Freezing The Graph

Here's an example of the freeze graph tool:

```
~/tensorflow/bazel-bin/tensorflow/python/tools/freeze_graph \
--input_graph=base_graph.pb \
--input_checkpoint=ckpt \
--input_binary=true \
--output_graph=frozen_graph.pb \
--output_node_names=Softmax
```

The *freeze_graph* command requires five inputs:

- The input graph *input_graph* saved in protobuf format.
- The input graph checkpoint, *input_checkpoint*.
- *input_binary* denotes whether the input graph is a binary file. Set this to true if the input is a *.pb* file instead of *.pbtxt*.
- The name of the output graph, *output_graph*, i.e. the frozen graph.
- The names of the output nodes. It's a good idea in general to name key nodes in the graph and these names will come in handy when using these tools as well.

The result is saved in the *frozen_graph.pb* file. This ends up removing many extraneous operations from the graph. We can get all a graph's operations with *.get_operations()* method. The *load_graph* method below takes a binary protobuf file as input and returns the graph and list of operations.

```
def load_graph(graph_file, use_xla=False):
    jit_level = 0
    config = tf.ConfigProto()
    if use_xla:
        jit_level = tf.OptimizerOptions.ON_1
        config.graph_options.optimizer_options.global_jit_level = jit_level

    with tf.Session(graph=tf.Graph(), config=config) as sess:
        gd = tf.GraphDef()
        with tf.gfile.Open(graph_file, 'rb') as f:
            data = f.read()
            gd.ParseFromString(data)
        tf.import_graph_def(gd, name='')
        ops = sess.graph.get_operations()
        n_ops = len(ops)
        return sess.graph, ops
```

```
from graph_utils import load_graph

sess, base_ops = load_graph('base_graph.pb')
print(len(base_ops)) # 2165
sess, frozen_ops = load_graph('frozen_graph.pb')
print(len(frozen_ops)) # 245
```

Optimizing for Inference

Once the graph is frozen there are a variety of transformations that can be performed; dependent on what we wish to achieve. TensorFlow has packaged up some inference optimizations in a tool aptly called *optimize_for_inference*.

optimize_for_inference does the following:

- Removes training-specific and debug-specific nodes
- Fuses common operations
- Removes entire sections of the graph that are never reached

Here's how it can be used:

```
~/tensorflow/bazel-bin/tensorflow/python/tools/optimize_for_inference \
--input=frozen_graph.pb \
--output=optimized_graph.pb \
--frozen_graph=True \
--input_names=image_input \
--output_names=Softmax
```

We'll use the graph we just froze as the input graph, *input*. *output* is the name of the output graph; we'll be creative and call it *optimized_graph.pb*.

The *optimize_for_inference* tool works for both frozen and unfrozen graphs, so we have to specify whether the graph is already frozen or not with *frozen_graph*.

input_names and *output_names* are the names of the input and output nodes respectively. As the option names suggest, there can be more than one input or output node, separated by commas.

Let's take a look at the effect this has on the number of operations.

```
from graph_utils import load_graph

sess, optimized_ops = load_graph('optimized_graph.pb')
print(len(optimized_ops)) # 200
```

8-bit Calculations

We've covered freezing the graph and optimizing for inference, but we haven't yet covered quantization. So the next optimization we'll discuss is converting the graph to perform 8-bit calculations. Here's an example using the *transform_graph* tool:

```
~/tensorflow/bazel-bin/tensorflow/tools/graph_transforms/transform_graph \
--in_graph=frozen_graph.pb \
--out_graph=eightbit_graph.pb \
--inputs=image_input \
--outputs=Softmax \
--transforms='
add_default_attributes
remove_nodes(op=Identity, op=CheckNumerics)
fold_constants(ignore_errors=true)
fold_batch_norms
fold_old_batch_norms
fuse_resize_and_conv
quantize_weights
quantize_nodes
strip_unused_nodes
sort_by_execution_order'
```

There's a lot going on here, which you can find more information in the TensorFlow Graph Transforms documentation.

The gist is that *fold* transforms look for subgraphs that always evaluate to to the same result. Then they consolidate each such subgraph into one *Constant* node.

quantize_weights quantizes values larger than 15 bits. It also adds nodes to convert back to floating point. The *quantize_weights* transform is mainly for reducing graph size. For the desired quantization computation behaviour we'll need to use *quantize_nodes* as well.

Ok, let's take a look:

```
from graph_utils import load_graph

sess, eightbit_ops = load_graph('eightbit_graph.pb')
print(len(eightbit_ops)) # 425
```

There are 425 operations, that's more than the original frozen graph! Quantization computation requires extra nodes in general so it's not a big deal. Nodes that have no quantization equivalent are keep as floating point.

AOI & JIT

TensorFlow supports both JIT (just in time) and AOT (ahead of time) compilation.

AOT compilation is the kind used in a C or C++ program; it compiles the program "ahead" of the actual use. A really cool aspect of AOT compilation is you can potentially create a static binary file, meaning it's entirely self contained. You can deploy it by simply downloading the file and executing it, without concern for downloading extra software, besides necessary hardware drivers, i.e. for GPU use.

JIT compilation doesn't compile code until it's actually run. You can imagine as a piece of code is being interpreted machine instructions are concurrently generated. Nothing will change during the initial interpretation, the JIT might as well not exist. However, on the second and all future uses that piece of code will no longer be interpreted. Instead the compiled machine instructions will be used.

Under the hood AOT and JIT compilation make use of XLA (Accelerated Linear Algebra). XLA is a compiler and runtime for linear algebra. XLA takes a TensorFlow graph and uses LLVM to generate machine code instructions. LLVM is itself a compiler which generates machine code from its IR (intermediate representation). So, in a nutshell:

TensorFlow $\rightarrow$ XLA $\rightarrow$ LLVM IR $\rightarrow$ Machine Code

which means TensorFlow can potentially be used on any architecture LLVM generates code for.

Both AOT and JIT compilation are experimental in TensorFlow. However, JIT compilation is fairly straightforward to apply. Note JIT compilation is NOT limited to inference but can be used during training as well.

Reusing the Graph

So once we've optimized the graph and saved it to a file how do we use it?

First load the graph from the binary protobuf file:

```
from graph_utils import load_graph

sess, _ = load_graph('your_graph.pb')
graph = sess.graph
```

Once the graph is loaded we can access operations and tensors with the *get_operation_by_name* and *get_tensor_by_name tf.Graph* methods respectively. We could also retrieve all the operations of a *tf.Graph* with the *get_operations* method, which is very useful if we are unsure which operation to use. In this case we want to pass an image as input and receive the softmax probabilities as output:

```
image_input = graph.get_tensor_by_name('image_input:0')
keep_prob = graph.get_tensor_by_name('keep_prob:0')
softmax = graph.get_tensor_by_name('Softmax:0')
```

Then we can take an image and run the computation in a session as we normally would. *keep_prob* is related to the dropout probability.

```
probs = sess.run(softmax, {image_input: img, keep_prob: 1.0})
```