real-world example. In this section, you'll implement a moving-average algorithm in TensorFlow, and then you'll carefully track the variables you care about for visualization in TensorBoard.

2.8.1. Implementing a moving average

In this section, you'll use TensorBoard to visualize how data changes. Suppose you're interested in calculating the average stock price of a company. Typically, computing the average is just a matter of adding up all the values and dividing by the total number seen: mean = $(x_1 + x_2 + \ldots + x_n) / n$. When the total number of values is unknown, you can use a technique called *exponential averaging* to estimate the average value of an unknown number of data points. The exponential average algorithm calculates the current estimated average as a function of the previous estimated average and the current value.

More succinctly, $Avg_t = f(Avg_{t-1}, x_t) = (1 - \alpha)\, Avg_{t-1} + \alpha\, x_t$. Alpha ($\alpha$) is a parameter that will be tuned, representing how strongly recent values should be biased in the calculation of the average. The higher the value of α, the more dramatically the calculated average will differ from the previously estimated average. Figure 2.8 (shown after listing 2.16) shows how TensorBoard visualizes the values and corresponding running average over time.

Figure 2.8. The summary display in TensorBoard created in listing 2.16. TensorBoard provides a user-friendly interface to visualize data produced in TensorFlow.



When you code this, it's a good idea to think about the main piece of computation that takes place in each iteration. In this case, each iteration will compute $Avg_t = (1 - \alpha) Avg_{t-1} + \alpha x_t$. As a result, you can design a TensorFlow operator (listing 2.12) that does exactly as the formula says. To run this code, you'll have to eventually define `alpha`, `curr_value`, and `prev_avg`.

Listing 2.12. Defining the average update operator

```
update_avg = alpha * curr_value + (1 - alpha) * prev_avg          1
```

- ***1*** **alpha is a tf.constant, curr_value is a placeholder, and prev_avg is a variable.**

You'll define the undefined variables later. The reason you're writing code in such a backward way is that defining the interface first forces you to implement the peripheral setup code to satisfy the interface. Skipping ahead, let's jump right to the session part to see how your algorithm should behave. The following listing sets up the primary loop and calls the `update_avg` operator on each iteration. Running the `update_avg` operator depends on the `curr_value`, which is fed using the `feed_dict` argument.

Listing 2.13. Running iterations of the exponential average algorithm

```
raw_data = np.random.normal(10, 1, 100)

with tf.Session() as sess:
    for i in range(len(raw_data)):
        curr_avg = sess.run(update_avg, feed_dict={curr_value:raw_data[i]}
        sess.run(tf.assign(prev_avg, curr_avg))
```

Great, the general picture is clear, because all that's left to do is to write out the undefined variables. Let's fill in the gaps and implement a working piece of TensorFlow code. Copy the following listing so you can run it.

Listing 2.14. Filling in missing code to complete the exponential average algorithm

```
import tensorflow as tf
import numpy as np

raw_data = np.random.normal(10, 1, 100)                                    1

alpha = tf.constant(0.05)                                                  2
curr_value = tf.placeholder(tf.float32)                                    3
prev_avg = tf.Variable(0.)                                                 4
update_avg = alpha * curr_value + (1 - alpha) * prev_avg

init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    for i in range(len(raw_data)):                                         5
        curr_avg = sess.run(update_avg, feed_dict={curr_value: raw_data[i]})
        sess.run(tf.assign(prev_avg, curr_avg))
        print(raw_data[i], curr_avg)
```

- ***1* Creates a vector of 100 numbers with a mean of 10 and standard deviation of 1**
- ***2* Defines alpha as a constant**
- ***3* A placeholder is just like a variable, but the value is injected from the session.**
- ***4* Initializes the previous average to zero**
- ***5* Loops through the data one by one to update the average**

2.8.2. Visualizing the moving average

Now that you have a working implementation of a moving-average algorithm, let's visualize the results by using TensorBoard. Visualization using TensorBoard is usually a two-step process:

1. Pick out which nodes you care about measuring by annotating them with a *summary op*.
2. Call `add_summary` on them to queue up data to be written to disk.

For example, let's say you have an `img` placeholder and a `cost` op, as shown in the following listing. You can annotate them (by giving each a name such as `img` or `cost`) so that they're capable of being visualized in TensorBoard. You'll do something similar with your moving-average example.

Listing 2.15. Annotating with a summary op

```
img = tf.placeholder(tf.float32, [None, None, None, 3])
cost = tf.reduce_sum(...)
```

```
my_img_summary = tf.summary.image("img", img)
my_cost_summary = tf.summary.scalar("cost", cost)
```

More generally, to communicate with TensorBoard, you must use a summary op, which produces serialized strings used by a `SummaryWriter` to save updates to a directory. Every time you call the `add_summary` method from `SummaryWriter`, TensorFlow will save data to disk for TensorBoard to use.

Warning

Be careful not to call the `add_summary` function too often! Although doing so will produce higher-resolution visualizations of your variables, it'll be at the cost of more computation and slightly slower learning.

Run the following command to make a directory called logs in the same folder as this source code:

```
$ mkdir logs
```

Run TensorBoard with the location of the logs directory passed in as an argument:

```
$ tensorboard --logdir=./logs
```

Open a browser and navigate to http://localhost:6006, which is the default URL for TensorBoard. The following listing shows how to hook up the `SummaryWriter` to your code. Run it and refresh the TensorBoard to see the visualizations.

Listing 2.16. Writing summaries to view in TensorBoard

```
import tensorflow as tf
import numpy as np

raw_data = np.random.normal(10, 1, 100)

alpha = tf.constant(0.05)
curr_value = tf.placeholder(tf.float32)
prev_avg = tf.Variable(0.)
update_avg = alpha * curr_value + (1 - alpha) * prev_avg

avg_hist = tf.summary.scalar("running_average", update_avg)           1
value_hist = tf.summary.scalar("incoming_values", curr_value)         2
merged = tf.summary.merge_all()                                       3
writer = tf.summary.FileWriter("./logs")                              4
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    sess.add_graph(sess.graph)                                        5
    for i in range(len(raw_data)):
        summary_str, curr_avg = sess.run([merged, update_avg],
            feed_dict={curr_value: raw_data[i]})                      6
        sess.run(tf.assign(prev_avg, curr_avg))
        print(raw_data[i], curr_avg)
        writer.add_summary(summary_str, i)                            7
```

- **1 Creates a summary node for the averages**
- **2 Creates a summary node for the values**
- **3 Merges the summaries to make it easier to run all at once**
- **4 Passes in the logs directory's location to the writer**
- **5 Optional, but allows you to visualize the computation graph in TensorBoard**
- **6 Runs the merged op and the update_avg op at the same time**
- **7 Adds the summary to the writer**

Tip

You may need to ensure that the TensorFlow session has ended before starting TensorBoard. If you rerun listing 2.16, you'll need to remember to clear the logs directory.

2.9. SUMMARY

- You should start thinking of mathematical algorithms in terms of a flowchart of computation. When you consider each node as an operation, and edges as data flow, writing TensorFlow code becomes trivial. After you define your graph, you evaluate it under a session, and you have your result.
- No doubt, there's more to TensorFlow than representing computations as a graph. As you'll see in the coming chapters, some of the built-in functions are tailored to the field of machine learning. In fact, TensorFlow has some of the best support for convolutional neural networks, a currently popular type of model for processing images (with promising results in audio and text as well).
- TensorBoard provides an easy way to visualize the way data changes in TensorFlow code as well as troubleshoot bugs by inspecting trends in data.
- TensorFlow works wonderfully with Jupyter notebooks, which are an elegant interactive medium for sharing and documenting Python code.