# 11-695: AI Engineering
# Eager Execution

Spring 2019

# Outline

# Eager Execution - First Steps

Carnegie Mellon

tf_import_eager.py

```
1  import tensoflow as tf
2  tf.enable_eager_execution()
3
4  x = tf.constant(1) # same constant declaration
5  y = tf.constant(2)
6  x # <tf.Tensor: id=0, shape=(), dtype=int32, numpy=1>
7  z = x + y
8  z # <tf.Tensor: id=6, shape=(), dtype=int32, numpy=3>
9  z.numpy() # 3
```

- Enable Eager Execution
  - At the very beginning
  - Tensor has the numpy values along
  - Same TF operators `add`, `multiply`, ...
- No session
- No placeholder
- But variables are the same (`tf.Variable`)

# Outline

# Deal with `tf.data.Dataset`

Carnegie Mellon

tf_eager_data.py

```
1  data = tf.data.Dataset.from_tensor_slices((img_path, img_labels))
2  # doing shuffling, batching, repeating
3
4  # then
5  for batch_data in data:
6    (img, labels) = batch_data
7    # do smth
8
9  # or
10 for (batch_num, (img, labels)) in enumerate(data):
11   # do smth
```

- How `tf.data.Dataset` consumes, converts, transforms data stays the same
- Iterate the Same as when you work with Python

# Outline

# tf.GradientTape

Carnegie Mellon

tf_gtape.py

```
1  x = tf.Variable([[1.0, 2.0]]) # float
2  with tf.GradientTape() as tape:
3    tape.watch(x) # x must be a Tensor first, it's auto done
4    # create 'graph' of ops here
5    y = tf.square(x)
6
7  # call gradient() outside the context
8  # <tf.Tensor: id=92, shape=(1, 2), dtype=float32, numpy=array([[2., 4.]], dtype=float32)>
9  dy_x = tape.gradient(y, x)
10
11 dy_x = tape.gradient(y, x) # error! tape is not persistent
```

- Tape is a sequential storage device
- Gradient Tape records gradient in order of operations
  - Under its umbrella of *context*
  - It records ops so that later can calculate grads
  - It costs memory

# tf.GradientTape

Carnegie Mellon

tf_gtape_fn.py

```
1  def grad(model, x, y):
2    with tf.GradientTape() as tape:
3      loss_value = loss(model, x, y)
4    return tape.gradient(loss_value, model.variables)
5
6  # take grads
7  grads = grad(model, x, y)
8
9  # Backprop: Apply the grads to the model's variables
10 optimizer.apply_gradients(zip(grads, model.variables))
```

- Calculate gradient of the `loss_value` *w.r.t* `x, y`
- Normally, avoid grads *calculation* inside tape's *context*

# tf.GradientTape

Carnegie Mellon

tf_gtape_fn_2.py

```
1  def Model1(tf.keras.Model):
2      ...
3
4  def Model2(tf.keras.Model):
5     ...
6
7  model1 = Model1()
8  model2 = Model2()
9
10 variables = model1.variables() + model2.variables()
11 grads = tape.gradient(loss, variables)
12
13 optimizer.apply_gradients(zip(gradients, variables))
```

- Calculate gradient of the `loss_value` *w.r.t* `x, y`
- Normally, avoid grads *calculation* inside tape's *context*

# tf.GradientTape



- Play more with code:
  - ▸ tf.GradientTape API
  - Tutorial: ▸ Auto Differentiation
  - Tutorial: ▸ TF Eager
  - `GradientTape` with ▸ MNIST

# Outline

# Save and Restore

Carnegie Mellon

tf_save_restore.py

```
model1 = Model1()
model2 = Model2()
opt = tf.train.AdamOptimizer()

# define a checkpoint object wrapping all objects in K-V type
checkpoint = tf.train.Checkpoint(model1=model1, model2=model2, opt=opt)
checkpoint_prefix = './ckpt' # directory to save

# save
checkpoint.save(checkpoint_prefix)  # ckpt-1.index, ckpt-1.data-00000-of-00001, ...

# normally restore the latest one
checkpoint.restore(tf.train.latest_checkpoint(checkpoint_path))
```

- Using `tf.train.Checkpoint` to wrap any object you have
- Can use validation score to save the best latest checkpoint
- Tutorial: ▸ Object-Based Saving
- More Tutorial: ▸ tf.keras-Based Save and Restore