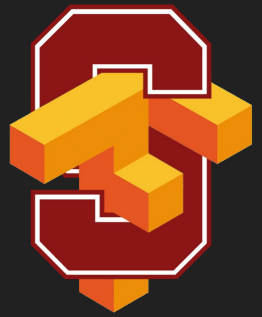


# TensorFlow Ops

CS 20: TensorFlow for Deep Learning Research
Lecture 2
1/17/2017

# Agenda

Basic operations

Tensor types

Importing data

Lazy loading

**Fun with TensorBoard!!!**

# Your first TensorFlow program

```
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))
```

# Your first TensorFlow program

```
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))
```

**Warning?**

```
The TensorFlow library wasn't
compiled to use SSE4.1 instructions,
but these are available on your
machine and could speed up CPU
computations.
```

# Your first TensorFlow program

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL']='2'
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))
```

**No more warning**

# Visualize it with TensorBoard

```
import tensorflow as tf

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

writer = tf.summary.FileWriter('./graphs', tf.get_default_graph())
with tf.Session() as sess:
      # 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

'graphs' or any location where you want to keep your event files

# Run it

Go to terminal, run:

```
$ python3 [yourprogram].py
$ tensorboard --logdir="./graphs" --port 6006
```

6006 or any port you want

Then open your browser and go to: `http://localhost:6006/`

TensorBoard GRAPHS INACTIVE

Fit to screen

Download PNG

Run (4) simple

Session runs (0)

Upload Choose File

Trace inputs

Color Structure

Device

Close legend.

Graph (* = expandable)

Namespace* ?

OpNode ?

Unconnected series* ?

Connected series* ?

Constant ?

Summary ?

Dataflow edge ?

Control dependency edge ?

Reference edge ?

Main GraphAuxiliary Nodes

# Visualize it with TensorBoard

```
import tensorflow as tf

a = tf.constant(2)
b = tf.constant(3)
x = tf.add(a, b)
writer = tf.summary.FileWriter('./graphs', tf.get_default_graph())
writer.close()
```

# Visualize it with TensorBoard

```
import tensorflow as tf

a = tf.constant(2)
b = tf.constant(3)
x = tf.add(a, b)
writer = tf.summary.FileWriter('./graphs', tf.get_default_graph())
writer.close()
```

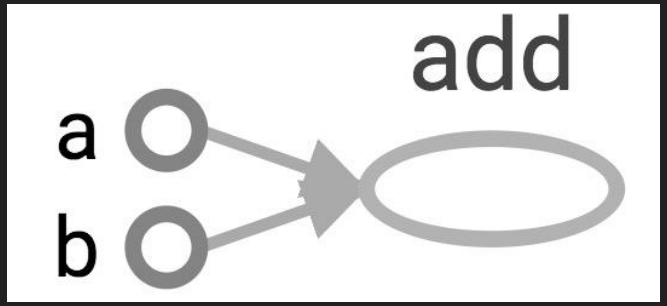

Question:
How to change Const, Const_1 to the names we give the variables?

# Explicitly name them

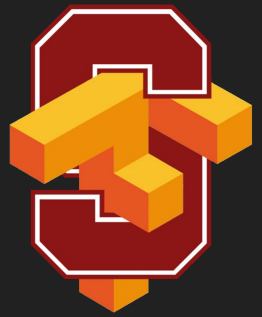
```
import tensorflow as tf

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

writer = tf.summary.FileWriter('./graphs', tf.get_default_graph())
with tf.Session() as sess:
    print(sess.run(x)) # >> 5
```

# Explicitly name them

```
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', tf.get_default_graph())
with tf.Session() as sess:
    print(sess.run(x)) # >> 5
```



**TensorBoard can do much more than just visualizing your graphs.**
**Learn to use TensorBoard well and often!**



# Constants, Sequences, Variables, Ops

# Constants

```
import tensorflow as tf

a = tf.constant([2, 2], name='a')
b = tf.constant([[0, 1], [2, 3]], name='b')
```

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

# Constants

```
import tensorflow as tf

a = tf.constant([2, 2], name='a')
b = tf.constant([[0, 1], [2, 3]], name='b')
x = tf.multiply(a, b, name='mul')

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

#  >>  [[0 2]
#       [4 6]]
```

Broadcasting similar to 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

Similar to numpy.zeros

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

# Tensors filled with a specific value

```
tf.zeros_like(input_tensor, dtype=None, name=None, optimize=True)
```

creates a tensor of shape and type (unless type is specified) as the input_tensor but all elements are zeros.

Similar to numpy.zeros_like

```
# input_tensor is [[0, 1], [2, 3], [4, 5]]

tf.zeros_like(input_tensor) ==> [[0, 0], [0, 0], [0, 0]]
```

# Tensors filled with a specific value

```
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.

Similar to NumPy.full

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

# Constants as sequences

**tf.lin_space(start, stop, num, name=None)**

```
tf.lin_space(10.0, 13.0, 4) ==> [10. 11. 12. 13.]
```

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

```
tf.range(3, 18, 3) ==> [3 6 9 12 15]
tf.range(5) ==> [0 1 2 3 4]
```

NOT THE SAME AS NUMPY SEQUENCES

Tensor objects are not iterable

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

# Randomly Generated Constants

```
tf.random_normal
tf.truncated_normal
tf.random_uniform
tf.random_shuffle
tf.random_crop
tf.multinomial
tf.random_gamma
```

# Randomly Generated Constants

```
tf.set_random_seed(seed)
```

# Operations

| 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 |

Buduma. *Fundamentals of Deep Learning*. O'Reilly, 2017

# Arithmetic Ops

- `tf.abs`
- `tf.negative`
- `tf.sign`
- `tf.reciprocal`
- `tf.square`
- `tf.round`
- `tf.sqrt`
- `tf.rsqrt`
- `tf.pow`
- `tf.exp`

Pretty standard, quite similar to numpy.

# Wizard of Div

```
a = tf.constant([2, 2], name='a')
b = tf.constant([[0, 1], [2, 3]], name='b')
with tf.Session() as sess:
    print(sess.run(tf.div(b, a)))           ⇒ [[0 0] [1 1]]
    print(sess.run(tf.divide(b, a)))        ⇒ [[0. 0.5] [1. 1.5]]
    print(sess.run(tf.truediv(b, a)))       ⇒ [[0. 0.5] [1. 1.5]]
    print(sess.run(tf.floordiv(b, a)))      ⇒ [[0 0] [1 1]]
    print(sess.run(tf.realdiv(b, a)))       ⇒ # Error: only works for real values
    print(sess.run(tf.truncatediv(b, a)))   ⇒ [[0 0] [1 1]]
    print(sess.run(tf.floor_div(b, a)))     ⇒ [[0 0] [1 1]]
```

# TensorFlow Data Types

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

```
t_0 = 19                    # scalars are treated like 0-d tensors
tf.zeros_like(t_0)          # ==> 0
tf.ones_like(t_0)           # ==> 1
```

# TensorFlow Data Types

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

```
t_0 = 19                    # scalars are treated like 0-d tensors
tf.zeros_like(t_0)          # ==> 0
tf.ones_like(t_0)           # ==> 1
```

# TensorFlow Data Types

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

```
t_0 = 19                                # scalars are treated like 0-d tensors
tf.zeros_like(t_0)                      # ==> 0
tf.ones_like(t_0)                       # ==> 1

t_1 = [b"apple", b"peach", b"grape"]    # 1-d arrays are treated like 1-d tensors
tf.zeros_like(t_1)                      # ==> ?????
```

# TensorFlow Data Types

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

```
t_0 = 19                                # scalars are treated like 0-d tensors
tf.zeros_like(t_0)                      # ==> 0
tf.ones_like(t_0)                       # ==> 1

t_1 = [b"apple", b"peach", b"grape"]    # 1-d arrays are treated like 1-d tensors
tf.zeros_like(t_1)                      # ==> [b'' b'' b'']
```

# TensorFlow Data Types

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

```
t_0 = 19                                # scalars are treated like 0-d tensors
tf.zeros_like(t_0)                      # ==> 0
tf.ones_like(t_0)                       # ==> 1

t_1 = [b"apple", b"peach", b"grape"]    # 1-d arrays are treated like 1-d tensors
tf.zeros_like(t_1)                      # ==> [b'' b'' b'']
tf.ones_like(t_1)                       # ==> ?????
```

# TensorFlow Data Types

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

```
t_0 = 19                                # scalars are treated like 0-d tensors
tf.zeros_like(t_0)                      # ==> 0
tf.ones_like(t_0)                       # ==> 1

t_1 = [b"apple", b"peach", b"grape"]    # 1-d arrays are treated like 1-d tensors
tf.zeros_like(t_1)                      # ==> [b'' b'' b'']
tf.ones_like(t_1)                       # ==> TypeError: Expected string, got 1 of type 'int' instead.
```

# TensorFlow Data Types

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

```
t_0 = 19                        # scalars are treated like 0-d tensors
tf.zeros_like(t_0)              # ==> 0
tf.ones_like(t_0)               # ==> 1

t_1 = [b"apple", b"peach", b"grape"]    # 1-d arrays are treated like 1-d tensors
tf.zeros_like(t_1)              # ==> [b'' b'' b'']
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]]    # 2-d arrays are treated like 2-d tensors

tf.zeros_like(t_2)              # ==> ?????
tf.ones_like(t_2)               # ==> ?????
```

# TensorFlow Data Types

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

```
t_0 = 19                                # scalars are treated like 0-d tensors
tf.zeros_like(t_0)                      # ==> 0
tf.ones_like(t_0)                       # ==> 1

t_1 = [b"apple", b"peach", b"grape"]    # 1-d arrays are treated like 1-d tensors
tf.zeros_like(t_1)                      # ==> [b'' b'' b'']
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]]            # 2-d arrays are treated like 2-d tensors

tf.zeros_like(t_2)                      # ==> 3x3 tensor, all elements are False
tf.ones_like(t_2)                       # ==> 3x3 tensor, all elements are True
```

# TensorFlow Data Types

- `tf.float16` : 16-bit half-precision floating-point.
- `tf.float32` : 32-bit single-precision floating-point.
- `tf.float64` : 64-bit double-precision floating-point.
- `tf.bfloat16` : 16-bit truncated floating-point.
- `tf.complex64` : 64-bit single-precision complex.
- `tf.complex128` : 128-bit double-precision complex.
- `tf.int8` : 8-bit signed integer.
- `tf.uint8` : 8-bit unsigned integer.
- `tf.uint16` : 16-bit unsigned integer.
- `tf.int16` : 16-bit signed integer.
- `tf.int32` : 32-bit signed integer.
- `tf.int64` : 64-bit signed integer.
- `tf.bool` : Boolean.
- `tf.string` : String.
- `tf.qint8` : Quantized 8-bit signed integer.
- `tf.quint8` : Quantized 8-bit unsigned integer.
- `tf.qint16` : Quantized 16-bit signed integer.
- `tf.quint16` : Quantized 16-bit unsigned integer.
- `tf.qint32` : Quantized 32-bit signed integer.
- `tf.resource` : Handle to a mutable resource.

# 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 , output will be a NumPy ndarray.

```
sess = tf.Session()
a = tf.zeros([2, 3], np.int32)
print(type(a))                  # ⇒ <class 'tensorflow.python.framework.ops.Tensor'>
a = sess.run(a)
print(type(a))                  # ⇒ <class 'numpy.ndarray'>
```

# 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 , output will be a NumPy ndarray.

```
sess = tf.Session()
a = tf.zeros([2, 3], np.int32)
print(type(a))
a = sess.run(a)                 <<<< Avoid doing this. Use a_out = sess.run(a)
print(type(a))
```

# Use TF DType when possible

- Python native types: TensorFlow has to infer Python type

# Use TF DType when possible

- Python native types: TensorFlow has to infer Python type
- NumPy arrays: NumPy is not GPU compatible

# What’s wrong with constants …

… other than being constant?

# What’s wrong with constants?

Constants are stored in the graph definition

# Print out the graph def

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

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

# What’s wrong with constants?

This makes loading graphs expensive when constants are big

# What’s wrong with constants?

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 variables with tf.Variable
s = tf.Variable(2, name="scalar")
m = tf.Variable([[0, 1], [2, 3]], name="matrix")
W = tf.Variable(tf.zeros([784,10]))
```

# Variables

```
# create variables with tf.Variable
s = tf.Variable(2, name="scalar")
m = tf.Variable([[0, 1], [2, 3]], name="matrix")
W = tf.Variable(tf.zeros([784,10]))

# create variables with tf.get_variable
s = tf.get_variable("scalar", initializer=tf.constant(2))
m = tf.get_variable("matrix", initializer=tf.constant([[0, 1], [2, 3]]))
W = tf.get_variable("big_matrix", shape=(784, 10), initializer=tf.zeros_initializer())
```

# Variables

```
# create variables with tf.Variable
s = tf.Variable(2, name="scalar")
m = tf.Variable([[0, 1], [2, 3]], name="matrix")
W = tf.Variable(tf.zeros([784,10]))

# create variables with tf.get_variable
s = tf.get_variable("scalar", initializer=tf.constant(2))
m = tf.get_variable("matrix", initializer=tf.constant([[0, 1], [2, 3]]))
W = tf.get_variable("big_matrix", shape=(784, 10), initializer=tf.zeros_initializer())
```

# Variables

```
# create variables with tf.Variable
s = tf.Variable(2, name="scalar")
m = tf.Variable([[0, 1], [2, 3]], name="matrix")
W = tf.Variable(tf.zeros([784,10]))

# create variables with tf.get_variable
s = tf.get_variable("scalar", initializer=tf.constant(2))
m = tf.get_variable("matrix", initializer=tf.constant([[0, 1], [2, 3]]))
W = tf.get_variable("big_matrix", shape=(784, 10), initializer=tf.zeros_initializer())
```

Why tf.constant but tf.Variable?

# Variables

```
# create variables with tf.Variable
s = tf.Variable(2, name="scalar")
m = tf.Variable([[0, 1], [2, 3]], name="matrix")
W = tf.Variable(tf.zeros([784,10]))

# create variables with tf.get_variable
s = tf.get_variable("scalar", initializer=tf.constant(2))
m = tf.get_variable("matrix", initializer=tf.constant([[0, 1], [2, 3]]))
W = tf.get_variable("big_matrix", shape=(784, 10), initializer=tf.zeros_initializer())
```

tf.constant is an op
tf.Variable is a class with many ops

# tf.Variable class

```
# create variables with tf.get_variable
s = tf.get_variable("scalar", initializer=tf.constant(2))
m = tf.get_variable("matrix", initializer=tf.constant([[0, 1], [2, 3]]))
W = tf.get_variable("big_matrix", shape=(784, 10), initializer=tf.zeros_initializer())
```

**tf.Variable holds several ops:**

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

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

# tf.Variable class

```
# create variables with tf.get_variable
s = tf.get_variable("scalar", initializer=tf.constant(2))
m = tf.get_variable("matrix", initializer=tf.constant([[0, 1], [2, 3]]))
W = tf.get_variable("big_matrix", shape=(784, 10), initializer=tf.zeros_initializer())

with tf.Session() as sess:
      print(sess.run(W))    >> FailedPreconditionError: Attempting to use uninitialized value Variable
```

# You have to initialize your variables

The easiest way is initializing all variables at once:

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

Initializer is an op. You need to execute it within the context of a session

# You have to initialize your variables

The easiest way is initializing all variables at once:

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

Initialize only a subset of variables:

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

# You have to initialize your variables

The easiest way is initializing all variables at once:

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

Initialize only a subset of variables:

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

Initialize a single variable

```
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())                    # Similar to print(sess.run(W))
```

```
>> [[-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())                 # >> ????
```

# tf.Variable.assign()

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

Ugh, why?

# tf.Variable.assign()

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

W.assign(100) creates an assign op.
That op needs to be executed in a session to take effect.

# 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(W.initializer)
     sess.run(assign_op)
     print(W.eval())                    # >> 100
```

# 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)                 # >> what’s the value of my_var now?
```

# 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)                # >> the value of my_var now is 4
     sess.run(my_var_times_two)                # >> the value of my_var now is ???
```

# 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)                # >> the value of my_var now is 4
     sess.run(my_var_times_two)                # >> the value of my_var now is 8
     sess.run(my_var_times_two)                # >> the value of my_var now is 16
```

It assign 2 * `my_var` to my_var every time `my_var`_times_two op is executed.

# 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
```

# Each session maintains its own copy of variables

```
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 variables

```
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
```

# Each session maintains its own copy of variables

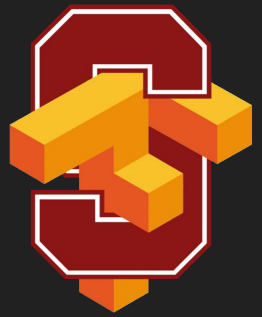
```
W = tf.Variable(10)

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()
```

# Control Dependencies

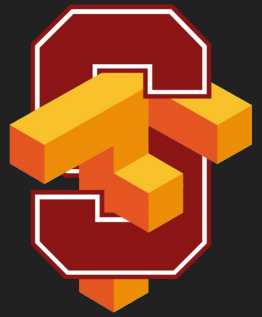
`tf.Graph.control_dependencies(control_inputs)`

# defines which ops should be run first

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



# Getting to know each other?



# Placeholder

# 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.

⇒ Assemble the graph first without knowing the values needed for computation

# Placeholders

A TF program often has 2 phases:

1. Assemble a graph
2. Use a session to execute operations in the graph.

⇒ Assemble the graph first without knowing the values needed for computation

Analogy:
Define the function f(x, y) = 2 * x + 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 for a vector of 3 elements, type tf.float32
a = tf.placeholder(tf.float32, shape=[3])

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))                        # >> ???
```

# Placeholders

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

```
# create a placeholder for a vector of 3 elements, type tf.float32
a = tf.placeholder(tf.float32, shape=[3])

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))                # >> InvalidArgumentError: a doesn’t an actual value
```

# Supplement the values to placeholders using a dictionary

# Placeholders

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

```
# create a placeholder for a vector of 3 elements, type tf.float32
a = tf.placeholder(tf.float32, shape=[3])

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, feed_dict={a: [1, 2, 3]}))    # the tensor a is the key, not the string ‘a’

# >> [6, 7, 8]
```

# Placeholders

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

```
# create a placeholder for a vector of 3 elements, type tf.float32
a = tf.placeholder(tf.float32, shape=[3])

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, feed_dict={a: [1, 2, 3]}))

# >> [6, 7, 8]
```

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

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

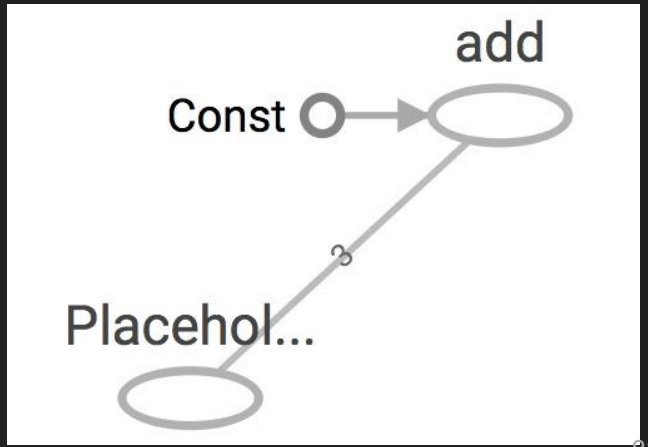
```
# create a placeholder of type float 32-bit, shape is a vector of 3 elements
a = tf.placeholder(tf.float32, shape=[3])

# 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, {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:
      print(sess.run(c, {a: [1, 2, 3]}))

# >> [6, 7, 8]
```



# What if want to feed multiple data points in?

You have to do it one at a time

```
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.

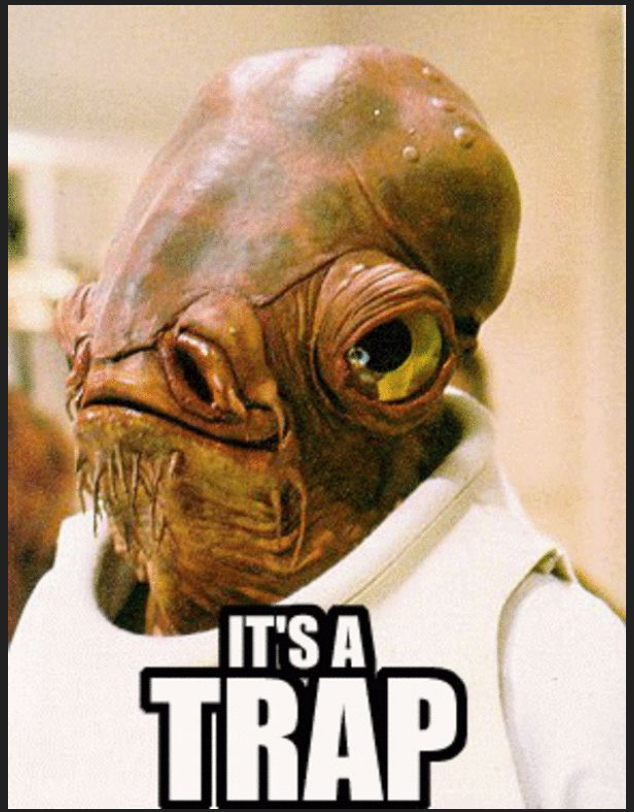
# Feeding values to TF ops

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

with tf.Session() as sess:
     # compute the value of b given a is 15
     sess.run(b, feed_dict={a: 15})                # >> 45
```

# Extremely helpful for testing

## Feed in dummy values to test parts of a large graph

# 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

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

writer = tf.summary.FileWriter('./graphs/normal_loading', tf.get_default_graph())
with tf.Session() as sess:
     sess.run(tf.global_variables_initializer())
     for _ in range(10):
          sess.run(z)
writer.close()
```

# Lazy loading Example

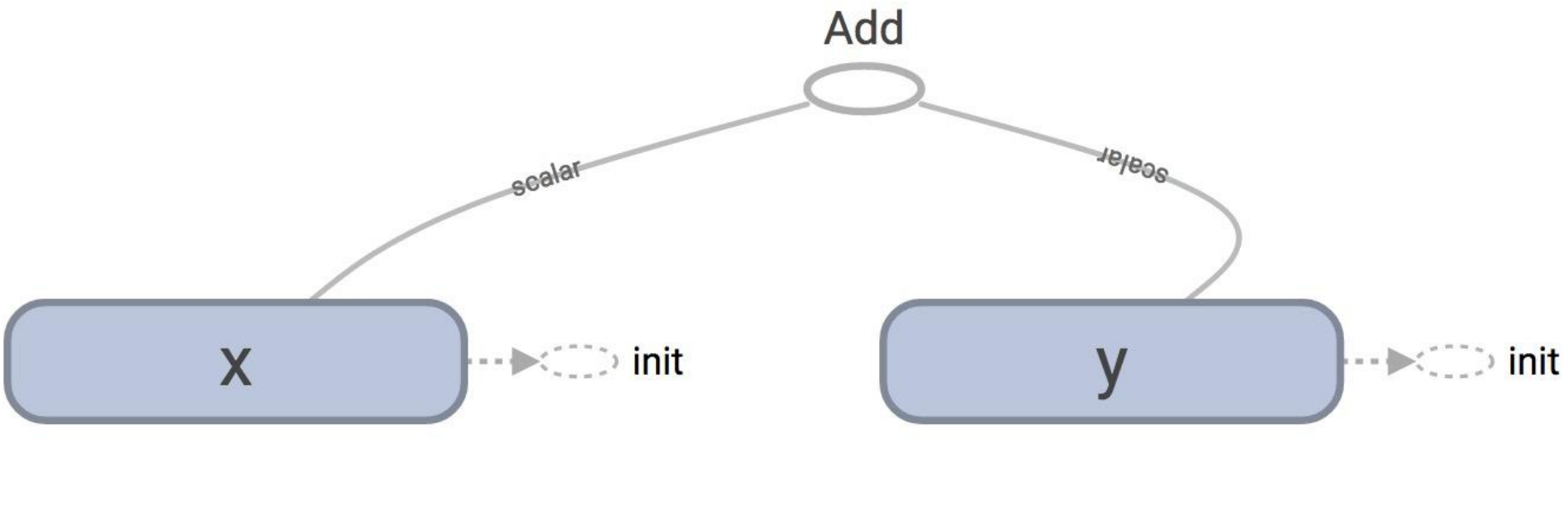
## Lazy loading

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

writer = tf.summary.FileWriter('./graphs/normal_loading', tf.get_default_graph())
with tf.Session() as sess:
     sess.run(tf.global_variables_initializer())
     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?**

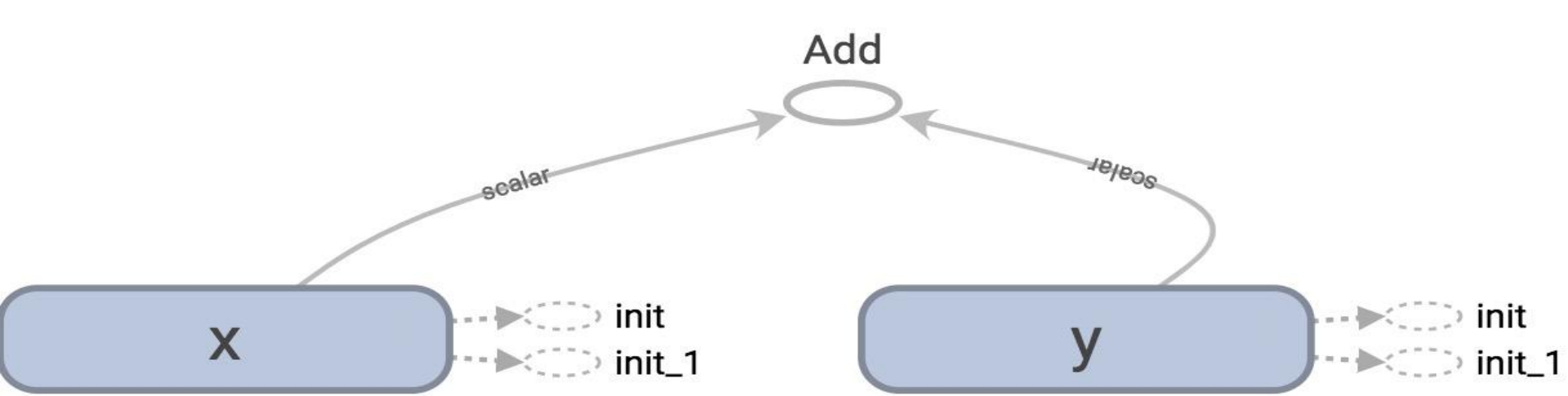
# TensorBoard

**Normal loading**



# TensorBoard

**Lazy loading**



# 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

# tf.get_default_graph().as_graph_def()

## Lazy loading

```
node {
  name: "Add_1"
  op: "Add"
  ...
  }
...
node {
  name: "Add_10"
  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, or millions 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

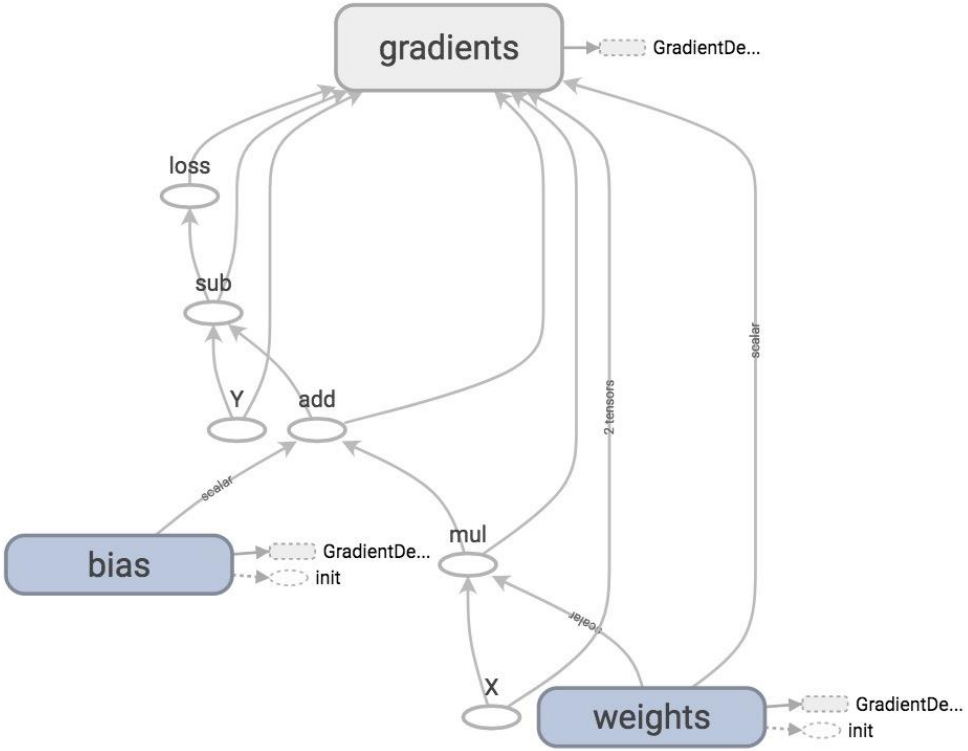
# Solution

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*

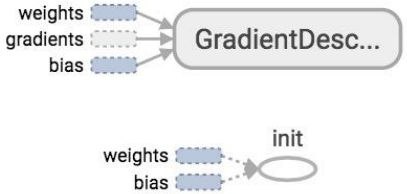
* This is not a Python class so I won't go into it here. But if you don't know how to use this property, you're welcome to ask me!

# Putting it together:
# Let’s build a machine learning model!

## Main Graph




## Auxiliary Nodes



**We will construct this model next time!!**

# Next class

Linear regression

Control Flow

tf.data

Optimizers

Logistic regression on MNIST

Feedback: huyenn@stanford.edu

Thanks!