



# An introduction to TensorFlow!

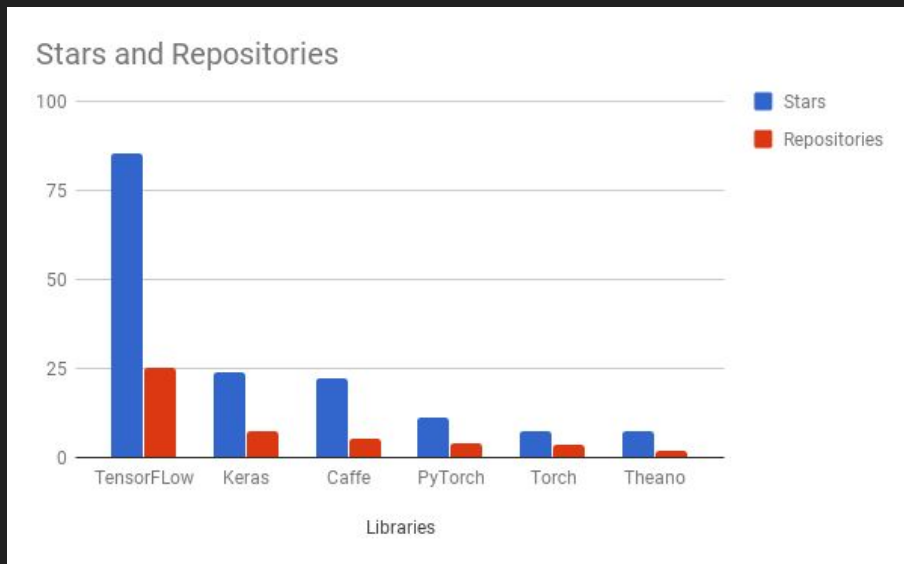
Chip Huyen ([chiphuyen@cs.stanford.edu](mailto:chiphuyen@cs.stanford.edu))

CS224N

1/25/2018

# Why TensorFlow?

- Flexibility + Scalability
- Popularity



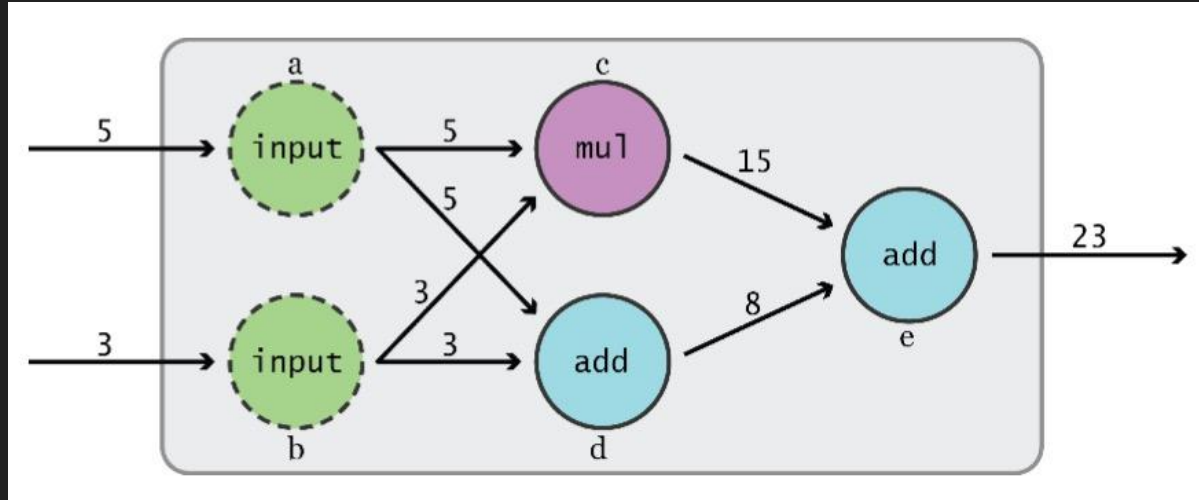
```
import tensorflow as tf
```



# Graphs and Sessions

# Data Flow Graphs

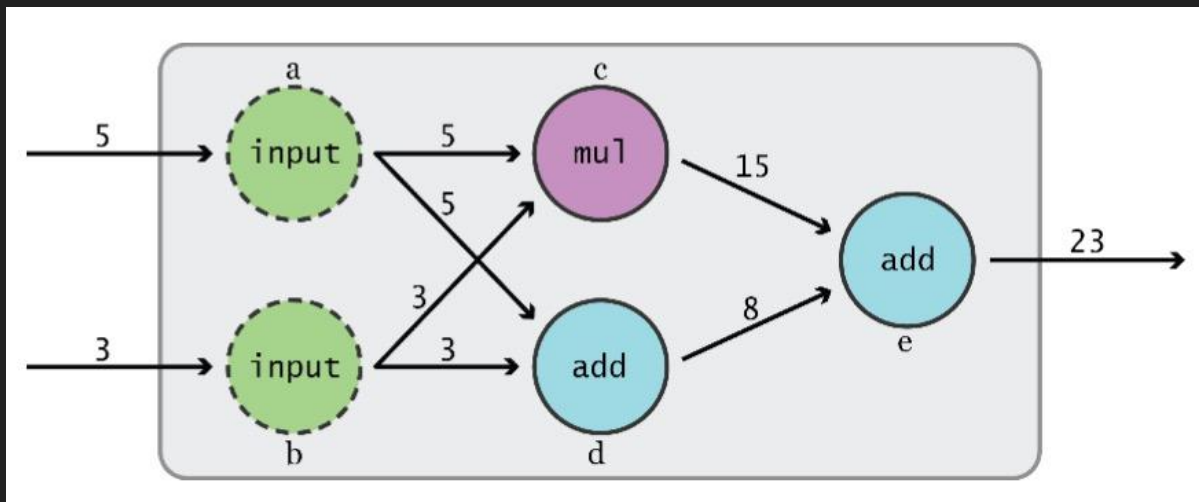
TensorFlow separates definition of computations from their execution



# Data Flow Graphs

Phase 1: assemble a graph

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

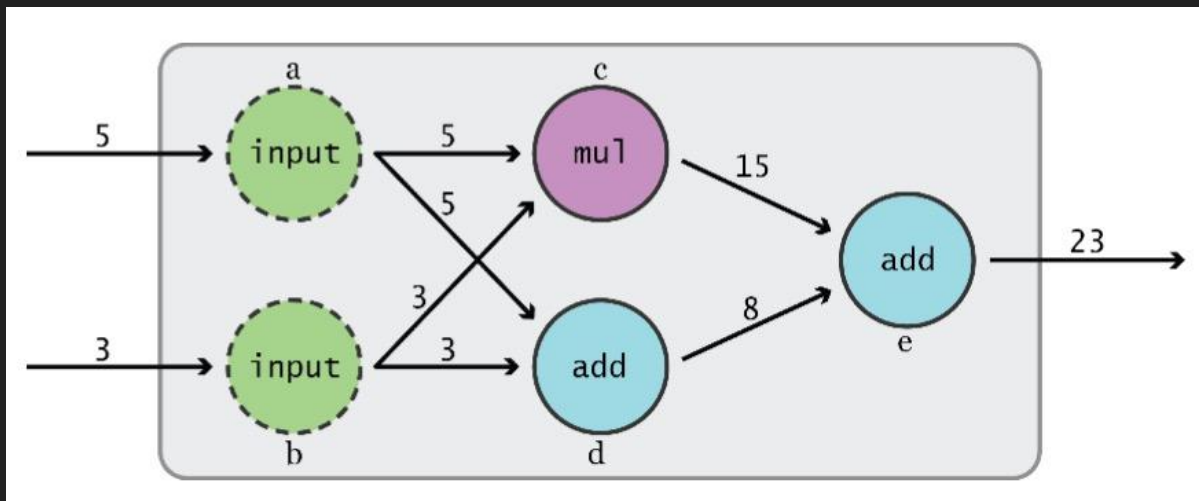


# Data Flow Graphs

Phase 1: assemble a graph

This might change in the future with eager mode!!

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



**What's a tensor?**



# What's a tensor?

## An n-dimensional array

0-d tensor: scalar (number)

1-d tensor: vector

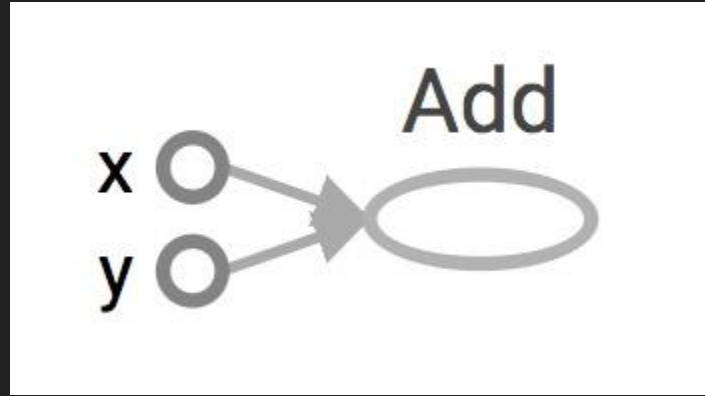
2-d tensor: matrix

and so on

# Data Flow Graphs

```
import tensorflow as tf  
a = tf.add(3, 5)
```

Visualized by TensorBoard



# Data Flow Graphs

```
import tensorflow as tf  
a = tf.add(3, 5)
```

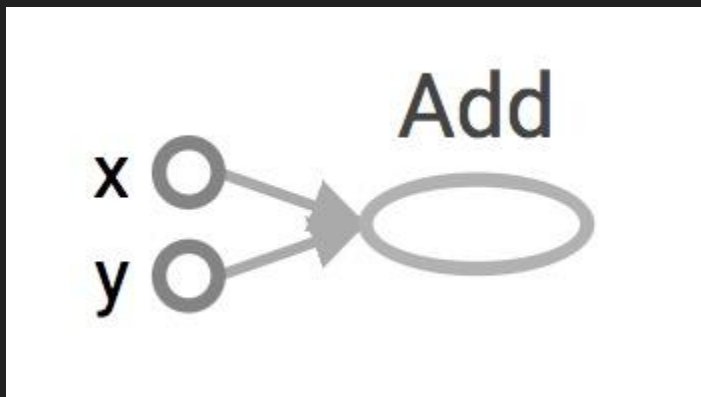
Why x, y?

TF automatically names the nodes when you don't explicitly name them.

x = 3

y = 5

Visualized by TensorBoard



# Data Flow Graphs

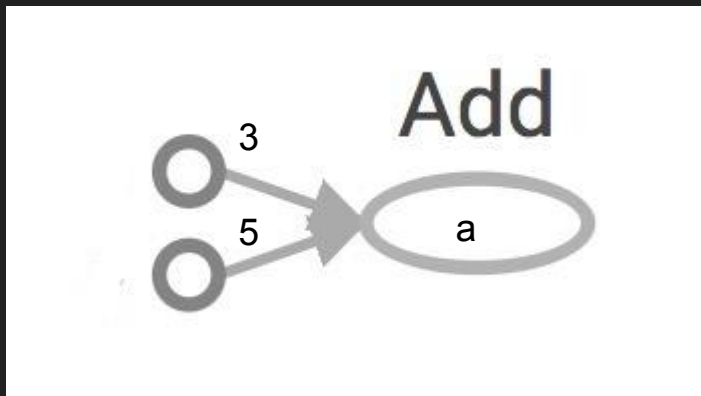
```
import tensorflow as tf  
a = tf.add(3, 5)
```

Nodes: operators, variables, and constants  
Edges: tensors

Tensors are data.

TensorFlow = tensor + flow = data + flow  
(I know, mind=blown)

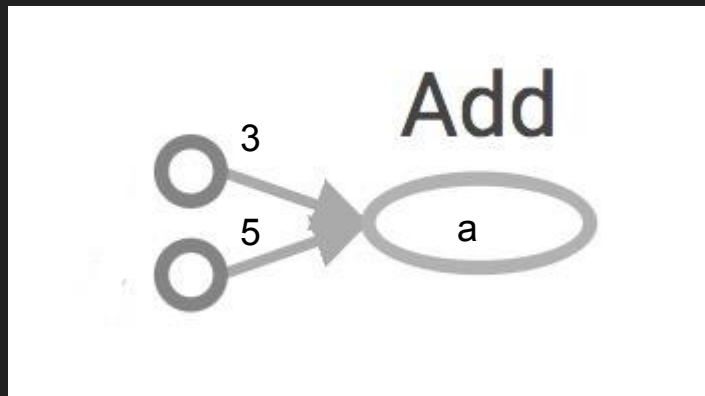
Interpreted?



# Data Flow Graphs

```
import tensorflow as tf  
a = tf.add(3, 5)  
print(a)
```

```
>> Tensor("Add:0", shape=(), dtype=int32)  
(Not 8)
```



# How to get the value of a?

Create a **session**, assign it to variable sess so we can call it later

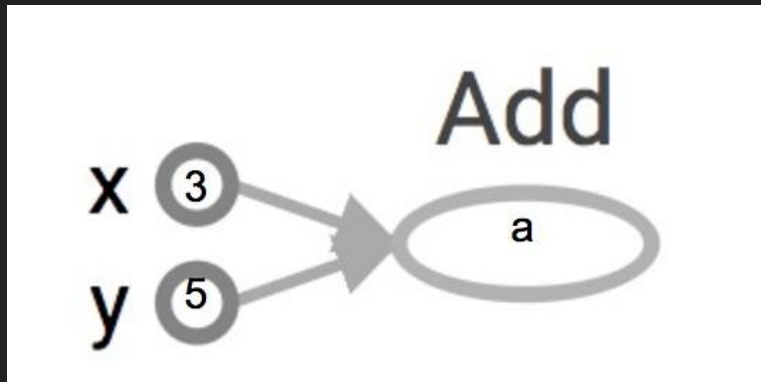
Within the session, evaluate the graph to fetch the value of a

# How to get the value of a?

Create a **session**, assign it to variable `sess` so we can call it later

Within the session, evaluate the graph to fetch the value of `a`

```
import tensorflow as tf
a = tf.add(3, 5)
sess = tf.Session()
print(sess.run(a))
sess.close()
```



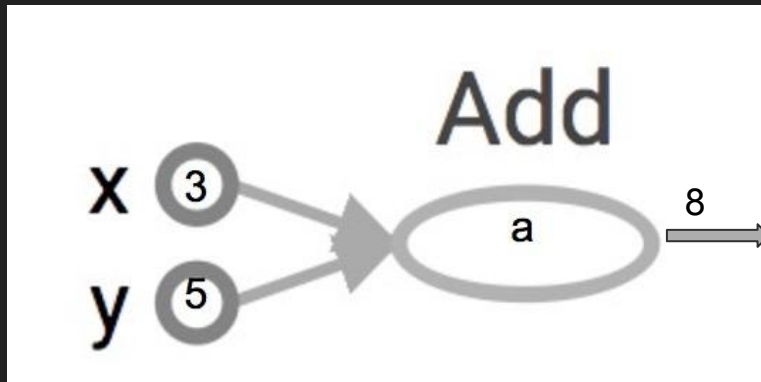
The session will look at the graph, trying to think: hmm, how can I get the value of `a`, then it computes all the nodes that leads to `a`.

# How to get the value of a?

Create a **session**, assign it to variable `sess` so we can call it later

Within the session, evaluate the graph to fetch the value of `a`

```
import tensorflow as tf
a = tf.add(3, 5)
sess = tf.Session()
print(sess.run(a))    >> 8
sess.close()
```



The session will look at the graph, trying to think: hmm, how can I get the value of `a`, then it computes all the nodes that leads to `a`.

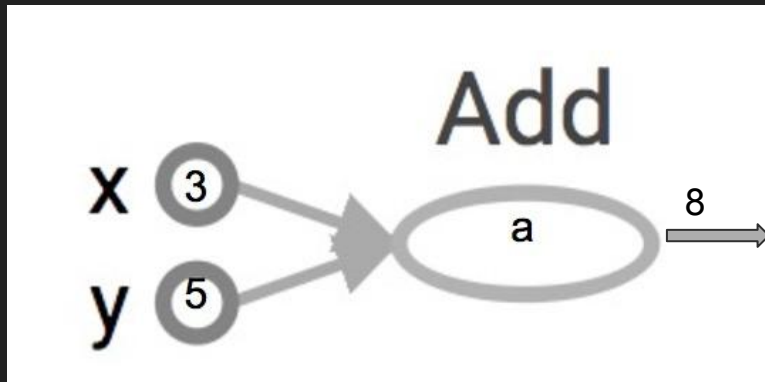


# How to get the value of a?

Create a **session**, assign it to variable `sess` so we can call it later

Within the session, evaluate the graph to fetch the value of `a`

```
import tensorflow as tf
a = tf.add(3, 5)
sess = tf.Session()
with tf.Session() as sess:
    print(sess.run(a))
sess.close()
```



# **tf.Session()**

A Session object encapsulates the environment in which Operation objects are executed, and Tensor objects are evaluated.

# tf.Session()

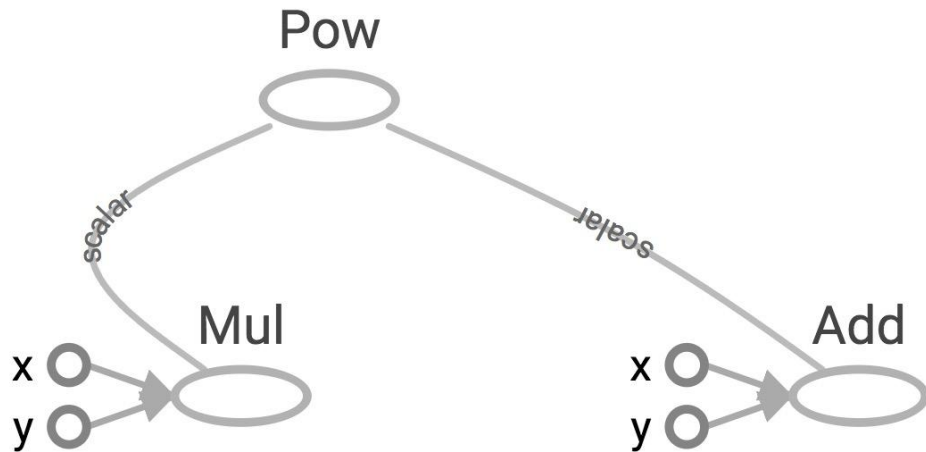
A Session object encapsulates the environment in which Operation objects are executed, and Tensor objects are evaluated.

Session will also allocate memory to store the current values of variables.

# More graph

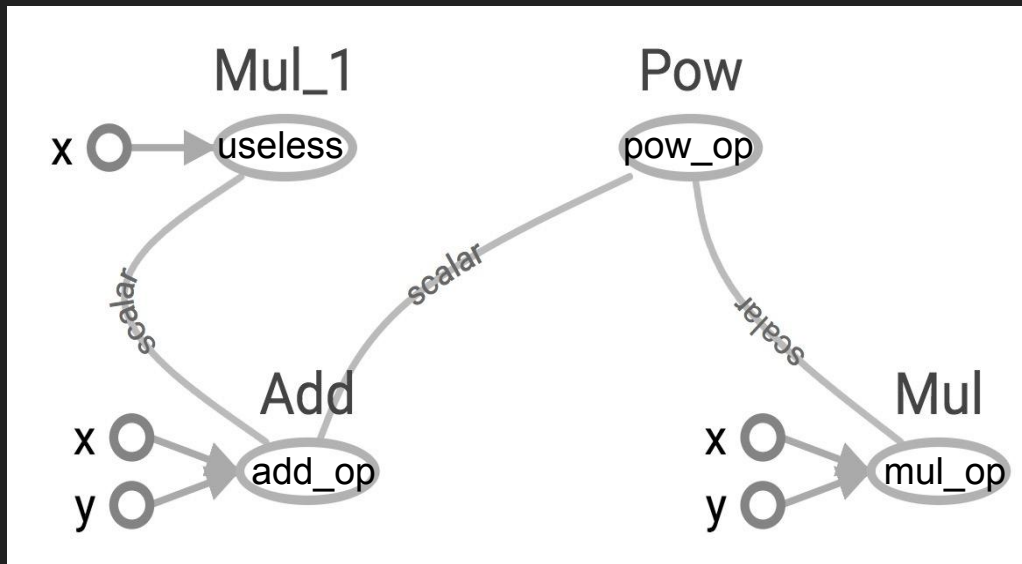
Visualized by TensorBoard

```
x = 2
y = 3
op1 = tf.add(x, y)
op2 = tf.multiply(x, y)
op3 = tf.pow(op2, op1)
with tf.Session() as sess:
    op3 = sess.run(op3)
```



# Subgraphs

```
x = 2
y = 3
add_op = tf.add(x, y)
mul_op = tf.multiply(x, y)
useless = tf.multiply(x, add_op)
pow_op = tf.pow(add_op, mul_op)
with tf.Session() as sess:
    z = sess.run(pow_op)
```

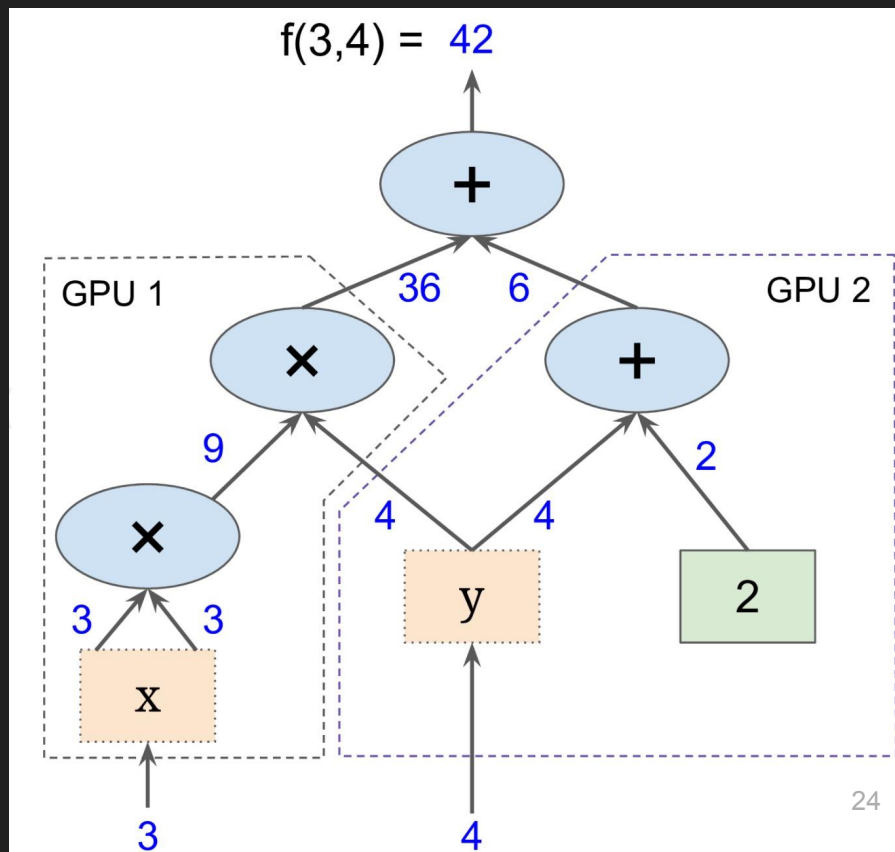


Because we only want the value of pow\_op and pow\_op doesn't depend on useless, session won't compute value of useless  
→ save computation

# Subgraphs

Possible to break graphs into several chunks and run them parallelly across multiple CPUs, GPUs, TPUs, or other devices

Example: AlexNet



# Distributed Computation

To put part of a graph on a specific CPU or GPU:

```
# Creates a graph.
with tf.device('/gpu:2'):
    a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], name='a')
    b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], name='b')
    c = tf.multiply(a, b)

# Creates a session with log_device_placement set to True.
sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))

# Runs the op.
print(sess.run(c))
```

# Why graphs

1. Save computation. Only run subgraphs that lead to the values you want to fetch.



# Why graphs

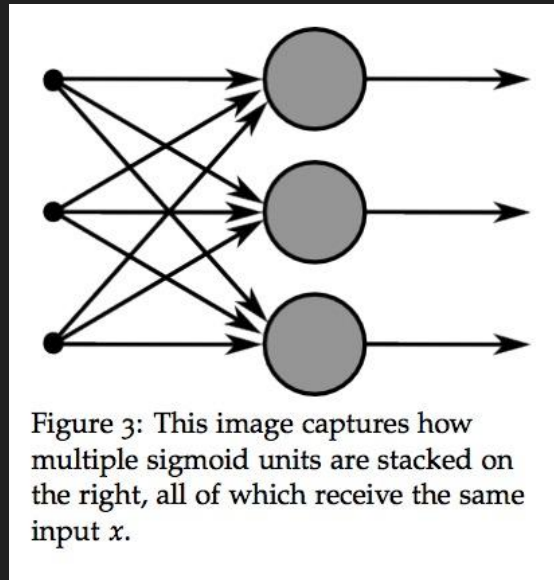
1. Save computation. Only run subgraphs that lead to the values you want to fetch.
2. Break computation into small, differential pieces to facilitate auto-differentiation

# Why graphs

1. Save computation. Only run subgraphs that lead to the values you want to fetch.
2. Break computation into small, differential pieces to facilitate auto-differentiation
3. Facilitate distributed computation, spread the work across multiple CPUs, GPUs, TPUs, or other devices

# Why graphs

1. Save computation. Only run subgraphs that lead to the values you want to fetch.
2. Break computation into small, differential pieces to facilitate auto-differentiation
3. Facilitate distributed computation, spread the work across multiple CPUs, GPUs, TPUs, or other devices
4. Many common machine learning models are taught and visualized as directed graphs



A neural net graph from Stanford's CS224N course



# 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')
x = tf.multiply(a, b, name='mul')
```

Broadcasting similar to NumPy

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

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

# Tensors filled with a specific value

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

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

Similar to NumPy

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

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

# Constants as sequences

```
tf.linspace(start, stop, num, name=None)
```

```
tf.linspace(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)
```

# 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_out = sess.run(a)
print(type(a_out))      # ⇒ <class 'numpy.ndarray'>
```

# 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?


Not trainable

# Constants are stored in graph definition

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

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

```
attr {  
  key: "value"  
  value {  
    tensor {  
      dtype: DT_FLOAT  
      tensor_shape {  
        dim {  
          size: 2  
        }  
      }  
      tensor_content: "\000\000\200?\000\000\000@"  
    }  
  }  
}
```



# Constants are stored in graph definition

This makes loading graphs expensive when constants are big

# Constants are stored in graph 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 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())
```



# 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)
```

# 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
```



# 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 and great when you have different batch sizes, 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)`**

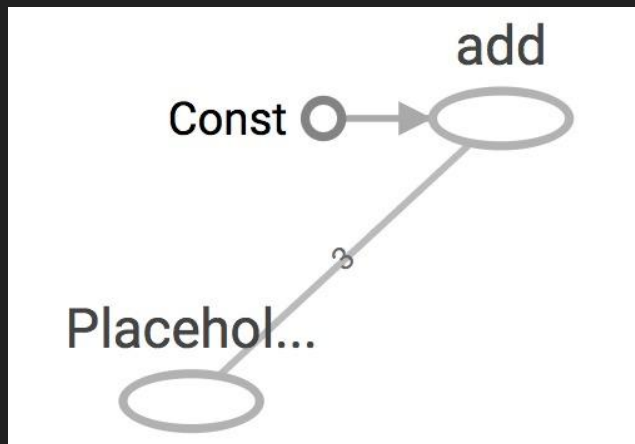
```
# create a placeholder of type float 32-bit, shape is a vector of 3 elements  
a = tf.placeholder(tf.float32, shape=[3])
```

```
# 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]
```



**How does TensorFlow know what variables  
to update?**



# Optimizers

# Optimizer

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01).minimize(loss)  
_, l = sess.run([optimizer, loss], feed_dict={X: x, Y:y})
```



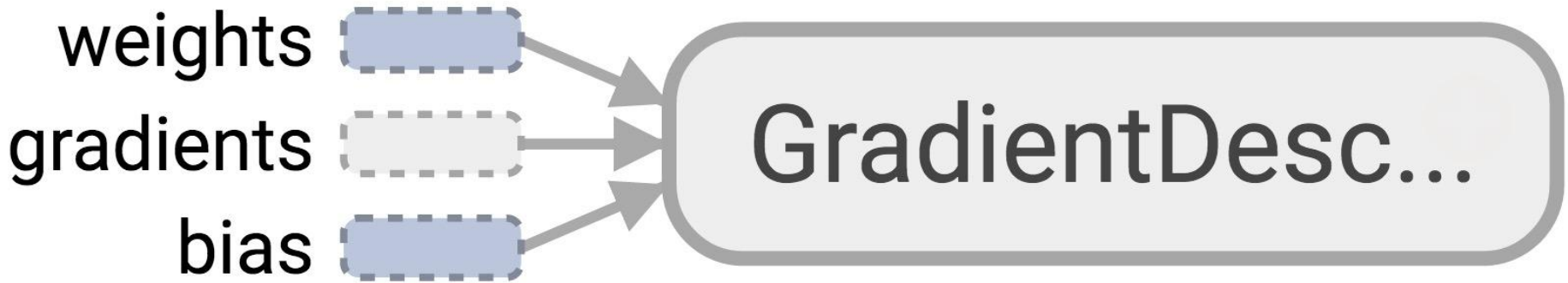
# Optimizer

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001).minimize(loss)  
_, l = sess.run([optimizer, loss], feed_dict={X: x, Y:y})
```

Session looks at all trainable variables that loss depends on and update them

# Optimizer

Session looks at all trainable variables that optimizer depends on and update them



# Trainable variables

```
tf.Variable(initial_value=None, trainable=True,...)
```

Specify if a variable should be trained or not  
By default, all variables are trainable

# List of optimizers in TF

`tf.train.GradientDescentOptimizer`

`tf.train.AdagradOptimizer`

`tf.train.MomentumOptimizer`

Usually Adam works out-of-the-box better than SGD

`tf.train.AdamOptimizer`

`tf.train.FtrlOptimizer`

`tf.train.RMSPropOptimizer`

...



# Manage Experiments

# **tf.train.Saver**

saves graph's variables in binary files

# Saves sessions, not graphs!

```
tf.train.Saver.save(sess, save_path, global_step=None...)  
tf.train.Saver.restore(sess, save_path)
```

# Save parameters after 1000 steps

```
# define model
model = SkipGramModel(params)

# create a saver object
saver = tf.train.Saver()

with tf.Session() as sess:
    for step in range(training_steps):
        sess.run([optimizer])

        # save model every 1000 steps
        if (step + 1) % 1000 == 0:
            saver.save(sess,
                        'checkpoint_directory/model_name',
                        global_step=step)
```



# Specify the step at which the model is saved

```
# define model
model = SkipGramModel(params)

# create a saver object
saver = tf.train.Saver()

with tf.Session() as sess:
    for step in range(training_steps):
        sess.run([optimizer])

        # save model every 1000 steps
        if (step + 1) % 1000 == 0:
            saver.save(sess,
                        'checkpoint_directory/model_name',
                        global_step=step)
```

# Global step

```
global_step = tf.Variable(0, dtype=tf.int32, trainable=False, name='global_step')
```

Very common in  
TensorFlow program














# Global step

```
global_step = tf.Variable(0,  
                          dtype=tf.int32,  
                          trainable=False,  
                          name='global_step')  
  
optimizer = tf.train.AdamOptimizer(lr).minimize(loss, global_step=global_step)
```

Need to tell optimizer to increment global step

This can also help your optimizer know when  
to decay learning rate

# Your checkpoints are saved in checkpoint\_directory

 checkpoint	265 bytes
 skip-gram-1000.data-00000-of-00001	51.4 MB
 skip-gram-1000.index	261 bytes
 skip-gram-1000.meta	87 KB
 skip-gram-2000.data-00000-of-00001	51.4 MB
 skip-gram-2000.index	261 bytes
 skip-gram-2000.meta	87 KB
 skip-gram-3000.data-00000-of-00001	51.4 MB
 skip-gram-3000.index	261 bytes
 skip-gram-3000.meta	87 KB
 skip-gram-4000.data-00000-of-00001	51.4 MB
 skip-gram-4000.index	261 bytes
 skip-gram-4000.meta	87 KB

# **tf.train.Saver**

Only save variables, not graph

Checkpoints map variable names to tensors

# tf.train.Saver

Can also choose to save certain variables

```
v1 = tf.Variable(..., name='v1')
```

```
v2 = tf.Variable(..., name='v2')
```

You can save your variables in one of three ways:

```
saver = tf.train.Saver({'v1': v1, 'v2': v2})
```

```
saver = tf.train.Saver([v1, v2])
```

```
saver = tf.train.Saver({v.op.name: v for v in [v1, v2]}) # similar to a dict
```

# Restore variables

```
saver.restore(sess, 'checkpoints/name_of_the_checkpoint')
```

```
e.g. saver.restore(sess, 'checkpoints/skip-gram-99999')
```

Still need to first build  
graph

# Restore the latest checkpoint

```
# check if there is checkpoint
ckpt = tf.train.get_checkpoint_state(os.path.dirname('checkpoints/checkpoint'))

# check if there is a valid checkpoint path
if ckpt and ckpt.model_checkpoint_path:
    saver.restore(sess, ckpt.model_checkpoint_path)
```

1. checkpoint file keeps track of the latest checkpoint
2. restore checkpoints only when there is a valid checkpoint path