11-695: Competitive Engineering
Dynamic Computational Graphs

Spring 2018

Where are we?

Carnegie Mellon

tf_feed_forward_net.py

```
1  def feed_forward_net(x, dims=[256, 512, 128], num_classes=10):
2    # x is a tensor of size [N, H, W, C]
3    N = tf.shape(x)[0]
4    H, W, C = [dim.value for dim in x.get_shape()[1:]]
5    x = tf.reshape(x, [N, H * W * C])  # flatten
6    for layer_id, next_dim in enumerate(dims):
7      curr_dim = x.get_shape()[-1].value  # get_shape() returns a <list>
8      with tf.variable_scope("layer_{}".format(layer_id)):
9        w = tf.get_variable("w", [curr_dim, next_dim])  # w's name: "layer_2/w"
10     x = tf.matmul(x, w)
11     x = tf.nn.relu(x)
12   curr_dim = x.get_shape()[-1].value  # get_shape() returns a <list>
13   with tf.variable_scope("logits"):
14     w = tf.get_variable("w", [curr_dim, num_classes])  # w's name: "logits/w"
15     logits = tf.matmul(x, w)
16   return logits
```

- Corrections from last time:
  - TF cannot infer the dimensions `H`, `W`, `C`
  - `tf.shape` is for the dimensions *not* known at graph construction
  - `x.get_shape` is for the dimensions *known* at graph construction

The Need for Dynamic Graphs

Carnegie Mellon

tf_feed_forward_net.py

```
1  def feed_forward_net(x, dims=[256, 512, 128], num_classes=10):
2    # x is a tensor of size [N, H, W, C]
3    N = tf.shape(x)[0]
4    H, W, C = [dim.value for dim in x.get_shape()[1:]]
5    x = tf.reshape(x, [N, H * W * C])  # flatten
6    for layer_id, next_dim in enumerate(dims):
7      curr_dim = x.get_shape()[-1].value  # get_shape() returns a <list>
8      with tf.variable_scope("layer_{}".format(layer_id)):
9        w = tf.get_variable("w", [curr_dim, next_dim])  # w's name: "layer_2/w"
10     x = tf.matmul(x, w)
11     x = tf.nn.relu(x)
12   curr_dim = x.get_shape()[-1].value  # get_shape() returns a <list>
13   with tf.variable_scope("logits"):
14     w = tf.get_variable("w", [curr_dim, num_classes])  # w's name: "logits/w"
15     logits = tf.matmul(x, w)
16   return logits
```

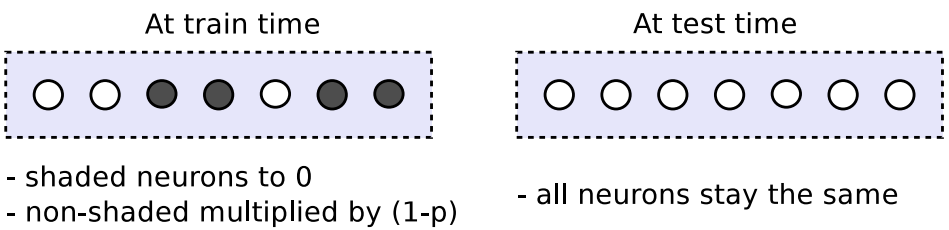
- Some behaviors of TF graphs cannot be foreseen until executing
  - E.g. the batch dimension `N = tf.shape(x)[0]`
  - E.g. dropout's behavior at training time and test time

Let's Code `tf.nn.dropout`

Carnegie Mellon

tf_feed_forward_net.py

```
1  def feed_forward_net(x, dims=[256, 512, 128], num_classes=10, dropout_keep=0.9):
2    # getting dimensions etc.
3    is_training = tf.placeholder(tf.bool, shape=None, name="is_training")
4    for layer_id, next_dim in enumerate(dims):
5      # create Variable w via tf.get_variable; perform tf.matmul
6      x = tf.cond(is_training,
7                  lambda: tf.nn.dropout(x, dropout_keep),
8                  lambda: x)
9      x = tf.nn.relu(x)
10   # compute logits...
11   ops = {"logits": logits, "is_training": is_training}
12   return ops
```



- Tell TF whether it's running the training graph or the test graph

Using feed_dict

Carnegie Mellon

tf_feed_forward_net.py

```
1  def feed_forward_net(x, dims=[256, 512, 128], num_classes=10, dropout_keep=0.9):
2    is_training = tf.placeholder(tf.bool, shape=None, name="is_training")
3    # tf.cond and other code
4    ops = {"logits": logits, "is_training": is_training}
5    return logits
6
7  def build_tf_graph():
8    # The mess we discussed
9    ops = {"loss": loss, "train_op": train_op, "preds": preds, "accus": accus,
10          "is_training": is_training }
11   return ops
12
13 def main(_args):
14   # boiler plate code
15   ops = build_tf_graph()
16   with tf.Session() as sess:
17     for train_step in range(10000):
18       output = sess.run([ops["train_op"]], feed_dict={ops["is_training"]: True})
19     preds = sess.run([ops["preds"]], feed_dict={ops["is_training"]: False})
```

- Specify whether you're training or testing via `feed_dict`

ConvNet and Batch Normalization

Carnegie Mellon

tf_conv_net.py

```
1  def batch_norm(x, is_training, scope, decay=0.9, eps=1e-3):
2    # do something
3
4  def conv_net(x, kernel_sizes=[3, 5, 7], num_channels=[128, 256, 512], num_classes=10):
5    # x: tensor of size [N, H, W, C]
6    N, H, W, C = tf.unstack(tf.shape(x))
7    is_training = tf.placeholder(tf.bool, shape=None, name="is_training")
8    for layer_id, (k_size, next_c) in enumerate(zip(kernal_sizes, num_channels)):
9      # create w
10     x = tf.nn.conv2d(x, w, padding="SAME")
11     x = batch_norm(x, is_training, "layer_{0}".format(layer_id))
12     x = tf.nn.relu(x)
13   # flatten, compute logits and return
```

- Pattern: $(\text{conv} \rightarrow \text{batchnorm} \rightarrow \text{ReLU}) \times N$
- Batchnorm at training time: $x = \alpha \cdot \frac{x - \hat{\mu}}{\sqrt{\varepsilon + \hat{\sigma}^2}} + \beta$
- Batchnorm at test time: $x = \alpha \cdot \frac{x - \bar{\mu}}{\sqrt{\varepsilon + \bar{\sigma}^2}} + \beta$

ConvNet and Batch Normalization

Carnegie Mellon

tf_batch_norm.py

```
1  def batch_norm(x, is_training, scope, decay=0.9, eps=1e-3):
2    shape = [x.get_shape()[3]]
3    with tf.variable_scope(name, reuse=None if is_training else True):
4      offset = tf.get_variable("offset", shape)
5      scale = tf.get_variable("scale", shape)
6      moving_mean = tf.get_variable("moving_mean", shape, trainable=False)
7      moving_var = tf.get_variable("moving_var", shape, trainable=False)
8
9      # train time
10     x_train, mean, var = tf.nn.fused_batch_norm(
11       x, scale, offset, epsilon=epsilon, data_format=data_format, is_training=True)
12     update_mean = moving_averages.assign_moving_average(moving_mean, mean, decay)
13     update_var = moving_averages.assign_moving_average(moving_var, var, decay)
14     with tf.control_dependencies([update_mean, update_var]):
15       x_train = tf.identity(x_train)
16
17     # test time: next slide
18   return x
```

- Meet our old friend: `tf.control_dependencies`

ConvNet and Batch Normalization

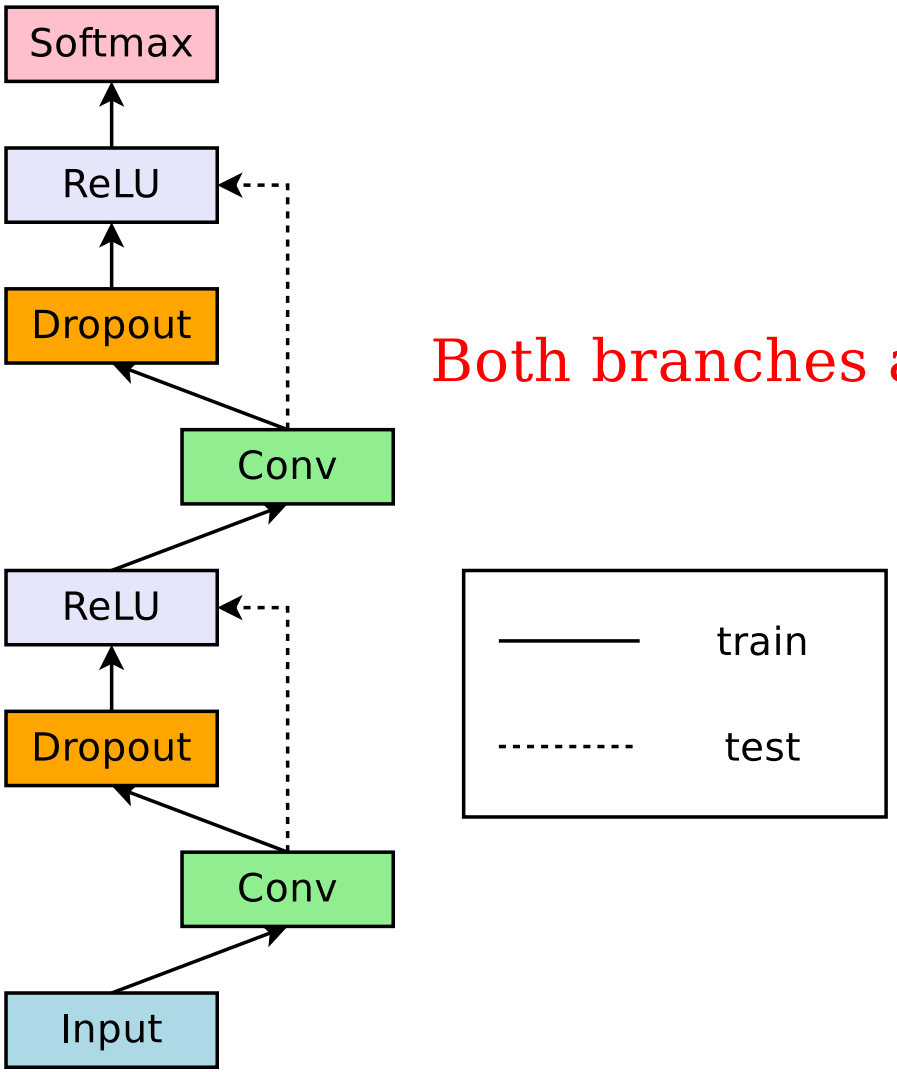
Carnegie Mellon

tf_batch_norm.py

```
1  def batch_norm(x, is_training, scope, decay=0.9, eps=1e-3):
2    shape = [x.get_shape()[3]]
3    with tf.variable_scope(name, reuse=None if is_training else True):
4      offset = tf.get_variable("offset", shape)
5      scale = tf.get_variable("scale", shape)
6      moving_mean = tf.get_variable("moving_mean", shape, trainable=False)
7      moving_var = tf.get_variable("moving_var", shape, trainable=False)
8
9      # train time: previous slide
10     x_test, _, _ = tf.nn.fused_batch_norm(x, scale, offset, mean=moving_mean,
11                                           var=moving_var,
12                                           epsilon=epsilon, data_format=data_format,
13                                           is_training=False)
14     x = tf.cond(is_training, lambda: x_train, lambda: x_test)
15   return x
```

- At test time: don't need $\hat{\mu}$ and $\hat{\sigma}$

Watch out for `tf.cond`



Both branches are excuted!!!