Basic tensorflow: Creating a graph

In [13]:
```
import tensorflow as tf
```

Right after we import TensorFlow (with import tensorflow as tf), a specific empty default graph is formed. All the nodes we create are automatically associated with that default graph.

Using the tf. methods, we will create six nodes assigned to arbitrarily named variables. The contents of these variables should be regarded as the output of the operations, and not the operations themselves. For now we refer to both the operations and their outputs with the names of their corresponding variables.

The first three nodes are each told to output a constant value. The values 5, 2, and 3 are assigned to a, b, and c, respectively:

In [14]:
```
a = tf.constant(5, name = "a")
b = tf.constant(2, name = "b")
c = tf.constant(3, name = "c")
```

Each of the next three nodes gets two existing variables as inputs, and performs simple arithmetic operations on them: Node d multiplies the outputs of nodes a and b. Node e adds the outputs of nodes b and c. Node f subtracts the output of node e from that of node d.

In [15]:
```
d = tf.multiply(a,b, name = "op_multi")
e = tf.add(c,b, name = "op_add")
f = tf.subtract(d,e, name = "op_subtract")
```

And we should get a tensorflow graph

In [16]:
```
sess = tf.Session()
outs = sess.run([f,e,d])
filewriter = tf.summary.FileWriter("output1", sess.graph)
```

graph_large_attrs_key=_too_large_attrs&limit_attr_size=1024&run=.png

For some arithmetic and logical operations it is possible to use operation shortcuts instead of having to apply tf.. For example, in this graph we could have used $*/+/-$ instead of tf.multiply()/tf.add()/tf.subtract() Some shortcuts for common operations are:

```
tf.add()      a + b
tf.multiply   a * b
```

Creating a Session and running it:

Once we are done describing the computation graph, we are ready to run the computations that it represents. For this to happen, we need to create and run a session. We do this by adding the following code:

In [19]:
```
sess = tf.Session()  #this is an operator called a constructor that creates an ob
outs = sess.run(f)
sess.close()
print("outs = {}".format(outs))
```

```
outs = 5
```

First, we launch the graph in a tf.Session. A Session object is the part of the TensorFlow API that communicates between Python objects and data on our end, and the actual computational system where memory is allocated for the objects we define, intermediate variables are stored, and finally results are fetched for us.

In [21]:
```
sess = tf.Session()
```

The execution itself is then done with the .run() method of the Session object. When called, this method completes one set of computations in our graph in the following manner: it starts at the requested output(s) and then works backward, computing nodes that must be executed according to the set of dependencies. Therefore, the part of the graph that will be computed depends on our output query.

In our example, we requested that node f be computed and got its value, 5, as output:

In [22]:
```
outs = sess.run(f)
```

When our computation task is completed, it is good practice to close the session using the sess.close() command, making sure the resources used by our session are freed up.

In [23]:
```
sess.close()
```

Constructing and Managing our graph:

As mentioned, as soon as we import TensorFlow, a default graph is automatically created for us with a session. We can create additional graphs and control their association with some given operations. tf.Graph() creates a new graph, represented as a TensorFlow object. In this example we create another graph and assign it to the variable g:

In [25]:
```
import tensorflow as tf
print(tf.get_default_graph())

g = tf.Graph()  #this is a constructor for Graph object
print(g)
```

```
<tensorflow.python.framework.ops.Graph object at 0x7f7d9c275210>
<tensorflow.python.framework.ops.Graph object at 0x7f7d9c205710>
```

You can have multiple graphs in your program.

At this point we have two graphs: the default graph and the empty graph in g. Both are revealed as TensorFlow objects when printed. Since g hasn't been assigned as the default graph, any operation we create will not be associated with it, but rather with the default one.

We can check which graph is currently set as the default by using tf.get_default_graph(). Also, for a given node, we can view the graph it's associated with by using the .graph attribute:

In [26]:
```
g = tf.Graph()  #new graph
a = tf.constant(5)  #a is a constant, but in which graph is it?

print(a.graph is g)
print(a.graph is tf.get_default_graph())
```

```
False
True
```

In [27]:
```
a.graph is g  #can ask if a belongs to our new graph g? But it doesn't -- it belo
```

Out[27]:
```
False
```

In [28]:
```
a.graph is tf.get_default_graph()
```

Out[28]:
```
True
```

In this code example we see that the operation we've created is associated with the default graph and not with the graph in g.

To make sure our constructed nodes are associated with the right graph we can construct them using a very useful Python construct: the with statement.

The with statement is used to wrap the execution of a block with methods defined by a context manager—an object that has the special method functions .**enter**() to set up a block of code and .**exit**() to exit the block.

In other words, it's very convenient in many cases to execute some code that requires "setting up" of some kind (like opening a file, SQL table, etc.) and then always "tearing it down" at the end, regardless of whether the code ran well or raised any kind of exception. In our case we use with to set up a graph and make sure every piece of code will be performed in the context of that graph.

We use the with statement together with the as_default() command, which returns a context manager that makes this graph the default one. This comes in handy when working with multiple graphs:

In [29]:
```
g1 = tf.get_default_graph()
g2 = tf.Graph()

print(g1 is tf.get_default_graph())   # output should be True because g1 is our d

with g2.as_default():       #with is a keyword that starts a new block or scope;
    print(g1 is tf.get_default_graph()) #in this block of code, use g2 as the def
                                        #is no longer default

print(g1 is tf.get_default_graph())  #True because our scope ended with the empty
```

```
True
False
True
```

The with statement can also be used to start a session without having to explicitly close it.

Fetches:

In our initial graph example, we request one specific node (node f) by passing the variable it was assigned to as an argument to the sess.run() method. This argument is called fetches, corresponding to the elements of the graph we wish to compute. We can also ask sess.run() for multiple nodes' outputs simply by inputting a list of requested nodes:

In [30]:
```
with tf.Session() as sess:
   fetches = [a,b,c,d,e,f]
   outs = sess.run(fetches)

print("outs = {}".format(outs))
print(type(outs[0]))
```

```
outs = [5, 2, 3, 10, 5, 5]
<type 'numpy.int32'>
```

We get back a list containing the outputs of the nodes according to how they were ordered in the input list. The data in each item of the list is of type NumPy.

TensorFlow computes only the essential nodes according to the set of dependencies. This is also manifested in our example: when we ask for the output of node d, only the outputs of nodes a and b are computed. This is a great advantage of TensorFlow—it doesn't matter how big and complicated our graph is as a whole, since we can run just a small portion of it as needed.

Opening a session using the with clause will ensure the session is automatically closed once all computations are done.

Flowing tensors:

Understanding how nodes and edges are actually represented in TensorFlow, and how we can control their characteristics. To demonstrate how they work, we will focus on source operations, which are used to initialize values.

Nodes are operations, Edges are tensor objects

When we construct a node in the graph, like we did with tf.add(), we are actually creating an operation instance. These operations do not produce actual values until the graph is executed, but rather reference their to-be-computed result as a handle that can be passed on—flow—to another node. These handles, which we can think of as the edges in our graph, are referred to as Tensor objects, and this is where the name TensorFlow originates from.

TensorFlow is designed such that first a skeleton graph is created with all of its components. At this point no actual data flows in it and no computations take place. It is only upon execution, when we run the session, that data enters the graph and computations occur

In a previous example, tf.constant() created a node with the corresponding passed value. Printing the output of the constructor, we see that it’s actually a Tensor object instance. These objects have methods and attributes that control their behavior and that can be defined upon creation.

In this example, the variable c stores a Tensor object with the name Const 5:0, designated to contain a 32-bit floating-point scalar:

In [33]:
```
c = tf.constant(4.0)  #in python and TF, you just give the value 4.0, and TF assu
print(c)
#If you wanted this to be float64, you would have to cast
#If you run this many times, the value will increase because it has many c consta
```

```
Tensor("Const_9:0", shape=(), dtype=float32)
```

Setting attributes with source operations

Each Tensor object in TensorFlow has attributes such as name, shape, and dtype that help identify and set the characteristics of that object. These attributes are optional when creating a node, and are set automatically by TensorFlow when missing.

We will take a look at these attributes. We will do so by looking at Tensor objects created by ops known as source operations. Source operations are operations that create data, usually without using any previously processed inputs. With these operations we can create scalars, as we already encountered with the tf.constant() method, as well as arrays and other types of data.

Data Types

The basic units of data that pass through a graph are numerical, Boolean, or string elements. When we print out the Tensor object c from our last code example, we see that its data type is a floating-point number. Since we didn’t specify the type of data, TensorFlow inferred it automatically. For example 5 is regarded as an integer, while anything with a decimal point, like 5.1, is regarded as a floating-point number.

We can explicitly choose what data type we want to work with by specifying it when we create the Tensor object. We can see what type of data was set for a given Tensor object by using the attribute dtype:

In [34]:
```
c = tf.constant(4.0, dtype=tf.float64)
print(c)
print(c.dtype)
```

```
Tensor("Const_10:0", shape=(), dtype=float64)
<dtype: 'float64'>
```

Tensorflow supports many data types.

- Complex numbers
- Booleans
- Strings
- Integers
- etc.

It is important to make sure our data types match throughout the graph—performing an operation with two nonmatching data types will result in an exception. To change the data type setting of a Tensor object, we can use the tf.cast() operation, passing the relevant Tensor and the new data type of interest as the first and second arguments, respectively:

In [35]:
```
x = tf.constant([1,2,3],name='x',dtype=tf.float32)
print(x.dtype)
x = tf.cast(x,tf.int64) #turn the float number above into integer
print(x.dtype)
```

```
<dtype: 'float32'>
<dtype: 'int64'>
```

Tensor Array and Shapes:

~A source of potential confusion is that two different things are referred to by the name, Tensor. As used in the previous sections, Tensor is the name of an object used in the Python API as a handle for the result of an operation in the graph.

However, tensor is also a mathematical term for n-dimensional arrays. For example, a 1×1 tensor is a scalar, a $1 \times n$ tensor is a vector, an n×n tensor is a matrix, and an n×n×n tensor is just a three-dimensional array.

TensorFlow regards all the data units that flow in the graph as tensors, whether they are multidimensional arrays, vectors, matrices, or scalars.

As with dtype, unless stated explicitly, TensorFlow automatically infers the shape of the data. When we printed out the Tensor object at the beginning of this section, it showed that its shape was (), corresponding to the shape of a scalar.

Using scalars is good for demonstration purposes, but most of the time it's much more practical to work with multidimensional arrays. To initialize high-dimensional arrays, we can use Python lists or NumPy arrays as inputs.

In the following example, we use as inputs a 2×3 matrix using a Python list and then a 3D NumPy array of size $2 \times 2 \times 3$ (two matrices of size 2×3)

Note: Numpy is a useful package for numerical computing and working with arrays. Tensorflow and Numpy are tightly coupled. For example, the output returned by sess.run() is a NumPy array. Many TF operations have the same syntax as funcions in NumPy.

In [36]:
```
import numpy as np
a = [[1,2,3],
     [4,5,6]]    # a is a matrix or a python list; you can use regular python matr
print (a)
print (type (a))
b = np.array(a)   #this is a numpy array; in your programs you can have various ty
print (type(b))
c = tf.constant([[1,2,3],     #you wanna make a tensor c; could have writte d - ty
                [4,5,6]])             #take numpy objects  and turn them into ten

print("Python List input: {}".format(c.get_shape()))

c = tf.constant(np.array([    #turning numpy array into tensor
                 [[1,2,3],
                  [4,5,6]],

                 [[1,1,1],
                  [2,2,2]]
                 ]))
#Another way to turn numpy array into tensor:
print("3d NumPy array input: {}".format(c.get_shape()))
bb = tf.convert_to_tensor(b, dtype = tf.float32) #turns numpy array into a real te
print (type(bb))          #you can always as for the type
```

```
[[1, 2, 3], [4, 5, 6]]
<type 'list'>
<type 'numpy.ndarray'>
Python List input: (2, 3)
3d NumPy array input: (2, 2, 3)
<class 'tensorflow.python.framework.ops.Tensor'>
```

The get_shape() method returns the shape of the tensor as a tuple of integers. The number of integers corresponds to the number of dimensions of the tensor, and each integer is the number of array entries along that dimension. For example, a shape of $(2, 3)$ indicates a matrix, since it has two integers, and the size of the matrix is 2×3.

Variables, Placeholders:

Variables can maintain a fixed state in the graph. This is important because their current state might influence how they change in the following iteration. Like other Tensors, Variables can be used as input for other operations in the graph.

Using Variables is done in two stages. First we call the tf.Variable() function in order to create a Variable and define what value it will be initialized with. We then have to explicitly perform an initialization operation by running the session with the tf.global_variables_initializer() method, which allocates the memory for the Variable and sets its initial values.

Like other Tensor objects, Variables are computed only when the model runs, as we can see in the following example:

In [37]:
```
init_val = tf.random_normal((1,5),0,1)  #random normal distribution that gives yo
                                        #with standard dev of 1
var = tf.Variable(init_val, name='var') #this is a constructor, it will be named
print("pre run: \n{}".format(var)) #\n is for new line

init = tf.global_variables_initializer()  #init is an operation that initializes
#but with this line you create a node called init, which is why you have to open
with tf.Session() as sess:
    sess.run(init)
    # print (var) everything that was supposed ot be initialized was, but the var
    post_var = sess.run(var)

print("\npost run: \n{}".format(post_var))
```

```
pre run:
<tf.Variable 'var:0' shape=(1, 5) dtype=float32_ref>

post run:
[[-0.30407318 -0.92795068  0.5859226   0.45946285  2.25937557]]
```

TensorFlow has designated built-in structures for feeding input values. These structures are called placeholders. Placeholders can be thought of as empty Variables that will be filled with data later on. We use them by first constructing our graph and only when it is executed feeding them with the input data.

Placeholders have an optional shape argument. If a shape is not fed or is passed as None, then the placeholder can be fed with data of any size. It is common to use None for the dimension of a matrix that corresponds to the number of samples (usually rows), while having the length of the features (usually columns) fixed:

In [40]:
```
ph = tf.placeholder(tf.float32,shape=(None,10))
```

Whenever we define a placeholder, we must feed it with some input values or else an exception will be thrown. The input data is passed to the session.run() method as a dictionary, where each key corresponds to a placeholder variable name, and the matching values are the data values given in the form of a list or a NumPy array:

In [41]:
```
sess.run(s,feed_dict={x: X_data,w: w_data})
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-41-d46a87107525> in <module>()
----> 1 sess.run(s,feed_dict={x: X_data,w: w_data})

NameError: name 'X_data' is not defined
```

Let’s see how it looks with another graph example, this time with placeholders for two inputs: a matrix x and a vector w. These inputs are matrix-multiplied to create a five-unit vector xw and added with a constant vector b filled with the value -1. Finally, the variable s takes the maximum value of that vector by using the tf.reduce_max() operation. The word reduce is used because we are reducing a five-unit vector to a single scalar:

In [42]:
```
import tensorflow as tf
import numpy as np
```

In [43]:
```

x_data = np.random.randn(5,10) #random normal matix of shape 5x10
w_data = np.random.randn(10,1) #vector

with tf.Graph().as_default():
    x = tf.placeholder(tf.float32,shape=(5,10))  #x is placeholder for variable
    w = tf.placeholder(tf.float32,shape=(10,1))
    b = tf.fill((5,1),-1.0) #matrix with one column and 5 rows filled with -1
    xw = tf.matmul(x,w) #matmul means matrix multiply matrix x and vector w

    xwb = xw + b   #a vertical vctor, product of matrix x and vector w plus vecto
    s = tf.reduce_max(xwb)  #computes maximum across dimensions of a tensor
    #up to here, this is all symbolic math -- there are no values
    #Now we want to calculate s
    with tf.Session() as sess:
        outs = sess.run(s,feed_dict={x: x_data,w: w_data}) #feed dictionary (in l
        #x is key, x_data is value
        #Finally now, we're giving values for x and w using feed dictionary
        print (sess.run(b))

print("outs = {}".format(outs))
```

```
[[-1.]
 [-1.]
 [-1.]
 [-1.]
 [-1.]]
outs = 2.84774780273
```

In [ ]: