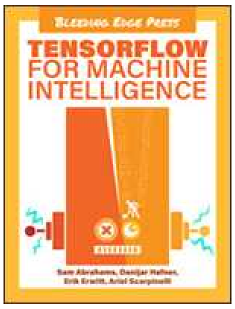


TensorFlow for Machine Intelligence: A Hands-On Introduction to Learning Algorithms

by Sam Abrahams, Danijar Hafner, Erik Erwitt and Ariel Scarpinelli
Bleeding Edge Press. (c) 2016. Copying Prohibited.

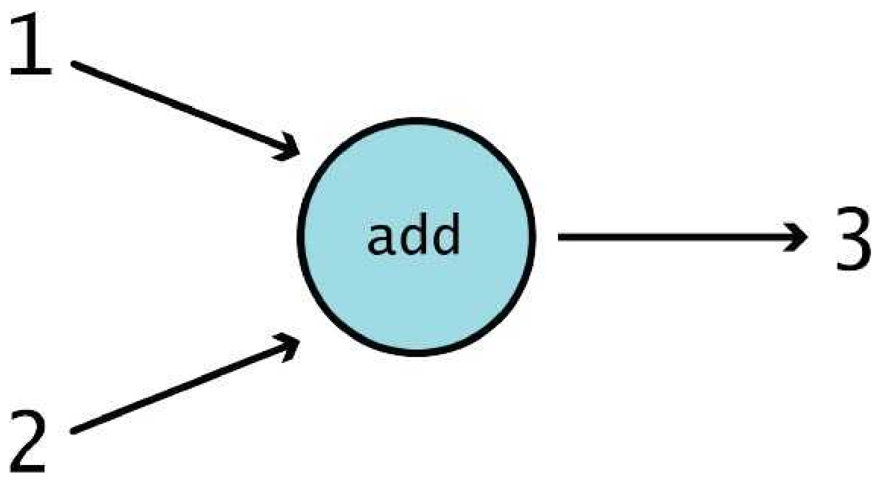
Chapter 3: TensorFlow Fundamentals

Introduction to Computation Graphs

This section covers the basics of computation graphs without the context of TensorFlow. This includes defining nodes, edges, and dependencies, and we also provide several examples to illustrate key principles. If you are experienced and/or comfortable with computation graphs, you may skip to the next section.

Graph basics

At the core of every TensorFlow program is the *computation graph* described in code with the TensorFlow API. A computation graph, is a specific type of directed graph that is used for defining, unsurprisingly, computational structure. In TensorFlow it is, in essence, a series of functions chained together, each passing its output to zero, one, or more functions further along in the chain. In this way, a user can construct a complex transformation on data by using blocks of smaller, well-understood mathematical functions. Let's take a look at a bare-bones example.



In the above example, we see the graph for basic addition. The function, represented by a circle, takes in two inputs, represented as arrows pointing into the function. It outputs the result of adding 1 and 2 together: 3, which is shown as an arrow pointing out of the function. The result could then be passed along to another function, or it might simply be returned to the client.

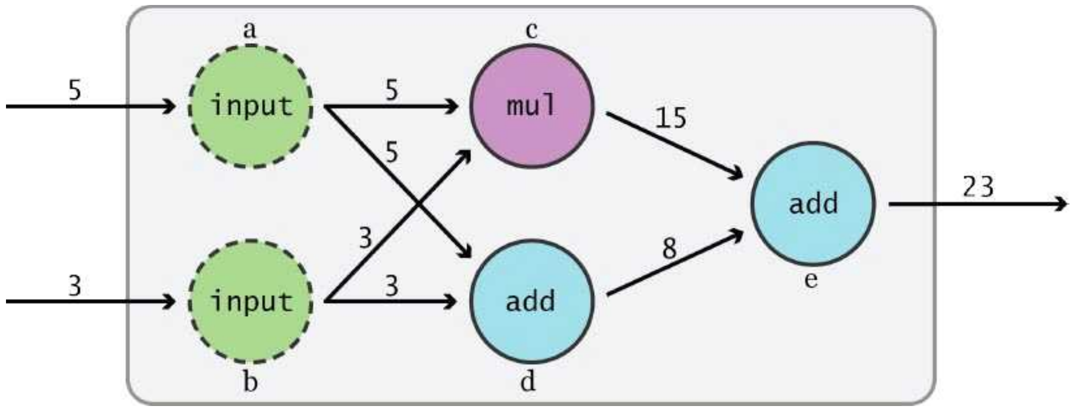
We can also look at this graph as a simple equation:

$$f(1, 2) = 1 + 2 = 3$$

The above illustrates how the two fundamental building blocks of graphs, nodes and edges, are used when constructing a computation graph. Let's go over their properties:

- **Nodes**, typically drawn as circles, ovals, or boxes, represent some sort of *computation* or *action* being done on or with data in the graph's context. In the above example, the operation "add" is the sole node.
- **Edges** are the actual values that get passed to and from Operations, and are typically drawn as arrows. In the "add" example, the inputs 1 and 2 are both edges leading into the node, while the output 3 is an edge leading out of the node. Conceptually, we can think of edges as the link between different Operations as they carry information from one node to the next.

Now, here's a slightly more interesting example:



There's a bit more going on in this graph! The data is traveling from left to right (as indicated by the direction of the arrows), so let's break down the graph, starting from the left.

1. At the very beginning, we can see two values flowing into the graph, 5 and 3. They may be coming from a different graph, being read in from a file, or entered directly by the client.
2. Each of these initial values is passed to one of two explicit "input" nodes, labeled a and b in the graphic. The "input" nodes simply pass on values given to them- node a receives the value 5 and outputs that same number to nodes c and d, while node b performs the same action with the value 3.
3. Node c is a multiplication operation. It takes in the values 5 and 3 from nodes a and b, respectively, and outputs its result of 15 to node e. Meanwhile, node d performs addition with the same input values and passes the computed value of 8 along to node e.
4. Finally, node e, the final node in our graph, is another "add" node. It receives the values of 15 and 8, adds them together, and spits out 23 as the final result of our graph.

Here's how the above graphical representation might look as a series of equations:

$$a = input_1;\ b = input_2$$

$$c = a \cdot b;\ d = a + b$$

$$e = c + d$$

If we wanted to solve *e* for *a* = 5 and *b* = 3, we can just work backwards from *e* and plug in!

$$a = 5;\ b = 3$$

$$e = (a \cdot b) + (a + b)$$

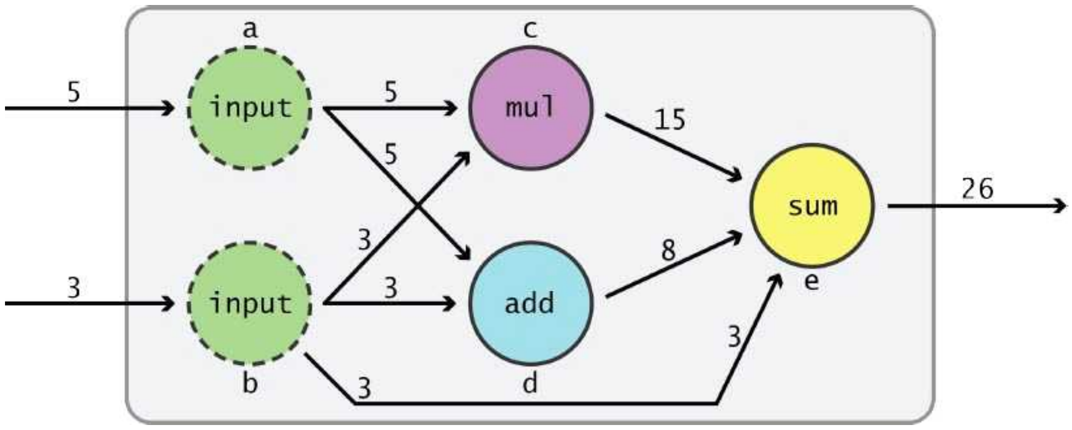
$$e = (5 \cdot 3) + (5 + 3)$$

$$e = 15 + 8 = 23$$

With that, the computation is complete! There are concepts worth pointing out here:

- The pattern of using "input" nodes is useful, as it allows us to relay a single input value to a huge amount of future nodes. If we didn't do this, the client (or whoever passed in the initial values) would have to explicitly pass each input value to multiple nodes in our graph. This way, the client only has to worry about passing in the appropriate values once and any repeated use of those inputs is abstracted away. We'll touch a little more on abstracting graphs shortly.
- Pop quiz: which node will run first- the multiplication node c, or the addition node d? The answer: you can't tell. From just this graph, it's impossible to know which of c and d will execute first. Some might read the graph from left-to-right *and* top-to-bottom and simply assume that node c would run first, but it's important to note that the graph could have easily been drawn with d on top of c. Others may think of these nodes as running concurrently, but that may not always be the case, due to various implementation details or hardware limitations. In reality, it's best to think of them as running *independently* of one another. Because node c doesn't rely on any information from node d, it doesn't have to wait for node d to do anything in order to complete its operation. The converse is also true: node d doesn't need any information from node c. We'll talk more about dependency later in this chapter.

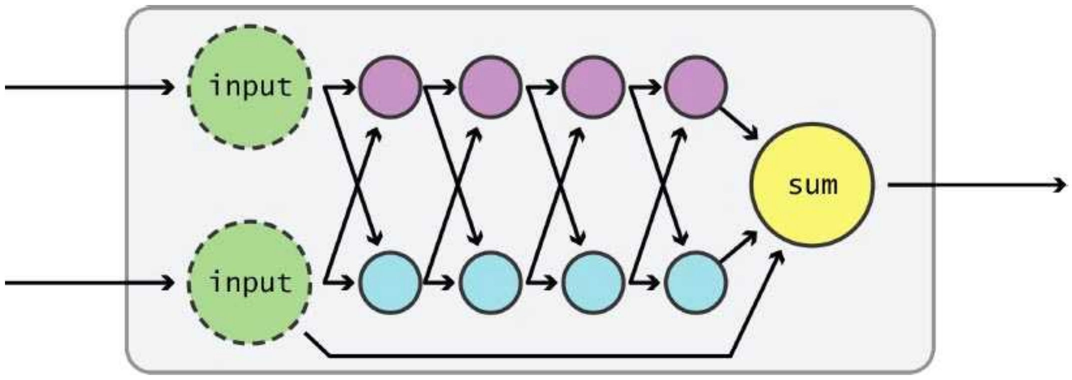
Next, here's a slightly modified version of the graph:



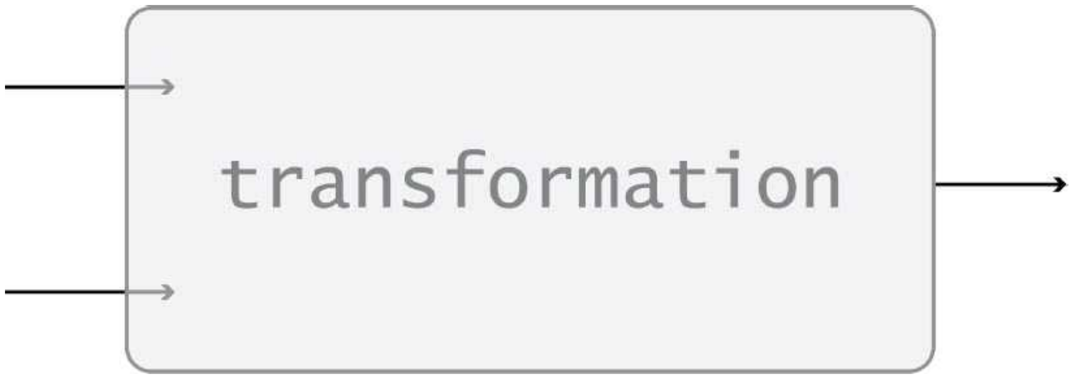
There are two main changes here:

1. The "input" value 3 from node b is now being passed on to node e.
2. The function "add" in node e has been replaced with "sum", to indicate that it adds more than two numbers.

Notice how we are able to add an edge between nodes that appear to have other nodes "in the way." In general, any node can pass its output to any future node in the graph, no matter how many computations take place in between. The graph could have looked like the following, and still be perfectly valid:



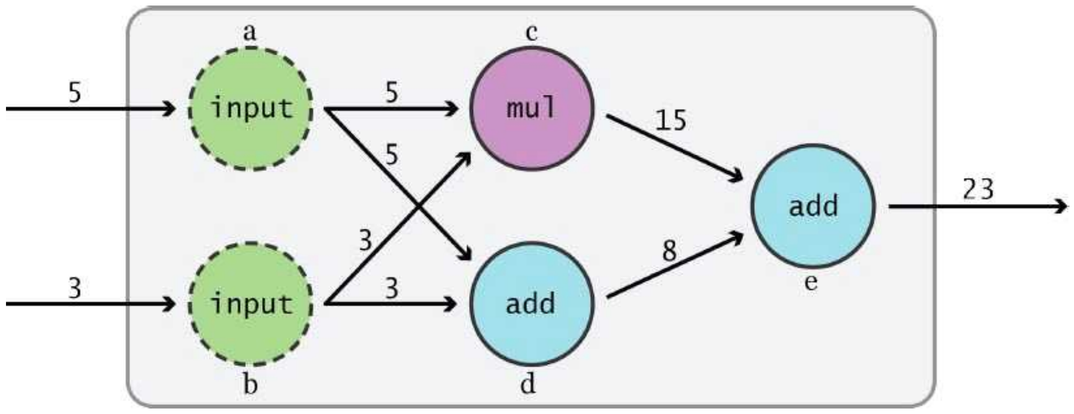
With both of these graphs, we can begin to see the benefit of abstracting the graph's input. We were able to manipulate the precise details of what's going on inside of our graph, but the client only has to know to send information to the same two input nodes. We can extend this abstraction even further, and can draw our graph like this:



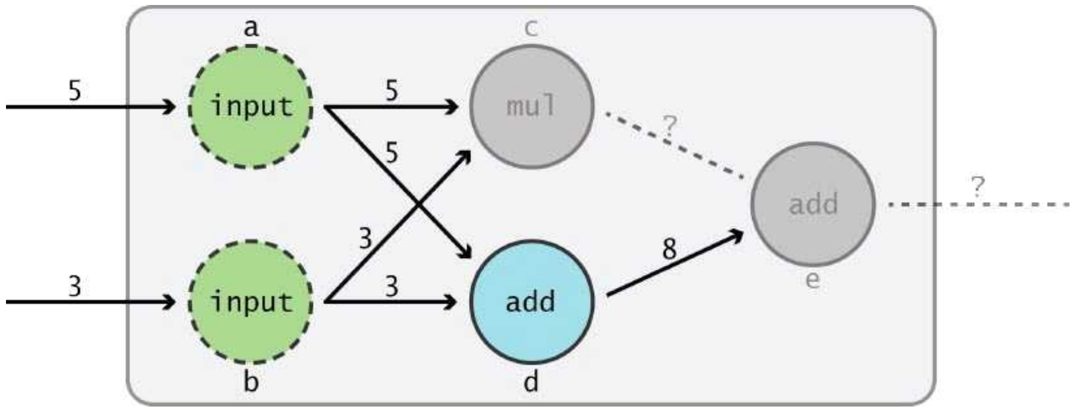
By doing this we can think of entire sequences of nodes as discrete building blocks with a set input and output. It can be easier to visualize chaining together groups of computations instead of having to worry about the specific details of each piece.

Dependencies

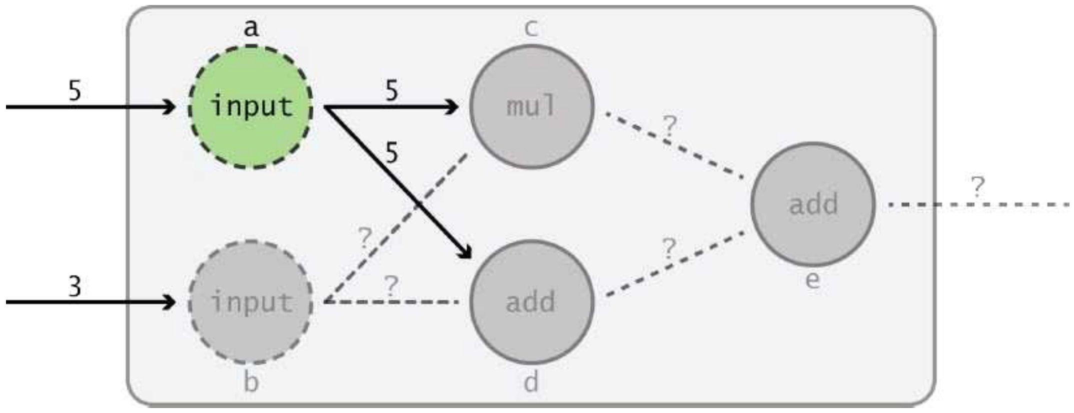
There are certain types of connections between nodes that aren't allowed, the most common of which is one that creates an unresolved *circular dependency*. In order to explain a circular dependency, we're going to illustrate what a dependency is. Let's take a look at this graph again:



The concept of a dependency is straight-forward: any node, A, that is required for the computation of a later node, B, is said to be a *dependency* of B. If a node A and node B do not need any information from one another, they are said to be *independent*. To visually represent this, let's take a look at what happens if the multiplication node c is unable to finish its computation (for whatever reason):



Predictably, since node e requires the output from node c, it is unable to perform its calculation and waits indefinitely for node c's data to arrive. It's pretty easy to see that nodes c and d are dependencies of node e, as they feed information directly into the final addition function. However, it may be slightly less obvious to see that the inputs a and b are also dependencies of e. What happens if one of the inputs fails to pass its data on to the next functions in the graph?



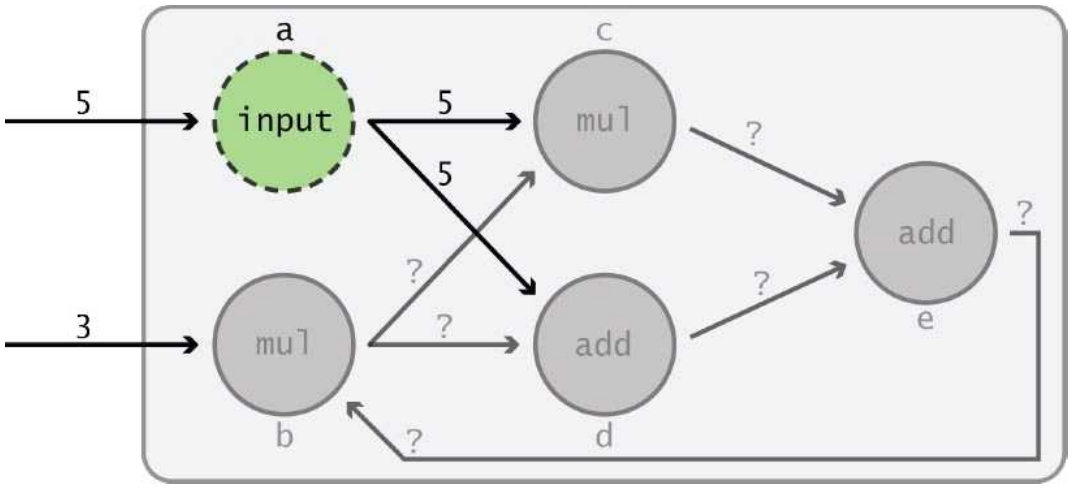
As you can see, removing one of the inputs halts most of the computation from actually occurring, and this demonstrates the *transitivity* of dependencies. That is to say, if A is dependent on B, and B is dependent on C, then A is dependent on C. In this case, the final node e is dependent on nodes c and d, and the nodes c and d are both dependent on input node b. Therefore, the final node e is dependent on the input node b. We can make the same reasoning for node e being dependent on node a, as well. Additionally, we can make a distinction between the different dependencies e has:

1. We can say that e is *directly dependent* on nodes c and d. By this, we mean that data must come *directly* from both node c and d in order for node e to execute.

2. We can say that e is *indirectly dependent* on nodes a and b. This means that the outputs of a and b do *not* feed directly into node e. Instead, their values are fed into an intermediary node(s) which is also a dependency of e, which can either be a direct dependency or

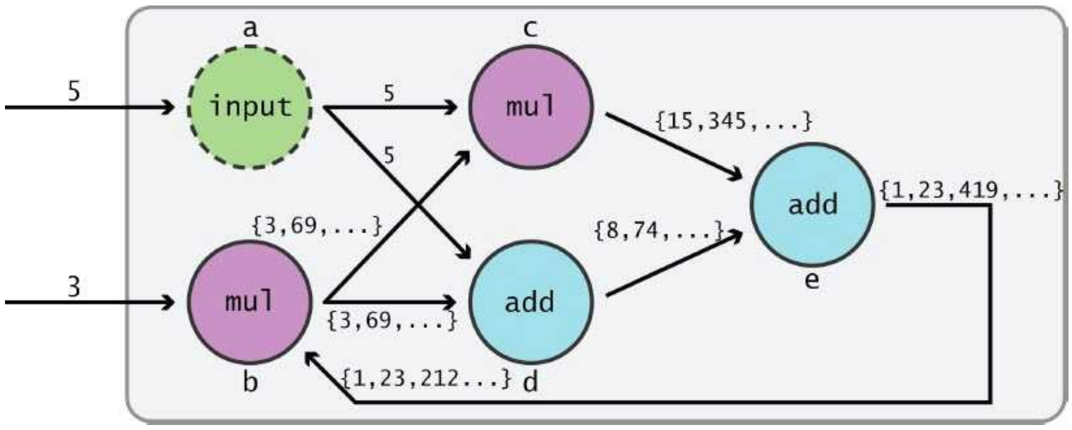
indirect dependency. This means that a node can be indirectly dependent on a node with many layers of intermediaries in-between (and each of those intermediaries is also a dependency).

Finally, let's see what happens if we redirect the output of a graph back into an earlier portion of it:



Well, unfortunately it looks like that isn't going to fly. We are now attempting to pass the output of node e back into node b and, hopefully, have the graph cycle through its computations. The problem here is that node b now has node e as a direct dependency, while at the same time, node e is dependent on node b (as we showed previously). The result of this is that neither b nor e can execute, as they are both waiting for the other node to complete its computation.

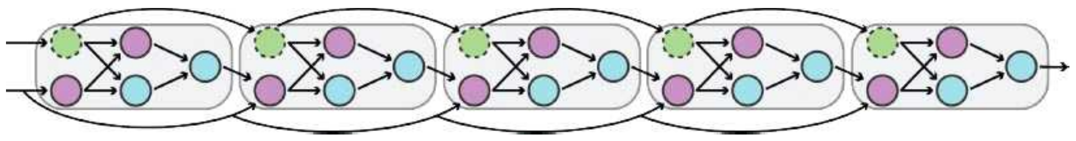
Perhaps you are clever and decide that we could provide some initial state to the value feeding into either b or e. It is our graph, after all. Let's give the graph a kick-start by giving the output of e an initial value of 1:



Here's what the first few loops through the graph look like. It creates an endless feedback loop, and most of the edges in the graph tend towards infinity. Neat! However, for software like TensorFlow, these sorts of infinite loops are bad for a number of reasons:

1. Because it's an infinite loop, the termination of the program isn't going to be graceful.
2. The number of dependencies becomes infinite, as each subsequent iteration is dependent on all previous iterations. Unfortunately, each node does not count as a single dependency- each time its output changes values it is counted again. This makes it impossible to keep track of dependency information, which is critical for a number of reasons (see the end of this section).
3. Frequently you end up in situations like this scenario, where the values being passed on either explode into huge positive numbers (where they will eventually overflow), huge negative numbers (where you will eventually underflow), or become close to zero (at which point each iteration has little additional meaning).

Because of this, truly circular dependencies can't be expressed in TensorFlow, which is not a bad thing. In practical use, we simulate these sorts of dependencies by copying a finite number of versions of the graph, placing them side-by-side, and feeding them into one another in sequence. This process is commonly referred to as "unrolling" the graph, and will be touched on more in the chapter on recurrent neural networks. To visualize what this unrolling looks like graphically, here's what the graph would look like after we've unrolled this circular dependency 5 times:



If you analyze this graph, you'll discover that this sequence of nodes and edges is identical to looping through the previous graph 5 times. Note how the original input values (represented by the arrows skipping along the top and bottom of the graph) get passed onto each copy as they are needed for each copied "iteration" through the graph. By unrolling our graph like this, we can simulate useful cyclical dependencies while maintaining a deterministic computation.

Now that we understand dependencies, we can talk about why it's useful to keep track of them. Imagine for a moment, that we only wanted to get the output of node `c` from the previous example (the multiplication node). We've already defined the entire graph, including node `d`, which is independent of `c`, and node `e`, which occurs after `c` in the graph. Would we have to calculate the entire graph, even though we don't need the values of `d` and `e`? No! Just by looking at the graph, you can see that it would be a waste of time to calculate all of the nodes if we only want the output from `c`. The question is: how do we make sure our computer only computes the necessary nodes without having to tell it by hand? The answer: use our dependencies!

The concept behind this is fairly simple, and the only thing we have to ensure is that each node has a list of the nodes it directly (*not* indirectly) depends on. We start with an empty stack, which will eventually hold all of the nodes we want to run. Start with the node(s) that you want to get the output from. Obviously it must execute, so we add it to our stack. We look at our output node's list of dependencies-which means that *those* nodes must run in order to calculate our output, so we add all of them to the stack. Now we look at all of those nodes and see what *their* direct dependencies are and add *those* to the stack. We continue this pattern all the way back in the graph until there are no dependencies left to run, and in this way we guarantee that we have all of the nodes we need to run the graph, and *only* those nodes. In addition, the stack will be ordered in a way that we are guaranteed to be able to run each node in the stack as we iterate through it. The main thing to look out for is to keep track of nodes that were already calculated and to store their value in memory- that way we don't calculate the same node over and over again. By doing this, we are able to make sure our computation is as lean as possible, which can save hours of processing time on huge graphs.

Defining Computation Graphs in TensorFlow

In this book, you're going to be exposed to diverse and fairly complex machine learning models. However, the process of defining each of them in TensorFlow follows a similar pattern. As you dive into various math concepts and learn how to implement them, understanding the core TensorFlow work pattern will keep yourself grounded. Luckily, this workflow is simple to remember- it's only two steps:

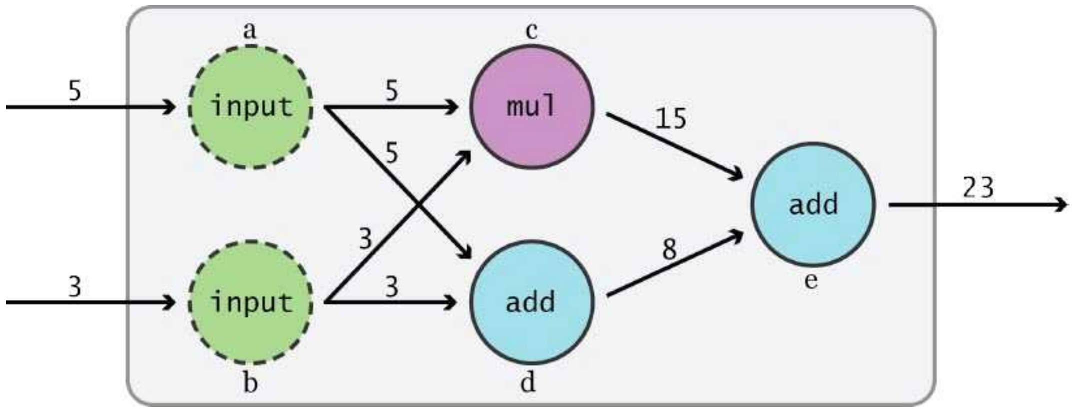
1. *Define* the computation graph
2. *Run* the graph (with data)

This seems obvious- you can't run a graph if it doesn't exist yet! But it's an important distinction to make as the sheer volume of functionality in TensorFlow can be overwhelming when writing your own code. By worrying about only one portion of this workflow at a time, it can help you structure your code more thoughtfully as well as aide in pointing you towards the next thing to work on.

This section will focus on the basics of *defining* graphs in TensorFlow, and the next section will go over *running* a graph once its created. At the end, we'll tie the two together, and show how we can create graphs that change over multiple runs and take in different data.

Building your first TensorFlow graph

We became pretty familiar with the following graph in the last section:



Here's what it looks like in TensorFlow code:

```
import tensorflow as tf
```

```
a = tf.constant(5, name="input_a")
b = tf.constant(3, name="input_b")
c = tf.mul(a,b, name="mul_c")
d = tf.add(a,b, name="add_d")
e = tf.add(c,d, name="add_e")
```

Let's break this code down line by line. First, you'll notice this import statement:

```
import tensorflow as tf
```

This, unsurprisingly, imports the TensorFlow library and gives it an alias of `tf`. This is by convention, as it's much easer to type "tf," rather than "tensorflow" over and over as we use its various functions!

Next, let's focus on our first two variable assignments:

```
a = tf.constant(5, name="input_a")
b = tf.constant(3, name="input_b")
```

Here, we're defining our "input" nodes, a and b. These lines use our first TensorFlow Operation: **tf.constant()**. In TensorFlow, any computation node in the graph is called an **Operation**, or **Op** for short. Ops take in zero or more `Tensor` objects as input and output zero or more `Tensor` objects. To create an Operation, you call its associated Python constructor- in this case, **tf.constant()** creates a "constant" Op. It takes in a single tensor value, and outputs that same value to nodes that are directly connected to it. For convenience, the function automatically converts the scalar numbers 5 and 3 into `Tensor` objects for us. We also pass in an optional string `name` parameter, which we can use to give an identifier to the nodes we create.

Don't worry if you don't fully understand what an Operation or `Tensor` object are at this time, since we'll be going into more detail later in this chapter.

```
c = tf.mul(a,b, name="mul_c")
d = tf.add(a,b, name="add_d")
```

Here, we are defining the next two nodes in our graph, and they both use the nodes we defined previously. Node c uses the **tf.mul**. Op, which takes in two inputs and outputs the result of multiplying them together. Similarly, node d uses **tf.add**, an Operation that outputs the result of adding two inputs together. We again pass in a `name` to both of these Ops (it's something you'll be seeing a lot of). Notice that we don't have to define the edges of the graph separately from the node- when you create a node in TensorFlow, you include all of the inputs that the Operation needs to compute, and the software draws the connections for you.

```
e = tf.add(c,d, name="add_e")
```

This last line defines the final node in our graph. e uses **tf.add** in a similar fashion to node d. However, this time it takes nodes c and d as input- exactly as its described in the graph above.

With that, our first, albeit small, graph has been fully defined! If you were to execute the above in a Python script or shell, it would run, but it wouldn't actually do anything. Remember- this is just the *definition* part of the process. To get a brief taste of what running a graph looks like, we could add the following two lines at the end to get our graph to output the final node:

```
sess = tf.Session()
sess.run(e)
```

If you ran this in an interactive environment, such as the `python` shell or the Jupyter/iPython Notebook, you would see the correct output:

```
...
>>> sess = tf.Session()
>>> sess.run(e)
23
```

That's enough talk for now: let's actually get this running in live code!

Exercise: Building a Basic Graph in TensorFlow

It's time to do it live! In this exercise, you'll code your first TensorFlow graph, run various parts of it, and get your first exposure to the incredibly useful tool **TensorBoard**. When you finish this, you should feel comfortable experimenting with and building basic TensorFlow graphs.

Now, let's actually define it in TensorFlow! Make sure you have TensorFlow installed, and start up your Python dependency environment (Virtualenv, Conda, Docker) if you're using one. In addition, if you installed TensorFlow from source, make sure that your console's present working directory is *not* the TensorFlow source folder, otherwise Python will get confused when we import the library. Now, start an interactive Python session, either using the Jupyter Notebook with the shell command `jupyter notebook`, or start a simple Python shell with `python`. If you have another preferred way of writing Python interactively, feel free to use that!

You could write this as a Python file and run it non-interactively, but the output of running a graph is not displayed by default when doing so. For the sake of seeing the result of your graph, getting immediate feedback on your syntax, and (in the case of the Jupyter Notebook) the ability to fix errors and change code on the fly, we highly recommend doing these examples in an

> interactive environment. Plus, interactive TensorFlow is fun!

First, we need to load up the TensorFlow library. Write out your import statement as follows:

```
import tensorflow as tf
```

It may think for a few seconds, but afterward it will finish importing and will be ready for the next line of code. If you installed TensorFlow with GPU support, you may see some output notifying you that CUDA libraries were imported. If you get an error that looks like this:

```
ImportError: cannot import name pywrap_tensorflow
```

Make sure that you didn't launch your interactive environment from the TensorFlow source folder. If you get an error that looks like this:

```
ImportError: No module named tensorflow
```

Double check that TensorFlow is installed properly. If you are using Virtualenv or Conda, ensure that your TensorFlow environment it is active when you start your interactive Python software. Note that if you have multiple terminals running, one terminal may have an environment active while the other does not.

Assuming the import worked without any hiccups, we can move on to the next portion of the code:

```
a = tf.constant(5, name="input_a")
b = tf.constant(3, name="input_b")
```

This is the same code that we saw above- feel free to change the values or `name` parameters of these constants. In this book, we'll stick to the same values we had for the sake of consistency.

```
c = tf.mul(a,b, name="mul_c")
d = tf.add(a,b, name="add_d")
```

Next up, we have the first Ops in our code that actually perform a mathematical function. If you're sick and tired of **tf.mul** and **tf.add**, feel free to swap in **tf.sub**, **tf.div**, or **tf.mod**, which perform subtraction, division, or modulo operations, respectively.

> **tf.div** performs either integer division or floating point division depending on the type of input provided. If you want to ensure floating point division, try out **tf.truediv**!

Then we can add in our final node:

```
e = tf.add(c,d, name="add_e")
```

You probably noticed that there hasn't been any output when calling these Operations. That's because they have been simply adding Ops to a graph behind the scenes, but no computation is actually taking place. In order to run the graph, we're going to need a TensorFlow `Session`:

```
sess = tf.Session()
```

`Session` objects are in charge of supervising graphs as they run, and are the primary interface for running graphs. We're going to discuss `Session` objects in depth after this exercise, but for now just know that in TensorFlow you need a `Session` if you want to run your code! We assign our `Session` to the variable `sess` so we can access it later.

> On `InteractiveSession`: There is a slight variation on `tf.Session` called `tf.InteractiveSession`. It's actually designed for use in interactive Python software, such as those you may be using, and it makes a few alternative ways of running code a little simpler. The downsides are that it's less useful for writing TensorFlow in a Python file, and that it abstracts away information that you should learn as a new user to TensorFlow. Besides, in the end it doesn't save *that* many keystrokes. In this book, we'll stick to the standard `tf.Session`

```
sess.run(e)
```

Here's where we finally can see the result! After running this code, you should see the output of your graph. In our example graph, the output was 23, but it will be different depending the exact functions and inputs you used. That's not all we can do however. Let's try plugging in one of the other nodes in our graph to `sess.run()`:

```
sess.run(c)
```

You should see the intermediary value of `c` as the output of this call (15, in the example code). TensorFlow doesn't make any assumptions about graphs you create, and for all the program cares node `c` could be the output you want! In fact, you can use the `run()` on any Operation in your graph. When you pass an Op into `sess.run()`, what you are essentially saying to TensorFlow is, "Here is a node I would like to output. Please run all operations necessary to calculate that node". Play around and try outputting some of the other nodes in your graph!

You can also save the output from running the graph- let's save the output from node `e` to a Python variable called `output`:

```
output = sess.run(e)
```

Great! Now that we have a `Session` active and our graph defined, let's visualize it to confirm that it's structured the same way we drew it out. To do that, we're going to use **TensorBoard**, which came installed with TensorFlow. To take advantage of TensorBoard, we're just going to add one line to our code:

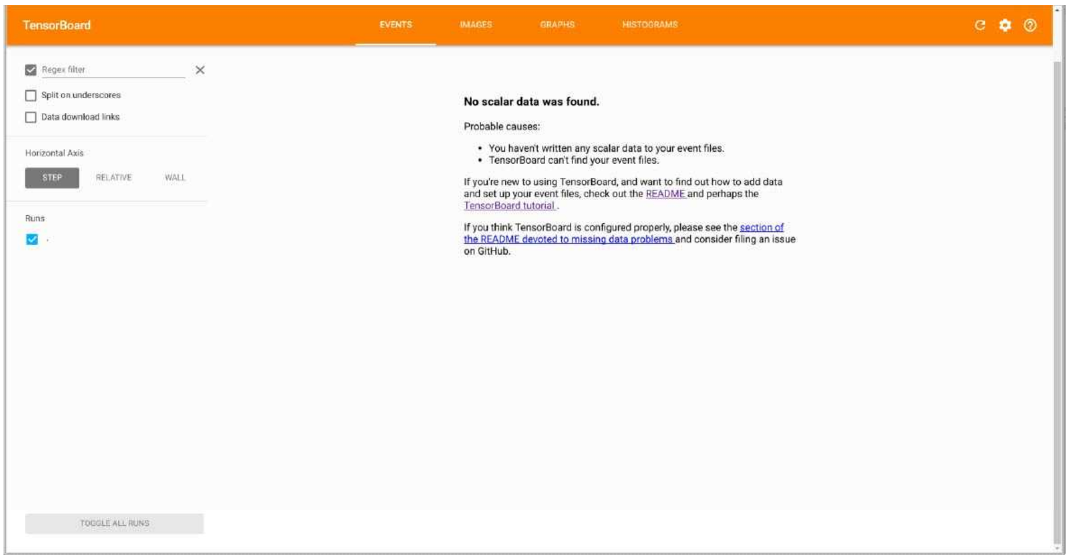
```
writer = tf.train.SummaryWriter('./my_graph', sess.graph)
```

Let's break down what this code does. We are creating a TensorFlow **SummaryWriter** object, and assigning it to the variable `writer`. In this exercise, we won't be performing any additional actions with the `SummaryWriter`, but in the future we'll be using them to save data and summary statistics from our graphs, so we assign it to a variable to get in the habit. We pass in two parameters to initialize `SummaryWriter`. The first is a string output directory, which is where the graph description will be stored on disk. In this case, the files created will be put in a directory called `my_graph`, and will be located inside the directory we are running our Python code. The second input we pass into `SummaryWriter` is the `graph` attribute of our `Session`. `tf.Session` objects, as managers of graphs defined in TensorFlow, have a `graph` attribute that is a reference to the graph they are keeping track of. By passing this on to `SummaryWriter`, the writer will output a description of the graph inside the "my_graph" directory. `SummaryWriter` objects write this data immediately upon initialization, so once you have executed this line of code, we can start up TensorBoard.

Go to your terminal and type in the following command, making sure that your present working directory is the same as where you ran your Python code (you should see the "my_graph" directory listed):

```
$ tensorboard --logdir="my_graph"
```

You should see some log info print to the console, and then the message "Starting TensorBoard on port 6006". What you've done is start up a TensorBoard server that is using data from the "my_graph" directory. By default, the server started on port 6006- to access TensorBoard, open up a browser and type `http://localhost:6006`. You'll be greeting with an orange-and-white-themed screen:



No scalar data was found.

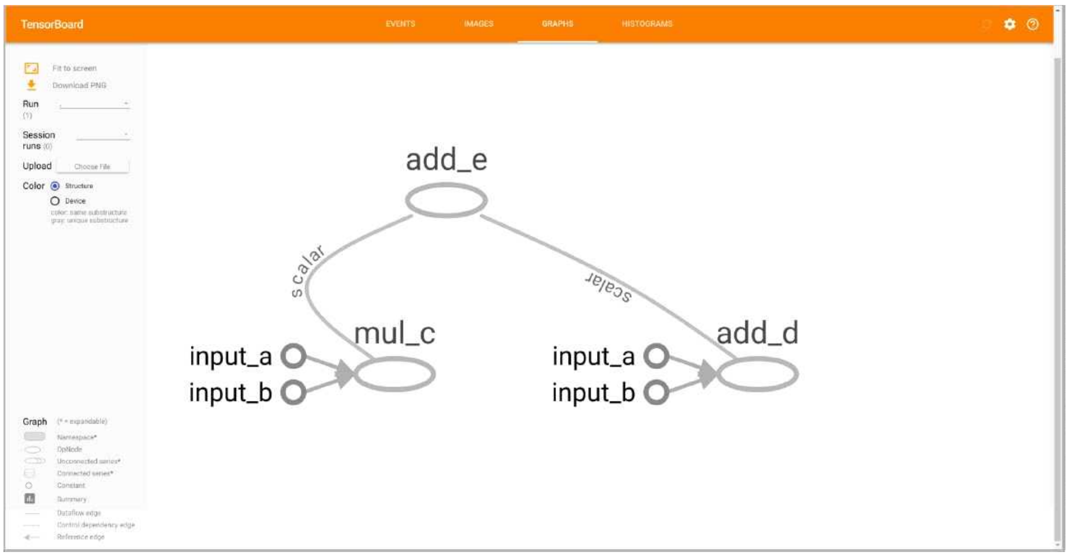
Probable causes:

- You haven't written any scalar data to your event files.
- TensorBoard can't find your event files.

If you're new to using TensorBoard, and want to find out how to add data and set up your event files, check out the README and perhaps the TensorBoard tutorial.

If you think TensorBoard is configured properly, please see the section of the README devoted to missing data problems and consider filing an issue on GitHub.

Don't be alarmed by the "No scalar data was found" warning message. That just means that we didn't save out any summary statistics for TensorBoard to display- normally, this screen would show us information that we asked TensorFlow to save using our `SummaryWriter`. Since we didn't write any additional stats, there's nothing to display. That's fine, though, as we're here to admire our beautiful graph. Click on the "Graphs" link at the top of the page, and you should see a screen similar to this:



That's more like it! If your graph is too small, you can zoom in on TensorBoard by scrolling your mousewheel up. You can see how each of the nodes is labeled based on the `name` parameter we passed into each Operation. If you click on the nodes, you can get information about them such as which other nodes they are attached to. You'll notice that the "inputs", `a` and `b` appear to be duplicated, but if you hover or click on either of the nodes labeled "input_a", you should see that they both get a highlighted together. This graph doesn't *look* exactly like the graph we drew above, but it is the same graph since the "input" nodes are simply shown twice. Pretty awesome!

And that's it! You've officially written and run your first ever TensorFlow graph, and you've checked it out in TensorBoard! Not bad for a few lines of code!

For more practice, try adding in a few more nodes, experimenting with some of the different math Ops talked about and adding in a few more `tf.constant` nodes. Run the different nodes you've added and make sure you understand exactly how data is moving through the graph.

Once you are done constructing your graph, let's be tidy and close the `Session` and `SummaryWriter`:

```
writer.close()
sess.close()
```

Technically, `Session` objects close automatically when the program terminates (or, in the interactive case, when you close/restart the Python kernel). However, it's best to explicitly close out of the `Session` to avoid any sort of weird edge case scenarios.

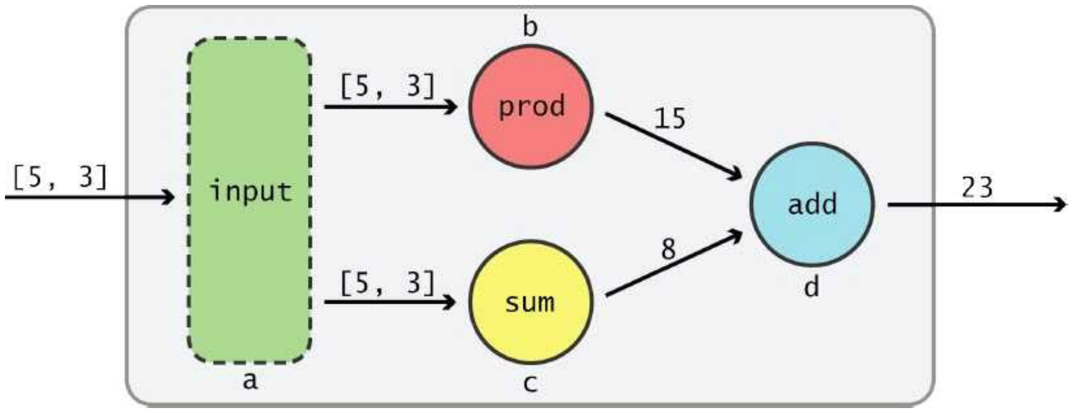
Here's the full Python code after going through this tutorial with our example values:

```
import tensorflow as tf

a = tf.constant(5, name="input_a")
b = tf.constant(3, name="input_b")
c = tf.mul(a,b, name="mul_c")
d = tf.add(a,b, name="add_d")
e = tf.add(c,d, name="add_e")

sess = tf.Session()
output = sess.run(e)
writer = tf.train.SummaryWriter('./my_graph', sess.graph)

writer.close()
sess.close()
```

Thinking with tensors

Simple, scalar numbers are great when learning the basics of computation graphs, but now that we have a grasp of the "flow", let's get acquainted with tensors.

Tensors, as mentioned before, are simply the n-dimensional abstraction of matrices. So a 1-D tensor would be equivalent to a vector, a 2-D tensor is a matrix, and above that you can just say "N-D tensor". With this in mind, we can modify our previous example graph to use tensors:



Now, instead of having two separate input nodes, we have a single node that can take in a vector (or 1-D tensor) of numbers. This graph has several advantages over our previous example:

1. The client only has to send input to a single node, which simplifies using the graph.
2. The nodes that directly depend on the input now only have to keep track of one dependency instead of two.
3. We now have the option of making the graph take in vectors of any length, if we'd like. This would make the graph more flexible. We can

also have the graph enforce a strict requirement, and force inputs to be of length two (or any length we'd like)

We can implement this change in TensorFlow by modifying our previous code:

```
import tensorflow as tf

a = tf.constant([5,3], name="input_a")
b = tf.reduce_prod(a, name="prod_b")
c = tf.reduce_sum(a, name="sum_c")
d = tf.add(b,c, name="add_d")
```

Aside from adjusting the variable names, we made two main changes here:

1. We replaced the separate nodes a and b with a consolidated input node (now just a). We passed in a list of numbers, which `tf.constant` is able to convert to a 1-D `Tensor`
2. Our multiplication and addition Operations, which used to take in scalar values, are now **tf.reduce_prod()** and **tf.reduce_sum()**. These functions, when just given a `Tensor` as input, take all of its values and either multiply or sum them up, respectively.

In TensorFlow, all data passed from node to node are `Tensor` objects. As we've seen, TensorFlow Operations are able to look at standard Python types, such as integers and strings, and automatically convert them into tensors. There are a variety of ways to create `Tensor` objects manually (that is, without reading it in from an external data source), so let's go over a few of them.

In this book, when discussing code we will use "tensor" and "`Tensor`" interchangeably.

PYTHON NATIVE TYPES

TensorFlow can take in Python numbers, booleans, strings, or lists of any of the above. Single values will be converted to a 0-D `Tensor` (or scalar), lists of values will be converted to a 1-D `Tensor` (vector), lists of lists of values will be converted to a 2-D `Tensor` (matrix), and so on. Here's a small chart showcasing this:

```
t_0 = 50                                # Treated as 0-D Tensor, or "scalar"

t_1 = [b"apple", b"peach", b"grape"] # Treated as 1-D Tensor, or "vector"

t_2 = [[True, False, False],            # Treated as 2-D Tensor, or "matrix"
       [False, False, True],
       [False, True, False]]

t_3 = [[ [0, 0], [0, 1], [0, 2] ],     # Treated as 3-D Tensor
       [ [1, 0], [1, 1], [1, 2] ],
       [ [2, 0], [2, 1], [2, 2] ]]
...
```

TensorFlow Data Types

We haven't seen booleans or strings yet, but you can think of tensors as a way to store any data in a structured format. Obviously, math functions don't work on strings, and string-parsing functions don't work on numbers, but it's good to know that TensorFlow can handle more than just numerics! Here's the **full list of data types available in TensorFlow**:

Data type (dtype)	Description
`tf.float32`	32-bit floating point
`tf.float64`	64-bit floating point
`tf.int8`	8-bit signed integer
`tf.int16`	16-bit signed integer
`tf.int32`	32-bit signed integer
`tf.int64`	64-bit signed integer
`tf.uint8`	8-bit unsigned integer
`tf.string`	String (as bytes array, *not* Unicode)
`tf.bool`	Boolean
`tf.complex64`	Complex number, with 32-bit floating point real portion, and 32-bit floating point imaginary portion
`tf.qint8`	8-bit signed integer (used in quantized Operations)
`tf.qint32`	32-bit signed integer (used in quantized Operations)
`tf.quint8`	8-bit unsigned integer (used in quantized Operations)

Using Python types to specify `Tensor` objects is quick and easy, and it is useful for prototyping ideas. However, there is an important and unfortunate downside to doing it this way. TensorFlow has a plethora of data types at its disposal, but basic Python types lack the ability to explicitly state what kind of data type you'd like to use. Instead, TensorFlow has to infer which data type you meant. With some types, such as strings, this is simple, but for others it may be impossible. For example, in Python all integers **are the same type**, but TensorFlow has 8-bit, 16-bit, 32-bit, and 64-bit integers available. There are ways to convert the data into the appropriate type when you pass it into TensorFlow, but certain data types still may be difficult to declare correctly, such as complex numbers. Because of this, it is common to see hand-defined `Tensor` objects as **NumPy** arrays.

NUMPY ARRAYS

TensorFlow is tightly integrated with **NumPy**, the scientific computing package designed for manipulating N-dimensional arrays. If you don't have experience with NumPy, we highly recommend looking at the wealth of tutorials and documentation available for the library, as it has become part of the *lingua franca* of data science. TensorFlow's data types are based on **those from NumPy**; in fact, the statement `np.int32 == tf.int32` returns `True`! Any NumPy array can be passed into any TensorFlow Op, and the beauty is that you can easily specify the data type you need with minimal effort.

String Data Types

There is a "gotcha" here for string data types. For numeric and boolean types, TensorFlow and NumPy dtypes match down the line. However, `tf.string` does not have an exact match in NumPy due to the way NumPy handles strings. That said, TensorFlow can import string arrays from NumPy perfectly fine- just don't specify a `dtype` in NumPy!

As a bonus, you can use the functionality of the `numpy` library both before and after running your graph, as the tensors returned from `Session.run` *are* NumPy arrays. Here's an example of how to create NumPy arrays, mirroring the above example.

```
import numpy as np   # Don't forget to import NumPy!

# 0-D Tensor with 32-bit integer data type
t_0 = np.array(50, dtype=np.int32)

# 1-D Tensor with byte string data type
# Note: don't explicitly specify dtype when using strings in NumPy
t_1 = np.array([b"apple", b"peach", b"grape"])

# 1-D Tensor with boolean data type
t_2 = np.array([[True, False, False],
                [False, False, True],
                [False, True, False]],
                dtype=np.bool)

# 3-D Tensor with 64-bit integer data type
t_3 = np.array([[ [0, 0], [0, 1], [0, 2] ],
                [ [1, 0], [1, 1], [1, 2] ],
                [ [2, 0], [2, 1], [2, 2] ]],
                dtype=np.int64)
...
```

Although TensorFlow is designed to understand NumPy data types natively, the converse is not true. Don't accidentally try to initialize a NumPy array with `tf.int32`!

[FOOTNOTE] Technically, NumPy is able to automatically detect data types as well, but it really is best to start getting in the habit of being explicit about the numeric properties you want your `Tensor` objects to have. When you're dealing with huge graphs, you *really* don't want to have to hunt down which objects are causing a `TypeMismatchError`! The one exception to this is when dealing with strings- don't bother specifying a `dtype` when creating a string `Tensor`.

Using NumPy is the recommended way of specifying `Tensor` objects by hand!

Tensor shape

Throughout the TensorFlow library, you'll commonly see functions and Operations that refer to a tensor's "shape". The shape, in TensorFlow terminology, describes both the number dimensions in a tensor as well as the length of each dimension. Tensor shapes can either be Python lists or tuples containing an ordered set of integers: there are as many numbers in the list as there are dimensions, and each number describes the length of its corresponding dimension. For example, the list `[2, 3]` describes the shape of a 2-D tensor of length 2 in its first dimension and length 3 in its second dimension. Note that either tuples (wrapped with parentheses `()`) or lists (wraped with brackets `[]`) can be used to define shapes. Let's take a look at more examples to illustrate this further:

```
# Shapes that specify a 0-D Tensor (scalar)
# e.g. any single number: 7, 1, 3, 4, etc.
```

```
s_0_list = []
s_0_tuple = ()

# Shape that describes a vector of length 3
# e.g. [1, 2, 3]
s_1 = [3]

# Shape that describes a 3-by-2 matrix
# e.g [[1 ,2],
#      [3, 4],
#      [5, 6]]
s_2 = (3, 2)
```

In addition to being able to specify fixed lengths to each dimension, you are also able assign a flexible length by passing in `None` as a dimension's value. Furthermore, passing in the value `None` as a shape (instead of using a list/tuple that contains `None`), will tell TensorFlow to allow a tensor of any shape. That is, a tensor with any amount of dimensions and any length for each dimension:

```
# Shape for a vector of any length:
s_1_flex = [None]

# Shape for a matrix that is any amount of rows tall, and 3 columns wide:
s_2_flex = (None, 3)

# Shape of a 3-D Tensor with length 2 in its first dimension, and variable-
# length in its second and third dimensions:
s_3_flex = [2, None, None]

# Shape that could be any Tensor
s_any = None
```

If you ever need to figure out the shape of a tensor in the middle of your graph, you can use the `tf.shape` Op. It simply takes in the Tensor object you'd like to find the shape for, and returns it as an `int32` vector:

```
import tensorflow as tf

# ...create some sort of mystery tensor

# Find the shape of the mystery tensor
shape = tf.shape(mystery_tensor, name="mystery_shape")
```

Remember that `tf.shape`, like any other Operation, doesn't run until it is executed inside of a Session.

Reminder!

Tensors are just a superset of matrices!

TensorFlow operations

As mentioned earlier, **TensorFlow Operations**, also known as **Ops**, are nodes that perform computations on or with Tensor objects. After computation, they return zero or more tensors, which can be used by other Ops later in the graph. To create an Operation, you call its constructor in Python, which takes in whatever Tensor parameters needed for its calculation, known as *inputs*, as well as any additional information needed to properly create the Op, known as *attributes*. The Python constructor returns a handle to the Operation's *output* (zero or more `Tensor` objects), and it is this output which can be passed on to other Operations or `Session.run`:

```
import tensorflow as tf
import numpy as np

# Initialize some tensors to use in computation
a = np.array([2, 3], dtype=np.int32)
b = np.array([4, 5], dtype=np.int32)

# Use `tf.add()` to initialize an "add" Operation
# The variable `c` will be a handle to the Tensor output of this Op
c = tf.add(a, b)
```

Zero-input, Zero-output Operations

Yes, that means there are Ops that technically take in zero inputs and return zero outputs. Ops are more than just mathematical computations, and are used for tasks such as initializing state. We'll be going over some of these non-mathematical Operations in this chapter, but for now just remember that not all nodes need to be connected to other nodes.

In addition to *inputs* and *attributes*, each Operation constructor accepts a string parameter, `name`, as input. As we saw in the exercise above, providing a `name` allows us to refer to a specific Op by a descriptive string:

```
c = tf.add(a, b, name="my_add_op")
```

In this example, we give the name "my_add_op" to the add Operation, which we'll be able to refer to when using tools such as TensorBoard.

You may find that you'll want to reuse the same `name` for different Operations in a graph. Instead of manually adding prefixes or suffixes to each `name`, you can use a `name_scope` to group operations together programmatically. We'll go over the basic use of name scopes in the exercise at the end of this chapter.

Overloaded Operators

TensorFlow also overloads common mathematical operators to make multiplication, addition, subtraction, and other common operations more concise. If one or more arguments to the operator is a `Tensor` object, a TensorFlow Operation will be called and added to the graph. For example, you can easily add two tensors together like this:

```
# Assume that `a` and `b` are `Tensor` objects with matching shapes
c = a + b
```

Here is a complete list of overloaded operators for tensors:

Unary operators

Operator	Related TensorFlow Operation	Description
-x	**tf.neg()**	Returns the negative value of each element in x
~x	**tf.logical_not()**	Returns the logical NOT of each element in x. Only compatible with `Tensor` objects with `dtype` of `tf.bool`
abs(x)	**tf.abs()**	Returns the absolute value of each element in x

Binary operators

Operator	Related TensorFlow Operation	Description
x + y	**tf.add()**	Add x and y, element-wise
x - y	**tf.sub()**	Subtract y from x, element-wise
x * y	**tf.mul()**	Multiply x and y, element-wise
x / y (Python 2)	**tf.div()**	Will perform element-wise integer division when given an integer type tensor, and floating point ("true") division on floating point tensors
x / y (Python 3)	**tf.truediv()**	Element-wise floating point division (including on integers)
x // y (Python 3)	**tf.floordiv()**	Element-wise floor division, not returning any remainder from the computation
x % y	**tf.mod()**	Element-wise modulo
x ** y	**tf.pow()**	The result of raising each element in x to its corresponding element y, element-wise
x < y	**tf.less()**	Returns the truth table of x < y, element-wise
x <= y	**tf.less_equal()**	Returns the truth table of x `<= y`, element-wise
x > y	**tf.greater()**	Returns the truth table of x > y, element-wise
x >= y	**tf.greater_equal()**	Returns the truth table of x >= y, element-wise
x & y	**tf.logical_and()**	Returns the truth table of x & y, element-wise. `dtype` must be `tf.bool`
x | y	**tf.logical_or()**	Returns the truth table of x | y, element-wise. `dtype` must be `tf.bool`
x ^ y	**tf.logical_xor()**	Returns the truth table of x ^ y, element-wise. `dtype` must be `tf.bool`

Using these overloaded operators can be great when quickly putting together code, but you will not be able to give `name` values to each of these Operations. If you need to pass in a `name` to the Op, call the TensorFlow Operation directly.

Technically, the == operator is overloaded as well, but it will not return a `Tensor` of boolean values. Instead, it will return `True` if the two tensors being compared are the same object, and `False` otherwise. This is mainly used for internal purposes. If you'd like to check for equality or inequality, check out **tf.equal()** and **tf.not_equal**, respectively.

TensorFlow graphs

Thus far, we've only referenced "the graph" as some sort of abstract, omni-presence in TensorFlow, and we haven't questioned how Operations are automatically attached to a graph when we start coding. Now that we've seen some examples, let's take a look at the **TensorFlow Graph object**, learn how to create more of them, use multiple graphs in conjunction with one another.

Creating a `Graph` is simple- its constructor doesn't take any variables:

```
import tensorflow as tf

# Create a new graph:
g = tf.Graph()
```

Once we have our `Graph` initialized, we can add Operations to it by using the `Graph.as_default()` method to access its context manager. In conjunction with the `with` statement, we can use the context manager to let TensorFlow know that we want to add Operations to a specific `Graph`:

```
with g.as_default():
    # Create Operations as usual; they will be added to graph `g`
    a = tf.mul(2, 3)
    ...
```

You might be wondering why we haven't needed to specify the graph we'd like to add our Ops to in the previous examples. As a convenience, TensorFlow automatically creates a `Graph` when the library is loaded and assigns it to be the default. Thus, any Operations, tensors, etc. defined outside of a `Graph.as_default()` context manager will automatically be placed in the default graph:

```
# Placed in the default graph
in_default_graph = tf.add(1,2)

# Placed in graph `g`
with g.as_default():
    in_graph_g = tf.mul(2,3)

# We are no longer in the `with` block, so this is placed in the default
graph
also_in_default_graph = tf.sub(5,1)
```

If you'd like to get a handle to the default graph, use the **tf.get_default_graph()** function:

```
default_graph = tf.get_default_graph()
```

In most TensorFlow programs, you will only ever deal with the default graph. However, creating multiple graphs can be useful if you are defining multiple models that do not have interdependencies. When defining multiple graphs in one file, it's best practice to either not use the default graph or immediately assign a handle to it. This ensures that nodes are added to each graph in a uniform manner:

Correct - Create new graphs, ignore default graph:

```
import tensorflow as tf

g1 = tf.Graph()
g2 = tf.Graph()

with g1.as_default():
    # Define g1 Operations, tensors, etc.
    ...

with g2.as_default():
    # Define g2 Operations, tensors, etc.
    ...
```

Correct - Get handle to default graph

```
import tensorflow as tf

g1 = tf.get_default_graph()
g2 = tf.Graph()

with g1.as_default():
    # Define g1 Operations, tensors, etc.
    ...
```

```
with g2.as_default():
    # Define g2 Operations, tensors, etc.
    ...
```

Incorrect: Mix default graph and user-created graph styles

```
import tensorflow as tf

g2 = tf.Graph()

# Define default graph Operations, tensors, etc.
...

with g2.as_default():
    # Define g2 Operations, tensors, etc.
    ...
```

Additionally, it is possible to load in previously defined models from other TensorFlow scripts and assign them to `Graph` objects using a combination of the **Graph.as_graph_def()** and **tf.import_graph_def** functions. Thus, a user can compute and use the output of several separate models in the same Python file. We will cover importing and exporting graphs later in this book.

TensorFlow Sessions

Sessions, as discussed in the previous exercise, are responsible for graph execution. The constructor https://www.tensorflow.org/versions/master/api_docs/ python/client.html#Session.*init*[`tf.Session()`] takes in three optional parameters:

- `target` specifies the execution engine to use. For most applications, this will be left at its default empty string value. When using sessions in a distributed setting, this parameter is used to connect to `tf.train.Server` instances (covered in the later chapters of this book).
- `graph` specifies the `Graph` object that will be launched in the `Session`. The default value is `None`, which indicates that the current default graph should be used. When using multiple graphs, it's best to explicitly pass in the `Graph` you'd like to run (instead of creating the `Session` inside of a `with` block).
- `config` allows users to specify options to configure the session, such as limiting the number of CPUs or GPUs to use, setting optimization parameters for graphs, and logging options.

In a typical TensorFlow program, `Session` objects will be created without changing any of the default construction parameters.

```
import tensorflow as tf

# Create Operations, Tensors, etc (using the default graph)
a = tf.add(2, 5)
b = tf.mul(a, 3)

# Start up a `Session` using the default graph
sess = tf.Session()
```

Note that these two calls are identical:

```
sess = tf.Session()
sess = tf.Session(graph=tf.get_default_graph())
```

Once a `Session` is opened, you can use its primary method, **run()**, to calculate the value of a desired `Tensor` output:

```
sess.run(b)   # Returns 21
```

`Session.run()` takes in one required parameter, `fetches`, as well as three optional parameters: `feed_dict`, `options`, and `run_metadata`. We won't cover `options` or `run_metadata`, as they are still experimental (thus prone to being changed) and are of limited use at this time. `feed_dict`, however, is important to understand and will be covered below.

FETCHES

`fetches` accepts any graph element (either an `Operation` or `Tensor` object), which specifies what the user would like to execute. If the requested object is a `Tensor`, then the output of `run()` will be a NumPy array. If the object is an `Operation`, then the output will be `None`.

In the above example, we set `fetches` to the tensor b (the output of the `tf.mul` Operation). This tells TensorFlow that the Session should find all of the nodes necessary to compute the value of b, execute them in order, and output the value of b. We can also pass in a list of graph elements:

```
sess.run([a, b])  # returns [7, 21]
```

When `fetches` is a list, the output of `run()` will be a list with values corresponding to the output of the requested elements. In this example, we ask for the values of a and b, in that order. Since both a and b are tensors, we receive their values as output.

In addition using fetches to get Tensor outputs, you'll also see examples where we give fetches a direct handle to an Operation which a useful side-effect when run. An example of this is **tf.initialize_all_variables()**, which prepares all TensorFlow Variable objects to be used (Variable objects will be covered later in this chapter). We still pass the Op as the fetches parameter, but the result of Session.run() will be None:

```
# Performs the computations needed to initialize Variables, but returns
`None`
sess.run(tf.initialize_all_variables())
```

FEED DICTIONARY

The parameter feed_dict is used to override Tensor values in the graph, and it expects a Python dictionary object as input. The keys in the dictionary are handles to Tensor objects that should be overridden, while the values can be numbers, strings, lists, or NumPy arrays (as described previously). The values must be of the same type (or able to be converted to the same type) as the Tensor key. Let's show how we can use feed_dict to overwrite the value of a in the previous graph:

```
import tensorflow as tf

# Create Operations, Tensors, etc (using the default graph)
a = tf.add(2, 5)
b = tf.mul(a, 3)

# Start up a `Session` using the default graph
sess = tf.Session()

# Define a dictionary that says to replace the value of `a` with 15
replace_dict = {a: 15}

# Run the session, passing in `replace_dict` as the value to `feed_dict`
sess.run(b, feed_dict=replace_dict)  # returns 45
```

Notice that even though a would normally evaluate to 7, the dictionary we passed into feed_dict replaced that value with 15. feed_dict can be extremely useful in a number of situations. Because the value of a tensor is provided up front, the graph no longer needs to compute any of the tensor's normal dependencies. This means that if you have a large graph and want to test out part of it with dummy values, TensorFlow won't waste time with unnecessary computations. feed_dict is also useful for specifying input values, as we'll cover in the upcoming placeholder section.

After you are finished using the Session, call its close() method to release unneeded resources:

```
# Open Session
sess = tf.Session()

# Run the graph, write summary statistics, etc.
...

# Close the graph, release its resources
sess.close()
```

As an alternative, you can also use the Session as a context manager, which will automatically close when the code exits its scope:

```
with tf.Session() as sess:
    # Run graph, write summary statistics, etc.
    ...

# The Session closes automatically
```

We can also use a Session as a context manager by using its **as_default()** method. Similarly to how Graph objects can be used implicitly by certain Operations, you can set a session to be used automatically by certain functions. The most common of such functions are **Operation.run()** and **Tensor.eval()**, which act as if you had passed them in to Session.run() directly.

```
# Define simple constant
a = tf.constant(5)

# Open up a Session
sess = tf.Session()

# Use the Session as a default inside of `with` block
with sess.as_default():
    a.eval()
```

```
# Have to close Session manually.
sess.close()
```

More on `Interactivesession`

Earlier in the book, we mentioned that `InteractiveSession` is another type of TensorFlow session, but that we wouldn't be using it. All `InteractiveSession` does is automatically make itself the default session in the runtime. This can be handy when using an interactive Python shell, as you can use `a.eval()` or `a.run()` instead of having to explicitly type out `sess.run([a])`. However, if you need to juggle multiple sessions, things can get a little tricky. We find that maintaining a consistent way of running graphs makes debugging much easier, so we're sticking with regular `Session` objects.

Now that we've got a firm understanding of running our graph, let's look at how to properly specify input nodes and use `feed_dict` in conjunction with them.

Adding Inputs with Placeholder nodes

You may have noticed that the graph we defined previously doesn't use true "input"; it always uses the same numbers, 5 and 3. What we would like to do instead is take values from the client so that we can reuse the transformation described by our graph with all sorts of different numbers. We do that with what is called a "placeholder". Placeholders, as the name implies, act as if they are Tensor objects, but they do not have their values specified when created. Instead, they hold the place for a Tensor that will be fed at runtime, in effect becoming an "input" node. Creating placeholders is done using the **tf.placeholder** Operation:

```
import tensorflow as tf
import numpy as np

# Creates a placeholder vector of length 2 with data type int32
a = tf.placeholder(tf.int32, shape=[2], name="my_input")

# Use the placeholder as if it were any other Tensor object
b = tf.reduce_prod(a, name="prod_b")
c = tf.reduce_sum(a, name="sum_c")

# Finish off the graph
d = tf.add(b, c, name="add_d")
```

`tf.placeholder` takes in a required parameter `dtype`, as well as the optional parameter `shape`:

- `dtype` specifies the data type of values that will be passed into the placeholder. This is required, as it is needed to ensure that there will be no type mismatch errors.
- `shape` specifies what shape the fed Tensor will be. See the discussion on Tensor shapes above. The default value of `shape` is `None`, which means a Tensor of any shape will be accepted.

Like any Operation, you can also specify a `name` identifier to `tf.placeholder`.

In order to actually give a value to the placeholder, we'll use the `feed_dict` parameter in `Session.run()`. We use the handle to the placeholder's output as the key to the dictionary (in the above code, the variable a), and the Tensor object we want to pass in as its value:

```
# Open a TensorFlow Session
sess = tf.Session()

# Create a dictionary to pass into `feed_dict`
# Key: `a`, the handle to the placeholder's output Tensor
# Value: A vector with value [5, 3] and int32 data type
input_dict = {a: np.array([5, 3], dtype=np.int32)}

# Fetch the value of `d`, feeding the values of `input_vector` into `a`
sess.run(d, feed_dict=input_dict)
```

You *must* include a key-value pair in `feed_dict` for each placeholder that is a dependency of the fetched output. Above, we fetched d, which depends on the output of a. If we had defined additional placeholders that d did not depend on, we would not need to include them in the `feed_dict`.

You cannot fetch the value of placeholders- it will simply raise an exception if you try to feed one into `Session.run()`.

Variables

CREATING VARIABLES

`Tensor` and `Operation` objects are immutable, but machine learning tasks, by their nature, need a mechanism to save changing values over time. This is accomplished in TensorFlow with **Variable** objects, which contain mutable tensor values that persist across multiple calls to `Session.run()`. You can create a `Variable` by using its constructor, `tf.Variable()`:

```
import tensorflow as tf

# Pass in a starting value of three for the variable
my_var = tf.Variable(3, name="my_variable")
```

Variables can be used in TensorFlow functions/Operations anywhere you might use a `Tensor`; its present value will be passed on to the Operation using it:

```
add = tf.add(5, my_var)
mul = tf.mul(8, my_var)
```

The initial value of Variables will often be large tensors of zeros, ones, or random values. To make it easier to create these common values, TensorFlow has a number of helper Ops, such as **tf.zeros()**, **tf.ones()**, **tf.random_normal()**, and **tf.random_uniform()**, each of which takes in a `shape` parameter which specifies the dimension of the desired `Tensor`:

```
# 2x2 matrix of  zeros
zeros = tf.zeros([2, 2])

# vector of length 6 of ones
ones = tf.ones([6])

# 3x3x3 Tensor of random uniform  values between 0 and 10
uniform = tf.random_uniform([3, 3, 3], minval=0, maxval=10)

# 3x3x3 Tensor of normally distributed numbers; mean 0 and standard devia-
tion 2
normal = tf.random_normal([3, 3, 3], mean=0.0, stddev=2.0)
```

Instead of using `tf.random_normal()`, you'll often see use of **tf.truncated_normal()** instead, as it doesn't create any values more than two standard deviations away from its mean. This prevents the possibility of having one or two numbers be significantly different than the other values in the tensor:

```
# No values below 3.0 or above 7.0 will be returned in this Tensor
trunc = tf.truncated_normal([2, 2], mean=5.0, stddev=1.0)
```

You can pass in these Operations as the initial values of Variables as you would a hand-written `Tensor`:

```
# Default value of mean=0.0
# Default value of stddev=1.0
random_var = tf.Variable(tf.truncated_normal([2, 2]))
```

VARIABLE INITIALIZATION

`Variable` objects live in the `Graph` like most other TensorFlow objects, but their state is actually managed by a `Session`. Because of this, Variables have an extra step involved in order to use them- you must *initialize* the `Variable` within a `Session`. This causes the `Session` to start keeping track of the ongoing value of the `Variable`. This is typically done by passing in the **tf.initialize_all_variables()** Operation to `Session.run()`:

```
init = tf.initialize_all_variables()
sess = tf.Session()
sess.run(init)
```

If you'd only like to initialize a subset of Variables defined in the graph, you can use `tf.initialize_variables()`, which takes in a list of Variables to be initialized:

```
var1 = tf.Variable(0, name="initialize_me")
var2 = tf.Variable(1, name="no_initialization")
init = tf.initialize_variables([var1], name="init_var1")
sess = tf.Session()
sess.run(init)
```

CHANGING VARIABLES

In order to change the value of the `Variable`, you can use the **Variable.assign()** method, which gives the `Variable` the new value to be. Note that `Variable.assign()` is an Operation, and must be run in a `Session` to take effect:

```
# Create variable with starting value of 1
my_var = tf.Variable(1)

# Create an operation that multiplies the variable by 2 each time it is run
my_var_times_two = my_var.assign(my_var * 2)
```

```
# Initialization operation
init = tf.initialize_all_variables()

# Start a session
sess = tf.Session()

# Initialize variable
sess.run(init)

# Multiply variable by two and return it
sess.run(my_var_times_two)
## OUT: 2

# Multiply again
sess.run(my_var_times_two)
## OUT: 4

# Multiply again
sess.run(my_var_times_two)
## OUT: 8
```

For simple incrementing and decrementing of Variables, TensorFlow includes the **Variable.assign_add()Variable.assign_sub()** methods:

```
# Increment by 1
sess.run(my_var.assign_add(1))

# Decrement by 1
sess.run(my_var.assign_sub(1))
```

Because Sessions maintain `Variable` values separately, each `Session` can have its own current value for a `Variable` defined in a graph:

```
# Create Ops
my_var = tf.Variable(0)
init = tf.initialize_all_variables()

# Start Sessions
sess1 = tf.Session()
sess2 = tf.Session()

# Initialize Variable in sess1, and increment value of my_var in that Session
sess1.run(init)
sess1.run(my_var.assign_add(5))
## OUT: 5

# Do the same with sess2, but use a different increment value
sess2.run(init)
sess2.run(my_var.assign_add(2))
## OUT: 2

# Can increment the Variable values in each Session independently

sess1.run(my_var.assign_add(5))
## OUT: 10

sess2.run(my_var.assign_add(2))
## OUT: 4
```

If you'd like to reset your Variables to their starting value, simply call `tf.initialize_all_variables()` again (or `tf.initialize_variables` if you only want to reset a subset of them):

```
# Create Ops
my_var = tf.Variable(0)
init = tf.initialize_all_variables()

# Start Session
sess = tf.Session()

# Initialize Variables
```

```
sess.run(init)

# Change the Variable
sess.run(my_var.assign(10))

# Reset the Variable to 0, its initial value
sess.run(init)
```

TRAINABLE

Later in this book, you'll see various `Optimizer` classes which automatically train machine learning models. That means that it will change values of `Variable` objects without explicitly asking to do so. In most cases, this is what you want, but if there are Variables in your graph that should *only* be changed manually and not with an `Optimizer`, you need to set their `trainable` parameter to `False` when creating them:

```
not_trainable = tf.Variable(0, trainable=False)
```

This is typically done with step counters or anything else that isn't going to be involved in the calculation of a machine learning model.

Organizing your graph with name scopes

We've now covered the core building blocks necessary to build any TensorFlow graph. So far, we've only worked with toy graphs containing a few nodes and small tensors, but real world models can contain dozens or hundreds of nodes, as well as millions of parameters. In order to manage this level of complexity, TensorFlow currently offers a mechanism to help organize your graphs: *name scopes*.

Name scopes are incredibly simple to use and provide great value when visualizing your graph with TensorBoard. Essentially, name scopes allow you to group Operations into larger, named blocks. Then, when you launch your graph with TensorBoard, each name scope will encapsulate its own Ops, making the visualization much more digestible. For basic name scope usage, simply add your Operations in a `with tf.name_scope(<name>)` block:

```
import tensorflow as tf

with tf.name_scope("Scope_A"):
    a = tf.add(1, 2, name="A_add")
    b = tf.mul(a, 3, name="A_mul")

with tf.name_scope("Scope_B"):
    c = tf.add(4, 5, name="B_add")
    d = tf.mul(c, 6, name="B_mul")

e = tf.add(b, d, name="output")
```

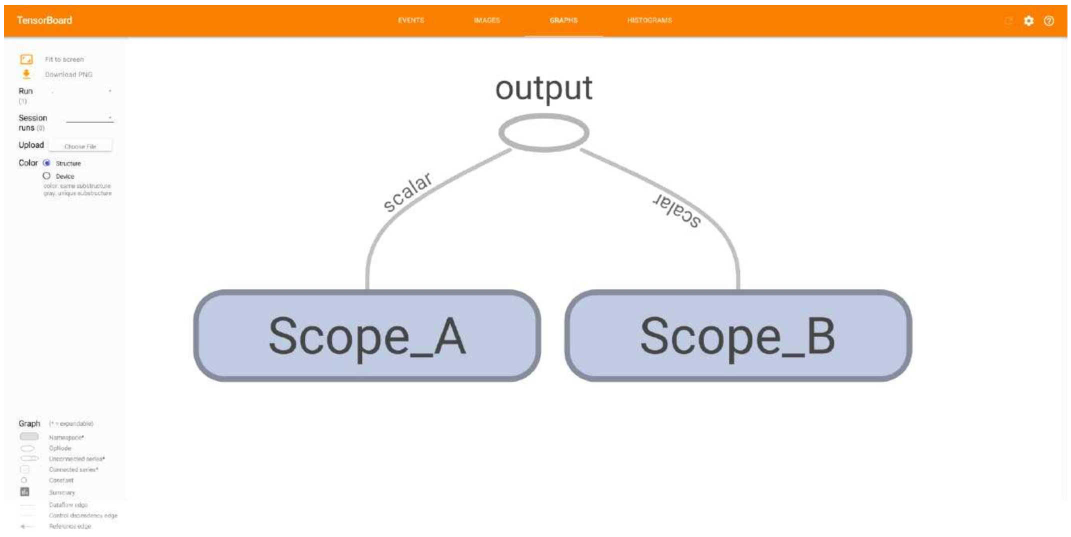
To see the result of these name scopes in TensorBoard, let's open up a `SummaryWriter` and write this graph to disk.

```
writer = tf.train.SummaryWriter('./name_scope_1', graph=tf.get_de-
fault_graph())
writer.close()
```

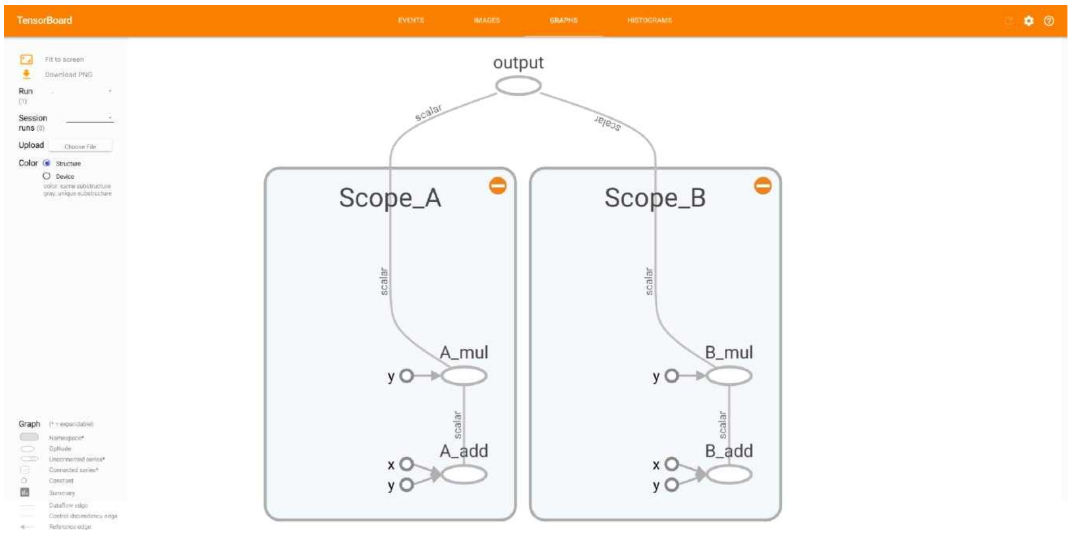
Because the `SummaryWriter` exports the graph immediately, we can simply start up TensorBoard after running the above code. Navigate to where you ran the previous script and start up TensorBoard:

```
$ tensorboard --logdir='./name_scope_1'
```

As before, this will start a TensorBoard server on your local computer at port 6006. Open up a browser and enter `localhost:6006` into the URL bar. Navigate to the "Graph" tab, and you'll see something similar to this:



You'll notice that the `add` and `mul` Operations we added to the graph aren't immediately visible- instead, we see their enclosing name scopes. You can expand the name scope boxes by clicking on the plus + icon in their upper right corner.



Inside of each scope, you'll see the individual Operations you've added to the graph. You can also nest name scopes within other name scopes:

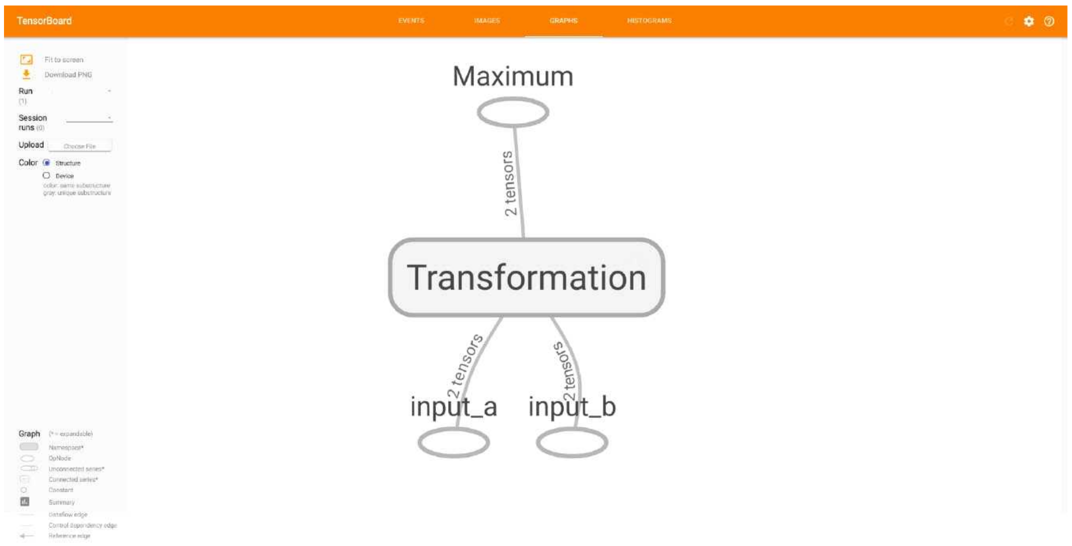
```
graph = tf.Graph()

with graph.as_default():
    in_1 = tf.placeholder(tf.float32, shape=[], name="input_a")
    in_2 = tf.placeholder(tf.float32, shape=[], name="input_b")
    const = tf.constant(3, dtype=tf.float32, name="static_value")

    with tf.name_scope("Transformation"):

        with tf.name_scope("A"):
            A_mul = tf.mul(in_1, const)
            A_out = tf.sub(A_mul, in_1)

        with tf.name_scope("B"):
            B_mul = tf.mul(in_2, const)
            A_out = tf.sub(B_mul, in_2)

        with tf.name_scope("C"):
            C_div = tf.div(A_out, B_out)
            C_out = tf.add(C_div, const)
        with tf.name_scope("D"):
            D_div = tf.div(B_out, A_out)
            D_out = tf.add(D_div, const)

    out = tf.maximum(C_out, D_out)

writer = tf.train.SummaryWriter('./name_scope_2', graph=graph)
writer.close()
```

To mix things up, this code explicitly creates a `tf.Graph` object instead of using the default graph. Let's look at the code and focus on the name scopes to see exactly how it's structured:

```
graph = tf.Graph()

with graph.as_default():
    in_1 = tf.placeholder(...)
    in_2 = tf.placeholder(...)
    const = tf.constant(...)

    with tf.name_scope("Transformation"):

        with tf.name_scope("A"):
            # Takes in_1, outputs some value
            ...

        with tf.name_scope("B"):
            # Takes in_2, outputs some value
            ...

        with tf.name_scope("C"):
            # Takes the output of A and B, outputs some value
            ...

        with tf.name_scope("D"):
            # Takes the output of A and B, outputs some value
            ...

    # Takes the output of C and D
    out = tf.maximum(...)
```

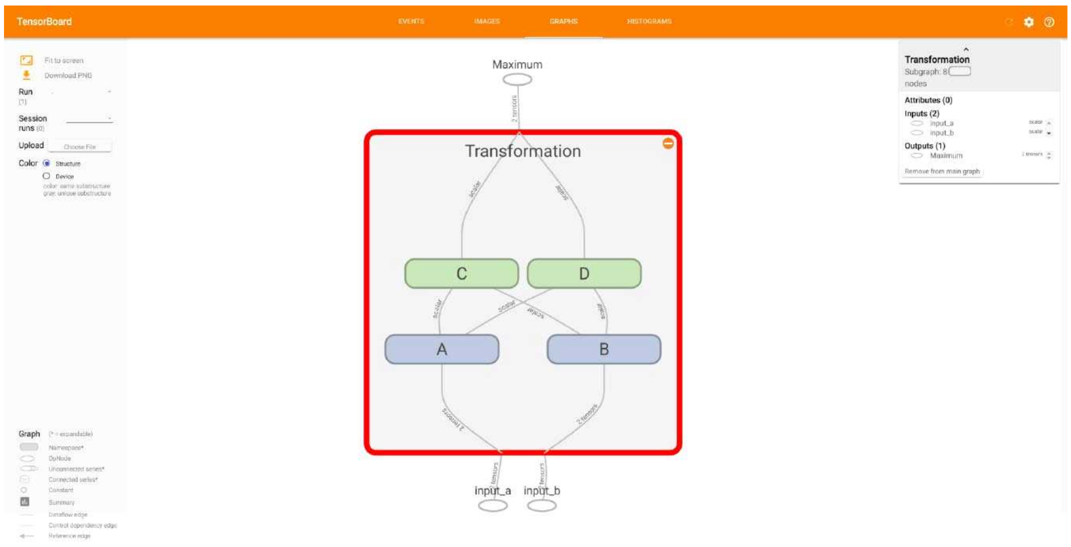
Now it's easier to dissect. This model has two scalar placeholder nodes as input, a TensorFlow constant, a middle chunk called "Transformation", and then a final output node that uses **tf.maximum()** as its Operation. We can see this high-level overview inside of TensorBoard:

```
# Start up TensorBoard in a terminal, loading in our previous graph
$ tensorboard --logdir='./name_scope_2'
```

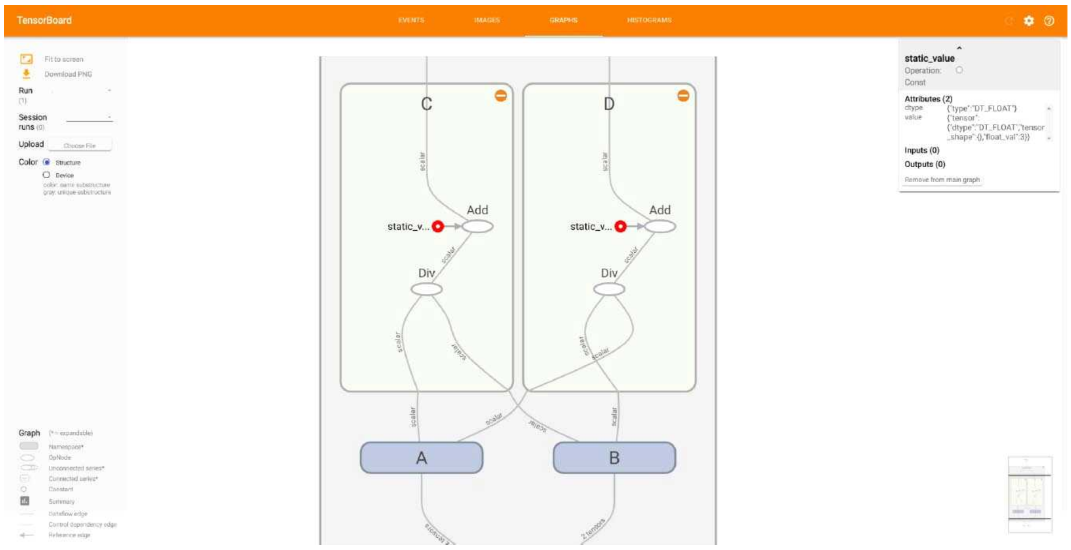


Inside of the Transformation name scope are four more name scopes arranged in two "layers". The first layer is comprised of scopes "A" and

"B", which pass their output values into the next layer of "C" and "D". The final node then uses the outputs from this last layer as its input. If you expand the Transformation name scope in TensorBoard, you'll get a look at this:



This also gives us a chance to showcase another feature in TensorBoard. In the above picture, you'll notice that name scopes "A" and "B" have matching color (blue), as do "C" and "D" (green). This is due to the fact that these name scopes have identical Operations setup in the same configuration. That is, "A" and "B" both have a `tf.mul()` Op feeding into a `tf.sub()` Op, while "C" and "D" have a `tf.div()` Op that feeds into a `tf.add()` Op. This becomes handy if you start using functions to create repeated sequences of Operations.



In this image, you can see that `tf.constant` objects don't behave quite the same way as other Tensors or Operations when displayed in TensorBoard. Even though we declared `static_value` outside of any name scope, it still gets placed inside them. Furthermore, instead of there being one icon for `static_value`, it creates a small visual whenever it is used. The basic idea for this is that constants can be used at any time and don't necessarily need to be used in any particular order. To prevent arrows flying all over the graph from a single point, it just makes a little small impression whenever a constant is used.

ASIDE: tf.maximum()

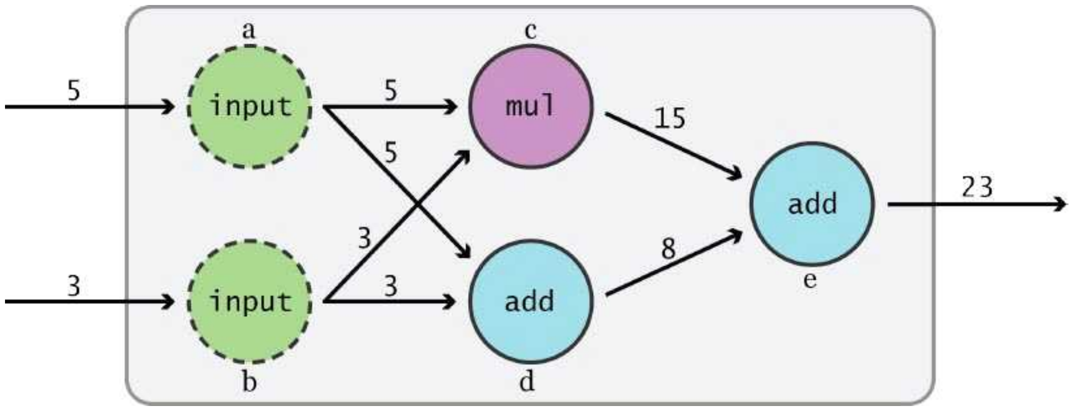
Separating a huge graph into meaningful clusters can make understanding and debugging your model a much more approachable task.

Logging with TensorBoard

Exercise: Putting it together

We'll end this chapter with an exercise that uses all of the components we've discussed: Tensors, Graphs, Operations, Variables, placeholders, Sessions, and name scopes. We'll also include some TensorBoard summaries so we can keep track of the graph as it runs. By the end of this, you should feel comfortable composing basic TensorFlow graphs and exploring it in TensorBoard.

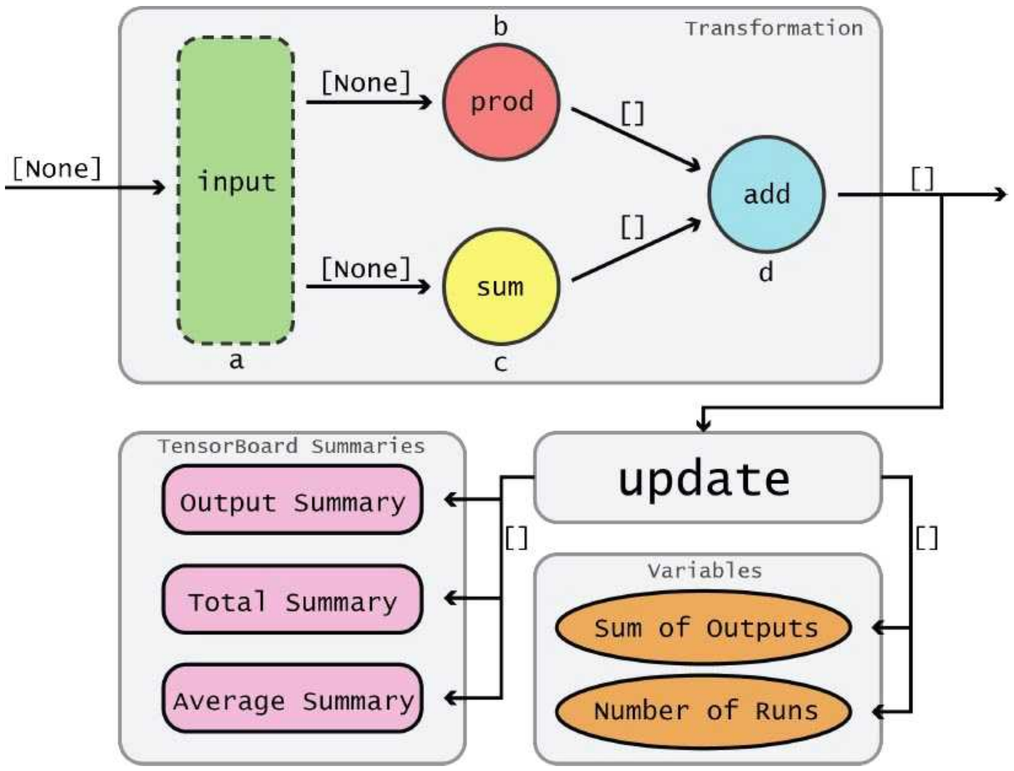
At its core, it is going to be the same sort of transformation as our first basic model:



But, this time it's going to have some important differences that use TensorFlow more fully:

- Our inputs will be placeholders instead of `tf.constant` nodes
- Instead of taking two discrete scalar inputs, our model will take in a single vector of any length
- We're going to accumulate the total value of all outputs over time as we use the graph
- The graph will be segmented nicely with name scopes
- After each run, we are going to save the output of the graph, the accumulated total of all outputs, and the average value of all outputs to disk for use in TensorBoard

Here's a rough outline of what we'd like our graph to look like:



Here are the key things to note about reading this model:

- Notice how each edge now has either a `[None]` or `[]` icon next to it. This represents the TensorFlow shape of the tensor flowing through that edge, with `None` representing a vector of any length, and `[]` representing a scalar.
- The output of node `d` now flows into an "update" section, which contains Operations necessary to update Variables as well as pass data through to the TensorBoard summaries.
- We have a separate name scope to contain our two Variables- one to store the accumulated sum of our outputs, the other to keep track of

how many times we've run the graph. Since these two Variables operate outside of the flow of our main transformation, it makes sense to put them in a separate space.

- There is a name scope dedicated to our TensorBoard summaries which will hold our `tf.scalar_summary` Operations. We place them after the "update" section to ensure that the summaries are added *after* we update our Variables, otherwise things could run out of order.

Let's get going! Open up your code editor or interactive Python environment.

Building the graph

The first thing we'll need to do, as always, is import the TensorFlow library:

```
import tensorflow as tf
```

We're going to explicitly create the graph that we'd like to use instead of using the default graph, so make one with `tf.Graph()`:

```
graph = tf.Graph()
```

And then we'll set our new graph as the default while we construct our model:

```
with graph.as_default():
```

We have two "global" style Variables in our model. The first is a "global_step", which will keep track of how many times we've run our model. This is a common paradigm in TensorFlow, and you'll see it used throughout the API. The second Variable is called "total_output"- it's going to keep track of the total sum of all outputs run on this model over time. Because these Variables are global in nature, we declare them separately from the rest of the nodes in the graph and place them into their own name scope:

```
with graph.as_default():

    with tf.name_scope("variables"):
        # Variable to keep track of how many times the graph has been run
        global_step = tf.Variable(0, dtype=tf.int32, trainable=False,
name="global_step")

        # Variable that keeps track of the sum of all output values over
time:
        total_output = tf.Variable(0.0, dtype=tf.float32, trainable=False,
name="total_output")
```

Note that we use the `trainable=False` setting- it won't have an impact in this model (we aren't training anything!), but it makes it explicit that these Variables will be set by hand.

Next up, we'll create the core transformation part of the model. We'll encapsulate the entire transformation in a name scope, "transformation", and separate them further into separate "input", "intermediate_layer", and "output" name scopes:

```
with graph.as_default():
    with tf.name_scope("variables"):
        ...

    with tf.name_scope("transformation"):

        # Separate input layer
        with tf.name_scope("input"):
            # Create input placeholder- takes in a Vector
            a = tf.placeholder(tf.float32, shape=[None], name="input_place-
holder_a")

        # Separate middle layer
        with tf.name_scope("intermediate_layer"):
            b = tf.reduce_prod(a, name="product_b")
            c = tf.reduce_sum(a, name="sum_c")

        # Separate output layer
        with tf.name_scope("output"):
            output = tf.add(b, c, name="output")
```

This is extremely similar to the code written for the previous model, with a few key differences:

- Our input node is a `tf.placeholder` that accepts a vector of any length (`shape=[None]`).
- Instead of using `tf.mul()` and `tf.add()`, we use `tf.reduce_prod()` and `tf.reduce_sum()`, respectively. This allows us to multiply and add across the entire input vector, as the earlier Ops only accept *exactly* 2 input scalars.

After the transformation computation, we're going to need to update our two Variables from above. Let's create an "update" name scope to hold these changes:

```
with graph.as_default():
    with tf.name_scope("variables"):
        ...
    with tf.name_scope("transformation"):
        ...

    with tf.name_scope("update"):
        # Increments the total_output Variable by the latest output
        update_total = total_output.assign_add(output)

        # Increments the above `global_step` Variable, should be run whenev-
er the graph is run
        increment_step = global_step.assign_add(1)
```

We use the `Variable.assign_add()` Operation to increment both `total_output` and `global_step`. We add on the value of `output` to `total_output`, as we want to accumulate the sum of all outputs over time. For `global_step`, we simply increment it by one.

After we have updated our Variables, we can create the TensorBoard summaries we're interested in. We'll place those inside a name scope called "summaries":

```
with graph.as_default():
        ...
    with tf.name_scope("update"):
        ...
    with tf.name_scope("summaries"):
        avg = tf.div(update_total, tf.cast(increment_step, tf.float32),
name="average")

        # Creates summaries for output node
        tf.scalar_summary(b'Output', output, name="output_summary")
        tf.scalar_summary(b'Sum of outputs over time', update_total,
name="total_summary")
        tf.scalar_summary(b'Average of outputs over time', avg, name="aver-
age_summary")
```

The first thing we do inside of this section is compute the average output value over time. Luckily, we have the total value of all outputs with `total_output` (we use the output from `update_total` to make sure that the update happens before we compute `avg`), as well as the total number of times we've run the graph with `global_step` (same thing- we use the output of `increment_step` to make sure the graph runs in order). Once we have the average, we save `output`, `update_total` and `avg` with separate `tf.scalar_summary` objects.

To finish up the graph, we'll create our Variable initialization Operation as well as a helper node to group all of our summaries into one Op. Let's place these in a name scope called "global_ops":

```
with graph.as_default():
        ...
    with tf.name_scope("summaries"):
        ...
    with tf.name_scope("global_ops"):
        # Initialization Op
        init = tf.initialize_all_variables()
        # Merge all summaries into one Operation
        merged_summaries = tf.merge_all_summaries()
```

You may be wondering why we placed the `tf.merge_all_summaries()` Op here instead of the "summaries" name scope. While it doesn't make a huge difference here, placing `merge_all_summaries()` with other global Operations is generally best practice. This graph only has one section for summaries, but you can imagine a graph having different summaries for Variables, Operations, name scopes, etc. By keeping `merge_all_summaries()` separate, it ensures that you'll be able to find the Operation without having to remember which specific "summary" code block you placed it in.

That's it for creating the graph! Now let's get things set up to execute the graph.

Running the graph

Let's open up a `Session` and launch the `Graph` we just made. We can also open up a `tf.train.SummaryWriter`, which we'll use to save our summaries. Here, we list `./improved_graph` as our destination folder for summaries to be saved.

```
sess = tf.Session(graph=graph)
writer = tf.train.SummaryWriter('./improved_graph', graph)
```

With a Session started, let's initialize our Variables before doing anything else:

```
sess.run(init)
```

To actually run our graph, let's create a helper function, `run_graph()` so that we don't have to keep typing the same thing over and over again. What we'd like is to pass in our input vector to the function, which will run the graph and save our summaries:

```
def run_graph(input_tensor):
    feed_dict = {a: input_tensor}
    _, step, summary = sess.run([output, increment_step, merged_summaries],
                                feed_dict=feed_dict)
    writer.add_summary(summary, global_step=step)
```

Here's the line-by-line breakdown of `run_graph()`:

1. First, we create a dictionary to use as a `feed_dict` in `Session.run()`. This corresponds to our `tf.placeholder` node, using its handle `a`.
2. Next, we tell our `Session` to run the graph using our `feed_dict`, and we want to make sure that we run `output`, `increment_step`, and our `merged_summaries` Ops. We need to save the `global_step` and `merged_summaries` values in order to write our summaries, so we save them to the `step` and `summary` Python variables. We use an underscore _ to indicate that we don't care about storing the value of `output`.
3. Finally, we add the summaries to our `SummaryWriter`. The `global_step` parameter is important, as it allows TensorBoard to graph data over time (it essentially creates the x-axis on a line chart coming up).

Let's actually use it! Call `run_graph` several times with vectors of various lengths- like this:

```
run_graph([2,8])
run_graph([3,1,3,3])
run_graph([8])
run_graph([1,2,3])
run_graph([11,4])
run_graph([4,1])
run_graph([7,3,1])
run_graph([6,3])
run_graph([0,2])
run_graph([4,5,6])
```

Do is as many times as you'd like. Once you've had your fill, go ahead and write the summaries to disk with the `SummaryWriter.flush()` method:

```
writer.flush()
```

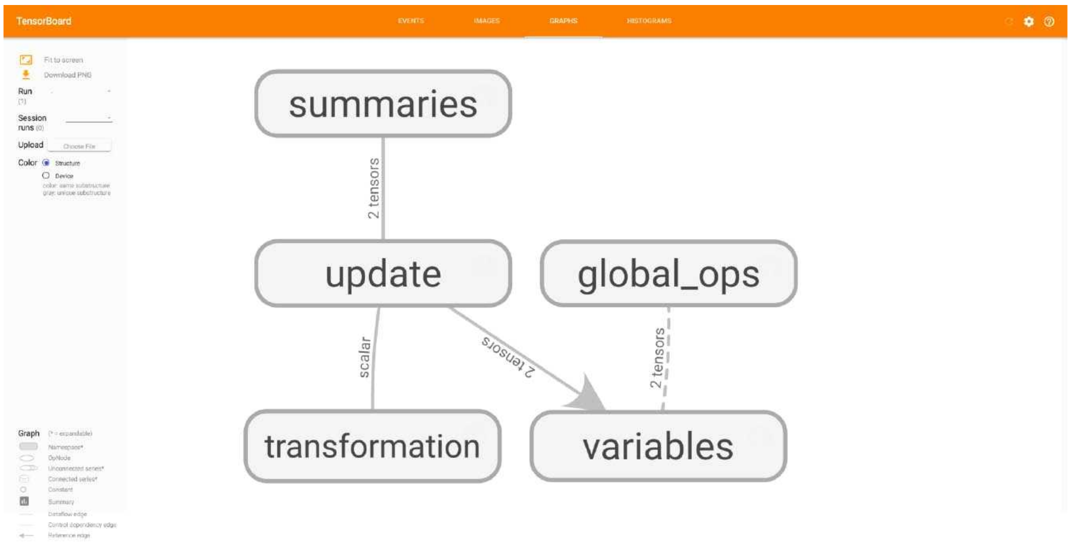
Finally, let's be tidy and close both our `SummaryWriter` and `Session`, now that we're done with them:

```
writer.close()
sess.close()
```

And that's it for our TensorFlow code! It was a little longer than the last graph, but it wasn't too bad. Let's open up TensorBoard and see what we've got. Fire up a terminal shell, navigate to the directory where you ran this code (make sure the "improved_graph" directory is located there), and run the following:

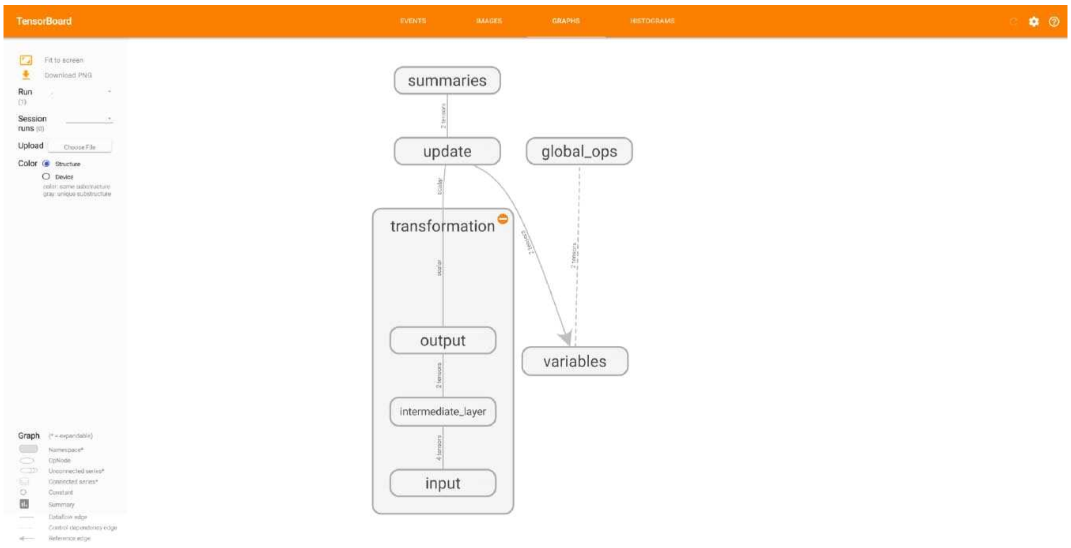
```
$ tensorboard --logdir="improved_graph"
```

As usual, this starts up a TensorBoard server on port 6006, hosting the data stored in "improved_graph". Type in "localhost:6006" into your web browser and let's see what we've got! Let's first check out the "Graph" tab:

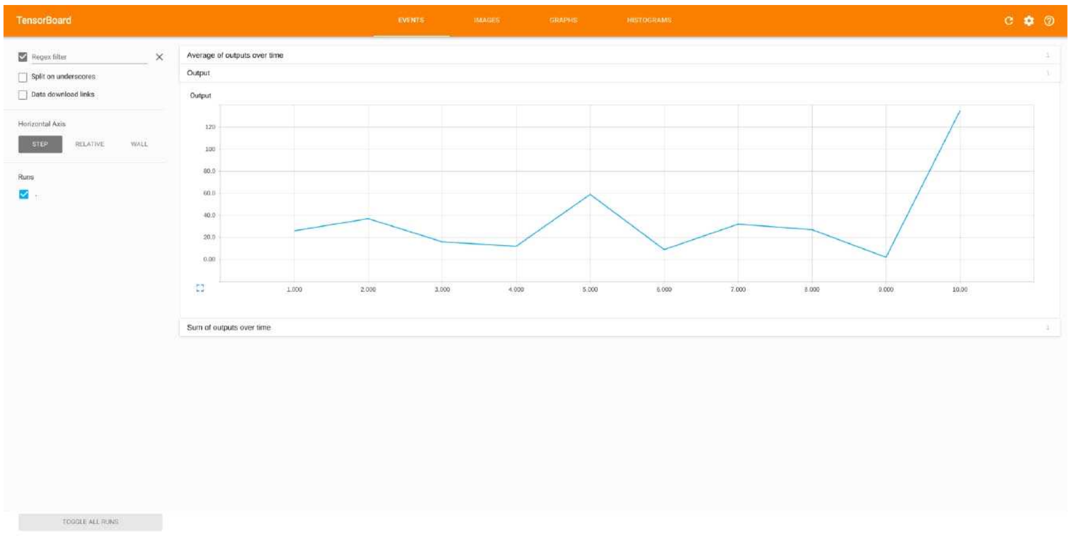


You'll see that this graph closely matches what we diagrammed out earlier. Our transformation operations flow into the update block, which then feeds into both the summary and variable name scopes. The main difference between this and our diagram is the "global_ops" name scope, which contains operations that aren't critical to the primary transformation computation.

You can expand the various blocks to get a more granular look at their structure:



Now we can see the separation of our input, the intermediate layer, and the output. It might be overkill on a simple model like this, but this sort of compartmentalization is extremely useful. Feel free to explore the rest of the graph. When you're ready, head over to the "Events" page.



When you open up the "Events" page, you should see three collapsed tabs, named based on the tags we gave our `scalar_summary` objects above. By clicking on any of them, you'll see a nice line chart showing the values we stored at various time steps. If you click the blue rectangle at the bottom left of the charts, they'll expand to look like the image above.

Definitely check out the results of your summaries, compare them, make sure that they make sense, and pat yourself on the back! That concludes this exercise- hopefully by now you have a good sense of how to create TensorFlow graphs based on visual sketches, as well as how to do some basic summaries with TensorBoard.

The entirety of the code for this exercise is below:

```
import tensorflow as tf

# Explicitly create a Graph object
graph = tf.Graph()

with graph.as_default():

    with tf.name_scope("variables"):
        # Variable to keep track of how many times the graph has been run
        global_step = tf.Variable(0, dtype=tf.int32, trainable=False,
name="global_step")

        # Variable that keeps track of the sum of all output values over
time:
        total_output = tf.Variable(0.0, dtype=tf.float32, trainable=False,
name="total_output")

    # Primary transformation Operations
    with tf.name_scope("transformation"):

        # Separate input layer
        with tf.name_scope("input"):
            # Create input placeholder- takes in a Vector
            a = tf.placeholder(tf.float32, shape=[None], name="input_place-
holder_a")

        # Separate middle layer
        with tf.name_scope("intermediate_layer"):
            b = tf.reduce_prod(a, name="product_b")
            c = tf.reduce_sum(a, name="sum_c")

        # Separate output layer
        with tf.name_scope("output"):
            output = tf.add(b, c, name="output")

    with tf.name_scope("update"):
        # Increments the total_output Variable by the latest output
        update_total = total_output.assign_add(output)
        # Increments the above `global_step` Variable, should be run whenev-
er the graph is run
        increment_step = global_step.assign_add(1)

    # Summary Operations
    with tf.name_scope("summaries"):
        avg = tf.div(update_total, tf.cast(increment_step, tf.float32),
name="average")

        # Creates summaries for output node
        tf.scalar_summary(b'Output', output, name="output_summary")
        tf.scalar_summary(b'Sum of outputs over time', update_total,
name="total_summary")
        tf.scalar_summary(b'Average of outputs over time', avg, name="aver-
age_summary")

    # Global Variables and Operations
    with tf.name_scope("global_ops"):
        # Initialization Op
        init = tf.initialize_all_variables()
        # Merge all summaries into one Operation
        merged_summaries = tf.merge_all_summaries()

# Start a Session, using the explicitly created Graph
sess = tf.Session(graph=graph)

# Open a SummaryWriter to save summaries
writer = tf.train.SummaryWriter('./improved_graph', graph)

# Initialize Variables
sess.run(init)

def run_graph(input_tensor):
    """
    Helper function; runs the graph with given input tensor and saves summa-
ries
    """
    feed_dict = {a: input_tensor}
    _, step, summary = sess.run([output, increment_step, merged_summaries],
                                feed_dict=feed_dict)
    writer.add_summary(summary, global_step=step)

# Run the graph with various inputs
run_graph([2,8])
run_graph([3,1,3,3])
run_graph([8])
run_graph([1,2,3])
run_graph([11,4])
run_graph([4,1])
run_graph([7,3,1])
run_graph([6,3])
run_graph([0,2])
run_graph([4,5,6])

# Write the summaries to disk
writer.flush()

# Close the SummaryWriter
writer.close()

# Close the session
sess.close()
```

Conclusion

That wraps up this chapter! There was a lot of information to absorb, and you should definitely play around with TensorFlow now that you have a grasp of the fundamentals. Get yourself fluent with Operations, Variables, and Sessions, and embed the basic loop of building and running graphs into your head.

Using TensorFlow for simple math problems is fun (for some people), but we haven't even touched on the primary use case for the library yet: machine learning. In the next chapter, you'll be introduced to some of the core concepts and techniques for machine learning and how to use them inside of TensorFlow.