TensorFlow Tutorial

CS 419(M) : Introduction to Machine Learning

Prof. Sunita Sarawagi

Divyansh Pareek and Karan Taneja

TensorFlow : What is it ?

- Open source software library
- Intensely optimized for fast Numerical computations
- Widely used for Machine Learning / Deep Learning

Getting Started

- Installation
  - OS dependent instructions available on https://www.tensorflow.org/install/
  - Install CPU version (instead of GPU) if you’re using a laptop
- Google Colab
  - Jupyter notebook environment
  - Good for getting off-the-ground
  - Go to https://colab.research.google.com and start typing!

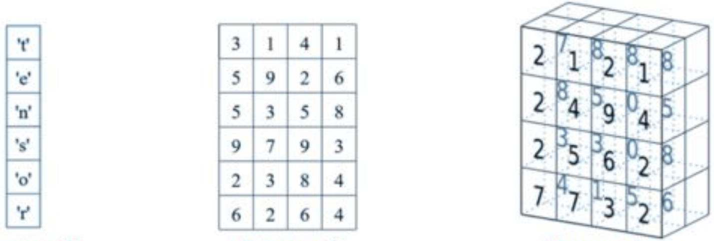
We’ll not cover the basics of python, we assume you are somewhat familiar with it.

Colab : Create a notebook

- Go to colab
- Create a python3 (Jupyter) notebook
- Go to *Edit > Notebook Settings*
    - Change Hardware Accelerator from None to GPU
- Click Connect to get connected to a machine in Google's datacenter
- Start typing!

But first … Tensor : What is it (in TF) ?

- Generalization of vectors and matrices to potentially higher dimensions
- Everything is a tensor!
    - A scalar is a tensor of dimension 0
    - A vector is a tensor of dimension 1
    - A matrix is a tensor of dimension 2

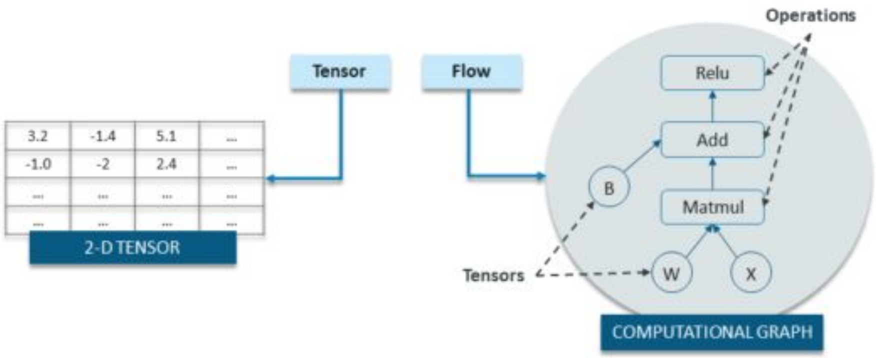


Tensor of dimension[1]   *Tensor of dimensions[2]*   *Tensor of dimensions[3]*

- Say you have N images, each of size H x W - represent that as a tensor of dimension N x H x W
    - Each entry would be a pixel value, say an *int* (base datatype) in the range [0, 256)
    - What about N videos, each of H x W spatial dimension and T units in time dimension ?
- Internally, TensorFlow represents tensors as n-dimensional arrays of base datatypes (tf.int32, tf.float64, etc)

TensorFlow = Tensor + Flow (Duh!)

- TensorFlow is quite literally the flow of tensors



- The idea of a **computational graph** is central to TensorFlow
  - We'll see this in more detail shortly

First TF Program : Adding two tensors

```
import tensorflow as tf

# Declare a tensor with : constant value = [1,2,3] | dimension = (3,) (inferred) | datatype = tf.float32 (inferred)
a = tf.constant([1.0,2.0,3.0])

# Declare a tensor with : constant value = [1,2,4] | dimension = (3,) (inferred) |  datatype = tf.float32 (inferred)
b = tf.constant([1.0,2.0,4.0])

# Add a & b
# c is a Tensor - the result of addition of tensors a & b
c = a + b # operator + is overloaded for tensors

with tf.Session() as sess:
  c_value = sess.run(c) # Could also have done : c_value = c.eval()
  print("Value of Node c : ", c_value)
  print("Attributes of Value of Node c : ", "Type:", type(c_value), " Shape:", c_value.shape)
  print("--- --- ---")
  print("Node c itself : ", c)
  print("Shape of Node c : ", c.shape)
  print("Dtype of Node c : ", c.dtype)
```

```
Value of Node c :  [2. 4. 7.]
Attributes of Value of Node c :  Type: <class 'numpy.ndarray'>  Shape: (3,)
--- --- ---
Node c itself :  Tensor("add_1:0", shape=(3,), dtype=float32)
Shape of Node c :  (3,)
Dtype of Node c :  <dtype: 'float32'>
```

Output of sess.run(c) is a np.ndarray

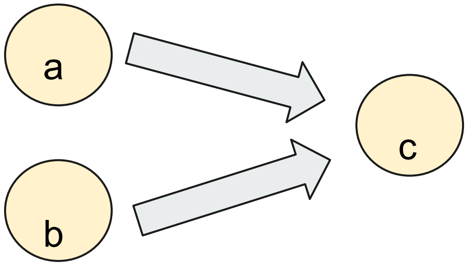
c itself is a tf.tensor

Computational Graph Model

- The fragment of the code before the initialization of tf.Session() was this

```
# Declare a tensor with : constant value = [1,2,3] | dimension = (3,) (inferred) | datatype = tf.float32 (inferred)
a = tf.constant([1.0,2.0,3.0])

# Declare a tensor with : constant value = [1,2,4] | dimension = (3,) (inferred) | datatype = tf.float32 (inferred)
b = tf.constant([1.0,2.0,4.0])

# Add a & b
# c is a Tensor - the result of addition of tensors a & b
c = a + b

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

- This defines a computation graph as follows



The arrows represent that the definition of c requires a & b (since c := a + b).
This computational graph is independent of the values that a & b hold (although here they are constants).
tf internally builds (and stores) this graph.

Computational Graph Model : tf.Session()

- Writing code in tf => defining such computation graphs (for whatever operations we want)
- UNLIKE normal python/numpy, this ONLY defines the computation graph
    - ie, it just defines all the operations on all the various tensors that we want
- Does NOT carry out any computation, just creates and stores a computation graph

- Then, instantiate a tf.Session()
- Call sess.run(tens)
    - Here sess is the name of the tf.Session() object we created
    - And tens is any tensor whose value we want to compute
- This call makes tf run a forward pass on the graph that it created, using actual values
    - This is the step where the computation actually happens

TensorFlow computations define a computation graph that has no numerical value until evaluated!

Computational Graph Model

- One of the central ideas of tensorflow

- **"TensorFlow programs are usually structured into a construction phase, that assembles a graph, and an execution phase that uses a session to execute ops in the graph."** - TensorFlow docs
- **"A Session object encapsulates the environment in which Operation objects are executed, and Tensor objects are evaluated."** - TensorFlow Docs
- All computations add nodes to global default graph (docs)
  - tf.Graph is the class handling such computation graphs
    - Lots of operations to modify and play around with computational graphs
    - To handle the default graph: *g = tf.get_default_graph()* ... then modify g
  - You can create multiple computation graphs
    - This will just multiple instances of tf.Graph

Computational Graph Model

```
import tensorflow as tf

# Declare a tensor with : constant value = [1,2,3] | dimension = (3,) (inferred) | datatype = tf.float32 (inferred)
a = tf.constant([1.0,2.0,3.0])

# Declare a tensor with : constant value = [1,2,4] | dimension = (3,) (inferred) |  datatype = tf.float32 (inferred)
b = tf.constant([1.0,2.0,4.0])

# Add a & b
# c is a Tensor - the result of addition of tensors a & b
c = a + b # operator + is overloaded for tensors

with tf.Session() as sess:
  c_value = sess.run(c) # Could also have done : c_value = c.eval()
  print("Value of Node c : ", c_value)
  print("Attributes of Value of Node c : ", "Type:", type(c_value), " Shape:", c_value.shape)
  print("--- --- ---")
  print("Node c itself : ", c)
  print("Shape of Node c : ", c.shape)
  print("Dtype of Node c : ", c.dtype)
```

Define a computation graph. At this point we can't see real value.

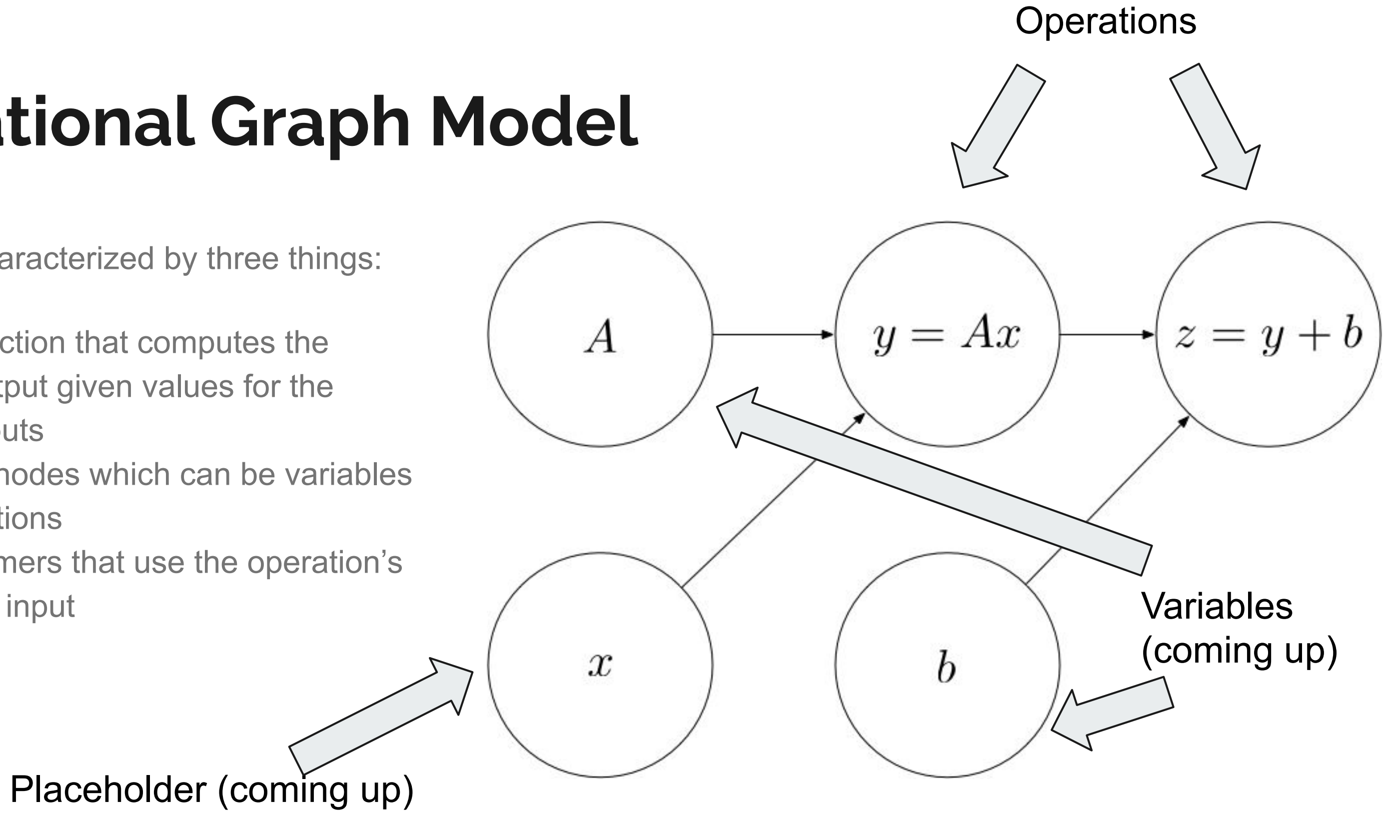
```
Value of Node c :  [2. 4. 7.]
Attributes of Value of Node c :  Type: <class 'numpy.ndarray'>  Shape: (3,)
--- --- ---
Node c itself :  Tensor("add_1:0", shape=(3,), dtype=float32)
Shape of Node c :  (3,)
Dtype of Node c :  <dtype: 'float32'>
```

Evaluate pre-defined comp graph. Graph eval must be preceded by a tf.Session() or tf.InteractiveSession() creation.

Computational Graph Model

Every operation is characterized by three things:

- A compute function that computes the operation's output given values for the operation's inputs
- A list of input_nodes which can be variables or other operations
- A list of consumers that use the operation's output as their input



Playing with Add : explicit shape def

```
[5] import tensorflow as tf

    # Declare a tensor with : constant value = [1,2,3] | dimension = (3,) (inferred) | datatype = tf.float32 (inferred)
    a = tf.constant([1.0,2.0,3.0])

    # Declare a tensor with : constant value = [1,2,4] | dimension = (3,1) (specified) |  datatype = tf.float32 (inferred)
    b = tf.constant([1.0,2.0,4.0], shape=(3,1)) # NO ERROR, but wasn't what we intended -- does elementwise addition
    # interpreted as [1,2,3] + [[1],[2],[4]] -- elementwise addition done on this

    # Add a & b
    # c is a Tensor - the result of addition of tensors a & b
    c = a + b # operator + is overloaded for tensors

    with tf.Session() as sess:
      c_value = sess.run(c) # Could also have done : c_value = c.eval()
      print("Value of Node c : ", c_value)
      print("Attributes of Value of Node c : ", "Type:", type(c_value), " Shape:", c_value.shape)
      print("--- --- ---")
      print("Node c itself : ", c)
      print("Shape of Node c : ", c.shape)
      print("Dtype of Node c : ", c.dtype)
```

```
Value of Node c :  [[2. 3. 4.]
 [3. 4. 5.]
 [5. 6. 7.]]
Attributes of Value of Node c :  Type: <class 'numpy.ndarray'>  Shape: (3, 3)
--- --- ---
Node c itself :  Tensor("add_3:0", shape=(3, 3), dtype=float32)
Shape of Node c :  (3, 3)
Dtype of Node c :  <dtype: 'float32'>
```

Playing with Add : explicit dtype def

```
[7] import tensorflow as tf

    # Declare a tensor with : constant value = [1,2,3] | dimension = (3,) (inferred) | datatype = tf.float32 (inferred)
    a = tf.constant([1.0,2.0,3.0])

    # Declare a tensor with : constant value = [1,2,4] | dimension = (3,) (specified) |  datatype = tf.float64 (specified)
    b = tf.constant([1.0,2.0,4.0], shape=(3,), dtype=tf.float64) # same as the original

    # Add a & b
    # c is a Tensor - the result of addition of tensors a & b
    c = a + b # operator + is overloaded for tensors

    with tf.Session() as sess:
      c_value = sess.run(c) # Could also have done : c_value = c.eval()
      print("Value of Node c : ", c_value)
      print("Attributes of Value of Node c : ", "Type:", type(c_value), " Shape:", c_value.shape)
      print("--- --- ---")
      print("Node c itself : ", c)
      print("Shape of Node c : ", c.shape)
      print("Dtype of Node c : ", c.dtype)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/op_def_library.py in _apply_op_helper(self, op_type_name, name, **keywords)
    509                 as_ref=input_arg.is_ref,
--> 510                 preferred_dtype=default_dtype)
    511           except TypeError as err:

/usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/ops.py in internal_convert_to_tensor(value, dtype, name, as_ref, preferred_dty
```

Playing with Add : using tf.add and node

```
[1] import tensorflow as tf

    # Declare a tensor with : constant value = [1,2,3] | dimension = (3,) (inferred) | datatype = tf.float32 (inferred)
    a = tf.constant([1.0,2.0,3.0])

    # Declare a tensor with : constant value = [1,2,4] | dimension = (3,) (inferred) | datatype = tf.float32 (inferred)
    b = tf.constant([1.0,2.0,4.0])

    # Add a & b
    # c is a Tensor - the result of addition of tensors a & b
    c = tf.add(a, b, name='MyAdd') # using tf.add instead of overloaded operator +

    with tf.Session() as sess:
      c_value = sess.run(c) # Could also have done : c_value = c.eval()
      print("Value of Node c : ", c_value)
      print("Attributes of Value of Node c : ", "Type:", type(c_value), " Shape:", c_value.shape)
      print("--- --- ---")
      print("Node c itself : ", c)
      print("Shape of Node c : ", c.shape)
      print("Dtype of Node c : ", c.dtype)
```

```
Value of Node c :  [2. 4. 7.]
Attributes of Value of Node c :  Type: <class 'numpy.ndarray'>  Shape: (3,)
--- --- ---
Node c itself :  Tensor("MyAdd:0", shape=(3,), dtype=float32)
Shape of Node c :  (3,)
Dtype of Node c :  <dtype: 'float32'>
```

Non-constant data : Placeholders and feed dictionaries

- What we did was pretty trivial, adding two constant vectors
- Let's say we want to still add two vectors, but now they are not constants
- How do we handle non-constant ? *tf.placeholder*!

- *tf.placeholder* defines dummy nodes that provide entry points for data to computational graph
- A *feed_dict* is a python dictionary mapping from *tf. placeholder* vars (or their names) to data (numpy arrays, lists, etc.)

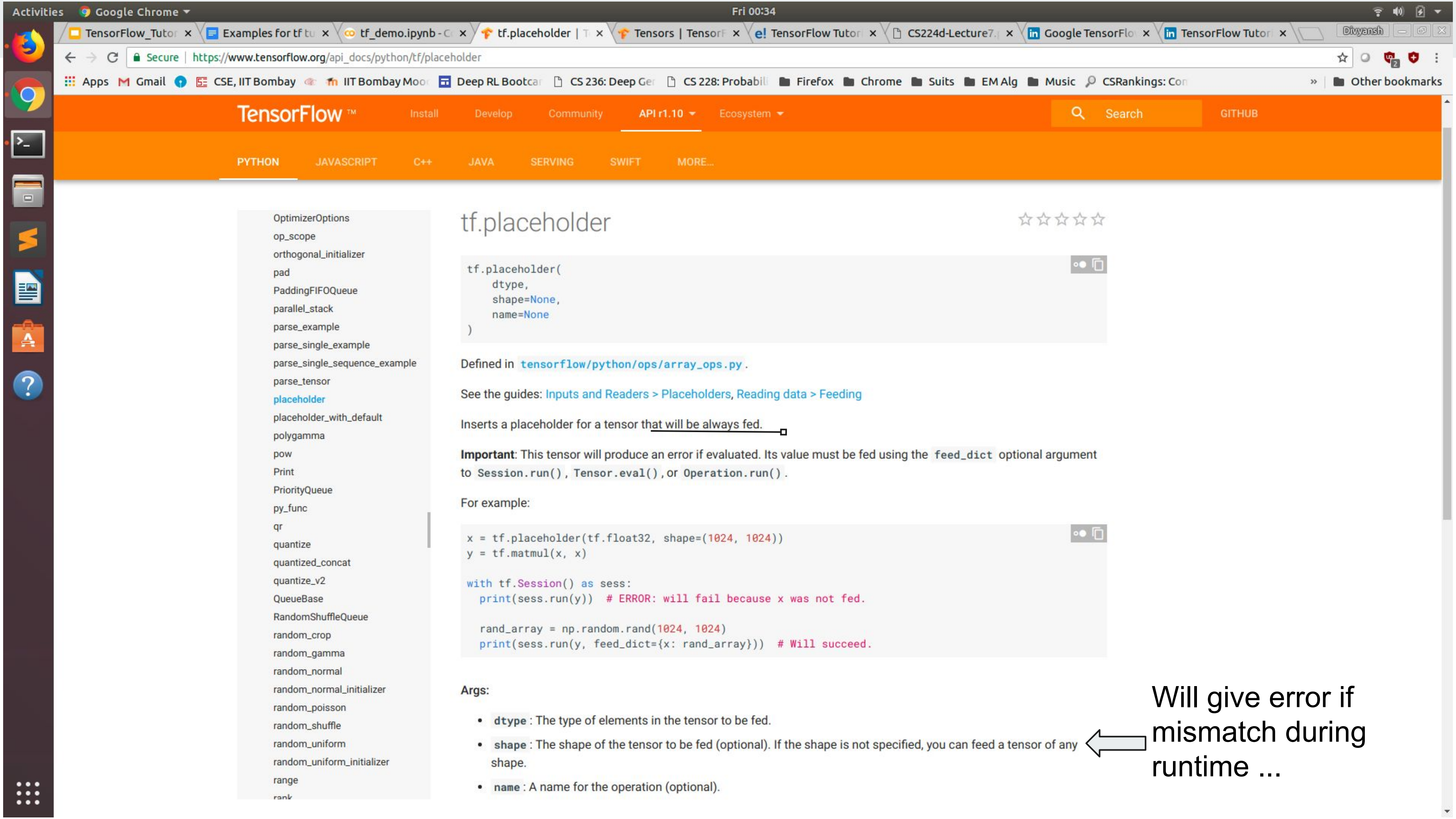
Placeholders and feed_dict : Example

```python
import tensorflow as tf

# Declare a placeholder for a : actual data provided at runtime
a = tf.placeholder(tf.float32)

# Declare a placeholder for b : actual data provided at runtime
b = tf.placeholder(tf.float32)

# Add a & b
# c is a Tensor - the result of addition of tensors a & b
c = a + b

with tf.Session() as sess:
  c_value = sess.run(c, feed_dict={a:[1.0,2.0,3.0], b:[1.0,2.0,4.0]}) # can give anything here ... in particular, read a dataset and give that here
  print("Value of Node c : ", c_value)
```

```
Value of Node c :  [2. 4. 7.]
```

Analogy: Can define the function f(x, y) = x*2 + y
without knowing value of x or y.
x, y are placeholders for the actual values.

Will give error if mismatch during computation (ie, forward pass on graph)


https://www.tensorflow.org/api_docs/python/tf/placeholder

TensorFlow™ Install Develop Community API r1.10 Ecosystem Search GITHUB

PYTHON JAVASCRIPT C++ JAVA SERVING SWIFT MORE...

OptimizerOptions
op_scope
orthogonal_initializer
pad
PaddingFIFOQueue
parallel_stack
parse_example
parse_single_example
parse_single_sequence_example
parse_tensor
placeholder
placeholder_with_default
polygamma
pow
Print
PriorityQueue
py_func
qr
quantize
quantized_concat
quantize_v2
QueueBase
RandomShuffleQueue
random_crop
random_gamma
random_normal
random_normal_initializer
random_poisson
random_shuffle
random_uniform
random_uniform_initializer
range
rank

tf.placeholder

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

Defined in `tensorflow/python/ops/array_ops.py`.

See the guides: Inputs and Readers > Placeholders, Reading data > Feeding

Inserts a placeholder for a tensor that will be always fed.

Important: This tensor will produce an error if evaluated. Its value must be fed using the `feed_dict` optional argument to `Session.run()`, `Tensor.eval()`, or `Operation.run()`.

For example:

```
x = tf.placeholder(tf.float32, shape=(1024, 1024))
y = tf.matmul(x, x)

with tf.Session() as sess:
  print(sess.run(y))  # ERROR: will fail because x was not fed.

  rand_array = np.random.rand(1024, 1024)
  print(sess.run(y, feed_dict={x: rand_array}))  # Will succeed.
```

Args:

- `dtype`: The type of elements in the tensor to be fed.
- `shape`: The shape of the tensor to be fed (optional). If the shape is not specified, you can feed a tensor of any shape.
- `name`: A name for the operation (optional).

Will give error if mismatch during runtime ...

Next Major piece of the puzzle : tf.Variable()

- We've seen
  - Computational graph model
  - Placeholders - special tensors that can be fed with input data
- With these, we can only do what all we could do in numpy
- The next major piece is *tf.Variable()*
  - **"A variable maintains state in the graph across calls to run()"** - Tensorflow Docs
  - **"When you train a model you use variables to hold and update parameters. Variables are in-memory buffers containing tensors"** - TensorFlow Docs

Variables

Why tf.constant but tf.Variable and not tf.variable?
tf.Variable is a class, but tf.constant is an op
tf.Variable holds several ops:
x = tf.Variable(...)
x.initializer # init op
x.value() # read op
x.assign(...) # write op

- Variable allows you to add such parameters or node to the graph that are trainable
  - i.e. the value can be modified over the period of a time
- **"The Variable() constructor requires an initial value for the variable, which can be a Tensor of any type and shape. The initial value defines the type and shape of the variable. After construction, the type and shape of the variable are fixed."** - TensorFlow Docs
- Example

```
import tensorflow as tf

# Create a variable.
w = tf.Variable(<initial-value>, name=<optional-name>)

# Use the variable in the graph like any Tensor.
y = tf.matmul(w, ...another variable or tensor...)

# The overloaded operators are available too.
z = tf.sigmoid(w + y)

# Assign a new value to the variable with `assign()` or a related method.
w.assign(w + 1.0)
w.assign_add(1.0)
```

Variables : Need to be initialized

```
import tensorflow as tf

# Declare a variable
w = tf.Variable(tf.zeros((2,2)), name="weight") # initialized by the 2 x 2 matrix of zeros

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

```
---------------------------------------------------------------------------
FailedPreconditionError                   Traceback (most recent call last)
/usr/local/lib/python3.6/dist-packages/tensorflow/python/client/session.py in _do_call(self, fn, *args)
   1277     try:
-> 1278       return fn(*args)
```

[8]

```
import tensorflow as tf

# Declare a variable
w = tf.Variable(tf.zeros((2,3)), name="weight") # initialized by the 2 x 2 matrix of zeros

with tf.Session() as sess:
  sess.run(w.initializer)
  w_value = sess.run(w)
  print("Value of w : ", w_value)
```

```
Value of w :  [[0. 0. 0.]
 [0. 0. 0.]]
```

Variables : initialization

The easiest way is initializing all variables at once:

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

Initialize only a subset of variables:

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

Initialize a single variable:

```
W = tf.Variable(tf.zeros([784,10]))
with tf.Session() as sess:
    sess.run(W.initializer)
```

Variables : Updation

- Since variables maintain state across runs, we naturally want them to update across runs
- How do we update them ?

```
W = tf.Variable(10)
W.assign(100)
with tf.Session() as sess:
        sess.run(W.initializer)
        print W.eval() # >> 10, not 100
```

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

- Note that we defined an operation for the variable updation
    - And we ran that op using sess.run() to update the variable

Variables across sessions

```
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()
```

Let’s do a proper example!

- Example 1 (Slide 27)
- Example 2 (Slide 31)

More awesome(!) things about tf

What we've covered are just the very basics. But this should enable you to read & understand tf code more easily.

tf.device -- multiple devices
tf.stop_gradient -- stop gradient somewhere in computational graph
tf.Print -- a node that prints out its contents at runtime
tf.trainable _variables -- make some variables trainable
tf.get_variable
tf.layers.* (conv2d, dropout, pooling, etc)
tf.nn.* (relu, etc)
tf.reduce_*
tf.train.* (optimizers, etc)

Lots and lots and lots more!

Selling points of tf:

1. Autodiff (nowadays all libraries have this)
2. Highly optimised for both graph creation/storage and also graph operations (this is the real power of tf)
3. Tensorboard - powerful visualisation tool

Many others...

References

https://www.tensorflow.org/api_docs/python/
https://www.edureka.co/blog/tensorflow-tutorial/
https://cs224d.stanford.edu/lectures/CS224d-Lecture7.pdf
https://www.slideshare.net/tw_dsconf/tensorflow-tutorial
https://www.slideshare.net/nmhkahn/tensorflow-tutorial-71896086
https://web.stanford.edu/class/cs20si/2017/lectures/slides_01.pdf

Thank You!