11-695: Competitive Engineering
Let's Code a Neural Net!

Spring 2018

TF Paradigm Recap

tf_graph_recap.py

```python
import tensorflow as tf

def build_tf_graph():
  # create ops, variables, etc.

def main(_args):
  g = tf.Graph()
  with g.as_default():
    build_tf_graph()
    with tf.Session() as sess:
      output = sess.run([train_op, ...])  # execute the operation z
```

- Computational Graphs:
  - nodes or *ops*
  - edges are *tensors*
- Execution: `sess.run([a, a, b, c])`
  - `a`, `b`, `c` *and their parents* will be run
  - Two `a`s will be run *once*

Let’s Code a Neural Network!

tf_graph_dict_trick.py

```python
import tensorflow as tf

def build_tf_graph():
  # create ops, variables, etc.
  ops = {
    "train": train_op,
    "grad_norm": gradient_norm,
    "preds": get_predictions,
  }
  return ops

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

- Graph names are *different* from Python names
  - Can be lost when you build large graphs
  - You should use a Python `dict` to store these Python *handles*

Step 1: Build a Model in a `TF` Graph

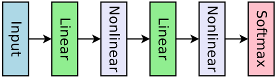
`tf_feed_forward_net.py`

```python
def feed_forward_net(x, dims=[256, 512, 128], num_classes=10):
  # x is a tensor of size [N, H, W, C]
  N, H, W, C = tf.unstack(tf.shape(x))

def build_tf_graph():
  images, labels = get_data_ops()
  logits = feed_forward_net(images)
  # LATER: loss function, train_op, etc.
```

- What are `images` and `labels`?
  - They are *TF ops*
- What happens in `N, H, W, C = tf.unstack(tf.shape(x))`?
  - `tf.shape(x)` is a *TF ops*.
  - `tf.unstack(...)` is a *TF ops*.
  - So are `N`, `H`, `W`, `C`
- Get used to them. Everything in TF is an ops.

Step 1: Build a Model in a TF Graph



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, H, W, C = tf.unstack(tf.shape(x))
  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
```

- Flatten → (Linear → Nonlinear) × 100 → logits

Step 2: Loss Function

`tf_feed_forward_net.py`

```
def feed_forward_net(x, dims=[256, 512, 128], num_classes=10):
  # The mess we just discussed

def build_tf_graph():
  images, labels = get_data_ops()  # images, labels: [N, H, W, C], [N]
  logits = feed_forward_net(images)
  # cross entropy loss function
  loss = tf.nn.sparse_softmax_cross_entropy_with_logits(
    logits=logits, labels=labels)
  loss = tf.reduce_mean(loss)
  # LATER: train_op, etc.
```

- `tf.nn.sparse_softmax_cross_entropy_with_logits(logits, labels)` computes *all* losses.
  - Divide them by the batch size
- `tf.nn.sparse_softmax_cross_entropy_with_logits(...)` is *the correct way* to implement cross entropy loss

Step 2: Loss Function

`tf_feed_forward_net.py`

```python
def build_tf_graph():
  images, labels = get_data_ops()  # images, labels: [N, H, W, C], [N]
  logits = feed_forward_net(images)
  # cross entropy loss function
  probs = tf.nn.softmax(logits)
  label_probs = tf.gather(probs, labels, axis=1)
  loss = tf.log(label_probs)
  # LATER: train_op, etc.
```

- Here is one (out of many) wrong ways!
  - Correct values, but much slower!
- Why?
  - $\log \frac{\exp\{\ell_i\}}{\sum_j \exp\{\ell_j\}}$ is *fused* into a kernel.

Step 3: Train op

Carnegie Mellon

`tf_feed_forward_net.py`

```python
def build_tf_graph():
  images, labels = get_data_ops()  # images, labels: [N, H, W, C], [N]
  logits = feed_forward_net(images)
  loss = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits, labels=labels)
  # train_op
  optimizer = tf.train.GradientDescentOptimizer(learning_rate=1.0)
  train_op = optimizer.minimize(loss)
  # LATER: predictions and accuracies
```

- `optimizer` is *not* the train op
- `optimizer.minimize(loss)` is the train op
- If you call `sess.run([train_op])`, it will perform
    - Forward pass `images` and `labels`
    - Back-propagation
    - Update all variables using gradient descents.

Step 4: Predictions and Accuracies

`tf_feed_forward_net.py`

```python
def build_tf_graph():
  images, labels = get_data_ops()  # images, labels: [N, H, W, C], [N]
  logits = feed_forward_net(images)
  loss = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits, labels=labels)
  optimizer = tf.train.GradientDescentOptimizer(learning_rate=1.0)
  train_op = optimizer.minimize(loss)
  # predictions and accuracies
  preds = tf.argmax(logits, axis=1)
  accus = tf.equal(preds, labels)  # TF boolean tensor
  accus = tf.cast(accus, dtype=tf.int32)
  accus = tf.reduce_sum(accus)
  ops = {
    "loss": loss,
    "train_op": train_op,
    "preds": preds,
    "accus": accus,
  }
  return ops
```

- Everything is a TF ops
    - Get used to this concept / idea / paradigm!
    - I will repeat this until you are used to it!

What have we done?

`tf_feed_forward_net.py`

```python
def build_tf_graph():
  # The mess we discussed
  ops = {
    "loss": loss, "train_op": train_op, "preds": preds, "accus": accus
  }
  return ops

def main(_args):
  g = tf.Graph()
  with g.as_default():
    ops = build_tf_graph()
    with tf.Session() as sess:
      sess.run(tf.global_variables_initializer()) # this is juts an ops!
      for train_step in range(10000):
        output = sess.run([ops["train_op"]])
```

- We built a TF graph with
  - A feed forward network
  - Essential ops *to train* and *to use* the network

Can we build a ConvNet?

tf_conv_net.py

```
def feed_forward_net(x, dims=[256, 512, 128], num_classes=10):
  # some mess you have seen

def conv_net(x, kernel_sizes=[3, 5, 7], num_channels=[128, 256, 512], num_classes=10):
  # some mess you will fill in

def build_tf_graph():
  images, labels = get_data_ops()
  logits_feed_forward = feed_forward_net(images)
  logits_conv = conv_net(images)
  # some more mess
  return ops

def main(_args):
  with tf.Graph().as_default:
    ops = build_tf_graph()
    # even more mess
```

- Everything is the same, but
  - Replace `feed_forward_net` with `conv_net`

Can we build a ConvNet?

tf_conv_net.py

```
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))
  for layer_id, (k_size, next_c) in enumerate(zip(kernal_sizes, num_channels)):
    curr_c = x.get_shape()[-1].value  # get_shape() returns a <list>
    with tf.variable_scope("layer_{}".format(layer_id)):
      # w’s name: "layer_2/w"
      w = tf.get_variable("w", [k_size, k_size, curr_c, next_c])
    x = tf.nn.conv2d(x, w, padding="SAME")
    x = tf.nn.relu(x)
  x = tf.reshape(x, [N, -1])
  curr_c = x.get_shape()[-1].value  # get_shape() returns a <list>
  with tf.variable_scope("logits"):
    w = tf.get_variable("w", [curr_c, num_classes])  # w’s name: "logits/w"
    logits = tf.matmul(x, w)
  return logits
```

- No more flatten!
- Just change your `tf.matmul` into `tf.nn.conv2d()`
  - `padding="SAME"`