11-695: Competitive Engineering
`TensorFlow`: Graphs, Execution, and Variables

Spring 2018

TensorFlow: Getting Started

CarnegieMellon

- Install: https://www.tensorflow.org/install/
- Usage:

tf_basic_program.py

```python
import tensorflow as tf

def main(_args):
  # your programs

if __name__ == "__main__":
  tf.app.run()
```

Structure of a tf program

- A program in *tf* always consists of:
  - Building a *computational graph*
  - Execute the relevant parts in the built graph

tf_basic_program.py

```
import tensorflow as tf

def main(_args):
  g = tf.Graph()         # create a computational graph
  with g.as_default():  # everything you do with TF happens in the graph g
    build_tf_graph()    # define the operations in g

    with tf.Session() as sess:                    # TF boiler-plate code
      sess.run(tf.global_variables_initializer())  # TF boiler-plate code

      # execute the TF graph, e.g.:
      sess.run([train_op, compute_loss])

if __name__ == "__main__":
  tf.app.run()
```

Outline

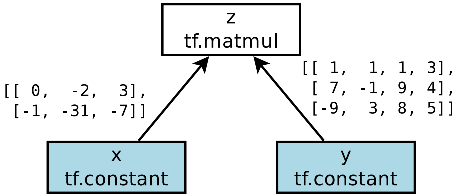
CarnegieMellon

1. Computational Graph
2. Execution Order
3. Variable

Computational Graph

tf_graph_demonstration_1.py

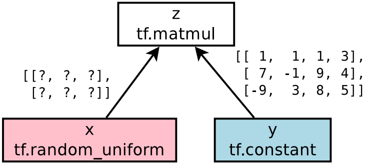
```python
import tensorflow as tf

def build_tf_graph():
  x = tf.constant([[0, -2, 3], [-1, -31, -7]], dtype=tf.int32)
  y = tf.constant([[1, 1, 1, 3], [7, -1, 9, 4], [-9, 3, 8, 5]], dtype=tf.int32)
  z = tf.matmul(x, y)
  return x, y, z

def main(_args):
  # other code...
  build_tf_graph()
  with tf.Session() as sess:
    output = sess.run([z])  # execute the operation z
    print(output)
```



Another Computational Graph

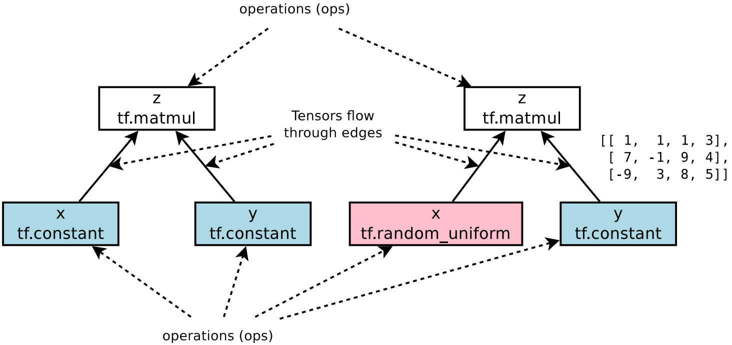
tf_graph_demonstration_2.py

```python
import tensorflow as tf

def build_tf_graph():
  x = tf.random_uniform([2, 3], minval=-5, maxval=5, dtype=tf.int32)
  y = tf.constant([[1, 1, 1, 3], [7, -1, 9, 4], [-9, 3, 8, 5]], dtype=tf.int32)
  z = tf.matmul(x, y)
  return x, y, z

def main(_args):
  # other code...
  x, y, z = build_tf_graph()
  with tf.Session() as sess:
    output = sess.run([z])  # execute the operation z
    print(output)
```



Formal Definition

Carnegie Mellon



- Formally speaking
  - `tf` computational graph is a *directed acycic graph* (DAG)
  - Nodes are called *operations*, or *ops*
  - Ops produce *tensors*
  - Tensors flow around through edges

Building TF Graphs

Carnegie Mellon

- Usually, `tf.some_thing()` creates a new ops and adds it to the computational graph.

Building `TF` Graphs

Carnegie Mellon

- Usually, `tf.some_thing()` creates a new ops and adds it to the computational graph.
- How does Python / TF know which graph are you referring to?

Building TF Graphs

- Usually, `tf.some_thing()` creates a new ops and adds it to the computational graph.
- How does Python / TF know which graph are you referring to?
  - `g.as_default()`

tf_graph_building.py

```
def build_tf_graph():
  # create ops

def main(_args):
  g = tf.Graph()
  with g.as_default():
    build_tf_graph()
```

Example TF Graphs

tf_graph_replication.py

```python
def build_tf_graph():
  x_values = np.random.uniform(-1.0, 1.0, [1000, 1000], dtype=np.float32)
  x = tf.constant(x_values, dtype=tf.float32)

  y = x ** 2  # creates an ops that takes x, returns x ** 2
  y = y + 1   # creates an ops that takes y, returns y + 1
  z = tf.nn.relu(y)  # creates an ops that takes y, returns max(y, 0)
```

- The variable names you see in Python has no meaning to TF.
- You can use them as *handles*, but TF doesn't care!

How to Build Complex Graphs?

Carnegie Mellon

- Same pattern as normal programming:
  - Use multiple files (and organize them appropriately)
  - Use functions, classes, inheritance, etc.

tf_graph_replication.py

```
from my_other_file import scary_network  # I made up all the the names

def complicated_neural_network(images):
  outputs = tf.convolution(images, ...)
  return outputs

def crazy_lstm_recurrent_convolution(inputs):
  outputs = tf.lstm(inputs, ...)
  outputs *= 100
  return outputs

def build_tf_graph():
  x = tf.input_images()  # I made this name up
  x = complicated_neural_network(x)
  x = crazy_lstm_recurrent_convolution(x)
  x = scary_network(x)
```

An Epic Failure

tf_graph_replication.py

```
def build_tf_graph():
  x_values = np.random.uniform(-1.0, 1.0, [1000, 1000], dtype=np.float32)
  x = tf.constant(x_values, dtype=tf.float32)

  # this fails
  for step in range(1000000000):
    x += 1.0

  # this works (but is very slow)
  for step in range(1000000000):
    x_values += 1.0

def main(_args):
  g = tf.Graph()
  with g.as_default():
    build_tf_graph()
```

An Epic Failure

tf_graph_replication.py

```python
def build_tf_graph():
  x_values = np.random.uniform(-1.0, 1.0, [1000, 1000], dtype=np.float32)
  x = tf.constant(x_values, dtype=tf.float32)

  # this fails
  for step in range(1000000000):
    x += 1.0

  # this works (but is very slow)
  for step in range(1000000000):
    x_values += 1.0

def main(_args):
  g = tf.Graph()
  with g.as_default():
    build_tf_graph()
```

- Each `x += 1` creates a new ops and does *not* override the old ops.

An Epic Failure

CarnegieMellon

tf_graph_replication.py

```
def build_tf_graph():
  x_values = np.random.uniform(-1.0, 1.0, [1000, 1000], dtype=np.float32)
  x = tf.constant(x_values, dtype=tf.float32)

  # this fails
  for step in range(1000000000):
    x += 1.0

  # this works (but is very slow)
  for step in range(1000000000):
    x_values += 1.0

def main(_args):
  g = tf.Graph()
  with g.as_default():
    build_tf_graph()
```

- Each `x += 1` creates a new ops and does *not* override the old ops.
- Out of memory (`TF` graphs need memory to store too).

A Final Note on TF Graph Building

CarnegieMellon

- Many functions that you use seems to take inputs and and return outputs
- But they actually just *add more ops* to your computational graphs
- and return the ops' *handles* so that you can make more ops
- Lesson: always check for `type` when you program with `TensorFlow` and `numpy`!!!

tf_graph_replication.py

```
def softmax(images):
  batch_size = tf.shape(images)[0]
  images = tf.reshape(images, [batch_size, -1])

  # don't care about these. we'll discuss them later
  images_dim = images.get_shape()[-1]
  w = tf.get_variable("w", [images_dim, 10])

  logits = tf.matmul(images, w)
  probs = tf.nn.softmax(logits)
  return probs
```

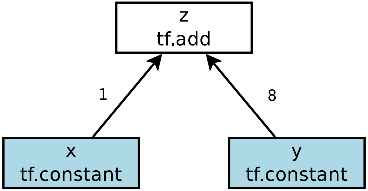
Outline

CarnegieMellon

Yet Another Computational Graph

tf_execution_order.py

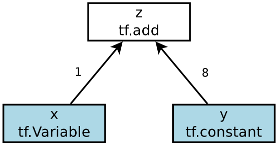
```
import tensorflow as tf

def build_tf_graph():
  x = tf.constant(1, dtype=tf.int32)
  y = tf.constant(8, dtype=tf.int32)
  z = x + y
  return x, y, z

def main(_args):
  # other code...
  x, y, z = build_tf_graph()
  with tf.Session() as sess:
    output = sess.run([x, y, z])  # execute all 3 operations
    print(output)  # output: [1, 8, 9]
```



What about now?

tf_execution_order_2.py

```python
def build_tf_graph():
  x = tf.Variable(1, dtype=tf.int32, name="x")
  y = tf.constant(8, dtype=tf.int32)
  z = x + y
  return x, y, z

def main(_args):
  # other code...
  with tf.Session() as sess:
    output = sess.run([x, y, z])  # execute all 3 operations
    print(output)                 # output: [1, 8, 9]
```

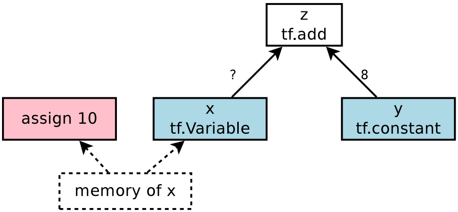


- Unlike `tf.constant`, `tf.Variable` can be changed

What about now?

tf_execution_order_3.py

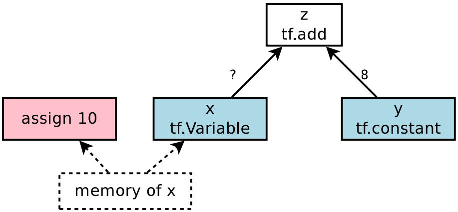
```
def build_tf_graph():
  x = tf.Variable(1, dtype=tf.int32, name="x")
  assign_x = tf.assign(x, 10)
  y = tf.constant(8, dtype=tf.int32)
  z = x + y
  return x, y, z, assign_x

def main(_args):
  # other code...
  with tf.Session() as sess:
    output = sess.run([z, assign_x])  # execute all 3 operations
    print(output)                     # output: ?
```



What about now?

tf_execution_order_3.py

```python
def build_tf_graph():
  x = tf.Variable(1, dtype=tf.int32, name="x")
  assign_x = tf.assign(x, 10)
  y = tf.constant(8, dtype=tf.int32)
  z = x + y
  return x, y, z, assign_x

def main(_args):
  # other code...
  with tf.Session() as sess:
    output = sess.run([z, assign_x])  # execute all 3 operations
    print(output)                     # output: ?
```

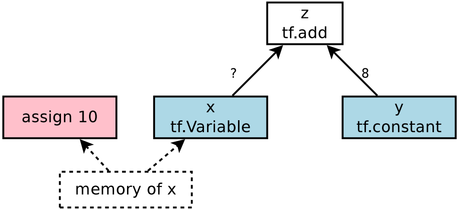


We don't know!

What about now?

tf_execution_order_3.py

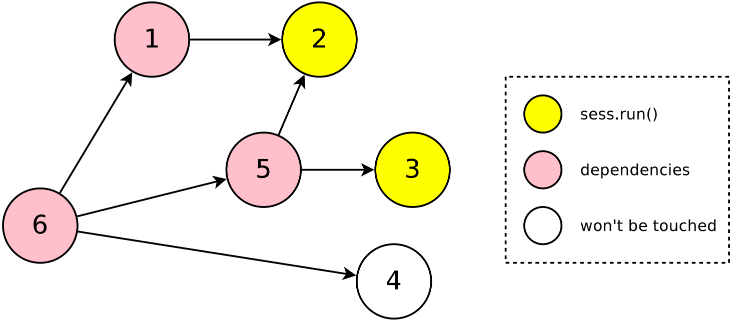
```
def build_tf_graph():
  x = tf.Variable(1, dtype=tf.int32, name="x")
  assign_x = tf.assign(x, 10)
  y = tf.constant(8, dtype=tf.int32)
  z = x + y
  return x, y, z, assign_x

def main(_args):
  # other code...
  with tf.Session() as sess:
    output = sess.run([z, assign_x])  # execute all 3 operations
    print(output)                     # output: ?
```



We don't know!

TF Execution Order

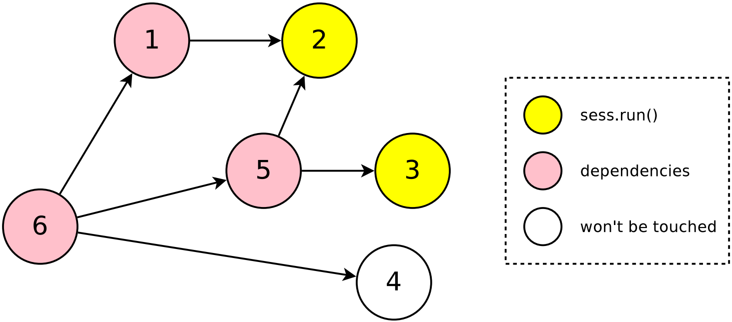
Carnegie Mellon



- Execution order follows the computational graph's topological order.

TF Execution Order

Carnegie Mellon



- Execution order follows the computational graph's topological order.
- and nothing else!

TF Race Conditions

Carnegie Mellon

tf_execution_order_4.py

```
def build_tf_graph():
  x = tf.Variable(1, dtype=tf.int32, name="x")
  assign_1 = tf.assign(x, 1)
  assign_2 = tf.assign(x, 2)

def main(_args):
  with tf.Session() as sess:
    _, _, x_value = sess.run([assign_1, assign_2, x])
    print(x_value)  # of course we don't know the output, but it's worse...
```

- Do we know the output of `print`?

TF Race Conditions

Carnegie Mellon

tf_execution_order_4.py

```
def build_tf_graph():
  x = tf.Variable(1, dtype=tf.int32, name="x")
  assign_1 = tf.assign(x, 1)
  assign_2 = tf.assign(x, 2)

def main(_args):
  with tf.Session() as sess:
    _, _, x_value = sess.run([assign_1, assign_2, x])
    print(x_value)  # of course we don't know the output, but it's worse...
```

- Do we know the output of `print`?
    - No!

TF Race Conditions

CarnegieMellon

tf_execution_order_4.py

```python
def build_tf_graph():
  x = tf.Variable(1, dtype=tf.int32, name="x")
  assign_1 = tf.assign(x, 1)
  assign_2 = tf.assign(x, 2)

def main(_args):
  with tf.Session() as sess:
    _, _, x_value = sess.run([assign_1, assign_2, x])
    print(x_value)  # of course we don't know the output, but it's worse...
```

- Do we know the output of `print`?
  - No!
- Do we know which value will be stored at `x`?

TF Race Conditions

Carnegie Mellon

tf_execution_order_4.py

```
def build_tf_graph():
  x = tf.Variable(1, dtype=tf.int32, name="x")
  assign_1 = tf.assign(x, 1)
  assign_2 = tf.assign(x, 2)

def main(_args):
  with tf.Session() as sess:
    _, _, x_value = sess.run([assign_1, assign_2, x])
    print(x_value)  # of course we don't know the output, but it's worse...
```

- Do we know the output of `print`?
  - No!
- Do we know which value will be stored at `x`?
  - No!

TF Race Conditions

tf_execution_order_4.py

```
def build_tf_graph():
  x = tf.Variable(1, dtype=tf.int32, name="x")
  assign_1 = tf.assign(x, 1)
  assign_2 = tf.assign(x, 2)

def main(_args):
  with tf.Session() as sess:
    _, _, x_value = sess.run([assign_1, assign_2, x])
    print(x_value)  # of course we don't know the output, but it's worse...
```

- Do we know the output of `print`?
  - No!
- Do we know which value will be stored at `x`?
  - No!
- Does the program even run *safely*?

TF Race Conditions

tf_execution_order_4.py

```python
def build_tf_graph():
  x = tf.Variable(1, dtype=tf.int32, name="x")
  assign_1 = tf.assign(x, 1)
  assign_2 = tf.assign(x, 2)

def main(_args):
  with tf.Session() as sess:
    _, _, x_value = sess.run([assign_1, assign_2, x])
    print(x_value)  # of course we don't know the output, but it's worse...
```

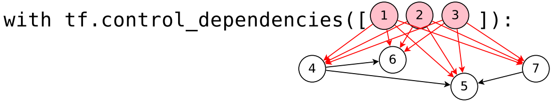
- Do we know the output of `print`?
  - No!
- Do we know which value will be stored at `x`?
  - No!
- Does the program even run *safely*?
  - No! `x` can become `NaN`

Imposing an Execution Order

tf_execution_dependency.py

```python
def build_tf_graph():
  x = tf.Variable(1, dtype=tf.int32, name="x")
  assign_1 = tf.assign(x, 1)
  with tf.control_dependencies([assign_1]):
    assign_5 = tf.assign(x, 5)

def main(_args):
  with tf.Session() as sess:
    sess.run([assign_1, assign_5])  # assign_1 is run first, then assign_5
    print(sess.run(x))              # output: 5
```

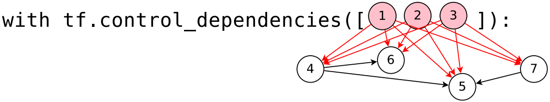
- `tf.control_dependencies([ops_1, ops_2, ops_3])`
  - `ops_1`, `ops_2`, `ops_3` are parents of everything in the `with` block.
  - `sess.run([anything_in_the_block])` will trigger them all



Imposing an Execution Order

tf_execution_dependency.py

```python
def build_tf_graph():
  x = tf.Variable(1, dtype=tf.int32, name="x")
  assign_1 = tf.assign(x, 1)
  with tf.control_dependencies([assign_1]):
    assign_5 = tf.assign(x, 5)

def main(_args):
  with tf.Session() as sess:
    sess.run([assign_1, assign_5])  # assign_1 is run first, then assign_5
    print(sess.run(x))              # output: 5
```



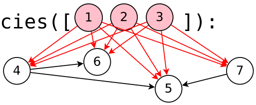
- What if you create a loop?

Imposing an Execution Order

tf_execution_dependency.py

```python
def build_tf_graph():
  x = tf.Variable(1, dtype=tf.int32, name="x")
  assign_1 = tf.assign(x, 1)
  with tf.control_dependencies([assign_1]):
    assign_5 = tf.assign(x, 5)

def main(_args):
  with tf.Session() as sess:
    sess.run([assign_1, assign_5])  # assign_1 is run first, then assign_5
    print(sess.run(x))              # output: 5
```

`with tf.control_dependencies([ ]):`



- What if you create a loop?
  - You cannot!
  - Only created ops can be passed to `tf.control_dependencies([...])`

Duplications in `sess.run()`

tf_execution_double_call.py

```
def build_tf_graph():
  x = tf.Variable(1, dtype=tf.int32, name="x")
  inc_1 = tf.assign_add(x, 1)

def main(_args):
  with tf.Session() as sess:
    sess.run([inc_1, inc_1, inc_1])
```

- What happens?

Duplications in `sess.run()`

tf_execution_double_call.py

```
def build_tf_graph():
  x = tf.Variable(1, dtype=tf.int32, name="x")
  inc_1 = tf.assign_add(x, 1)

def main(_args):
  with tf.Session() as sess:
    sess.run([inc_1, inc_1, inc_1])
```

- What happens?
  - Nothing unusual. `x` is increased by `1`.
  - No race conditions!

Duplications in `sess.run()`

tf_execution_double_call.py

```python
def build_tf_graph():
  x = tf.Variable(1, dtype=tf.int32, name="x")
  inc_1 = tf.assign_add(x, 1)

def main(_args):
  with tf.Session() as sess:
    sess.run([inc_1, inc_1, inc_1])
```

- What happens?
  - Nothing unusual. `x` is increased by `1`.
  - No race conditions!
- Why?

Duplications in `sess.run()`

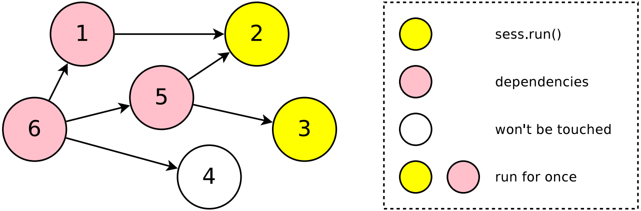
tf_execution_double_call.py

```python
def build_tf_graph():
  x = tf.Variable(1, dtype=tf.int32, name="x")
  inc_1 = tf.assign_add(x, 1)

def main(_args):
  with tf.Session() as sess:
    sess.run([inc_1, inc_1, inc_1])
```

- What happens?
  - Nothing unusual. `x` is increased by `1`.
  - No race conditions!
- Why?
  - `TF` runs everything in the induced graph *exactly once.*

Execution Order: Summary

Carnegie Mellon



- When you call `sess.run(ops_to_run)`:
  - `TF` looks for all parents of `ops_to_run`
  - `TF` marks all these parent nodes (i.e. the *induced graph*)
  - `TF` *forgets* what you put in `ops_to_run`
  - `TF` runs all the nodes the induced graph, once
  - `TF` preserves the dependencies in the induced graph, if any

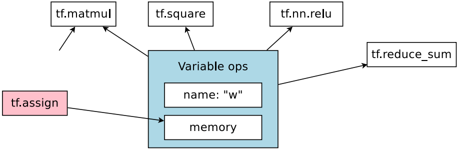
Outline

CarnegieMellon

Creating variable with `tf.get_variable`

tf_variable.py

```python
def build_tf_graph():
  w = tf.get_variable("w", [500, 1000])  # create a variable called "w"
                                         # with shape [500, 1000]
```



- `tf.Variable` can be “written” to
  - Unlike other `tf` ops
- `tf.Variable` stores trainable parameters of machine learning models
  - or whatever you wish :)

Variable Name

Carnegie Mellon

tf_another_variable.py

```python
def build_tf_graph():
  w = tf.get_variable("w", [500, 1000])
  another_w = tf.get_variable("w", [500, 1000])
  assign_w = tf.assign(w, np.ones([500, 1000]))
  assign_another_w = tf.assign(w, np.ones([500, 1000]))
```

- `w` and `another_w` are `the same tf.Variable`

Variable Name

Carnegie Mellon

tf_another_variable.py

```python
def build_tf_graph():
  w = tf.get_variable("w", [500, 1000])
  another_w = tf.get_variable("w", [500, 1000])
  assign_w = tf.assign(w, np.ones([500, 1000]))
  assign_another_w = tf.assign(w, np.ones([500, 1000]))
```

- `w` and `another_w` are `the same tf.Variable`
- `assign_w` and `assign_another_w` are *two different ops*

Variable Name

Carnegie Mellon

tf_another_variable.py

```python
def build_tf_graph():
  w = tf.get_variable("w", [500, 1000])
  another_w = tf.get_variable("w", [500, 1000])
  assign_w = tf.assign(w, np.ones([500, 1000]))
  assign_another_w = tf.assign(w, np.ones([500, 1000]))
```

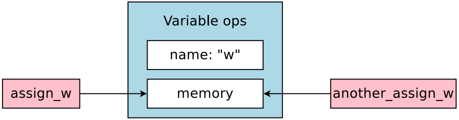
- `w` and `another_w` are `the same tf.Variable`
- `assign_w` and `assign_another_w` are *two different ops*
  - but do the same thing!

Variable Name

tf_another_variable.py

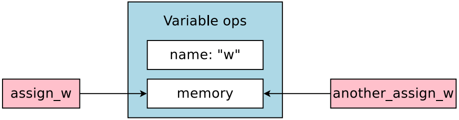
```
def build_tf_graph():
  w = tf.get_variable("w", [500, 1000])
  another_w = tf.get_variable("w", [500, 1000])
  assign_w = tf.assign(w, np.ones([500, 1000]))
  assign_another_w = tf.assign(w, np.ones([500, 1000]))
```

- `w` and `another_w` are `the same tf.Variable`
- `assign_w` and `assign_another_w` are *two different ops*
  - but do the same thing!



Variable Name

Carnegie Mellon



- `TF` computational graphs store their variables using `TF` names
- These are different from Python names
- You can use a variable's `TF` name to retrieve it

Retrieving a Variable by Name

Carnegie Mellon

tf_get_var_no_shape.py

```
def build_tf_graph():
  w = tf.get_variable("w", [500, 1000])
  the_same_w = tf.get_variable("w", [500, 1000])
  another_same_w = tf.get_variable("w")  # this is also okay!
```

- `TF` knows which `"w"` you are calling, no need to tell the shape again

Retrieving a Variable by Name

Carnegie Mellon

tf_get_var_with_shape.py

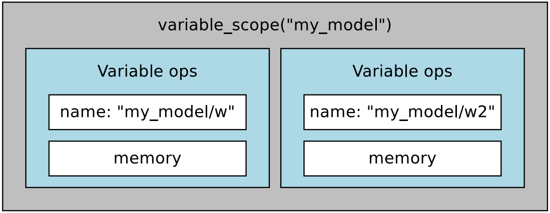
```
def build_tf_graph():
  w = tf.get_variable("w", [500, 1000])
  the_same_w = tf.get_variable("w", [500, 1000])         # the same w as above
  another_same_w = tf.get_variable("w")                  # this is okay
  yet_another_same_w = tf.get_variable("w", [501, 1000])  # this is not!
```

- TF knows which `"w"` you are calling, no need to tell the shape again
- TF knows you are trying to trick it!
  - `tf.get_variable` with an existing name ignores the shape
  - `tf.get_variable` with an existing name and a different shape will complain!

Variable Scope

tf_variable_scope.py

```python
def build_tf_graph():
  with tf.variable_scope("my_model"):
    w = tf.get_variable("w", [500, 1000])
    w2 = tf.get_variable("w2", [50, 50])
```



- Pad variables' `TF` names with prefixes
- Used when there are many variables to organize
  - e.g. `"mat_mul/w"` and `"convolution/w"` are weights for a matrix multiplication and a convolution.

Variable Scope and reuse

Carnegie Mellon

tf_var_scope_reuse.py

```
def build_tf_graph():
  with tf.variable_scope("my_model", reuse=True):
    w = tf.get_variable("w")    # "my_model/w" must be created before
    w2 = tf.get_variable("w2")  # "my_model/w2" must be created before
```

- You can use `reuse=True` in a `variable_scope` to force all `tf.get_variable("var_name")` to look up created variables.
- Throw errors if the variable with the name is not created before
- Used when building multiple graphs, loading variables, etc.