11-695: Competitive Engineering
Dynamic Computational Graphs (cont)
Recurrent Neural Networks

Spring 2018

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(x, scale, offset,
11       epsilon=epsilon, data_format=data_format, is_training=True)
12     update_mean = tf.assign_add(moving_mean, (1.0 - decay) * (mean - moving_mean))
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
```

- Moving avarage update: $\bar{\mu} \leftarrow m\bar{\mu} + (1-m)\hat{\mu} = \bar{\mu} + (1-m)(\hat{\mu} - \bar{\mu})$
- Meet our old friend: `tf.control_dependencies`

ConvNet and Batch Normalization

Carnegie Mellon

tf.nn.fused_batch_norm

```
tf.nn.fused_batch_norm(
  x, scale, offset, mean=None, variance=None,
  epsilon=0.001, data_format='NHWC', is_training=True, name=None)

  """
    x: Input Tensor of 4 dimensions.
    scale: A Tensor of 1 dimension for scaling.
    offset: A Tensor of 1 dimension for bias.
    mean: A Tensor of 1 dimension for population mean used for inference.
    variance: A Tensor of 1 dimension for population variance used
      for inference.
    epsilon: A small float number added to the variance of x.
    data_format: The data format for x. Either "NHWC" (default) or "NCHW".
    is_training: A bool value to specify if the operation is used for
      training or inference.
    name: A name for this operation (optional).
  """
```

- TF Docs: https://www.tensorflow.org/api_docs/python/

ConvNet and Batch Normalization

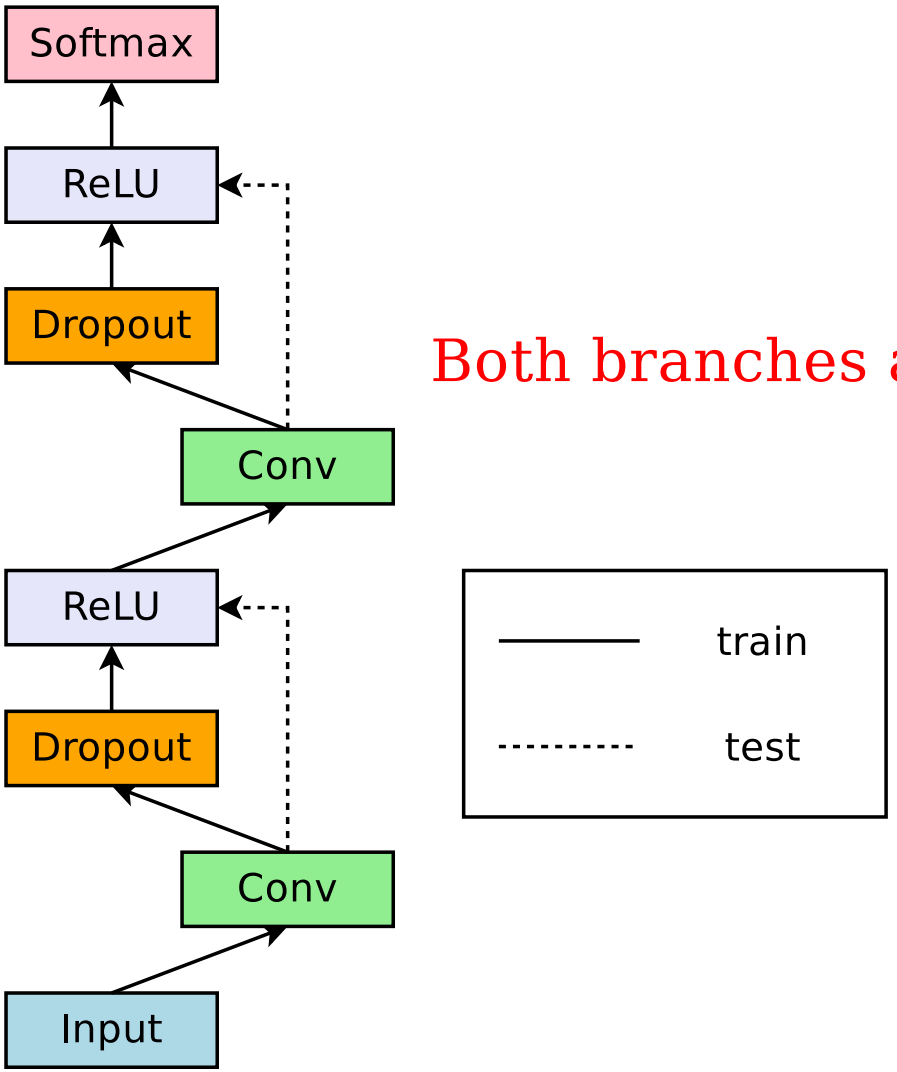
Carnegie Mellon

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!!!

A Generalization of `tf.cond`

Carnegie Mellon

tf_case.py

```
1  def build_tf_graph:
2    inp = get_input()
3    branch_id = tf.placeholder(tf.int32, shape=None, name="branch_id")
4    output = tf.case({
5        tf.equal(branch_id, 0): lambda: inp,
6        tf.equal(branch_id, 1): lambda: tf.nn.relu(inp),
7        tf.equal(branch_id, 1): lambda: tf.tanh(inp),
8      },
9      default=lambda: tf.zeros_like(inp))
```

- `tf.case` allows multiple branches. `tf.cond` allows 2 branches.
- `tf.case` also executes all branches

Best of TF's Dynamic: `tf.while_loop`

Carnegie Mellon

tf_while_loop.py

```
1  def build_tf_graph:
2    def condition(i, a, b):
3      return tf.less(i, 10)
4
5    def body(i, a, b):
6      return i+1, b, a+b
7
8    loop_vars = [
9      tf.constant(0, dtype=tf.int32),   # step
10     tf.constant(1, dtype=tf.int32),   # a
11     tf.constant(1, dtype=tf.int32),   # b
12   ]
13
14   loop_outputs = tf.while_loop(condition, body, loop_vars)
```

- What does this do?

Best of TF's Dynamic: `tf.while_loop`

Carnegie Mellon

tf_while_loop.py

```
1  def build_tf_graph:
2    def condition(i, a, b):
3      return tf.less(i, 10)
4
5    def body(i, a, b):
6      return i+1, b, a+b
7
8    loop_vars = [
9      tf.constant(0, dtype=tf.int32),   # step
10     tf.constant(1, dtype=tf.int32),   # a
11     tf.constant(1, dtype=tf.int32),   # b
12   ]
13
14   loop_outputs = tf.while_loop(condition, body, loop_vars)
```

- What does this do?
  - Computes the Fibonacci numbers.

Best of TF's Dynamic: `tf.while_loop`

Carnegie Mellon

tf_while_loop.py

```
1  def build_tf_graph:
2    def condition(i, a, b):
3      return tf.less(i, 10)
4
5    def body(i, a, b):
6      return i+1, b, a+b
7
8    loop_vars = [
9      tf.constant(0, dtype=tf.int32),   # step
10     tf.constant(1, dtype=tf.int32),   # a
11     tf.constant(1, dtype=tf.int32),   # b
12   ]
13
14   loop_outputs = tf.while_loop(condition, body, loop_vars)
```

- What does this do?
  - Computes the Fibonacci numbers.
- But we can do it in an easier way.

Can't we do something else?

Carnegie Mellon

tf_fibonacci_alternate.py

```
1  def build_tf_graph:
2    a, b = tf.constant(1, dtype=tf.int32), tf.constant(1, dtype=tf.int32)
3    for _ in range(10):
4      a, b = b, a + b
```

- How many TF ops are created?

Can't we do something else?

Carnegie Mellon

tf_fibonacci_alternate.py

```
1  def build_tf_graph:
2    a, b = tf.constant(1, dtype=tf.int32), tf.constant(1, dtype=tf.int32)
3    for _ in range(10):
4      a, b = b, a + b
```

- How many TF ops are created?
  - $O(n)$, where n is the index of the Fibonacci number you want

Can't we do something else?

Carnegie Mellon

tf_fibonacci_alternate.py

```
1  def build_tf_graph:
2    a, b = tf.constant(1, dtype=tf.int32), tf.constant(1, dtype=tf.int32)
3    for _ in range(10):
4      a, b = b, a + b
```

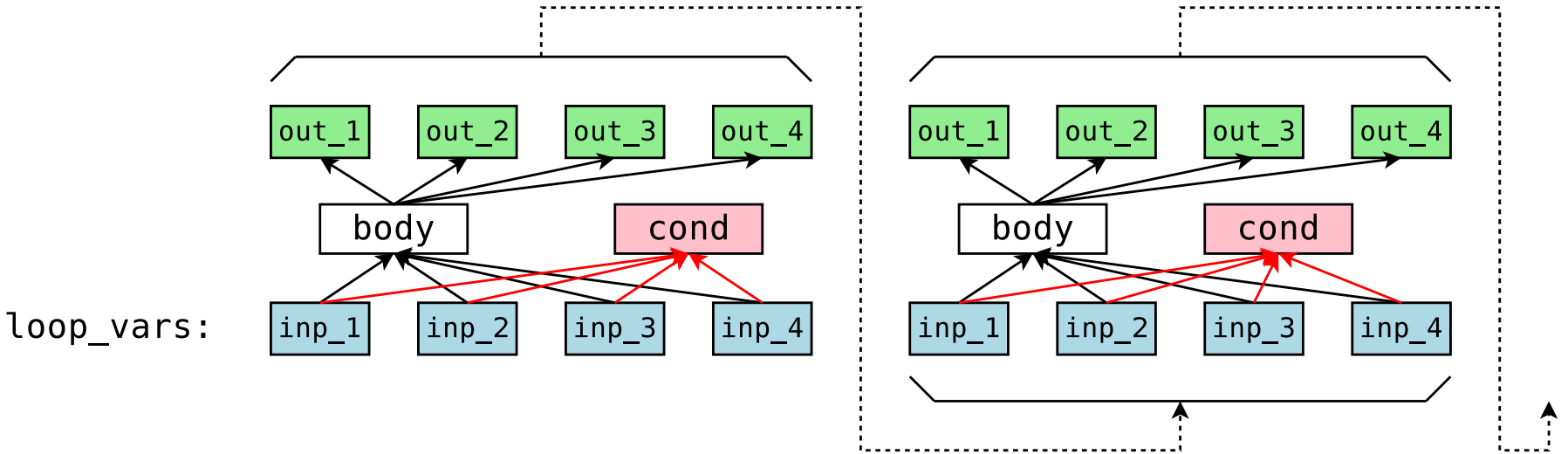
- How many TF ops are created?
  - $O(n)$, where n is the index of the Fibonacci number you want
- `tf.while_loop` creates $O(1)$ ops :-)

How does `tf.while_loop` work?

Carnegie Mellon

tf_while_loop.py

```
1  def build_tf_graph:
2    def condition(i, a, b): return tf.less(i, 10)
3    def body(i, a, b): return i+1, b, a+b
4    loop_vars = [tf.constant(0, dtype=tf.int32), tf.constant(1, dtype=tf.int32),
5                 tf.constant(1, dtype=tf.int32)]
6    loop_outputs = tf.while_loop(condition, body, loop_vars)
```



- `tf.while_loop` creates as many ops as `loop_vars` and `loop_vars`

An (abusive) use case

Carnegie Mellon

tf_while_loop.py

```
1  def build_tf_graph:
2    w = tf.get_variable("weight", [1000, 3, 3, 128, 128])
3    def condition(i, *args): return tf.less(i, 10)
4
5    def body(i, x, w):
6      weight = tf.reshape(w[i, :, :, :, :], [3, 3, 128, 128])
7      inp = x
8      x = tf.nn.conv2d(x, weight, padding="SAME")
9      x = tf.relu(x)
10     return i + 1, x + inp
11
12   # x: tensor of size [N, H, W, C]
13   loop_vars = [tf.constant(0, dtype=tf.int32), x, w]
14   loop_outputs = tf.while_loop(condition, body, loop_vars)
```

- What does this do?

An (abusive) use case

Carnegie Mellon

tf_while_loop.py

```
1  def build_tf_graph:
2    w = tf.get_variable("weight", [1000, 3, 3, 128, 128])
3    def condition(i, *args): return tf.less(i, 10)
4
5    def body(i, x, w):
6      weight = tf.reshape(w[i, :, :, :, :], [3, 3, 128, 128])
7      inp = x
8      x = tf.nn.conv2d(x, weight, padding="SAME")
9      x = tf.relu(x)
10     return i + 1, x + inp
11
12   # x: tensor of size [N, H, W, C]
13   loop_vars = [tf.constant(0, dtype=tf.int32), x, w]
14   loop_outputs = tf.while_loop(condition, body, loop_vars)
```

- What does this do?
  - (Vanilla) Residual network

An (abusive) use case

Carnegie Mellon

tf_while_loop.py

```
1  def build_tf_graph:
2    w = tf.get_variable("weight", [1000, 3, 3, 128, 128])
3    def condition(i, *args): return tf.less(i, 10)
4
5    def body(i, x, w):
6      weight = tf.reshape(w[i, :, :, :, :], [3, 3, 128, 128])
7      inp = x
8      x = tf.nn.conv2d(x, weight, padding="SAME")
9      x = tf.relu(x)
10     return i + 1, x + inp
11
12   # x: tensor of size [N, H, W, C]
13   loop_vars = [tf.constant(0, dtype=tf.int32), x, w]
14   loop_outputs = tf.while_loop(condition, body, loop_vars)
```

- What does this do?
  - (Vanilla) Residual network
- With much fewer ops, i.e. a much smaller graph.