TensorFlow Ops

CS 20SI:
TensorFlow for Deep Learning Research
Lecture 2
1/18/2017
Agenda

Basic operations
Tensor types
Project speed dating
Placeholders and feeding inputs
Lazy loading

Fun 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))
**Visualize it with TensorBoard**

```python
import tensorflow as tf

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

with tf.Session() as sess:
    # add this line to use TensorBoard.
    writer = tf.summary.FileWriter('./graphs', sess.graph)

    print sess.run(x)

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

Create the summary writer after graph definition and before running your session

Where you want to keep your event files
Run it

Go to terminal, run:

$ python [yourprogram].py

$ tensorboard --logdir=./graphs --port 6006

Then open your browser and go to: http://localhost:6006/
import tensorflow as tf

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

# add this line to use TensorBoard

writer = tf.summary.FileWriter("./graphs", sess.graph)

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

Visualize it with TensorBoard
Visualize it with TensorBoard

```python
import tensorflow as tf

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

writer = tf.summary.FileWriter("./graphs", sess.graph)

with tf.Session() as sess:
    print sess.run(x)
```

Question:
How to change Const, Const_1 to the names we give the variables?
import tensorflow as tf

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

writer = tf.summary.FileWriter("./graphs", sess.graph)

with tf.Session() as sess:
    print sess.run(x) # >> 5
import tensorflow as tf

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

writer = tf.summary.FileWriter("./graphs", sess.graph)

with tf.Session() as sess:
    print sess.run(x) # >> 5
Learn to use TensorBoard well and often. It will help a lot when you build complicated models.
More constants

tf.constant(value, dtype=None, shape=None, name='Const', verify_shape=False)
import tensorflow as tf
a = tf.constant([2, 2], name="a")
b = tf.constant([[0, 1], [2, 3]], name="b")
x = tf.add(a, b, name="add")
y = tf.mul(a, b, name="mul")

with tf.Session() as sess:
    x, y = sess.run([x, y])
    print x, y
# >> [5 8] [6 12]

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

Similar to how you can create constants in numpy
Tensors filled with a specific value

```
tf.zeros(shape, dtype=tf.float32, name=None)
```

creates a tensor of shape and all elements will be zeros (when ran in session)

```
Similar to numpy.zeros
```

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

more compact than other constants in the graph def
→ faster startup (esp. in distributed)
Tensors filled with a specific value

\[
\text{tf.zeros}\_\text{like}(\text{input\_tensor}, \text{dtype=}\text{None}, \text{name=}\text{None}, \text{optimize=}\text{True})
\]

creates 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) \Rightarrow \([0, 0], [0, 0], [0, 0]\]

Similar to numpy.zeros\_like
Tensors filled with a specific value

Same:

```python
tf.ones(shape, dtype=tf.float32, name=None)
tf.ones_like(input_tensor, dtype=None, name=None, optimize=True)
```

Similar to numpy.ones, numpy.ones_like
Tensors filled with a specific value

tf.fill(dims, value, name=None)

creates a tensor filled with a scalar value.

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

In numpy, this takes two step:
1. Create a numpy array a
2. a.fill(value)
Constants as sequences

```
tf.linspace(start, stop, num, name=None) # slightly different from np.linspace
```

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

```
tf.range(start, limit=None, delta=1, dtype=None, name='range')
```

```
# 'start' is 3, 'limit' is 18, 'delta' is 3
```

```
tf.range(start, limit, delta) ==> [3, 6, 9, 12, 15]
```

```
# 'limit' is 5
```

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

Tensor objects are not iterable

```
for _ in tf.range(4): # TypeError
```

Randomly Generated Constants

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)
Randomly Generated Constants

tf.set_random_seed(seed)
## Operations

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

*Table from “Fundamental of Deep Learning”*
Operations

Pretty standard, quite similar to numpy.

See TensorFlow documentation

```
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, [1, 2]), tf.reshape(b, [2, 1])) # >> [[18]]

tf.div(a, b) # >> [1 3]

tf.mod(a, b) # >> [1 0]
```
TensorFlow Data Types

TensorFlow takes Python natives types: boolean, numeric (int, float), strings

0-d tensor, or "scalar"

\[ t_0 = 19 \]
\[ \text{tf.zeros_like}(t_0) \# \Rightarrow 0 \]
\[ \text{tf.ones_like}(t_0) \# \Rightarrow 1 \]
TensorFlow Data Types

TensorFlow takes Python natives types: boolean, numeric (int, float), strings

0-d tensor, or "scalar"

t_0 = 19

tf.zeros_like(t_0) # ==> 0

tf.ones_like(t_0) # ==> 1

1-d tensor, or "vector"

t_1 = ['apple', 'peach', 'banana']

tf.zeros_like(t_1) # ==> ????????
TensorFlow Data Types

TensorFlow takes Python natives types: boolean, numeric (int, float), strings

0-d tensor, or "scalar"

\[ t_0 = 19 \]
\[ \text{tf.zeros_like}(t_0) \# ==> 0 \]
\[ \text{tf.ones_like}(t_0) \# ==> 1 \]

1-d tensor, or "vector"

\[ t_1 = ['apple', 'peach', 'banana'] \]
\[ \text{tf.zeros_like}(t_1) \# ==> ['' '' ''] \]
\[ \text{tf.ones_like}(t_1) \# ==> ???????? \]
TensorFlow Data Types

TensorFlow takes Python natives types: boolean, numeric (int, float), strings

0-d tensor, or "scalar"

\[
t_0 = 19
\]
\[
tf.zeros_like(t_0) \ # \ ==> \ 0
\]
\[
tf.ones_like(t_0) \ # \ ==> \ 1
\]

1-d tensor, or "vector"

\[
t_1 = ['apple', 'peach', 'banana']
\]
\[
tf.zeros_like(t_1) \ # \ ==> \ ['' '' ''']
\]
\[
tf.ones_like(t_1) \ # \ ==> \ TypeError: Expected string, got 1 of type 'int' instead.
\]

2x2 tensor, or "matrix"

\[
t_2 = [[True, False, False],
        [False, False, True],
        [False, True, False]]
\]
\[
tf.zeros_like(t_2) \ # \ ==> \ 2x2 \ tensor, \ all \ elements \ are \ False
\]
\[
tf.ones_like(t_2) \ # \ ==> \ 2x2 \ tensor, \ all \ elements \ are \ True
\]
## TensorFlow Data Types

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

TensorFlow integrates seamlessly with NumPy

tf.int32 == np.int32  # True

Can pass numpy types to TensorFlow ops

tf.ones([2, 2], np.float32)  # ⇒ [[1.0 1.0], [1.0 1.0]]

For tf.Session.run(fetches):

If the requested fetch is a Tensor, then the output of will be a NumPy ndarray.
TensorFlow Data Types

Do not use Python native types for tensors because TensorFlow has to infer Python type
Beware when using NumPy arrays because NumPy and TensorFlow might become not so compatible in the future!
What’s wrong with constants?
Other than being constant ...
What’s wrong with constants?

Constants are stored in the graph definition
import tensorflow as tf

my_const = tf.constant([1.0, 2.0], name="my_const")

with tf.Session() as sess:
    print sess.graph.as_graph_def()

# you will see value of my_const stored in the graph’s definition
This makes loading graphs expensive when constants are big
Only use constants for primitive types. Use variables or readers for more data that requires more memory.
# Variables

# 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]))
Variables?

# 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]))
How about variables?

# 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 is a class, but tf.constant is an op
# create variable a with scalar value
a = tf.Variable(2, name="scalar")

tf.Variable holds several ops:
x = tf.Variable(...)
x.value() # read op
x.assign(...) # write op
x.assign_add(...) # and more

# 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]))

How about variables?
You have to **initialize** your variables

The easiest way is initializing all variables at once:

```python
init = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init)
```
You have to **initialize** your variables

The easiest way is initializing all variables at once:

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

Initialize only a subset of variables:

```python
init_ab = tf.variables_initializer([a, b], name="init_ab")
with tf.Session() as sess:
    sess.run(init_ab)
```
You have to **initialize** your variables

The easiest way is initializing all variables at once:

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

Initialize only a subset of variables:

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

Initialize a single variable

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

# 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)
Eval() a variable

# 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]]
tf.Variable.assign()

W = tf.Variable(10)
W.assign(100)

with tf.Session() as sess:
    sess.run(W.initializer)
    print W.eval() # >> ????

What do you think this will return?
tf.Variable.assign()

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

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

W.assign(100) doesn’t assign the value 100 to W. It creates an assign op, and that op needs to be run to take effect.
tf.Variable.assign()

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

--------

W = tf.Variable(10)
assign_op = W.assign(100)
with tf.Session() as sess:
    sess.run(W.initializer)
    sess.run(assign_op)
    print W.eval() # >> 100
```
You don’t need to initialize variable because assign_op does it for you
tf.Variable.assign()

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

--------

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

In fact, initializer op is the assign op that assigns the variable’s initial value to the variable itself.
tf.Variable.assign()

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

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

with tf.Session() as sess:
    sess.run(my_var.initializer)
    sess.run(my_var_times_two)  # >> 4
tf.Variable.assign()

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

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

with tf.Session() as sess:
    sess.run(my_varinitializer)
    sess.run(my_var_times_two) # >> 4
    sess.run(my_var_times_two) # >> 8

What do you think this will return?
# create a variable whose original value is 2
my_var = tf.Variable(2, name="my_var")

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

with tf.Session() as sess:
    sess.run(my_var.initializer)
    sess.run(my_var_times_two) # >> 4
    sess.run(my_var_times_two) # >> 8
    sess.run(my_var_times_two) # >> 16

It assign 2 * my_var to a every time my_var_times_two is fetched.
assign_add() and assign_sub()

my_var = tf.Variable(10)

With tf.Session() as sess:

    sess.run(my_var.initializer)
    # increment by 10
    sess.run(my_var.assign_add(10))  # >> 20
    # decrement by 2
    sess.run(my_var.assign_sub(2))  # >> 18

assign_add() and assign_sub() can’t initialize the variable my_var for you because these ops need the original value of my_var.
Each session maintains its own copy of variable

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)) # >> ?
Each session maintains its own copy of variable

```python
W = tf.Variable(10)

d1 = tf.Session()
d2 = tf.Session()

d1.run(W.initializer)
d2.run(W.initializer)

print d1.run(W.assign_add(10)) # >> 20
print d2.run(W.assign_sub(2)) # >> 8
```
Each session maintains its own copy of variable

```python
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()
```
Use a variable to initialize another variable

Want to declare $U = 2 \cdot W$

# W is a random $700 \times 100$ tensor
$W = \text{tf.Variable}(\text{tf.truncated_normal}([700, 10]))$
$U = \text{tf.Variable}(2 \cdot W)$

Not so safe (but quite common)
Use a variable to initialize another variable

Want to declare $U = W \times 2$

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

# ensure that W is initialized before its value is used to initialize U
Safer
Session vs InteractiveSession

You sometimes see InteractiveSession instead of Session

The only difference is an InteractiveSession makes itself the default

```python
sess = tf.InteractiveSession()
a = tf.constant(5.0)
b = tf.constant(6.0)
c = a * b
# We can just use 'c.eval()' without specifying the context 'sess'
print(c.eval())
sess.close()
```
Control Dependencies

tf.Graph.control_dependencies(control_inputs)

# defines which ops should be run first

# 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 = ...
Project speed dating
Project Speed Dating

“OK CLASS, FIND A PARTNER”

OH NO
A quick reminder

A TF program often has 2 phases:
1. Assemble a graph
2. Use a session to execute operations in the graph.
Placeholders

A TF program often has 2 phases:
1. Assemble a graph
2. Use a session to execute operations in the graph.

⇒ Can assemble the graph first without knowing the values needed for computation
A TF program often has 2 phases:
1. Assemble a graph
2. Use a session to execute operations in the graph.

⇒ Can assemble the graph first without knowing the values needed for computation

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.
Why placeholders?

We, or our clients, can later supply their own data when they need to execute the computation.
Placeholders

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

# 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)

with tf.Session() as sess:
    print sess.run(c)  # Error because a doesn’t have any value
Feed the values to placeholders using a dictionary
Placeholders

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

# 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)

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]})  # the tensor a is the key, not the string ‘a’

# >> [6, 7, 8]
Placeholders

\texttt{tf.placeholder(dtype, shape=None, name=None)}

# 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 \texttt{tf.add(a, b)}

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]

\textbf{Quirk:}
shape=None means that tensor of any shape will be accepted as value for placeholder.

shape=None is easy to construct graphs, but nightmarish for debugging
**Placeholders**

\[ \text{tf.placeholder(dtype, shape=None, name=None)} \]

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

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

# use the placeholder as you would a constant or a variable
\[ c = a + b \quad \# \text{Short for tf.add(a, b)} \]

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]

**Quirk:**
shape=None also breaks all following shape inference, which makes many ops not work because they expect certain rank.
Placeholders are valid ops

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

# 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)

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]
What if want to feed multiple data points in?

We feed all the values in, one at a time

```python
with tf.Session() as sess:
    for a_value in list_of_values_for_a:
        print sess.run(c, {a: a_value})
```
You can feed_dict any feedable tensor. Placeholder is just a way to indicate that something must be fed.
tf.Graph.is_feedable(tensor)
# True if and only if tensor is feedable.
Feeding values to TF ops

# create operations, tensors, etc (using the default graph)
a = tf.add(2, 5)
b = tf.mul(a, 3)

with tf.Session() as sess:
    # 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
Extremely helpful for testing too
The trap of lazy loading*

*I might have made this term up
What’s lazy loading?
Defer creating/initializing an object until it is needed
Lazy loading Example

Normal loading:

```python
x = tf.Variable(10, name='x')
y = tf.Variable(20, name='y')
z = tf.add(x, y) # you create the node for add node before executing the graph

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    writer = tf.summary.FileWriter('./my_graph/l2', sess.graph)
    for _ in range(10):
        sess.run(z)
    writer.close()
```
Lazy loading Example

Lazy loading:

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

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    writer = tf.summary.FileWriter('./my_graph/l2', sess.graph)
    for _ in range(10):
        sess.run(tf.add(x, y)) # someone decides to be clever to save one line of code
    writer.close()
```
Both give the same value of z 
What’s the problem?
TensorBoard

Normal loading
TensorBoard

Lazy loading (just missing the node Add, bad for reading graph, but not a bug)
Lazy loading

TensorBoard
tf.get_default_graph().as_graph_def()

Normal loading:

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

Node “Add” added once to the graph definition
Lazy loading:

defnode {  
    name: "Add"  
    op: "Add"  
    ...
}
...

defnode {  
    name: "Add_9"  
    op: "Add"  
    ...
}

Node “Add” added 10 times to the graph definition

Or as many times as you want to compute z
Imagine you want to compute an op thousands of times!
Your graph gets bloated
Slow to load
Expensive to pass around
One of the most common TF non-bug bugs I’ve seen on GitHub
1. Separate definition of ops from computing/running ops
2. Use Python property to ensure function is also loaded once the first time it is called*

* This is not a Python class so I won’t go into it here. But if you don’t know how to use this property, you’re welcome to ask me!
Putting it together:
A simple linear regression example
We will construct this model next time!!
Next class

Linear regression in TensorFlow

Optimizers

Logistic regression on MNIST

Feedback: huyenn@stanford.edu

Thanks!