Tensorflow Review Session

Karl Cobbe and Joshua Achiam

OpenAI

February 1, 2019

Automatic Differentiation

What is Tensorflow?

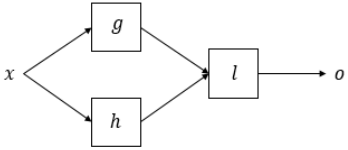
Tensorflow is a library for **building and manipulating computation graphs on tensors**.

What is a computation graph? A directed, acyclic graph where

- Bottom nodes (no arrows going in) are **inputs** (external inputs (data) or internal inputs (variables))
- Top nodes (no arrows going out) are **outputs**
- Middle nodes are **functions**.

In Tensorflow, all inputs, outputs, and function outputs are **tensors** (multi-dimensional arrays).

Computation Graphs



- Simple example: input x, output o, various intermediate functions.
- Chain rule gives us a way to compute gradient of o with respect to x by going *backward* through intermediate nodes:
  - Note $do/dl = 1$.
  - First, find $\partial l/\partial g$ and $\partial l/\partial h$.
  - Get gradients dg/dx and dh/dx.
  - Aggregate at x:

$$\frac{do}{dx} = \frac{\partial l}{\partial g}\frac{dg}{dx} + \frac{\partial l}{\partial h}\frac{dh}{dx}$$

Automatic Differentiation

- Given a computation graph $f : x \to y$, we can automatically generate a new computation graph that returns dy/dx
- (Very) roughly, each op in the graph is replaced by a gradients op that runs in the reverse-direction. Instead of the original op

$$\mathcal{O} : x_1, ..., x_n \to y,$$

we place

$$\mathcal{O}' : \frac{\partial *}{\partial y} \to \frac{\partial *}{\partial x_1}, ..., \frac{\partial *}{\partial x_n}$$

- This procedure allows us to get gradients of any scalar signal (e.g. loss function) with respect to any inputs to a graph (e.g. trainable parameters)!

Automatic Differentiation

- You will almost never have to worry about what happens under the hood here
- Libraries like Tensorflow, Theano, PyTorch, Caffe, and others take care of this for you
- Takeaway: computation graph libraries let you define really complicated graphs (especially neural net architectures), get gradients for no extra programming effort, and easily optimize via first-order methods!

Tensorflow Basics

Basic Graph Building

- Create points of data entry through `tf.placeholder()`
- Create parameter variables through `tf.Variable()` or `tf.get_variable()`[1]
- Apply **operations** to data
  - Can be simple/atomic
    - In some places numpy syntax is supported (including broadcasting)
    - e.g. `a + b` and `tf.add(a,b)` produce same output
  - Or composite: see `tf.layers` package for neural network layers
    - e.g. `tf.layers.dense()`, which creates standard feedforward 'densely connected' neural net layer
    - Great for fast prototyping—most of the work already done for you! Snap pieces together like Lego

[1] See https://stackoverflow.com/questions/37098546/difference-between-variable-and-get-variable-in-tensorflow

Example Graph Building

From MNIST tutorial:

```
# Create the model
x = tf.placeholder(
      dtype=tf.float32,
      shape=[None, 784]
      )
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
y = tf.matmul(x, W) + b
```

Operations Do Not Run At Define Time

Caution! Operations produce *tensors* as outputs, not data.

```
In [3]: np.add(5,5)
Out[3]: 10

In [4]: tf.add(5,5)
Out[4]: <tf.Tensor 'Add:0' shape=() dtype=int32>
```

Session, Run, and Initialization

- To compute outputs of CGs in TF, you need a Session. Several ways to get one:
  - `tf.Session()`
  - `tf.InteractiveSession()` (automatically sets as default)
  - `tf.get_default_session()` (only if a default session exists)
- Use the `run()` command from a session to perform computations
- `run()` requires a **feed dict** for placeholders

```
sess = tf.InteractiveSession()
x_ph = tf.placeholder(shape=(None,100) ,dtype=tf.float32)
y_ph = tf.placeholder(shape=(None,10), dtype=tf.float32)
loss, update_op = build_network(x_ph, y_ph)
x_batch, y_batch = data_gen.next()
out = sess.run(
        [loss, update_op],
        feed_dict={x_ph: x_batch, y_ph: y_batch}
        )
```

Further Notes on Run

- `run()` will only compute necessary pieces of computation graph to get outputs you ask for (nice—avoids excess compute!)
- As on previous slide, `run()` can get multiple outputs at once (with a single pass through necessary nodes in computation graph)
- If you have variables in your computation graph, nothing will work until you initialize them
  - To do this easily, after making session and graph, but before training:

```
sess.run(tf.global_variables_initializer())
```

Syntactic Sugar for Run

- If just running one op / evaluating one Tensor, **and a default Session exists**, you can use `.run()` and `.eval()`[2]
- `.run()` works for **operations**

```
sess = tf.Session()
with sess.as_default():
  tf.global_variables_initializer().run()
```

- `.eval()` works for **tensors**

```
sess = tf.InteractiveSession()
x, y = make_inputs()
accuracy = build_network(x, y)
x_batch, y_batch = data_gen.next()
print(accuracy.eval(feed_dict={x : x_batch, y : y_batch}))
```

(In above snippet, recall that `InteractiveSession` becomes default automatically!)

[2] See https://stackoverflow.com/questions/38987466/eval-and-run-in-tensorflow

Loss Functions

- Tensorflow makes common loss functions easy! Example, cross-entropy loss for classification:

```
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
y = build_logits_network(x)

# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])

cross_entropy = tf.reduce_mean(
  tf.nn.softmax_cross_entropy_with_logits(
    labels=y_,
    logits=y
    )
  )
```

- See `tf.losses` for more (`huber_loss`, `hinge_loss`, etc.)

Loss Functions

- Write your own custom losses. For instance, in policy gradients:

```
lr = tf.exp( logli - logli_old )
adv = tf.placeholder(shape=(None,), dtype=tf.float32)
surrogate_loss = -tf.reduce_mean( lr * adv )
```

Optimizers

- After building networks and loss functions, **add an optimizer** to minimize loss.
- Make an optimzer object, and set hyperparameters via constructor method (like momentum, RMSprop coefficients, Adam coefficients) or leave at safe defaults
- Call minimize on loss to get training op:

```
optimizer = tf.train.AdamOptimizer(learning_rate=1e-3)
train_op = optimizer.minimize(loss)
```

- To perform one step of training, just run training op!

```
sess.run(train_op, feed_dict)
```

- NB: If you want to, you can specify which variables the optimizer acts on as an argument to `minimize`

MNIST Example

```python
def main(_):
  # Import data
  mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)

  # Create the model
  x = tf.placeholder(tf.float32, [None, 784])
  W = tf.Variable(tf.zeros([784, 10]))
  b = tf.Variable(tf.zeros([10]))
  y = tf.matmul(x, W) + b

  # Define loss and optimizer
  y_ = tf.placeholder(tf.float32, [None, 10])

  cross_entropy = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y)
    )
  train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)

  sess = tf.InteractiveSession()
  tf.global_variables_initializer().run()
  # Train
  for _ in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})

  # Test trained model
  correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
  accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
  print(sess.run(
              accuracy,
              feed_dict={x: mnist.test.images, y_: mnist.test.labels}
              ))
```

Building Advanced Computation Graphs

Common Neural Network Operations

- Many standard neural network layers are already implemented, and enable high customization (for instance, custom initializers)
- Dense / Fully-Connected layers ($Wx + b$):
  - `tf.layers.dense`
  - `tf.contrib.layers.fully_connected`
- Conv layers
  - `tf.layers.conv2d`
  - `tf.contrib.layers.conv2d`
- Activation functions: `tf.nn.{relu, sigmoid, tanh, elu}`
- Neural Network tricks: `tf.layers.dropout`
- Tensor utilities: `tf.reshape`, `tf.contrib.layers.flatten`, `tf.concat`, `tf.reduce_sum`, `tf.reduce_mean`

Recurrent Neural Networks

- To build a recurrent neural network (RNN), first specify an **RNN Cell**, which defines the computation at each time step. Note, this is *not* the output Tensor you will run!
  - `tf.nn.rnn_cell.BasicRNNCell` ($h_t = \sigma(Wx_t + Rh_{t-1} + b)$)
  - `tf.nn.rnn_cell.GRUCell`
  - `tf.nn.rnn_cell.LSTMCell`
- Get a sequence of outputs and the final hidden state of an RNN computed with that Cell by calling `tf.nn.dynamic_rnn`

```
In [1]: import tensorflow as tf
In [2]: x = tf.placeholder(shape=(None,None,10),dtype=tf.float32)
In [3]: cell = tf.nn.rnn_cell.GRUCell(20)
In [4]: outputs, final = tf.nn.dynamic_rnn(cell, x, time_major=False, dtype=tf.float32)
In [5]: sess = tf.InteractiveSession()
In [6]: tf.global_variables_initializer().run()
In [7]: import numpy as np
In [8]: o, f = sess.run([outputs, final], {x: np.random.rand(32,50,10)})
In [9]: o.shape
Out[9]: (32, 50, 20)
In [10]: f.shape
Out[10]: (32, 20)
```

Variable Scoping and Reuse

- Organize variables by name with variable scopes

```
out = input

with tf.variable_scope('conv1'):
    out = tf.layers.batch_normalization(out,...)
    out = tf.conv2d(out,...)

with tf.variable_scope('conv2'):
    out = tf.layers.batch_normalization(out,...)
    out = tf.conv2d(out,...)
```

- Scopes can nest
- Scope names become part of directory structure for variable names; makes possible to access them through more arcane methods

- Can reuse variables (ie make two nodes with tied weights) via reuse

```
In [1]: import tensorflow as tf

In [2]: x = tf.placeholder(shape=(None,10),dtype=tf.float32)
   ...: y = tf.placeholder(shape=(None,10),dtype=tf.float32)
   ...: with tf.variable_scope('test'):
   ...:     o1 = tf.layers.dense(x, 10)
   ...: with tf.variable_scope('test',reuse=True):
   ...:     o2 = tf.layers.dense(x, 10)
   ...:

In [3]: o2
Out[3]: <tf.Tensor 'test_1/dense/BiasAdd:0' shape=(?, 10) dtype=float32>

In [4]: o1
Out[4]: <tf.Tensor 'test/dense/BiasAdd:0' shape=(?, 10) dtype=float32>

In [5]: tf.global_variables()
Out[5]:
[<tf.Variable 'test/dense/kernel:0' shape=(10, 10) dtype=float32_ref>,
 <tf.Variable 'test/dense/bias:0' shape=(10,) dtype=float32_ref>]
```

o1 and o2 are different dense operations, there is only one set of variables!

Gradients and Stop Gradients

- If you want to write a custom optimizer you may want to work with gradients directly; can do this using `tf.gradients`
- You may want to prevent backpropagation through a particular tensor: can do this with `tf.stop_gradient`
  - Example: in DQN, where we want to minimize a mean-square Bellman error with respect to params of current network but **not** target network

```
o = tf.placeholder(shape=(None,dim_o),dtype=tf.float32)
a = tf.placeholder(shape=(None,),dtype=tf.int32)
o2 = tf.placeholder(shape=(None,dim_o),dtype=tf.float32)
r = tf.placeholder(shape=(None,),dtype=tf.float32)
with tf.variable_scope('main'):
    q = build_network(o)                              # b x a
with tf.variable_scope('target'):
    q_targ = build_network(o2)                        # b x a
q_a = tf.reduce_sum(q * tf.one_hot(a, num_actions),1)   # b
q_targ_a = tf.reduce_max(q_targ,1)                      # b
target = r + gamma * q_targ_a
target = tf.stop_gradient(target)
loss = tf.reduce_mean(tf.square(q_a - target))
# later on, make assign statement so q_targ lags q
```

Logging and Debugging

Logging

- Tensorflow has native operations for saving data through `tf.summary`
- Declare summary ops as functions of other tensors or ops[3]

```
def variable_summaries(var):
  """Attach a lot of summaries to a Tensor (for TensorBoard visualization)."""
  with tf.name_scope('summaries'):
    mean = tf.reduce_mean(var)
    tf.summary.scalar('mean', mean)
    with tf.name_scope('stddev'):
      stddev = tf.sqrt(tf.reduce_mean(tf.square(var - mean)))
    tf.summary.scalar('stddev', stddev)
    tf.summary.scalar('max', tf.reduce_max(var))
    tf.summary.scalar('min', tf.reduce_min(var))
    tf.summary.histogram('histogram', var)
```

- Summary ops are never called unless you run them
- For convenience, merge all summary ops via

```
merged_summary_op = tf.summary.merge_all()
```

[3] Example from `https://www.tensorflow.org/get_started/summaries_and_tensorboard`

Logging

- Make a tf.summary.FileWriter to save summaries to file.[4]

```
...create a graph...
# Launch the graph in a session.
sess = tf.Session()
# Create a summary writer, add the 'graph' to the event file.
writer = tf.summary.FileWriter(<some-directory>, sess.graph)
```

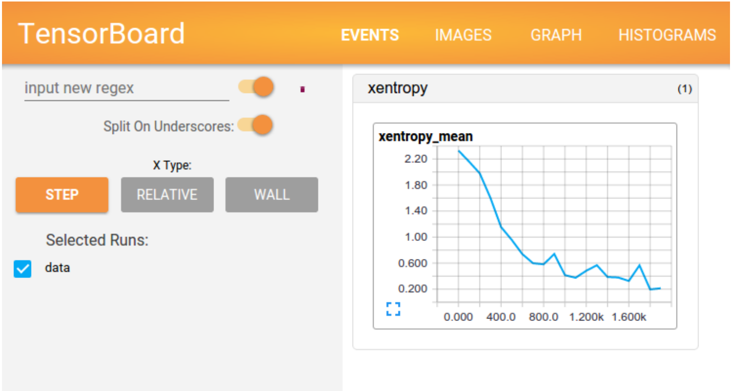
  Passing the graph into FileWriter allows you to inspect the computation graph later when you visualize in TensorBoard.

- To save summaries with the FileWriter, run the merged summary op and then use add_summary:

```
for i in range(n_steps):
  feed_dict = get_next_feed_dict()
  summary, _ = sess.run([merged_summary_op, train_op], feed_dict)
  writer.add_summary(summary, i)
```

[4] https://www.tensorflow.org/api_docs/python/tf/summary/FileWriter

Logging



Invoke Tensorboard with

`tensorboard --logdir=path/to/log-directory`

Debugging

- Explore with `InteractiveSession` in IPython
- Common issue—Tensor shapes are wrong. Can check with `<tensor>.get_shape().as_list()`.
- Want to look at the list of all variables? `tf.global_variables()`.

```
In [1]: import tensorflow as tf

In [2]: x = tf.placeholder(shape=(None,10), dtype=tf.float32)

In [3]: with tf.variable_scope('test'):
   ...:     y = tf.layers.dense(x, 20)
   ...:

In [4]: tf.global_variables()
Out[4]:
[<tf.Variable 'test/dense/kernel:0' shape=(10, 20) dtype=float32_ref>,
 <tf.Variable 'test/dense/bias:0' shape=(20,) dtype=float32_ref>]
```

- Good scoping makes it easier to find problem areas

Debugging

- Sometimes, look at raw inputs and outputs of networks!
  - If outputs all look the same despite different inputs, maybe a hidden layer's activations are saturating
- Be *super* careful about BatchNorm and other neural network tricks that have different behavior at training time and test time
  - See reference[5] for a good guide on using BatchNorm correctly.
- Make sure the scale of inputs to your network is reasonable (empirically it helps sometimes for data to have mean zero and std=1)
- If you are using default values anywhere (in layers, optimizers, etc.), **check them** and make sure they make sense

[5] http://ruishu.io/2016/12/27/batchnorm/

What Else is Out There?

Computation Graph Libraries

- Tensorflow (Google) - Huge community, well-supported, somewhat clunky API but great distributed performance
- Theano (U of Montreal) - Long history, widely-used, slow to compile
- Caffe (Berkeley / Facebook)
- Torch (Facebook and others) - Now available in Python (previously just Lua), define-by-run makes for fast and flexible prototyping, comparable speed to Tensorflow
- Chainer (Preferred Networks) - Another define-by-run like Torch

For some performance comparisons:
`https://github.com/soumith/convnet-benchmarks`

Alternate APIs/Wrappers for Tensorflow

- tf-contrib (active development code in Tensorflow available to user, usually stabilizes into main code eventually)
- Keras (officially supported by Google)
- TF-Slim (also supported by Go—wait a second... how many APIs did Google make for this thing?)
- Sonnet (supported by Deepmind! which is owned by Google)
- TFLearn
- skflow

Lots of fragmentation, but things seem to have stabilized around core Tensorflow / Keras. Sonnet may be worth watching, though, because of DeepMind.

That's All Folks

Questions?