11-695: AI Engineering
Dynamic Computational Graphs

Spring 2019

Meet our old *inference* friend

tf_feed_forward_net.py

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

tf_feed_forward_net.py

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

The Need for Dynamic Graphs

Carnegie Mellon

tf_multiple_graphs.py

```
train_logits = _inference(train_imgs,
                          model_type=model_type,
                          is_train=True,
                          n_outputs=n_outputs)
val_logits = _inference(val_imgs,
                        model_type=model_type,
                        is_train=False,
                        n_outputs=n_outputs)
test_logits = _inference(test_imgs,
                         model_type=model_type,
                         is_train=False,
                         n_outputs=n_outputs)
```

- You call `_inference` multiple times

The Need for Dynamic Graphs

tf_multiple_graphs.py

```
train_logits = _inference(train_imgs,
                          model_type=model_type,
                          is_train=True,
                          n_outputs=n_outputs)
val_logits = _inference(val_imgs,
                        model_type=model_type,
                        is_train=False,
                        n_outputs=n_outputs)
test_logits = _inference(test_imgs,
                         model_type=model_type,
                         is_train=False,
                         n_outputs=n_outputs)
```

- You call `_inference` multiple times
- You almost use 3 times memory

The Need for Dynamic Graphs

Carnegie Mellon

tf_multiple_graphs.py

```
train_logits = _inference(train_imgs,
                          model_type=model_type,
                          is_train=True,
                          n_outputs=n_outputs)
val_logits = _inference(val_imgs,
                        model_type=model_type,
                        is_train=False,
                        n_outputs=n_outputs)
test_logits = _inference(test_imgs,
                         model_type=model_type,
                         is_train=False,
                         n_outputs=n_outputs)
```

- You call `_inference` multiple times
- You almost use 3 times memory
- Most of 3 graphs are shared weights, only one `op` is different.

The Need for Dynamic Graphs

CarnegieMellon

tf_multiple_graphs.py

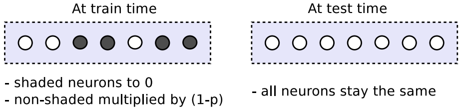
```
train_logits = _inference(train_imgs,
                          model_type=model_type,
                          is_train=True,
                          n_outputs=n_outputs)
val_logits = _inference(val_imgs,
                        model_type=model_type,
                        is_train=False,
                        n_outputs=n_outputs)
test_logits = _inference(test_imgs,
                         model_type=model_type,
                         is_train=False,
                         n_outputs=n_outputs)
```

- You call `_inference` multiple times
- You almost use 3 times memory
- Most of 3 graphs are shared weights, only one `op` is different.
- Why do not we just have 1 graph, and make it *dynamic* based on that `op`? → Yes, `TF` says Yes!

Let’s Code Dropout: use `tf.cond`

tf_feed_forward_net.py

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

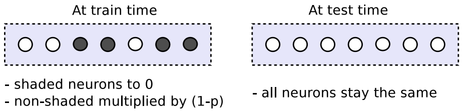


- Tell TF whether it’s running the training graph or the test graph

Why using `tf.cond`?

tf_feed_forward_net_short.py

```python
# omitted: create Variable w via tf.get_variable; perform tf.matmul
x = tf.cond(is_training,
            lambda: tf.nn.dropout(x, dropout_keep),
            lambda: x)
```



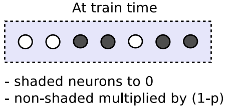
- Remember: we are talking in `TF` language
- We need *switching* control for our graph as well, such as `if`, `while`, `repeat`

Why using tf.cond?

Carnegie Mellon

tf_feed_forward_net_short.py

```
# omitted: create Variable w via tf.get_variable; perform tf.matmul
x = tf.cond(is_training,
            lambda: tf.nn.dropout(x, dropout_keep),
            lambda: x)
```



- Remember: we are talking in `TF` language
- We need *switching* control for our graph as well, such as `if`, `while`, `repeat`
- `tf.cond` in TF is similar to `if` in Python, with `lambda` for short-hand branching functions

Next, use feed_dict

Carnegie Mellon

tf_feed_forward_net.py

```python
def feed_forward_net(x, dims=[256, 512, 128], num_classes=10, dropout_keep=0.9):
  is_training = tf.placeholder(tf.bool, shape=None, name="is_training")
  # omitted: tf.cond and other code
  ops = {"logits": logits, "is_training": is_training}
  return logits

def build_tf_graph():
  # omitted ...
  ops = {"loss": loss, "train_op": train_op, "preds": preds, "accus": accus,
         "is_training": is_training }
  return ops

def main(_args):
  # boiler plate code
  ops = build_tf_graph()
  with tf.Session() as sess:
    for train_step in range(10000):
      output = sess.run([ops["train_op"]], feed_dict={ops["is_training"]: True})
    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 (BN)

tf_conv_net.py

```python
def batch_norm(x, is_training, scope, decay=0.9, eps=1e-3):
  # do something - next slide

def conv_net(x, kernel_sizes=[3, 5, 7], num_channels=[128, 256, 512], num_classes=10):
  # x: tensor of size [N, H, W, C]
  N, H, W, C = tf.unstack(tf.shape(x))
  is_training = tf.placeholder(tf.bool, shape=None, name="is_training")
  for layer_id, (k_size, next_c) in enumerate(zip(kernal_sizes, num_channels)):
    # create w
    x = tf.nn.conv2d(x, w, padding="SAME")
    x = batch_norm(x, is_training, "layer_{0}".format(layer_id))
    x = tf.nn.relu(x)
  # flatten, compute logits and return
```

- Pattern: (conv $\rightarrow$ batchnorm $\rightarrow$ ReLU) $\times N$
- Batch Norm at training time: $x = \alpha \cdot \frac{x - \hat{\mu}}{\sqrt{\hat{\sigma}^2 + \varepsilon}} + \beta$
- Batch Norm at test time: $x = \alpha \cdot \frac{x - E[x]}{\sqrt{Var(x) + \varepsilon}} + \beta$

BN with tf.nn.fused_batch_norm[1]

tf_batch_norm.py

```python
def batch_norm(x, is_training, scope, decay=0.9, eps=1e-3):
  shape = [x.get_shape()[3]]
  with tf.variable_scope(name, reuse=None if is_training else True):
    offset = tf.get_variable("offset", shape)
    scale = tf.get_variable("scale", shape)
    moving_mean = tf.get_variable("moving_mean", shape, trainable=False)
    moving_var = tf.get_variable("moving_var", shape, trainable=False)

    # train time
    x_train, mean, var = tf.nn.fused_batch_norm(
      x, scale, offset, epsilon=epsilon, data_format=data_format, is_training=True)
    update_mean = moving_averages.assign_moving_average(moving_mean, mean, decay)
    update_var = moving_averages.assign_moving_average(moving_var, var, decay)
    # why do we need this?
    with tf.control_dependencies([update_mean, update_var]):
      x_train = tf.identity(x_train)
    # test time: next slide
  return x
```

- $update\_mean = moving\_mean * decay + mean * (1 - decay)$
- Meet our old friend: `tf.control_dependencies`

[1] https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/kernels/fused_batch_norm_op.cc

ConvNet and Batch Normalization

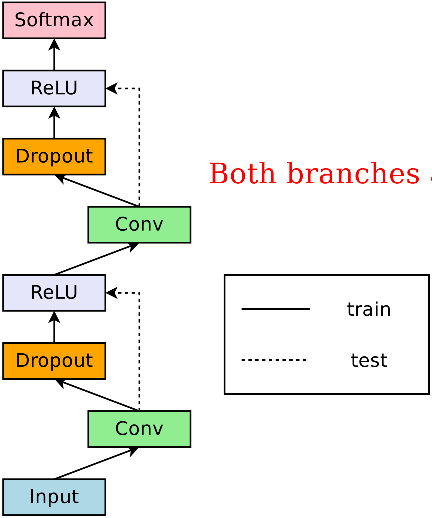
tf_batch_norm.py

```
def batch_norm(x, is_training, scope, decay=0.9, eps=1e-3):
  shape = [x.get_shape()[3]]
  with tf.variable_scope(name, reuse=None if is_training else True):
    offset = tf.get_variable("offset", shape)
    scale = tf.get_variable("scale", shape)
    moving_mean = tf.get_variable("moving_mean", shape, trainable=False)
    moving_var = tf.get_variable("moving_var", shape, trainable=False)

    # omitted: train time: previous slide
    # test time here:
    x_test, _, _ = tf.nn.fused_batch_norm(x, scale, offset, mean=moving_mean,
                                          var=moving_var,
                                          epsilon=epsilon, data_format=data_format,
                                          is_training=False)
    x = tf.cond(is_training, lambda: x_train, lambda: x_test)
  return x
```

- At test time: don't need $\hat{\mu}$ and $\hat{\sigma}$
- And also don't have
- Use global expected mean and variance estimated during training

Watch out for `tf.cond`



Both branches are excuted!!!