

11-695: AI Engineering

Introduction to TensorFlow

Spring 2019

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

tf_basic_program.py

```
1 import tensorflow as tf
2
3 def main(_args):
4     # your programs
5
6 if __name__ == "__main__":
7     tf.app.run()
```

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

tf_basic_program.py

```
1 import tensorflow as tf
2
3 def main(_args):
4     g = tf.Graph()           # create a computational graph
5     with g.as_default():    # everything you do with TF happens in the graph g
6         build_tf_graph()    # define the operations in g
7
8     with tf.Session() as sess:           # TF boiler-plate code
9         sess.run(tf.global_variables_initializer()) # TF boiler-plate code
10
11     # execute the TF graph, e.g.:
12     sess.run([train_op, compute_loss])
13
14 if __name__ == "__main__":
15     tf.app.run()
```

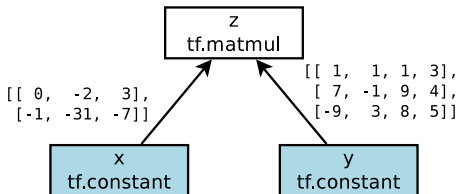
① Computational Graph

② Execution Order

③ Variable

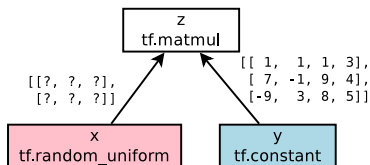
tf_graph_demonstration_1.py

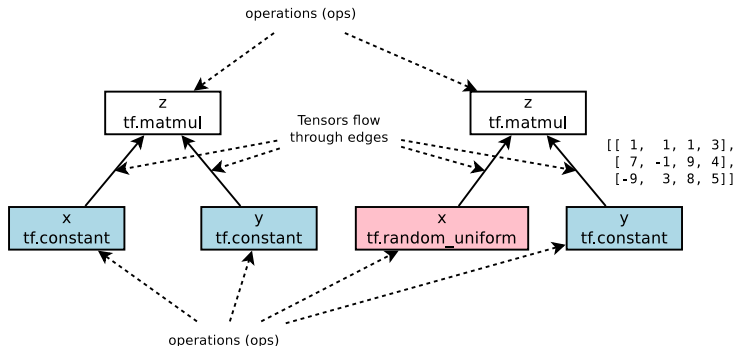
```
1 import tensorflow as tf
2
3 def build_tf_graph():
4     x = tf.constant([[0, -2, 3], [-1, -31, -7]], dtype=tf.int32)
5     y = tf.constant([[1, 1, 1, 3], [7, -1, 9, 4], [-9, 3, 8, 5]], dtype=tf.int32)
6     z = tf.matmul(x, y)
7     return x, y, z
8
9 def main(_args):
10     # other code...
11     build_tf_graph()
12     with tf.Session() as sess:
13         output = sess.run([z]) # execute the operation z
14         print(output)
```



tf_graph_demonstration_2.py

```
1 import tensorflow as tf
2
3 def build_tf_graph():
4     x = tf.random_uniform([2, 3], minval=-5, maxval=5, dtype=tf.int32)
5     y = tf.constant([[1, 1, 1, 3], [7, -1, 9, 4], [-9, 3, 8, 5]], dtype=tf.int32)
6     z = tf.matmul(x, y)
7     return x, y, z
8
9 def main(_args):
10     # other code...
11     x, y, z = build_tf_graph()
12     with tf.Session() as sess:
13         output = sess.run([z]) # execute the operation z
14         print(output)
```





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

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

- 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?

- 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

```
1 def build_tf_graph():
2     # create ops
3
4 def main(_args):
5     g = tf.Graph()
6     with g.as_default():
7         build_tf_graph()
```

tf_graph_replication.py

```
1 def build_tf_graph():
2     x_values = np.random.uniform(-1.0, 1.0, [1000, 1000], dtype=np.float32)
3     x = tf.constant(x_values, dtype=tf.float32)
4
5     y = x ** 2 # creates an ops that takes x, returns x ** 2
6     y = y + 1  # creates an ops that takes y, returns y + 1
7     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.
- And in most cases, TF doesn't care!

tf_graph_replication.py

```
1 def build_tf_graph():
2     x_values = np.random.uniform(-1.0, 1.0, [1000, 1000], dtype=np.float32)
3     x = tf.constant(x_values, dtype=tf.float32)
4
5     y = x ** 2 # creates an ops that takes x, returns x ** 2
6     y = y + 1  # creates an ops that takes y, returns y + 1
7     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.
- And in most cases, TF doesn't care!
- Why?

- Same as many things: as a rule of thumb, divide and conquer!

- Same as many things: as a rule of thumb, divide and conquer!
- i.e. you modularize your big graph into easily-controlled sub-modules
 - Use multiple files (and organize them appropriately)
 - Use functions, classes, inheritance, etc.

tf_graph_replication.py

```
1 from my_other_file import scary_network # I made up all the the names
2
3 def complicated_neural_network(images):
4     outputs = tf.convolution(images, ...)
5     return outputs
6
7 def crazy_lstm_recurrent_convolution(inputs):
8     outputs = tf.lstm(inputs, ...)
9     outputs *= 100
10    return outputs
11
12 def build_tf_graph():
13     x = tf.input_images() # I made this name up
14     x = complicated_neural_network(x)
15     x = crazy_lstm_recurrent_convolution(x)
```

tf_graph_replication.py

```
1 def build_tf_graph():
2     x_values = np.random.uniform(-1.0, 1.0, [1000, 1000])
3     x = tf.constant(x_values, dtype=tf.float32)
4
5     # this fails
6     for _ in range(1000000000):
7         x += 1.0
8
9     # this is okay (but should not be here)
10    for _ in range(1000000000):
11        x_values += 1.0
12
13 def main(_args):
14     g = tf.Graph()
15     with g.as_default():
16         build_tf_graph()
17
18 # first, anything does not look right?
```

tf_graph_replication.py

```
1 def build_tf_graph():
2     x_values = np.random.uniform(-1.0, 1.0, [1000, 1000])
3     x = tf.constant(x_values, dtype=tf.float32)
4
5     # this fails
6     for _ in range(1000000000):
7         x += 1.0
8
9     # this is okay (but should not be here)
10    for _ in range(1000000000):
11        x_values += 1.0
12
13 def main(_args):
14     g = tf.Graph()
15     with g.as_default():
16         build_tf_graph()
17
18 # first, anything does not look right?
```

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

tf_graph_replication.py

```
1 def build_tf_graph():
2     x_values = np.random.uniform(-1.0, 1.0, [1000, 1000])
3     x = tf.constant(x_values, dtype=tf.float32)
4
5     # this fails
6     for _ in range(1000000000):
7         x += 1.0
8
9     # this is okay (but should not be here)
10    for _ in range(1000000000):
11        x_values += 1.0
12
13 def main(_args):
14     g = tf.Graph()
15     with g.as_default():
16         build_tf_graph()
17
18 # first, anything does not look right?
```

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

- 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

```
1 def softmax(images):
2     batch_size = tf.shape(images)[0]
3     images = tf.reshape(images, [batch_size, -1])
4
5     # don't care about these. we'll discuss them later
6     images_dim = images.get_shape()[-1]
7     w = tf.get_variable("w", [images_dim, 10])
8
9     logits = tf.matmul(images, w)
10    probs = tf.nn.softmax(logits)
11    return probs
```

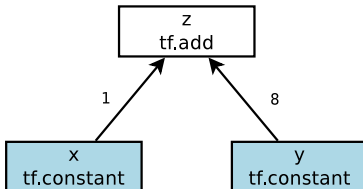
① Computational Graph

② Execution Order

③ Variable

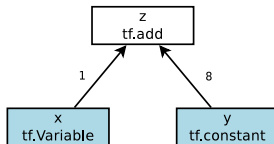
tf_execution_order.py

```
1 import tensorflow as tf
2
3 def build_tf_graph():
4     x = tf.constant(1, dtype=tf.int32)
5     y = tf.constant(8, dtype=tf.int32)
6     z = x + y
7     return x, y, z
8
9 def main(_args):
10     # other code...
11     x, y, z = build_tf_graph()
12     with tf.Session() as sess:
13         output = sess.run([x, y, z]) # execute all 3 operations
14         print(output) # output: [1, 8, 9]
```



tf_execution_order_2.py

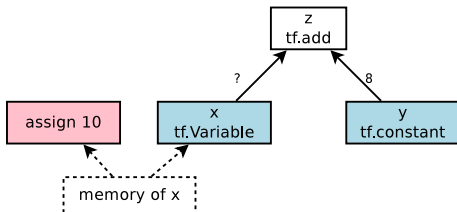
```
1 def build_tf_graph():
2     x = tf.Variable(1, dtype=tf.int32, name="x")
3     y = tf.constant(8, dtype=tf.int32)
4     z = x + y
5     return x, y, z
6
7 def main(_args):
8     # other code...
9     with tf.Session() as sess:
10         # we skip one important line here, not relevant right now tho
11
12         output = sess.run([x, y, z]) # execute all 3 operations
13         print(output)                # output: [1, 8, 9]
```



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

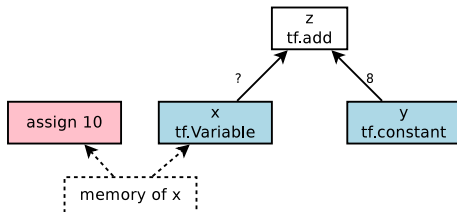
tf_execution_order_3.py

```
1 def build_tf_graph():
2     x = tf.Variable(1, dtype=tf.int32, name="x")
3     assign_x = tf.assign(x, 10)
4     y = tf.constant(8, dtype=tf.int32)
5     z = x + y
6     return x, y, z, assign_x
7
8 def main(_args):
9     # other code...
10    with tf.Session() as sess:
11        output = sess.run([z, assign_x]) # execute all 3 operations
12        print(output)                   # output: ?
```



tf_execution_order_3.py

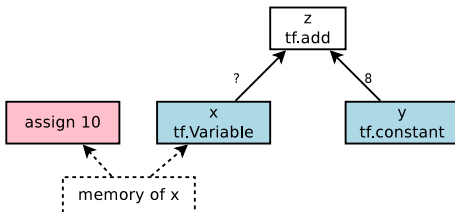
```
1 def build_tf_graph():
2     x = tf.Variable(1, dtype=tf.int32, name="x")
3     assign_x = tf.assign(x, 10)
4     y = tf.constant(8, dtype=tf.int32)
5     z = x + y
6     return x, y, z, assign_x
7
8 def main(_args):
9     # other code...
10    with tf.Session() as sess:
11        output = sess.run([z, assign_x]) # execute all 3 operations
12        print(output)                   # output: ?
```



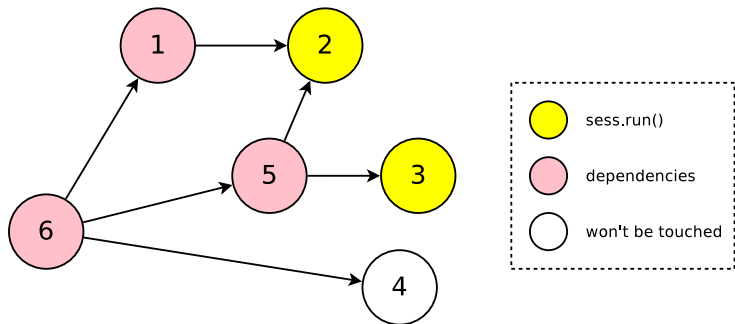
We don't know!

tf_execution_order_3.py

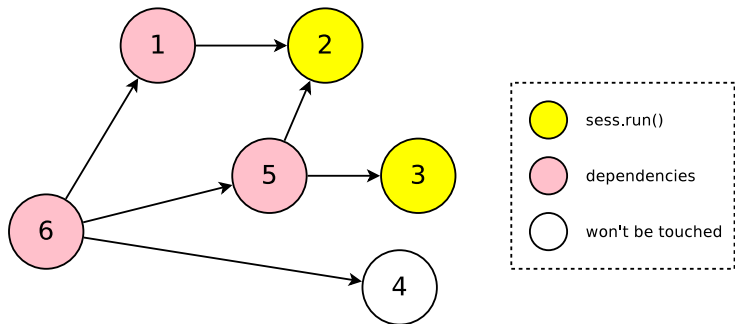
```
1 def build_tf_graph():
2     x = tf.Variable(1, dtype=tf.int32, name="x")
3     assign_x = tf.assign(x, 10)
4     y = tf.constant(8, dtype=tf.int32)
5     z = x + y
6     return x, y, z, assign_x
7
8 def main(_args):
9     # other code...
10    with tf.Session() as sess:
11        output = sess.run([z, assign_x]) # execute all 3 operations
12        print(output)                   # output: ?
```



We don't know!



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



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

tf_execution_order_4.py

```
1 def build_tf_graph():
2     x = tf.Variable(1, dtype=tf.int32, name="x")
3     assign_1 = tf.assign(x, 1)
4     assign_2 = tf.assign(x, 2)
5     return x, assign_1, assign_2
6
7 def main(_args):
8     x, assign_1, assign_2 = build_tf_graph()
9
10    with tf.Session() as sess:
11        _, _, x_value = sess.run([assign_1, assign_2, x])
12        print(x_value) # of course we don't know the output, but it's worse...
```

- Do we know the output of print?

tf_execution_order_4.py

```
1 def build_tf_graph():
2     x = tf.Variable(1, dtype=tf.int32, name="x")
3     assign_1 = tf.assign(x, 1)
4     assign_2 = tf.assign(x, 2)
5     return x, assign_1, assign_2
6
7 def main(_args):
8     x, assign_1, assign_2 = build_tf_graph()
9
10    with tf.Session() as sess:
11        _, _, x_value = sess.run([assign_1, assign_2, x])
12        print(x_value) # of course we don't know the output, but it's worse...
```

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

tf_execution_order_4.py

```
1 def build_tf_graph():
2     x = tf.Variable(1, dtype=tf.int32, name="x")
3     assign_1 = tf.assign(x, 1)
4     assign_2 = tf.assign(x, 2)
5     return x, assign_1, assign_2
6
7 def main(_args):
8     x, assign_1, assign_2 = build_tf_graph()
9
10    with tf.Session() as sess:
11        _, _, x_value = sess.run([assign_1, assign_2, x])
12        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_execution_order_4.py

```
1 def build_tf_graph():
2     x = tf.Variable(1, dtype=tf.int32, name="x")
3     assign_1 = tf.assign(x, 1)
4     assign_2 = tf.assign(x, 2)
5     return x, assign_1, assign_2
6
7 def main(_args):
8     x, assign_1, assign_2 = build_tf_graph()
9
10    with tf.Session() as sess:
11        _, _, x_value = sess.run([assign_1, assign_2, x])
12        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_execution_order_4.py

```
1 def build_tf_graph():
2     x = tf.Variable(1, dtype=tf.int32, name="x")
3     assign_1 = tf.assign(x, 1)
4     assign_2 = tf.assign(x, 2)
5     return x, assign_1, assign_2
6
7 def main(_args):
8     x, assign_1, assign_2 = build_tf_graph()
9
10    with tf.Session() as sess:
11        _, _, x_value = sess.run([assign_1, assign_2, x])
12        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_execution_order_4.py

```
1 def build_tf_graph():
2     x = tf.Variable(1, dtype=tf.int32, name="x")
3     assign_1 = tf.assign(x, 1)
4     assign_2 = tf.assign(x, 2)
5     return x, assign_1, assign_2
6
7 def main(_args):
8     x, assign_1, assign_2 = build_tf_graph()
9
10    with tf.Session() as sess:
11        _, _, x_value = sess.run([assign_1, assign_2, x])
12        print(x_value) # of course we don't know the output, but it's worse...
```

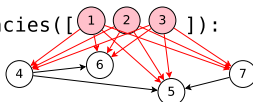
- Do we know the output of `print`?
 - No!
- Do we know which value will be stored at `x`?
 - No!
- Does the program even run *safely*?
 - No!

tf_execution_dependency.py

```
1 def build_tf_graph():
2     x = tf.Variable(1, dtype=tf.int32, name="x")
3     assign_1 = tf.assign(x, 1)
4     with tf.control_dependencies([assign_1]):
5         assign_5 = tf.assign(x, 5)
6
7 def main(_args):
8     # build graph here
9     with tf.Session() as sess:
10         # the important execution line is omitted
11         sess.run([assign_1, assign_5]) # assign_1 is run first, then assign_5
12         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

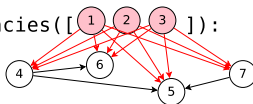
`with tf.control_dependencies([`



tf_execution_dependency.py

```
1 def build_tf_graph():
2     x = tf.Variable(1, dtype=tf.int32, name="x")
3     assign_1 = tf.assign(x, 1)
4     with tf.control_dependencies([assign_1]):
5         assign_5 = tf.assign(x, 5)
6
7 def main(_args):
8     # build graph here
9     with tf.Session() as sess:
10         # the important execution line is omitted
11         sess.run([assign_1, assign_5]) # assign_1 is run first, then assign_5
12         print(sess.run(x))             # output: 5
```

with tf.control_dependencies([

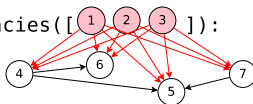


- What if you create a loop?

tf_execution_dependency.py

```
1 def build_tf_graph():
2     x = tf.Variable(1, dtype=tf.int32, name="x")
3     assign_1 = tf.assign(x, 1)
4     with tf.control_dependencies([assign_1]):
5         assign_5 = tf.assign(x, 5)
6
7 def main(_args):
8     # build graph here
9     with tf.Session() as sess:
10         # the important execution line is omitted
11         sess.run([assign_1, assign_5]) # assign_1 is run first, then assign_5
12         print(sess.run(x))             # output: 5
```

with tf.control_dependencies([1, 2, 3]):



- What if you create a loop?
 - You cannot!
 - Only created ops can be passed to

tf_execution_double_call.py

```
1 def build_tf_graph():
2     x = tf.Variable(1, dtype=tf.int32, name="x")
3     inc_1 = tf.assign_add(x, 1)
4
5 def main(_args):
6     # build graph here
7     with tf.Session() as sess:
8         # the important execution line is omitted
9         sess.run([inc_1, inc_1, inc_1])
```

- What happens?

tf_execution_double_call.py

```
1 def build_tf_graph():
2     x = tf.Variable(1, dtype=tf.int32, name="x")
3     inc_1 = tf.assign_add(x, 1)
4
5 def main(_args):
6     # build graph here
7     with tf.Session() as sess:
8         # the important execution line is omitted
9         sess.run([inc_1, inc_1, inc_1])
```

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

tf_execution_double_call.py

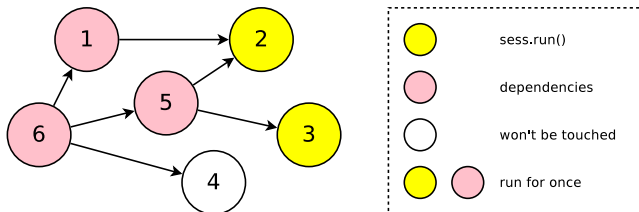
```
1 def build_tf_graph():
2     x = tf.Variable(1, dtype=tf.int32, name="x")
3     inc_1 = tf.assign_add(x, 1)
4
5 def main(_args):
6     # build graph here
7     with tf.Session() as sess:
8         # the important execution line is omitted
9         sess.run([inc_1, inc_1, inc_1])
```

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

tf_execution_double_call.py

```
1 def build_tf_graph():
2     x = tf.Variable(1, dtype=tf.int32, name="x")
3     inc_1 = tf.assign_add(x, 1)
4
5 def main(_args):
6     # build graph here
7     with tf.Session() as sess:
8         # the important execution line is omitted
9         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*.



- 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

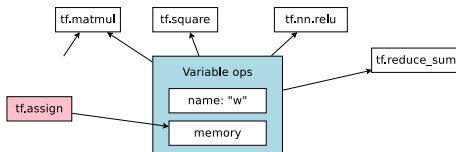
① Computational Graph

② Execution Order

③ Variable

`tf_variable.py`

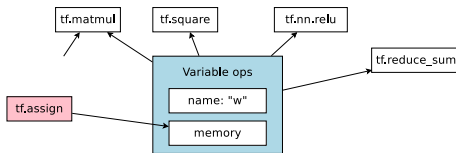
```
1 def build_tf_graph():
2     w = tf.get_variable("w", [500, 1000]) # create a variable called "w"
3                                           # with shape [500, 1000]
```



- `tf.Variable` can be “written” to
 - Unlike other `tf` ops
 - Notice something different in the name?

`tf_variable.py`

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



- `tf.Variable` can be “written” to
 - Unlike other `tf` ops
 - Notice something different in the name?
- `tf.Variable` stores trainable parameters of machine learning models
 - or whatever you wish :)

tf_another_variable.py

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

- w and another_w are the same tf.Variable

tf_another_variable.py

```
1 def build_tf_graph():
2     w = tf.get_variable("w", [500, 1000])
3     another_w = tf.get_variable("w", [500, 1000])
4     assign_w = tf.assign(w, np.ones([500, 1000]))
5     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*

tf_another_variable.py

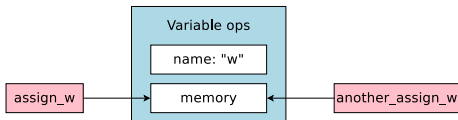
```
1 def build_tf_graph():
2     w = tf.get_variable("w", [500, 1000])
3     another_w = tf.get_variable("w", [500, 1000])
4     assign_w = tf.assign(w, np.ones([500, 1000]))
5     assign_another_w = tf.assign(w, np.ones([500, 1000]))
```

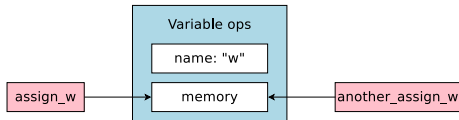
- `w` and `another_w` are the same `tf.Variable`
- `assign_w` and `assign_another_w` are *two different ops*
 - but do the same thing!

tf_another_variable.py

```
1 def build_tf_graph():
2     w = tf.get_variable("w", [500, 1000])
3     another_w = tf.get_variable("w", [500, 1000])
4     assign_w = tf.assign(w, np.ones([500, 1000]))
5     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!





- 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

tf_get_var_no_shape.py

```
1 def build_tf_graph():
2     w = tf.get_variable("w", [500, 1000])
3     the_same_w = tf.get_variable("w", [500, 1000])
4     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

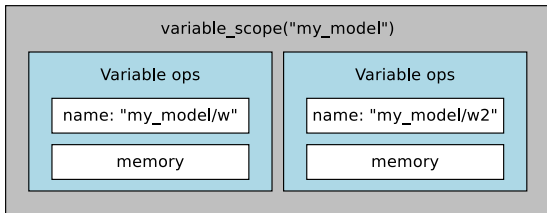
tf_get_var_with_shape.py

```
1 def build_tf_graph():
2     w = tf.get_variable("w", [500, 1000])
3     the_same_w = tf.get_variable("w", [500, 1000])           # the same w as above
4     another_same_w = tf.get_variable("w")                   # this is okay
5     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!

tf_variable_scope.py

```
1 def build_tf_graph():
2     with tf.variable_scope("my_model"):
3         w = tf.get_variable("w", [500, 1000])
4         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.

tf_var_scope_reuse.py

```
1 def build_tf_graph():
2     with tf.variable_scope("my_model", reuse=True):
3         w = tf.get_variable("w")      # "my_model/w" must be created before
4         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,
- or TF is confused about creating a new var or reusing the current one ¹
- Used when building multiple graphs, loading variables, etc.

¹<https://www.tensorflow.org/guide/variables>