TensorFlow Tutorial

https://docs.google.com/presentation/d/1_H1_XxknAWCwnscVJciN505y7G5hvDSNeF6Ej_3kfYo

```
git clone
https://github.com/wgchang/POSTECH_CV_Tutorials.git
```

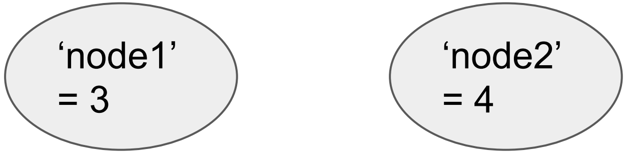
Contents

1. Tensorflow Basics
   a. Build Computational Graph
   b. Variables
   c. Sharing Variables
2. Image Classification Example
   a. CIFAR-10 classification

Build Computational Graph

Compute ‘3+4’

```
import tensorflow as tf

node1 = tf.constant(3.0, tf.float32)

node2 = tf.constant(4.0) # also tf.float32 implicitly

print(node1, node2)

=>   Tensor("Const:0", shape=(), dtype=float32) Tensor("Const_1:0", shape=(), dtype=float32)

sess = tf.Session()

print(sess.run([node1, node2]))

=>   [3.0, 4.0]
```



‘node1’
= 3

‘node2’
= 4

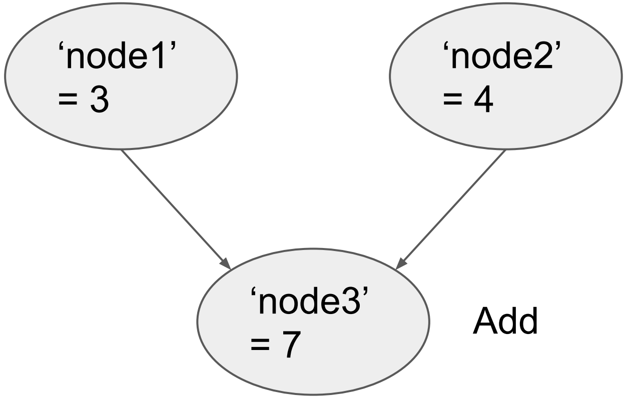
sess.run(node) gets the output of the ‘node’

Build Computational Graph

Compute ‘3+4’

```
node3 = tf.add(node1, node2)

print("node3: ", node3)

print("sess.run(node3): ",sess.run(node3))
```

```
node3:  Tensor("Add_2:0", shape=(), dtype=float32)

sess.run(node3):  7.0
```

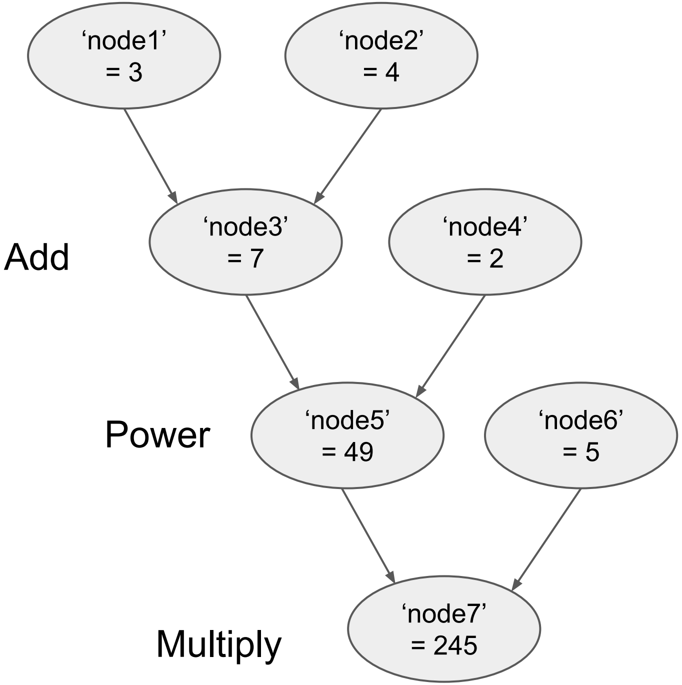


Build Computational Graph

Compute ‘((3+4)^2)*5’

```
node1 = tf.constant(3.0, tf.float32)

node2 = tf.constant(4.0, tf.float32)

node3 = tf.add(node1, node2)

node4 = tf.constant(2.0, tf.float32)

node5 = tf.pow(node3, node4)

node6 = tf.constant(5.0, tf.float32)

node7 = tf.multiply(node5, node6)
```

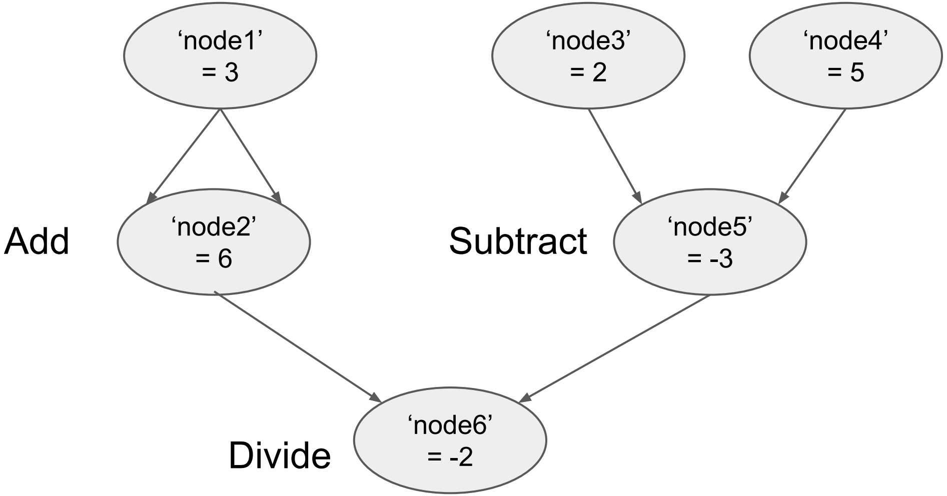
```
print(sess.run(node3))  =>  7

print(sess.run(node5))  =>  49
```



Build Computational Graph

Compute ‘((3+4)/(2-5))’

```
node1 = tf.constant(3.0, tf.float32)

node2 = tf.add(node1, node1)

node3 = tf.constant(2.0, tf.float32)

node4 =tf.constant(5.0, tf.float32)

node5 = tf.subtract(node4, node5)

node6 = tf.divide(node3, node6)
```



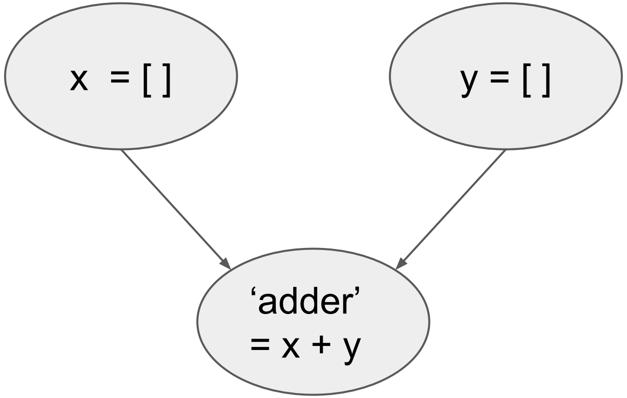
Build Computational Graph

Compute ‘x+y’

=> Define placeholder and graph first

=> feed value when sess.run

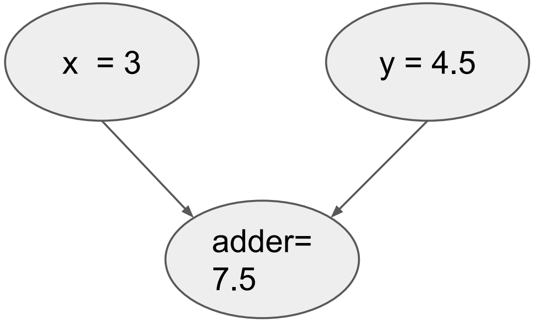
```
x = tf.placeholder(tf.float32)

y = tf.placeholder(tf.float32)

adder = tf.add(x, y)   # or just ‘x+y’ is okay
```



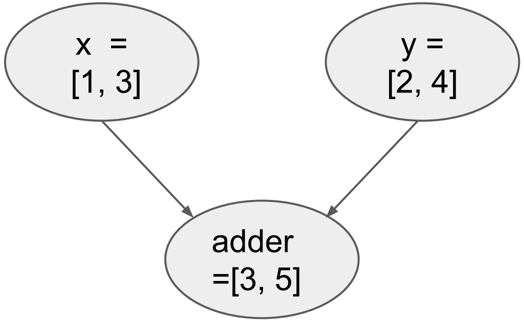
Build Computational Graph

Compute ‘x+y’

```
z = sess.run(adder, {x: 3, y: 4.5})

print(z)

=>   7.5
```

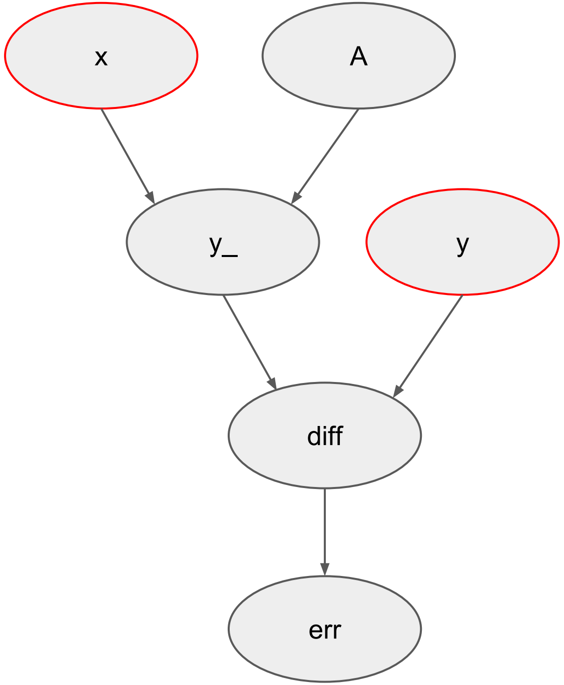


```
z = sess.run(adder, {x: [1,3], y: [2, 4]})

print(z)

=>   [ 3.  7.]
```



Build Computational Graph

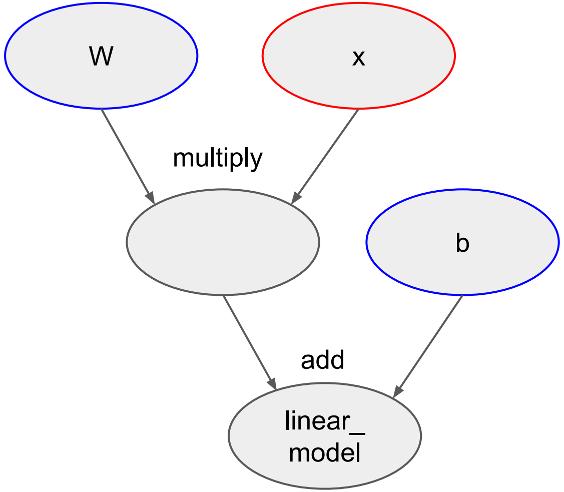
```
x = tf.placeholder(tf.float32)

y = tf.placeholder(tf.float32)

A = tf.constant([[1., 2.], [3., 4.]])

y_ = tf.matmul(A,x)

diff = y - y_

err = tf.norm(diff)

<=>         err = tf.norm(A*x-y)


print(sess.run(y_, {x: [[1.], [2.]]}))  =>  [[5.], [11.]]

print(sess.run(err, {x: [[1.], [2.]], y:[[8.], [7.]]}))  => 5.0
```



Learn Computational Graph

Learn linear model f(x) ≒ Wx+b

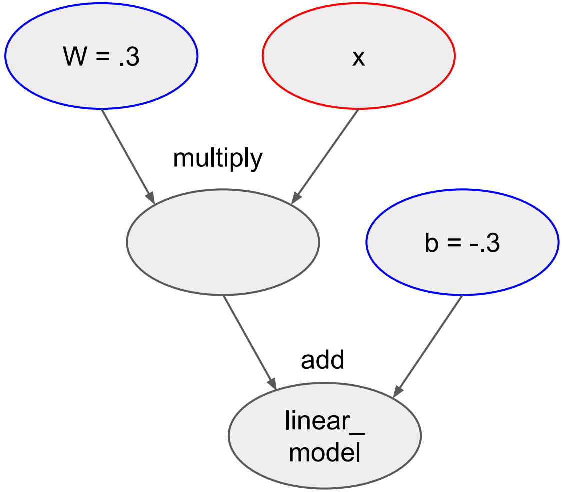
```
W = tf.Variable([.3], tf.float32)  # Variable adds learnable parameters in graph

b = tf.Variable([-.3], tf.float32)

x = tf.placeholder(tf.float32)

linear_model = W * x + b

print(sess.run(linear_model, {x:[1,2,3,4]}))  =>     ERROR!
```



Learn Computational Graph

Learn linear model f(x) ≒ Wx+b

```
W = tf.Variable([.3], tf.float32)  # Variable adds learnable parameters in graph

b = tf.Variable([-.3], tf.float32)

x = tf.placeholder(tf.float32)

linear_model = W * x + b

print(sess.run(linear_model, {x:[1,2,3,4]}))  =>     ERROR!
```



```
init = tf.global_variables_initializer()

sess.run(init)
```

Must initialize variables using initializers!

global_variables_initializer() initializes all the variables in a TensorFlow program

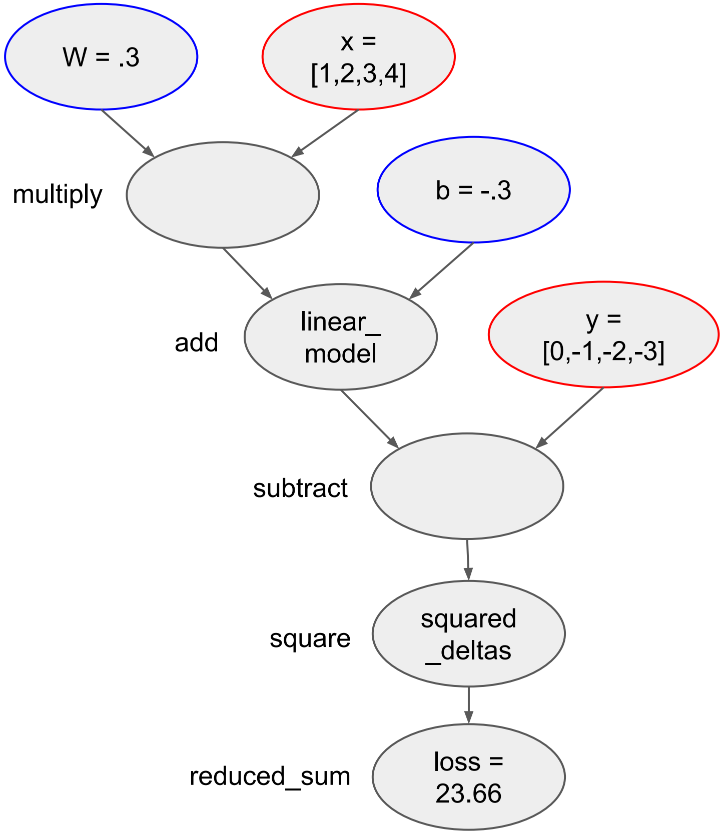
```
print(sess.run(linear_model, {x:[1,2,3,4]}))  =>     [ 0.          0.30000001  0.60000002  0.90000004]
```

Learn Computational Graph

Learn linear model f(x) ≒ Wx+b

Loss function : ∑||f(x)-y||^2

```
y = tf.placeholder(tf.float32)

squared_deltas = tf.square(linear_model - y)

loss = tf.reduce_sum(squared_deltas)


print(sess.run(loss, {x:[1,2,3,4], y:[0,-1,-2,-3]}))

23.66
```



Learn Computational Graph

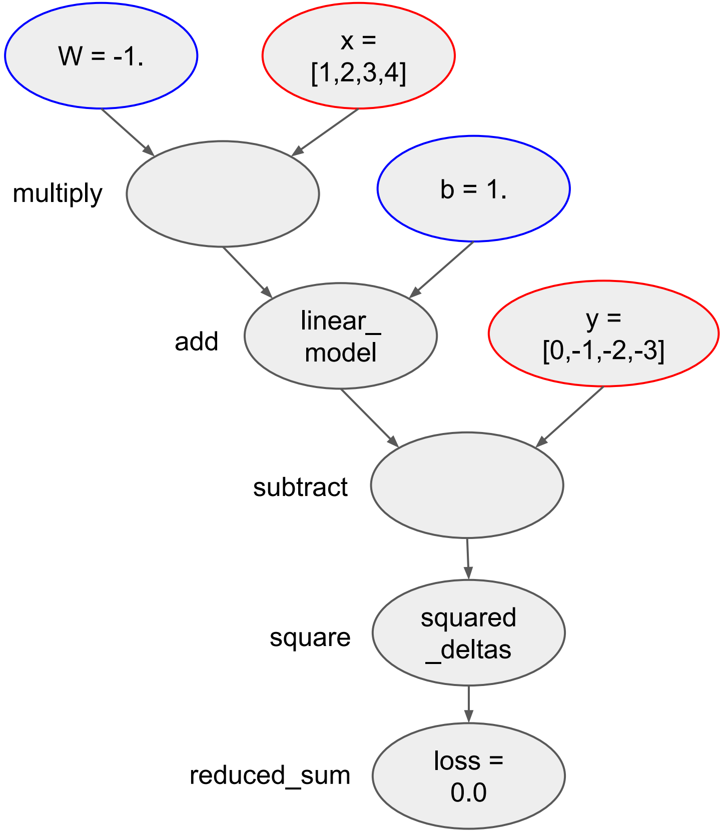
Learn linear model f(x) ≒ Wx+b

Loss function : ∑||f(x)-y||^2

Changing values of Variables

```
fixW = tf.assign(W, [-1.])

fixb = tf.assign(b, [1.])

sess.run([fixW, fixb])


print(sess.run(loss, {x:[1,2,3,4], y:[0,-1,-2,-3]}))

0.0
```

Same input & different output!



Learn Computational Graph

Minimize loss function by Optimizer

Simple Example : Stochastic Gradient Descent

```
optimizer = tf.train.GradientDescentOptimizer(0.01)

train = optimizer.minimize(loss)
```

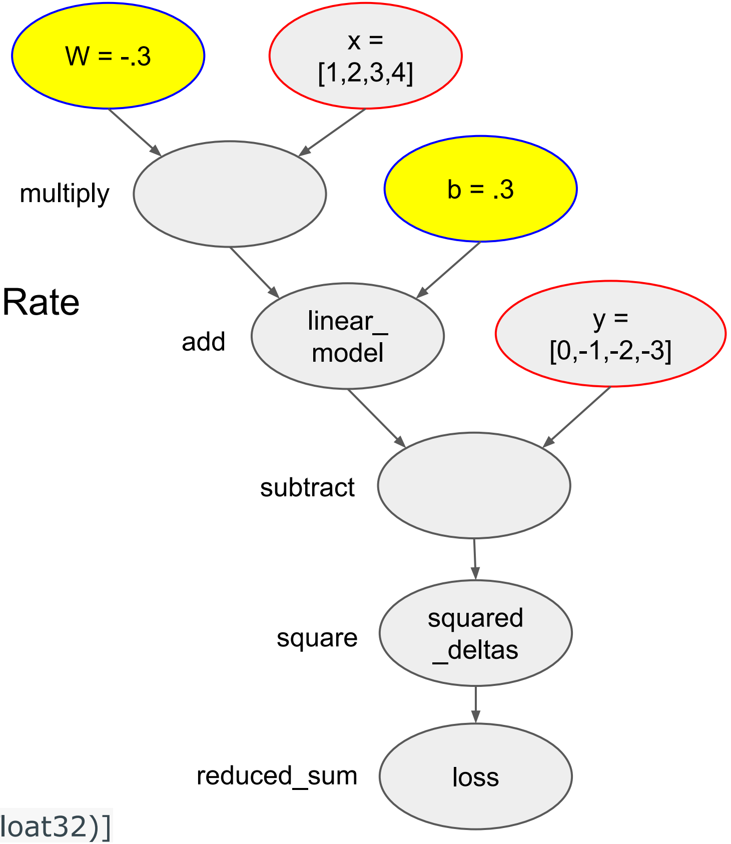
Learning Rate

```
sess.run(init) # reset values to incorrect defaults.
```

```
for i in range(1000):

  sess.run(train, {x:[1,2,3,4], y:[0,-1,-2,-3]})
```

```
print(sess.run([W, b]))
```

```
[array([-0.9999969], dtype=float32), array([ 0.99999082],  dtype=float32)]
```



Details about Variables ...

Creation : tf.Variables(Tensor, name)

size 784 X 200 tensor with random normal values

```
weights = tf.Variable(tf.random_normal([784, 200], stddev=0.35), name="weights")

biases = tf.Variable(tf.zeros([200]), name="biases")
```

size 200 tensor with zero values

Set device

```
# Pin a variable to CPU.

with tf.device("/cpu:0"):

  v = tf.Variable(...)
```

```
# Pin a variable to GPU.

with tf.device("/gpu:0"):

  v = tf.Variable(...)
```

```
# Pin a variable to a particular parameter server task.

with tf.device("/job:ps/task:7"):

  v = tf.Variable(...)
```

Operations modifying Variables (tf.Variable.assign / tf.train.Optimizer) should run on the same device with the mutating Variables

Initializing Variables

tf.global_variable_initializer() : An op to initialize ALL the Variables in the model.

```
weights = tf.Variable(tf.random_normal([784, 200], stddev=0.35), name="weights")

biases = tf.Variable(tf.zeros([200]), name="biases")        # Create two variables.

...

init_op = tf.global_variables_initializer()                 # Add an op to initialize the variables.

# Later, when launching the model

with tf.Session() as sess:

    sess.run(init_op)                # Run the init operation.

  ...

  # Use the model

  ...
```

Initializing Variables

If you want to initialize only specific variables :

Use `variables_initialize(var_list)` function, where `var_list` is the set of variables to initialize.

```
W1 = tf.Variable([.2], tf.float32)

W2 = tf.Variable([.4], tf.float32)

b = tf.Variable([-.3], tf.float32)


init = tf.global_variables_initializer()

sess.run(init)

print(sess.run([W1, W2, b]))
```

```
=>  [array([ 0.2], dtype=float32), array([ 0.40000001], dtype=float32), array([-0.30000001], dtype=float32)]
```

Initializing Variables

If you want to initialize only specific variables :

Use `variables_initialize(var_list)` function, where `var_list` is the set of variables to initialize.

```
W1 = tf.Variable([.2], tf.float32)

W2 = tf.Variable([.4], tf.float32)

b = tf.Variable([-.3], tf.float32)


init = tf.global_variables_initializer()

sess.run(init)

print(sess.run([W1, W2, b]))
```

```
assn_W1 = W1.assign([.1])

assn_W2 = W2.assign([.1])

assn_b = b.assign([-.1])


sess.run([assn_W1, assn_W2, assn_b])

print(sess.run([W1, W2, b]))
```

```
=>  [array([ 0.1], dtype=float32), array([ 0.1], dtype=float32), array([-0.1], dtype=float32)]
```

Initializing Variables

If you want to initialize only specific variables :

Use `variables_initialize(var_list)` function, where `var_list` is the set of variables to initialize.

```
W1 = tf.Variable([.2], tf.float32)

W2 = tf.Variable([.4], tf.float32)

b = tf.Variable([-.3], tf.float32)


init = tf.global_variables_initializer()

sess.run(init)

print(sess.run([W1, W2, b]))
```

```
assn_W1 = W1.assign([.1])

assn_W2 = W2.assign([.1])

assn_b = b.assign([-.1])


sess.run([assn_W1, assn_W2, assn_b])

print(sess.run([W1, W2, b]))
```

```
init_W = tf.variables_initializer(set([W1, b]))

sess.run(init_W)

print(sess.run([W1, W2, b]))
```

```
=>  [array([ 0.2], dtype=float32), array([-0.30000001], dtype=float32)]
```

Initializing Variables

If you want to initialize two Variables A, B to the same values:

Use `initialized_value()` property in the Variable!

```
A = tf.Variable(tf.random_normal([784, 200], stddev=0.35), name="A")

# Create another variable ‘B’ with the value of ‘A’

B = tf.Variable(A.initialized_value(), name="B")

# Create another variable ‘C’ with twice the value of ‘A’

C = tf.Variable(A.initialized_value() * 2.0, name=”C")
```

Saving and Loading Variables

Use `tf.train.Saver()`.

1. Save model

```
# Create some variables.
v1 = tf.Variable(..., name="v1")
v2 = tf.Variable(..., name="v2")
...
# Add an op to initialize the variables.
init_op = tf.global_variables_initializer()

# Add ops to save and restore all the variables.
saver = tf.train.Saver()
```

You can specify variable name in saving checkpoint file, or use tf.Varable.name as default.

```
# Later, launch the model, initialize the variables, do some work
# , then save the variables to disk.
with tf.Session() as sess:
    sess.run(init_op)
    # Do some work with the model.
    ..
    # Save the variables to disk.
    save_path = saver.save(sess, "model.ckpt")
    print("Model saved in file: %s" % save_path)
```

Saving and Loading Variables

Use `tf.train.Saver()`.

2. Load model

```
# Create some variables.

v1 = tf.Variable(..., name="v1")

v2 = tf.Variable(..., name="v2")

...

# Add ops to save and restore all the variables.

saver = tf.train.Saver()
```

When you restore variables from a file, you don’t have to initialize variables beforehand.

```
# Later, launch the model, use the saver to restore variables from disk

# , and do some work with the model.

with tf.Session() as sess:

    # Restore variables from disk.

    saver.restore(sess, "model.ckpt")

    print("Model restored.")

    # Do some work with the model

    ...
```

Saving and Loading Variables

Save and load only specific variables.

Use python dictionary to pass variables to save/load, key: name, value: variable

```
# Create some variables.

v1 = tf.Variable(..., name="v1")

v2 = tf.Variable(..., name="v2")

...

# Add ops to save and restore only 'v2' using the name "my_v2"

saver = tf.train.Saver({"my_v2": v2})

# Use the saver object normally after that.

...
```

Saving and Loading Variables

Save and load only specific variables.

```
v1 = tf.Variable(..., name="v1")
v2 = tf.Variable(..., name="v2")
...
# train v1, v2
saver = tf.train.Saver({“w1": v1, “w2": v2})
# Use the saver object normally after that.
save.save(sess, “model.ckpt")
...
```

```
w1 = tf.Variable(..., name=“w1")
w2 = tf.Variable(..., name=“w2")
w3 = tf.Variable(..., name=“w3")
w4 = tf.Variable(..., name=“w3")
restorer = tf.train.Saver({“w1": w1, “w2": w2})
# restore w1, w2 then initialize w3, w4
restorer.restore(sess, “model.ckpt”)
init = tf.variables_initializer(set([w3, w4]))
sess.run(init)
```

Sharing Variables

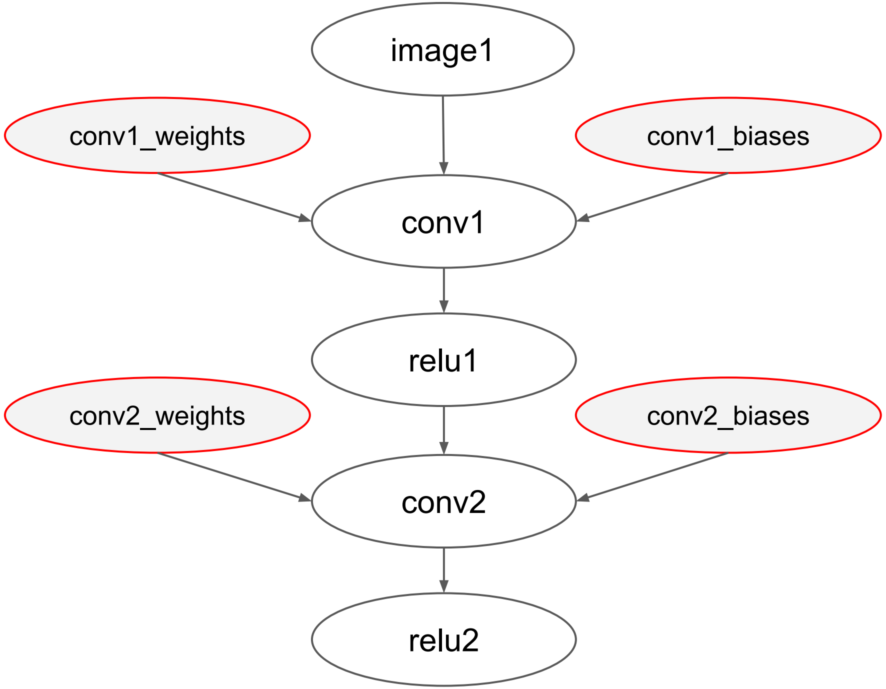
Imagine you created image filters with 2 convolutional layers this way:

```
def my_image_filter(input_images):
    conv1_weights = tf.Variable(tf.random_normal([5, 5, 32, 32]), name="conv1_weights")
    conv1_biases = tf.Variable(tf.zeros([32]), name="conv1_biases")
    conv1 = tf.nn.conv2d(input_images, conv1_weights, strides=[1, 1, 1, 1], padding='SAME')
    relu1 = tf.nn.relu(conv1 + conv1_biases)

    conv2_weights = tf.Variable(tf.random_normal([5, 5, 32, 32]), name="conv2_weights")
    conv2_biases = tf.Variable(tf.zeros([32]), name="conv2_biases")
    conv2 = tf.nn.conv2d(relu1, conv2_weights,  strides=[1, 1, 1, 1], padding='SAME')
    return tf.nn.relu(conv2 + conv2_biases)
```

Sharing Variables

Problem: You want to process two images (image1, image2) to same filters but whenever you call ‘my_image_filter’, new variables are created

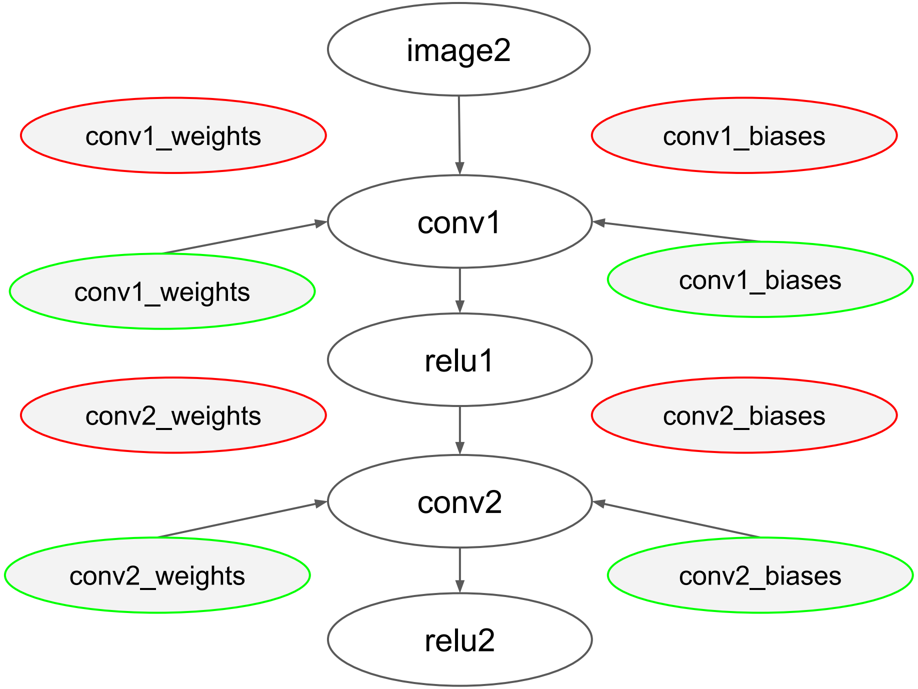
```
# First call creates one set of 4 variables.
result1 = my_image_filter(image1)
```



Sharing Variables

Problem: You want to process two images (image1, image2) to same filters but whenever you call ‘my_image_filter’, new variables are created

```
# First call creates one set of 4 variables.

result1 = my_image_filter(image1)

# Another set of 4 variables is created in the second call.

result2 = my_image_filter(image2)
```



Sharing Variables

One way to solve the problem… : Use seperate code for filters (ex. dictionary)

```
variables_dict = { "conv1_weights": tf.Variable(tf.random_normal([5, 5, 32, 32]), name="conv1_weights")
                   "conv1_biases": tf.Variable(tf.zeros([32]), name="conv1_biases")
   ... etc. ...}
```

```
def my_image_filter(input_images, variables_dict):
   conv1 = tf.nn.conv2d(input_images, variables_dict["conv1_weights"], strides=[1, 1, 1, 1], padding='SAME')
   relu1 = tf.nn.relu(conv1 + variables_dict["conv1_biases"])
   conv2 = tf.nn.conv2d(relu1, variables_dict["conv2_weights"], strides=[1, 1, 1, 1], padding='SAME')
   return tf.nn.relu(conv2 + variables_dict["conv2_biases"])

result1 = my_image_filter(image1, variables_dict)
result2 = my_image_filter(image2, variables_dict)   # Both calls to my_image_filter() now use the same variables
```

Sharing Variables

This is BAD solution… since,

Breaks Encapsulation!!

- The code that builds the graph must document the names, types, and shapes of variables to create.
- When the code changes, the callers may have to create more, or less, or different variables.

Solution : Use **tf.get_variable()** and **tf.variable_scope()**

Sharing Variables

tf.get_variable(<name>, <shape>, <initializer>)

- Get or create a variable instead of tf.Variable()
- Case I : Create a variable if a variable named with <name> doesn't exist, and raise error if does exist.
- Case II : Raise error if a variable named with <name> doesn't exist, but if does exists, then return that variable.

tf.variable_scope(<scope_name>)

- adds <scope_name> as the prefix of the <name> passed to get_variable
- Decide whether to reuse variables inside the scope by 'reuse' flag.
    - reuse == False => Case I.   reuse == True => Case II.

Sharing Variables

Some initializers in get_variable…

- `tf.constant_initializer(value)` : initializes everything to the provided value

- `tf.random_uniform_initializer(a, b)` : initializes uniformly from [a, b]

- `tf.random_normal_initializer(mean, stddev)` : initializes from the normal distribution with the given mean and standard deviation.

Sharing Variables

Adds <scope_name> in front of the variable name <name>

```
with tf.variable_scope("foo"):
    v = tf.get_variable("v", [1])
assert v.name == "foo/v:0"
```

```
with tf.variable_scope("foo"):
    with tf.variable_scope("bar"):
        v = tf.get_variable("v", [1])
assert v.name == "foo/bar/v:0"
```

By setting ‘reuse’ flag of variable_scope true, we can reuse variables by get_variable. The default flag is False.

```
with tf.variable_scope("foo"):
    v = tf.get_variable("v", [1])
with tf.variable_scope("foo", reuse=True):
    v1 = tf.get_variable("v", [1])
assert v1 is v        # v1 = v.
```

Sharing Variables

tf.get_variable_scope() retrieves the current variable scope and by using reuse_variables(), you can change the reuse flag to True to reuse variables inside the scope.

```
with tf.variable_scope("foo"):                      # tf.get_variable_scope().reuse == False
    v = tf.get_variable("v", [1])
    tf.get_variable_scope().reuse_variables()       # tf.get_variable_scope().reuse == True
    v1 = tf.get_variable("v", [1])
assert v1 is v
```

Setting ‘reuse’ back to False is impossible! => Breaking contract inside the scope.

Sharing Variables

Enter inside the reusing variable scope,

then exit back to non-reusing one.

```
with tf.variable_scope("root"):
    # At start, the scope is not reusing.
    assert tf.get_variable_scope().reuse == False
    with tf.variable_scope("foo"):
        # Opened a sub-scope, still not reusing.
        assert tf.get_variable_scope().reuse == False
    with tf.variable_scope("foo", reuse=True):
        # Explicitly opened a reusing scope.
        assert tf.get_variable_scope().reuse == True
        with tf.variable_scope("bar"):
            # Now sub-scope inherits the reuse flag.
            assert tf.get_variable_scope().reuse == True
    # Exited the reusing scope, back to a non-reusing one.
    assert tf.get_variable_scope().reuse == False
```

Sharing Variables

Alternative way... : Capturing Variable Scope

Can reuse variables without changing scope name

```
with tf.variable_scope("foo") as foo_scope:
    v = tf.get_variable("v", [1])
with tf.variable_scope(foo_scope):
    w = tf.get_variable("w", [1])
with tf.variable_scope(foo_scope, reuse=True):
    v1 = tf.get_variable("v", [1])
    w1 = tf.get_variable("w", [1])
assert v1 is v
assert w1 is w
```

```
# Doesn’t change scope name even though it is nested inside.
with tf.variable_scope("foo") as foo_scope:
    assert foo_scope.name == "foo"
with tf.variable_scope("bar"):
    with tf.variable_scope("baz") as other_scope:
        assert other_scope.name == "bar/baz"
        with tf.variable_scope(foo_scope) as foo_scope2:
            assert foo_scope2.name == "foo"  # Not changed.
```

Sharing Variables

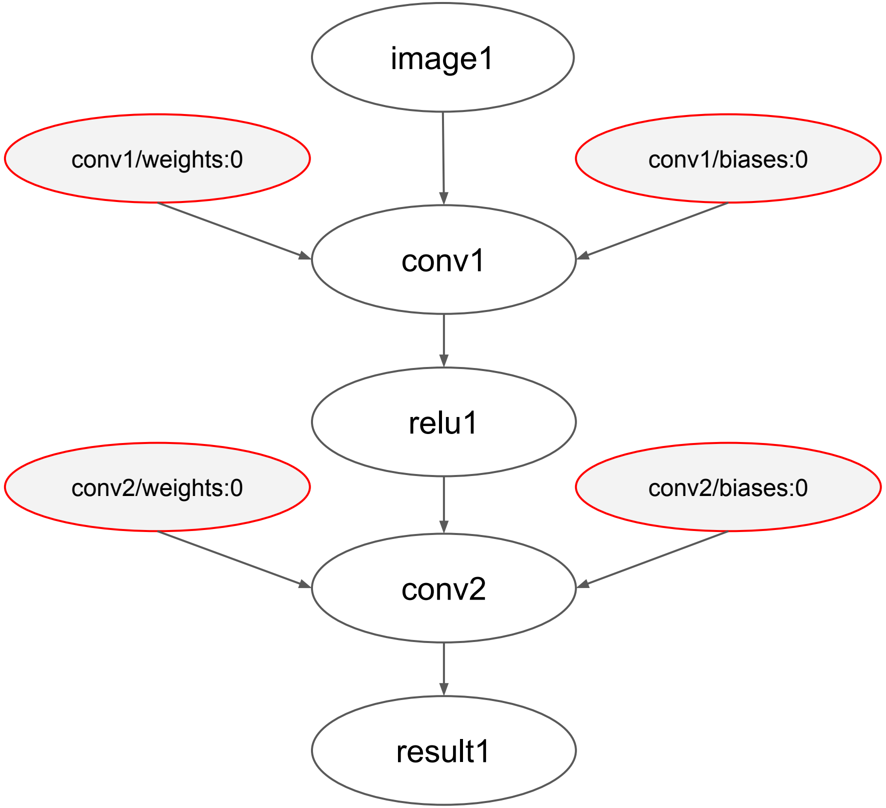
How to solve the previous problem?

```
def conv_relu(input, kernel_shape, bias_shape):
    # Create variable named "weights", “biases”.
    weights = tf.get_variable("weights", kernel_shape,  initializer=tf.random_normal_initializer())
    biases = tf.get_variable("biases", bias_shape, initializer=tf.constant_initializer(0.0))
    conv = tf.nn.conv2d(input, weights, strides=[1, 1, 1, 1], padding='SAME')
    return tf.nn.relu(conv + biases)

 def my_image_filter(input_images):
    with tf.variable_scope("conv1"):
        # Variables created here will be named "conv1/weights", "conv1/biases".
        relu1 = conv_relu(input_images, [5, 5, 32, 32], [32])
    with tf.variable_scope("conv2"):
        # Variables created here will be named "conv2/weights", "conv2/biases".
    return conv_relu(relu1, [5, 5, 32, 32], [32])
```

Sharing Variables

```
result1 = my_image_filter(image1)

result2 = my_image_filter(image2)

# Raises ValueError(... conv1/weights already exists ...)
```

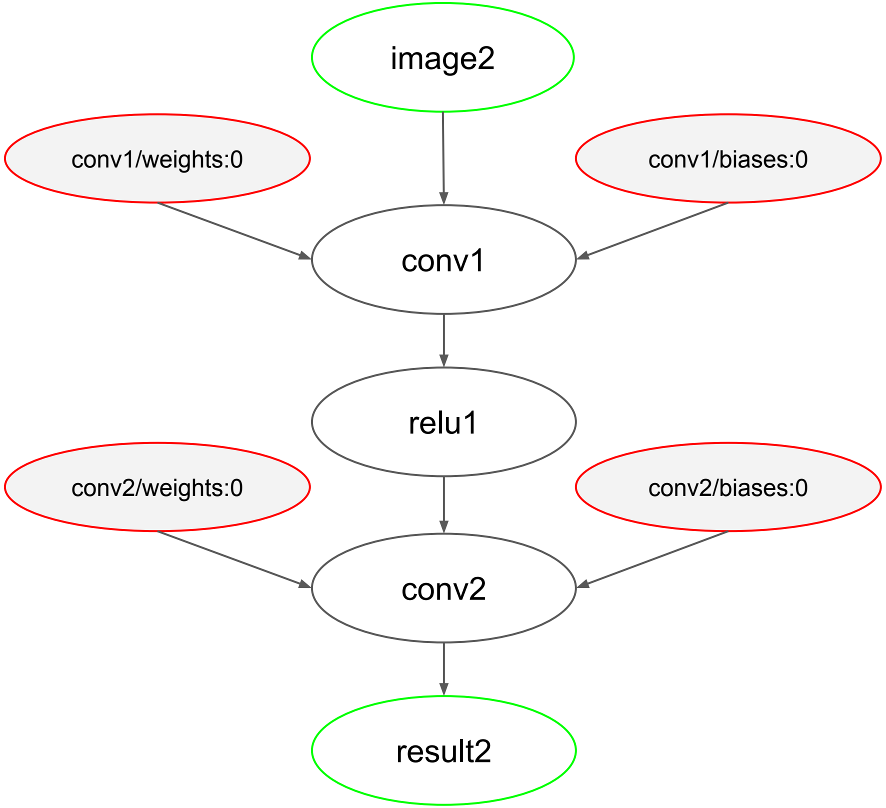
```
with tf.variable_scope("image_filters") as scope:

        result1 = my_image_filter(image1)
```



Sharing Variables

```
result1 = my_image_filter(image1)

result2 = my_image_filter(image2)

# Raises ValueError(... conv1/weights already exists ...)
```

```
with tf.variable_scope("image_filters") as scope:

        result1 = my_image_filter(image1)

        scope.reuse_variables()

        result2 = my_image_filter(image2)
```



More about variable scope...

tf.variable_scope() can carry the **default initializer** for get_variable!

```
with tf.variable_scope(“foo”, initializer=tf.constant_initializer(0.4)):
    v = tf.get_variable("v", [1])
    assert v.eval() == 0.4  # Default initializer as set above.
    w = tf.get_variable("w", [1], initializer=tf.constant_initializer(0.3)):
    assert w.eval() == 0.3  # Specific initializer overrides the default.
    with tf.variable_scope("bar"):
        v = tf.get_variable("v", [1])
        assert v.eval() == 0.4  # Inherited default initializer.
    with tf.variable_scope("baz", initializer=tf.random_normal_initializer()):
        v = tf.get_variable("v", [1])
        assert v.eval() == 0.176  # Changed default initializer.
```

Name scope

tf.name_scope(<scope_name>) governs operation name only,

while variable_scope governs both operation and tf.Variable name, but not tf.get_variable

```
with tf.variable_scope("foo"):
    x = 1.0 + tf.get_variable("v", [1])
assert x.op.name == "foo/add"
```

```
with tf.variable_scope("foo"):
    with tf.name_scope("bar"):
        v = tf.get_variable("v", [1])
        w = tf.Variable([.1], tf.float32)
        x = 1.0 + v
assert v.name == "foo/v:0"
assert w.name == “foo/bar/w:0”
assert x.op.name == "foo/bar/add"
```

Reference

Build Computational Graph

https://www.tensorflow.org/get_started/get_started

Variables

https://www.tensorflow.org/programmers_guide/variables

Sharing Variables

https://www.tensorflow.org/programmers_guide/variable_scope