Lecture note 2: TensorFlow Ops

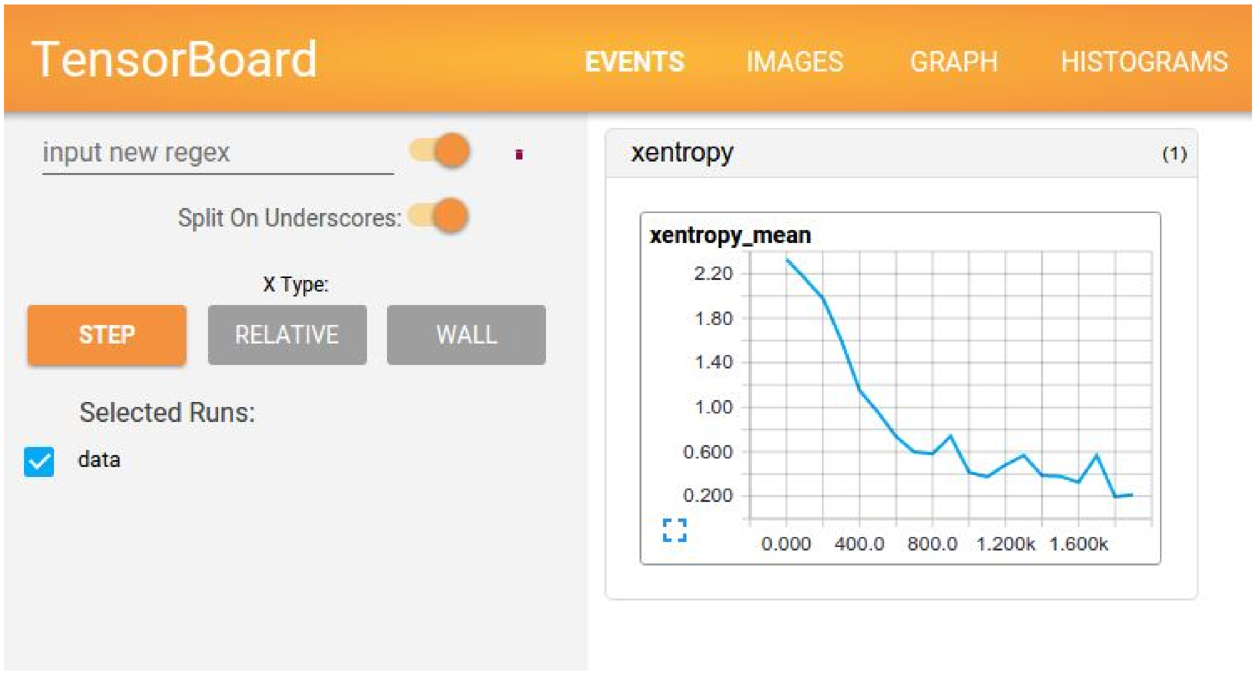
"CS 20SI: TensorFlow for Deep Learning Research" (cs20si.stanford.edu)
Prepared by Chip Huyen (huyenn@stanford.edu)
Reviewed by Danijar Hafner

1. Fun with TensorBoard

In TensorFlow, you collectively call constants, variables, operators as ops. TensorFlow is not just a software library, but a suite of softwares that include TensorFlow, TensorBoard, and TensorServing. To make the most out of TensorFlow, we should know how to use all of the above in conjunction with one another. In this lecture, we will first cover TensorBoard.

TensorBoard is graph visualization software included with any standard TensorFlow installation. In Google's own words: "The computations you'll use TensorFlow for - like training a massive deep neural network - can be complex and confusing. To make it easier to understand, debug, and optimize TensorFlow programs, we've included a suite of visualization tools called TensorBoard."

TensorBoard, when fully configured, will look something like this. Image from TensorBoard's website.



When a user perform certain operations in a TensorBoard-activated TensorFlow program, these operations are exported to an event file. TensorBoard is able to convert these event files to graphs that can give insight into a model's behavior. Learning to use TensorBoard early and often will make working with TensorFlow that much more enjoyable and productive.

Let's write your first TensorFlow program and visualize it with TensorBoard.

```
import tensorflow as tf
a = tf.constant(2)
b = tf.constant(3)
x = tf.add(a, b)
with tf.Session() as sess:
	print sess.run(x)
```

To activate TensorBoard on this program, add this line after you've built your graph, right before running the train loop.

```
writer = tf.summary.FileWriter(logs_dir, sess.graph)
```

The line above is to create a writer object to write operations to the event file, stored in the folder logs_dir. You can choose logs_dir to be something such as './graphs'.

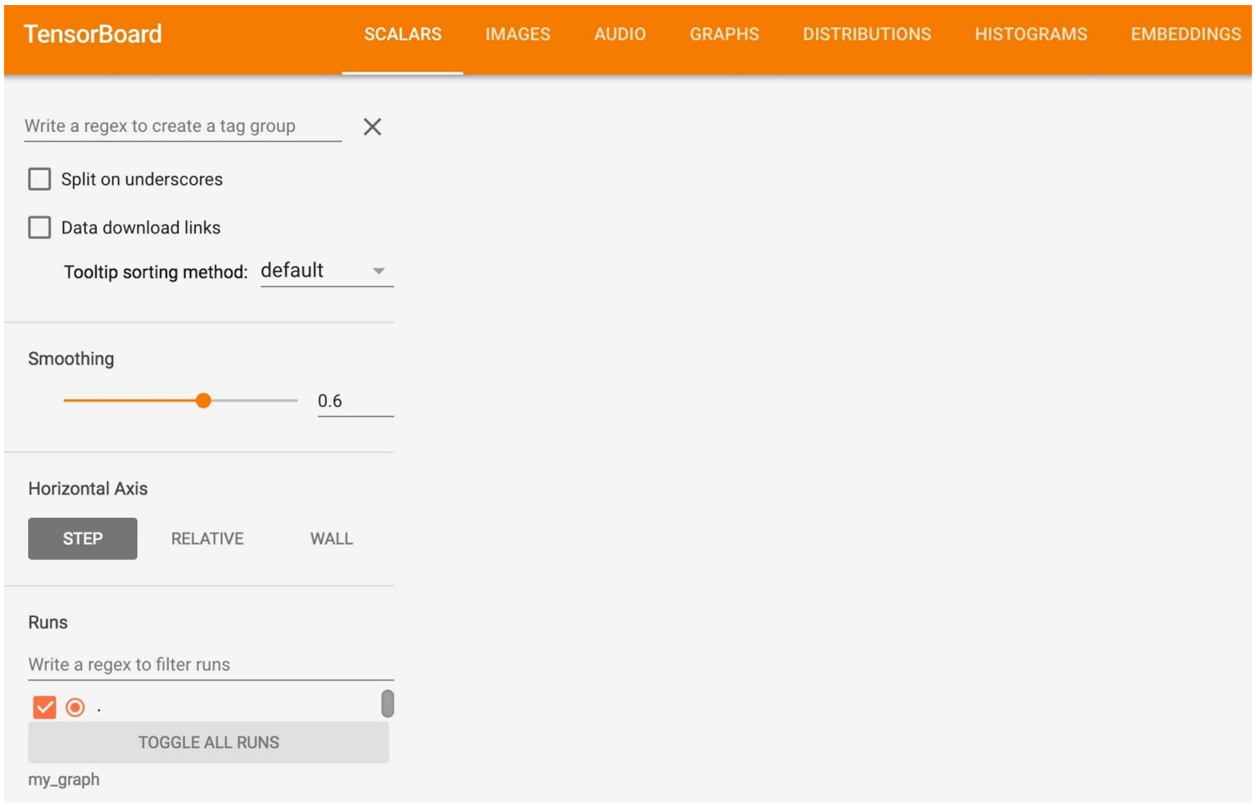
```
import tensorflow as tf
a = tf.constant(2)
b = tf.constant(3)
x = tf.add(a, b)

with tf.Session() as sess:
	writer = tf.summary.FileWriter('./graphs', sess.graph)
	print sess.run(x)

# close the writer when you're done using it
writer.close()
```

Next, go to Terminal, run the program. Make sure that your present working directory is the same as where you ran your Python code.

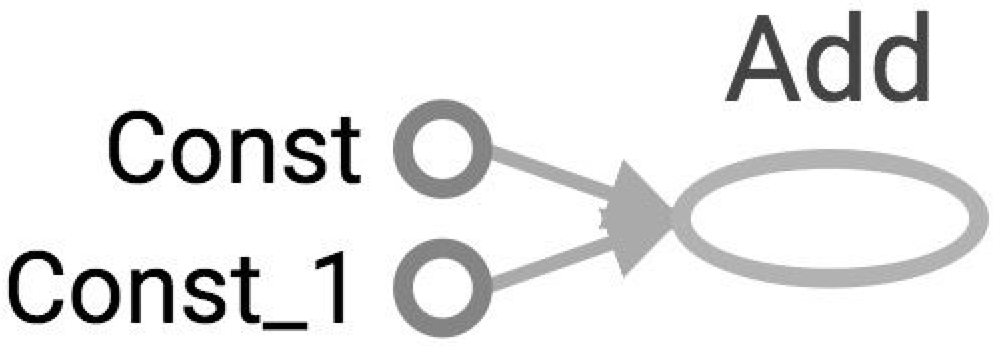
```
$ python [yourprogram.py]
$ tensorboard --logdir="./graphs"
```

Open your browser and go to http://localhost:6006/ (or the link you get back after running tensorboard command).

Go to the tab graph and you will see something like this:



Go to the Graph tab and I can see the graph with 3 nodes:

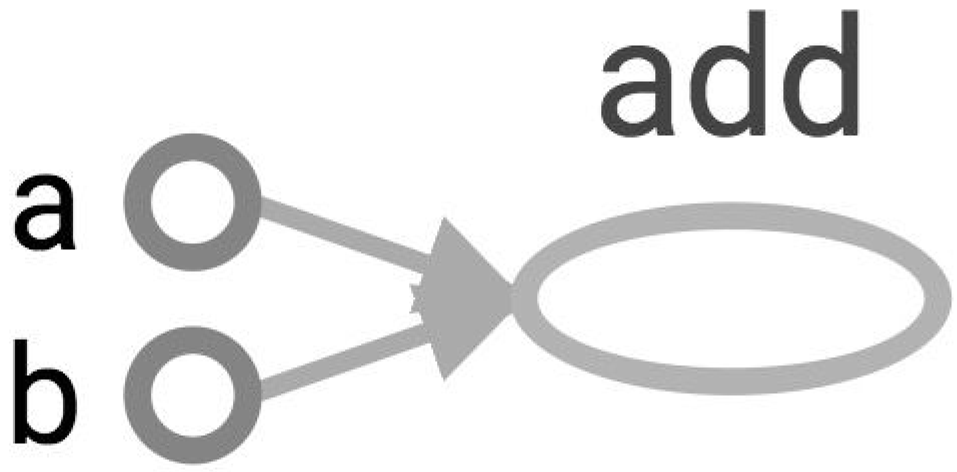


```
a = tf.constant(2)
b = tf.constant(3)
x = tf.add(a, b)
```

"Const" and "Const_1" correspond to a and b, and the node "Add" corresponds to x. The names we give them (a, b, and x) are for us to access them when we need. They mean nothing for the internal TensorFlow. To make TensorBoard display the names of your ops, you have to explicitly name them.

```
a = tf.constant([2, 2], name="a")
b = tf.constant([3, 6], name="b")
x = tf.add(a, b, name="add")
```

Now if you run TensorBoard again, you see this graph:



The graph itself defines the ops and dependencies, but not displays the values. It only cares about the values when we run the session with some values to fetch in mind. Quick reminder if you've forgot:

```
tf.Session.run(fetches, feed_dict=None, options=None, run_metadata=None)
```

Note: If you've run your code several times, there will be multiple event files in '~/dev/cs20si/graphs/lecture01', TF will show only the latest graph and display the warning of multiple event files. To get rid of the warning, delete all the event files you no longer need.

2. Constant types

Link to documentation: https://www.tensorflow.org/api_docs/python/constant_op/

You can create constants of scalar or tensor values.

```
tf.constant(value, dtype=None, shape=None, name='Const', verify_shape=False)
```

```
# constant of 1d tensor (vector)
a = tf.constant([2, 2], name="vector")

# constant of 2x2 tensor (matrix)
b = tf.constant([[0, 1], [2, 3]], name="b")
```

You can create tensors whose elements are of a specific value

Note the similarity to numpy.zeros, numpy.zeros_like, numpy.ones, numpy.ones_like.

```
tf.zeros(shape, dtype=tf.float32, name=None)
# create a tensor of shape and all elements are zeros

tf.zeros([2, 3], tf.int32) ==> [[0, 0, 0], [0, 0, 0]]
```

```
tf.zeros_like(input_tensor, dtype=None, name=None, optimize=True)
# create a tensor of shape and type (unless type is specified) as the input_tensor
but all elements are zeros.

# input_tensor is [0, 1], [2, 3], [4, 5]]
tf.zeros_like(input_tensor) ==> [[0, 0], [0, 0], [0, 0]]
```

```
tf.ones(shape, dtype=tf.float32, name=None)
# create a tensor of shape and all elements are ones

tf.ones([2, 3], tf.int32) ==> [[1, 1, 1], [1, 1, 1]]
```

```
tf.ones_like(input_tensor, dtype=None, name=None, optimize=True)
# create a tensor of shape and type (unless type is specified) as the input_tensor
but all elements are ones.

# input_tensor is [0, 1], [2, 3], [4, 5]]
tf.ones_like(input_tensor) ==> [[1, 1], [1, 1], [1, 1]]
```

```
tf.fill(dims, value, name=None)
# create a tensor filled with a scalar value.

tf.ones([2, 3], 8) ==> [[8, 8, 8], [8, 8, 8]]
```

You can create constants that are sequences

```
tf.linspace(start, stop, num, name=None)
# create a sequence of num evenly-spaced values are generated beginning at start. If
num > 1, the values in the sequence increase by stop - start / num - 1, so that the
last one is exactly stop.
# start, stop, num must be scalars
# comparable to but slightly different from numpy.linspace
# numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)

tf.linspace(10.0, 13.0, 4, name="linspace") ==> [10.0 11.0 12.0 13.0]
```

```
tf.range(start, limit=None, delta=1, dtype=None, name='range')
# create a sequence of numbers that begins at start and extends by increments of
delta up to but not including limit
# slight different from range in Python

# 'start' is 3, 'limit' is 18, 'delta' is 3
tf.range(start, limit, delta) ==> [3, 6, 9, 12, 15]

# 'start' is 3, 'limit' is 1,  'delta' is -0.5
tf.range(start, limit, delta) ==> [3, 2.5, 2, 1.5]

# 'limit' is 5
tf.range(limit) ==> [0, 1, 2, 3, 4]
```

Note that unlike NumPy or Python sequences, TensorFlow sequences are not iterable.

```
for _ in np.linspace(0, 10, 4): # OK
for _ in tf.linspace(0, 10, 4): # TypeError("'Tensor' object is not iterable.")

for _ in range(4): # OK
for _ in tf.range(4): # TypeError("'Tensor' object is not iterable.")
```

You can also generate random constants from certain distributions.

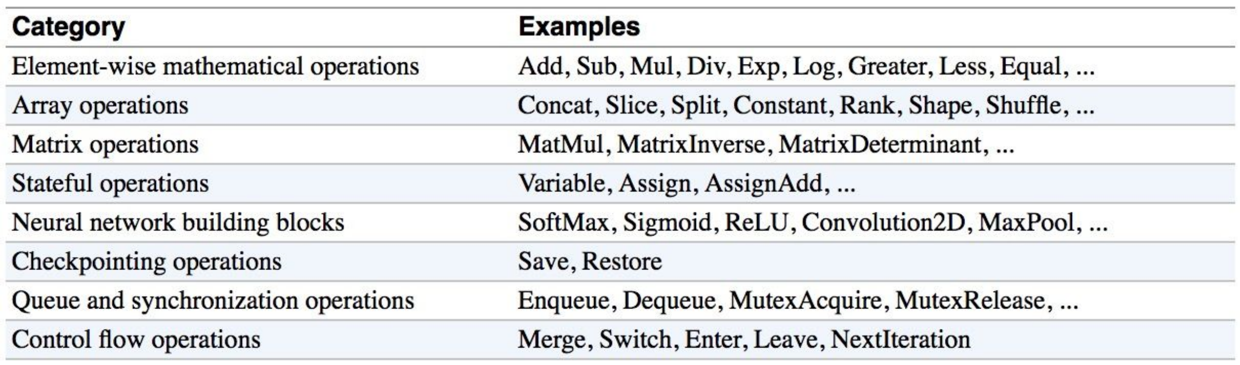
```
tf.random_normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32, seed=None, name=None)
tf.truncated_normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32, seed=None,
name=None)
tf.random_uniform(shape, minval=0, maxval=None, dtype=tf.float32, seed=None,
name=None)
tf.random_shuffle(value, seed=None, name=None)
tf.random_crop(value, size, seed=None, name=None)
tf.multinomial(logits, num_samples, seed=None, name=None)
tf.random_gamma(shape, alpha, beta=None, dtype=tf.float32, seed=None, name=None)
```

3. Math Operations

TensorFlow math ops are pretty standard, quite similar to NumPy. Visit https://www.tensorflow.org/api_docs/python/math_ops/arithmetic_operators for more because listing math ops is boring.

```
a = tf.constant([3, 6])
b = tf.constant([2, 2])
tf.add(a, b) # >> [5 8]
tf.add_n([a, b, b]) # >> [7 10]. Equivalent to a + b + b
tf.mul(a, b) # >> [6 12] because mul is element wise
tf.matmul(a, b) # >> ValueError
tf.matmul(tf.reshape(a, shape=[1, 2]), tf.reshape(b, shape=[2, 1])) # >> [[18]]
tf.div(a, b) # >> [1 3]
tf.mod(a, b) # >> [1 0]
```

Below is the table of ops in Python, courtesy of the authors of the book "Fundamentals of Deep Learning".



Category	Examples
Element-wise mathematical operations	Add, Sub, Mul, Div, Exp, Log, Greater, Less, Equal, ...
Array operations	Concat, Slice, Split, Constant, Rank, Shape, Shuffle, ...
Matrix operations	MatMul, MatrixInverse, MatrixDeterminant, ...
Stateful operations	Variable, Assign, AssignAdd, ...
Neural network building blocks	SoftMax, Sigmoid, ReLU, Convolution2D, MaxPool, ...
Checkpointing operations	Save, Restore
Queue and synchronization operations	Enqueue, Dequeue, MutexAcquire, MutexRelease, ...
Control flow operations	Merge, Switch, Enter, Leave, NextIteration

4. Data Types

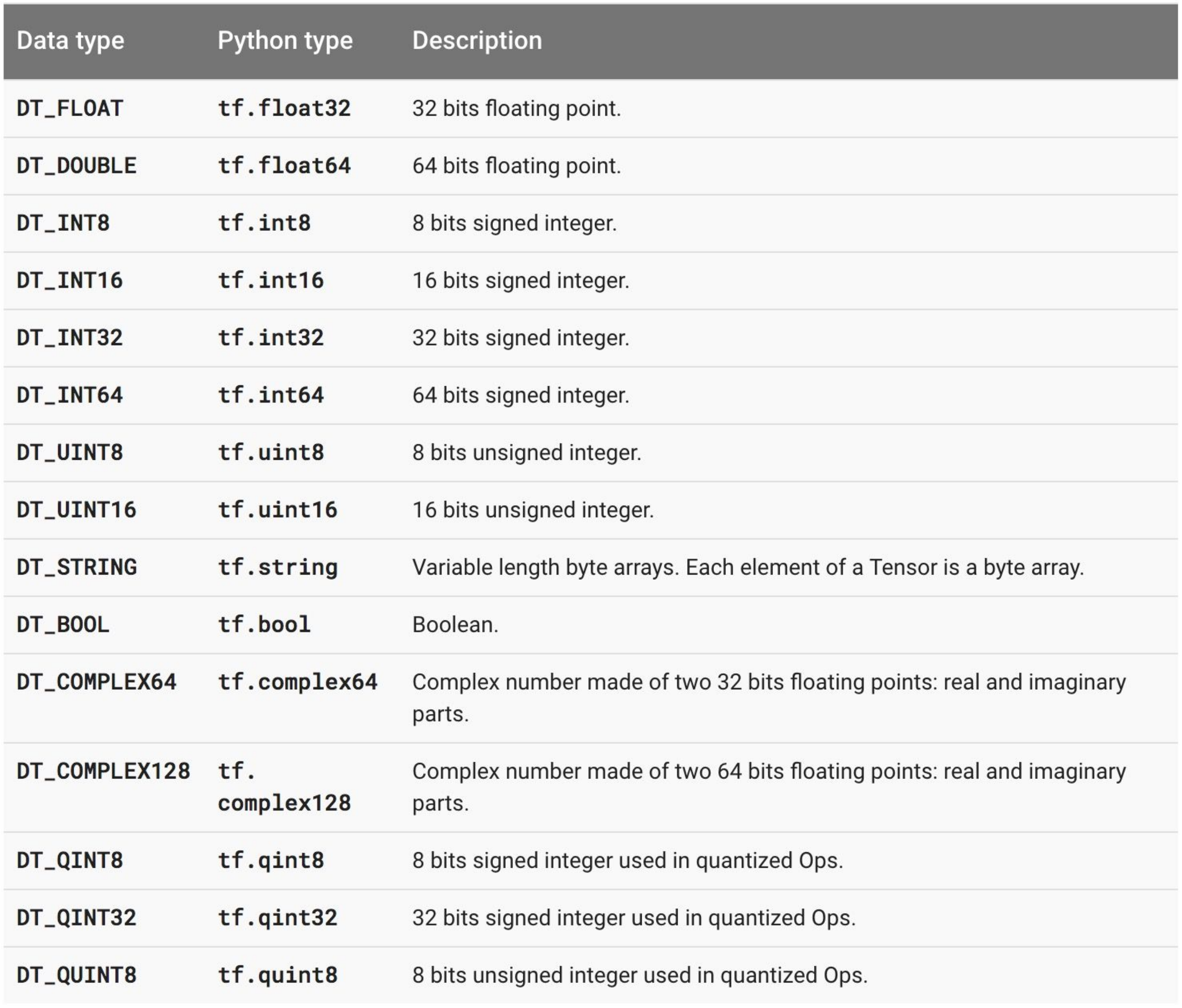
Python Native Types

TensorFlow takes in Python native types such as Python boolean values, numeric values (integers, floats), and strings. Single values will be converted to 0-d tensors (or scalars), lists of values will be converted to 1-d tensors (vectors), lists of lists of values will be converted to 2-d tensors (matrices), and so on. Example below is adapted and modified from the book “TensorFlow for Machine Intelligence”.

```
t_0 = 19 # Treated as a 0-d tensor, or "scalar"
tf.zeros_like(t_0) # ==> 0
tf.ones_like(t_0) # ==> 1

t_1 = [b"apple", b"peach", b"grape"] # treated as a 1-d tensor, or "vector"
tf.zeros_like(t_1) # ==> ['' '' '']
tf.ones_like(t_1) # ==> TypeError: Expected string, got 1 of type 'int' instead.

t_2 = [[True, False, False],
       [False, False, True],
       [False, True, False]] # treated as a 2-d tensor, or "matrix"

tf.zeros_like(t_2) # ==> 2x2 tensor, all elements are False
tf.ones_like(t_2) # ==> 2x2 tensor, all elements are True
```

TensorFlow Native Types

Like NumPy, TensorFlow also its own data types as you've seen tf.int32, tf.float32.Below is a list of current TensorFlow data types, taken from TensorFlow's official documentation.



Data type	Python type	Description
`DT_FLOAT`	`tf.float32`	32 bits floating point.
`DT_DOUBLE`	`tf.float64`	64 bits floating point.
`DT_INT8`	`tf.int8`	8 bits signed integer.
`DT_INT16`	`tf.int16`	16 bits signed integer.
`DT_INT32`	`tf.int32`	32 bits signed integer.
`DT_INT64`	`tf.int64`	64 bits signed integer.
`DT_UINT8`	`tf.uint8`	8 bits unsigned integer.
`DT_UINT16`	`tf.uint16`	16 bits unsigned integer.
`DT_STRING`	`tf.string`	Variable length byte arrays. Each element of a Tensor is a byte array.
`DT_BOOL`	`tf.bool`	Boolean.
`DT_COMPLEX64`	`tf.complex64`	Complex number made of two 32 bits floating points: real and imaginary parts.
`DT_COMPLEX128`	`tf.complex128`	Complex number made of two 64 bits floating points: real and imaginary parts.
`DT_QINT8`	`tf.qint8`	8 bits signed integer used in quantized Ops.
`DT_QINT32`	`tf.qint32`	32 bits signed integer used in quantized Ops.
`DT_QUINT8`	`tf.quint8`	8 bits unsigned integer used in quantized Ops.

NumPy Data Types

By now, you've probably noticed the similarity between NumPy and TensorFlow. TensorFlow was designed to integrate seamlessly with Numpy, the package that has become the *lingua franca* of data science.

TensorFlow's data types are based on those of NumPy; in fact, np.int32 == tf.int32 returns True. You can pass NumPy types to TensorFlow ops.

Example:

```
tf.ones([2, 2], np.float32) ==> [[1.0 1.0], [1.0 1.0]]
```

Remember our best friend **tf.Session.run(fetches)**? If the requested object is a Tensor, the output of will be a NumPy array.

TL;DR: Most of the times, you can use TensorFlow types and NumPy types interchangeably.

Note 1: There is a catch 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. TensorFlow can still import string arrays from NumPy perfectly fine -- just don't specify a dtype in NumPy!

Note 2: Both TensorFlow and NumPy are n-d array libraries. NumPy supports ndarray, but doesn't offer methods to create tensor functions and automatically compute derivatives, nor GPU support. So TensorFlow still wins!

Note 3: Using Python types to specify TensorFlow objects is quick and easy, and it is useful for prototyping ideas. However, there is an important pitfall in doing it this way. Python types lack the ability to explicitly state the data type, but TensorFlow's data types are more specific. For example, all integers are the same type, but TensorFlow has 8-bit, 16-bit, 32-bit, and 64-bit integers available. Therefore, if you use a Python type, TensorFlow has to infer which data type you mean.

It's possible 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 create hand-defined Tensor objects as NumPy arrays. However, always use TensorFlow types when possible, because both TensorFlow and NumPy can evolve to a point that such compatibility no longer exists.

5. Variables

Constants have been fun, but I think now you guys are old enough to know about variables. The difference between a constant and a variable:

1. A constant is constant. A variable can be assigned to, its value can be changed.
2. A constant's value is stored in the graph and its value is replicated wherever the graph is loaded. A variable is stored separately, and may live on a parameter server.

Point 2 basically means that constants are stored in the graph definition. When constants are memory expensive, it will be slow each time you have to load the graph. To see the graph's definition and what's stored in the graph's definition, simply print out the graph's protobuf. Protobuf stands for protocol buffer, "Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data – think XML, but smaller, faster, and simpler."

```
import tensorflow as tf

my_const = tf.constant([1.0, 2.0], name="my_const")
print tf.get_default_graph().as_graph_def()
```

Output:

```
node {
  name: "my_const"
  op: "Const"
  attr {
    key: "dtype"
    value {
      type: DT_FLOAT
    }
  }
  attr {
    key: "value"
    value {
      tensor {
        dtype: DT_FLOAT
        tensor_shape {
          dim {
            size: 2
          }
        }
        tensor_content: "\000\000\200?\000\000\000@"
      }
    }
  }
}
versions {
  producer: 17
}
```

Declare variables

To declare a variable, you create an instance of the class tf.Variable. Note that it's tf.constant but tf.Variable and not tf.variable because tf.constant is an op, while tf.Variable is a class.

```
#create variable a with scalar value
a = tf.Variable(2, name="scalar")

#create variable b as a vector
b = tf.Variable([2, 3], name="vector")
```

```
#create variable c as a 2x2 matrix
c = tf.Variable([[0, 1], [2, 3]], name="matrix")

# create variable W as 784 x 10 tensor, filled with zeros
W = tf.Variable(tf.zeros([784,10]))
```

tf.Variable holds several ops:

```
x = tf.Variable(...)

x.initializer # init
x.value() # read op
x.assign(...) # write op
x.assign_add(...)
# and more
```

You have to initialize variables before using them. If you try to evaluate the variables before initializing them you'll run into FailedPreconditionError: Attempting to use uninitialized value tensor.

The easiest way is initializing all variables at once using: tf.global_variables_initializer()

```
init = tf.global_variables_initializer()

with tf.Session() as sess:
       tf.run(init)
```

Note that you use tf.run() to run the initializer, not fetching any value.

To initialize only a subset of variables, you use tf.variables_initializer() with a list of variables you want to initialize:

```
init_ab = tf.variables_initializer([a, b], name="init_ab")
with tf.Session() as sess:
       tf.run(init_ab)
```

You can also initialize each variable separately using tf.Variable.initializer

```
# create variable W as 784 x 10 tensor, filled with zeros
W = tf.Variable(tf.zeros([784,10]))
with tf.Session() as sess:
      tf.run(W.initializer)
```

Another way to initialize a variable is to restore it from a save file. We will talk about in a few weeks.

Evaluate values of variables

If we print the initialized variable, we only see the tensor object.

```
# W is a random 700 x 100 variable object
W = tf.Variable(tf.truncated_normal([700, 10]))
with tf.Session() as sess:
      sess.run(W.initializer)
      print W

>> Tensor("Variable/read:0", shape=(700, 10), dtype=float32)
```

To get the value of a variable, we need to evaluate it using eval()

```
# W is a random 700 x 100 variable object
W = tf.Variable(tf.truncated_normal([700, 10]))
with tf.Session() as sess:
      sess.run(W.initializer)
      print W.eval()

>> [[-0.76781619 -0.67020458  1.15333688 ..., -0.98434633 -1.25692499
  -0.90904623]
 [-0.36763489 -0.65037876 -1.52936983 ...,  0.19320194 -0.38379928
   0.44387451]
 [ 0.12510735 -0.82649058  0.4321366  ..., -0.3816964   0.70466036
   1.33211911]
 ...,
 [ 0.9203397  -0.99590844  0.76853162 ..., -0.74290705  0.37568584
   0.64072722]
 [-0.12753558  0.52571583  1.03265858 ...,  0.59978199 -0.91293705
  -0.02646019]
 [ 0.19076447 -0.62968266 -1.97970271 ..., -1.48389161  0.68170643
   1.46369624]]
```

Assign values to variables

We can assign a value to a variable using tf.Variable.assign()

```
W = tf.Variable(10)
W.assign(100)
with tf.Session() as sess:
	sess.run(W.initializer)
	print W.eval() # >> 10
```

Why 10 and not 100? W.assign(100) doesn't assign the value 100 to W, but instead create an assign op to do that. For this op to take effect, we have to run this op in session.

```
W = tf.Variable(10)
assign_op = W.assign(100)
with tf.Session() as sess:
	sess.run(assign_op)
	print W.eval() # >> 100
```

Note that we don't have initialize W in this case, because assign() does it for us. In fact, initializer op is the assign op that assigns the variable's initial value to the variable itself.

```
# in the source code
self._initializer_op = state_ops.assign(self._variable, self._initial_value,
                                        validate_shape=validate_shape).op
```

Interesting example:

```
# create a variable whose original value is 2
a = tf.Variable(2, name="scalar")

# assign a * 2 to a and call that op a_times_two
a_times_two = a.assign(a * 2)

init = tf.global_variables_initializer()

with tf.Session() as sess:
	sess.run(init)
	# have to initialize a, because a_times_two op depends on the value of a
	sess.run(a_times_two) # >> 4
	sess.run(a_times_two) # >> 8
	sess.run(a_times_two) # >> 16
```

TensorFlow assigns a*2 to a every time a_times_two is fetched.

For simple incrementing and decrementing of variables, TensorFlow includes the tf.Variable.assign_add() and tf.Variable.assign_sub() methods. Unlike tf.Variable.assign(), tf.Variable.assign_add() and tf.Variable.assign_sub() don't initialize your variables for you because these ops depend on the initial values of the variable.

```
W = tf.Variable(10)

with tf.Session() as sess:
	sess.run(W.initializer)
	print sess.run(W.assign_add(10)) # >> 20
	print sess.run(W.assign_sub(2)) # >> 18
```

Because TensorFlow sessions maintain values separately, each Session can have its own current value for a variable defined in a graph.

```
W = tf.Variable(10)

sess1 = tf.Session()
sess2 = tf.Session()

sess1.run(W.initializer)
sess2.run(W.initializer)

print sess1.run(W.assign_add(10)) # >> 20
print sess2.run(W.assign_sub(2)) # >> 8

print sess1.run(W.assign_add(100)) # >> 120
print sess2.run(W.assign_sub(50)) # >> -42

sess1.close()
sess2.close()
```

You can, of course, declare a variable that depends on other variables.

Suppose you want to declare U = W * 2

```
# W is a random 700 x 100 tensor
W = tf.Variable(tf.truncated_normal([700, 10]))
U = tf.Variable(W * 2)
```

In this case, you should use initialized_value() to make sure that W is initialized before its value is used to initialize W.

```
U = tf.Variable(W.intialized_value() * 2)
```

Documentation on the class tf.Variable()

6. InteractiveSession

You sometimes see InteractiveSession instead of Session. The only difference is an InteractiveSession makes itself the default session so you can call run() or eval() without explicitly call the session. This is convenient in interactive shells and IPython notebooks, as it avoids having to pass an explicit Session object to run ops. However, it is complicated when you have multiple sessions to run.

```
sess = tf.InteractiveSession()
a = tf.constant(5.0)
b = tf.constant(6.0)
c = a * b
# We can just use 'c.eval()' without passing 'sess'
print(c.eval())
sess.close()
```

tf.InteractiveSession.close() closes an InteractiveSession.

tf.get_default_session() returns the default session for the current thread. The returned Session will be the innermost session on which a Session or Session.as_default() context has been entered.

7. Control Dependencies

Sometimes, we will have two independent ops but you'd like to specify which op should be run first, then you use tf.Graph.control_dependencies(control_inputs)

Example:

```
# your graph g have 5 ops: a, b, c, d, e
with g.control_dependencies([a, b, c]):
         # `d` and `e` will only run after `a`, `b`, and `c` have executed.
         d = ...
         e = ...
```

8. Placeholders and feed_dict

Remember from the lecture 1 that a TensorFlow program often has 2 phases:

Phase 1: assemble a graph
Phase 2: use a session to execute operations in the graph.

Therefore, we can assemble the graphs first without knowing the values needed for computation. This is equivalent to defining the function of x, y without knowing the values of x, y. For example, f(x, y) = x*2 + y.
x, y are placeholders for the actual values.

With the graph assembled, we, or our clients, can later supply their own data when they need to execute the computation.

To define a placeholder, we use:

```
tf.placeholder(dtype, shape=None, name=None)
```

Dtype is the required parameter that specifies of the data type of the value of the placeholder.

Shape specifies the shape of the tensor that can be accepted as actual value for the placeholder. shape=None means that tensors of any shape will be accepted. Using shape=None is easy to construct graphs, but nightmarish for debugging. You should always define the shape of your placeholders as detailed as possible.

You can also give your placeholder a name as you can any other op in TensorFlow.

More information on placeholders in the official documentation.

```
# create a placeholder of type float 32-bit, shape is a vector of 3 elements
a = tf.placeholder(tf.float32, shape=[3])

# create a constant of type float 32-bit, shape is a vector of 3 elements
b = tf.constant([5, 5, 5], tf.float32)

# use the placeholder as you would a constant or a variable
c = a + b  # Short for tf.add(a, b)

If we try to fetch c, we will run into error.

with tf.Session() as sess:
       print(sess.run(c))

>> NameError
```

This runs into an error because to compute c, we need the value of a, but a is just a placeholder without actual value. We have to first feed actual value into a.

```
with tf.Session() as sess:
       # feed [1, 2, 3] to placeholder a via the dict {a: [1, 2, 3]}
       # fetch value of c
       print(sess.run(c, {a: [1, 2, 3]}))

>> [6. 7. 8.]
```

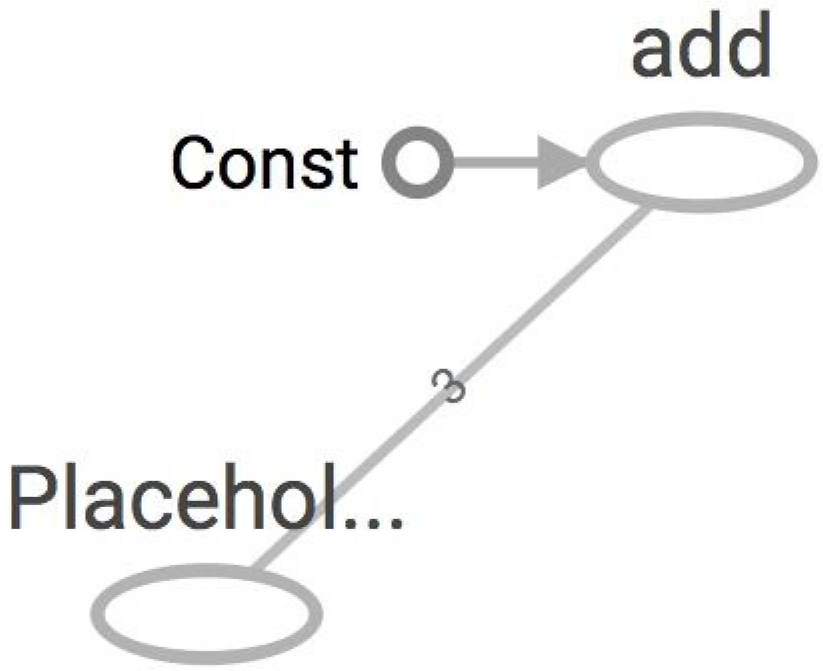
Let’s see how it looks in TensorBoard. Add

```
writer = tf.summary.FileWriter('./my_graph', sess.graph)
```

and type the following in the terminal:

```
$ tensorboard --logdir='my_graph'
```

As you can see, placeholder are treated like any other op. 3 is the shape of placeholder.



In the previous example, we feed one single value to the placeholder. What if we want to feed multiple data points to placeholder? It's a reasonable assumption since we often have to run computation through multiple data points in our training or testing set.

We can feed as any data points to the placeholder as we want by iterating through the data set and feed in the value one at a time.

```
with tf.Session() as sess:
       for a_value in list_of_a_values:
       print(sess.run(c, {a: a_value}))
```

You can feed values to tensors that aren't placeholders. Any tensors that are feedable can be fed. To check if a tensor is feedable or not, use:

```
tf.Graph.is_feedable(tensor)
```

```
# 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
```

feed_dict can be extremely useful to test your model. When you have a large graph and just want to test out certain parts, you can provide dummy values so TensorFlow won't waste time doing unnecessary computations.

9. The trap of lazy loading*

** I might have made this term up*

One of the most common TensorFlow non-bug bugs I see (and I used to commit) is what my friend Danijar and I call "lazy loading". Lazy loading is a term that refers to a programming pattern when you defer declaring/initializing an object until it is loaded. In the context of

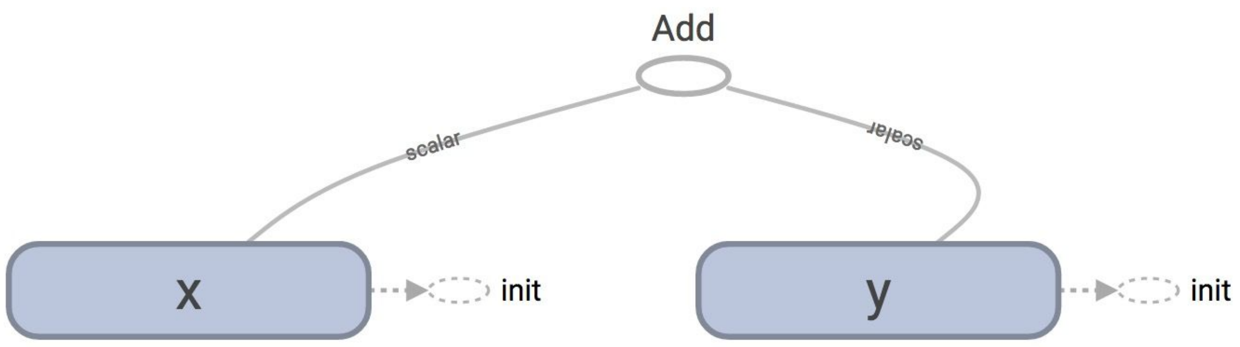
TensorFlow, it means you defer creating an op until you need to compute it. For example, this is normal loading: you create the op z when you assemble the graph.

```
x = tf.Variable(10, name='x')
y = tf.Variable(20, name='y')
z = tf.add(x, y)

with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        for _ in range(10):
                sess.run(z)
        writer.close()
```

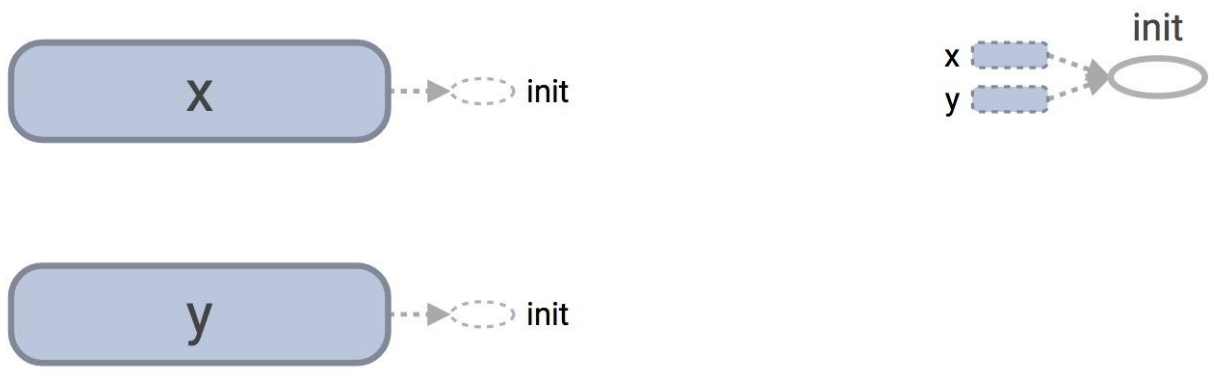
This is what happens when someone decides to be clever and use lazy loading to save one line of code:

```
x = tf.Variable(10, name='x')
y = tf.Variable(20, name='y')

with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        for _ in range(10):
                sess.run(tf.add(x, y)) # create the op add only when you need to compute it
        writer.close()
```

Let's see the graphs for them on TensorBoard.
Normal loading graph looks just like we expected.



Lazy loading



Well, the node "Add" is missing, which is understandable since we added the note "Add" after we've written the graph to FileWriter. This makes it harder to read the graph but it's not a bug. So, what's the big deal?

Let's look at the graph definition. Remember that to print out the graph definition, we use:

```
print tf.get_default_graph().as_graph_def()
```

The protobuf for the graph in normal loading has only 1 node "Add":

```
node {
  name: "Add"
  op: "Add"
  input: "x/read"
  input: "y/read"
  attr {
    key: "T"
    value {
      type: DT_INT32
    }
  }
}
```

On the other hand, the protobuf for the graph in lazy loading has 10 copies of the node "Add". It adds a new node "Add" every time you want to compute z!

```
node {
  name: "Add"
  op: "Add"
  ...
}
node {
  name: "Add_9"
  op: "Add"
  ...
}
```

You probably think: “This is stupid. Why would I want to compute the same value more than once?” and think that it’s a bug that nobody will ever commit. It happens more often than you think. For example, you might want to compute the same loss function or make some prediction after a certain number of training samples. Before you know it, you’ve computed it for thousands of times, and added thousands of unnecessary nodes to your graph. Your graph definition becomes bloated, slow to load and expensive to pass around.

There are two ways to avoid this bug. First, always separate the definition of ops and their execution when you can. But when it is not possible because you want to group related ops into classes, you can use Python property to ensure that your function is only loaded once when it’s first called. This is not a Python course so I won’t dig into how to do it. But if you want to know, check out this wonderful blog post by Danijar Hafner.