Lecture note 5: How to manage your experiments in TensorFlow

"CS 20SI: TensorFlow for Deep Learning Research" (cs20si.stanford.edu)
Prepared by Chip Huyen (huyenn@stanford.edu)
Reviewed by Danijar Hafner

We've built our word2vec model and it seems to be working pretty well for the small dataset that it uses. We know that it'd take much longer time for a larger dataset, and we also know that training more complicated models can take an ungodly amount of time. For example, future assignments in CS 224N will often take up to 4 hours. Models to do abstractive summarization can take days to get even results that only make sense in the slightest sense, even on pretty powerful GPUs. Many computer vision tasks can take much longer time to train.

We can't afford to let our models to run for days, wait to see how they go, then make adjustment. Or if our computer crashes, the training is interrupted and we'll have to run our model all over again. It's crucial to be able to stop training at any point, for any reason, and resume training as if nothing happens. It will be especially helpful for analyzing our models, as this allows us close inspection of our models after any number of training steps.

Another problem that researchers often face is how to produce our work in such a work that other researchers can replicate/verify our results. In training neural networks, we often use randomization. For example, we randomize the weights for our models, or we shuffle the order of our training samples. It's important to learn how to control this random factor in our models.

In this lecture, we will go over the excellent set of tools that TensorFlow provides to help us manage our experiments. The topics we cover today include: tf.train.Saver() class, TensorFlow's random seed and NumPy's random state, and visualization our training progress (aka more TensorBoard).

tf.train.Saver()

A good practice is to periodically save the model's parameters after a certain number of steps so that we can restore/retrain our model from that step if need be. The tf.train.Saver() class allows us to do so by saving the graph's variables in binary files.

```
tf.train.Saver.save(sess, save_path, global_step=None, latest_filename=None,
meta_graph_suffix='meta', write_meta_graph=True, write_state=True)
```

For example, if we want to save the variables of the graph after every 1000 training steps, we do the following:

```
# define model

# create a saver object
saver = tf.train.Saver()

# launch a session to compute the graph
with tf.Session() as sess:
    # actual training loop
        for step in range(training_steps):
                sess.run([optimizer])

                if (step + 1) % 1000==0:
                        saver.save(sess, 'checkpoint_directory/model_name',
                                     global_step=model.global_step)
```

In TensorFlow lingo, the step at which you save your graph's variables is called a checkpoint. Since we will be creating many checkpoints, it's helpful to append the number of training steps our model has gone through in a variable called global_step. It's a very common variable to see in TensorFlow program. We first need to create it, initialize it to 0 and set it to be not trainable, since we don't want to TensorFlow to optimize it.

```
self.global_step = tf.Variable(0, dtype=tf.int32, trainable=False, name='global_step')
```

We need to pass global_step as a parameter to the optimizer so it knows to increment global_step by one with each training step:

```
self.optimizer = tf.train.GradientDescentOptimizer(self.lr).minimize(self.loss,
                                                                      global_step=self.global_step)
```

To save the session's variables in the folder 'checkpoints' with name model-name-global-step, we use this:

```
saver.save(sess, 'checkpoints/skip-gram', global_step=model.global_step)
```

So our training loop for word2vec now looks like this:

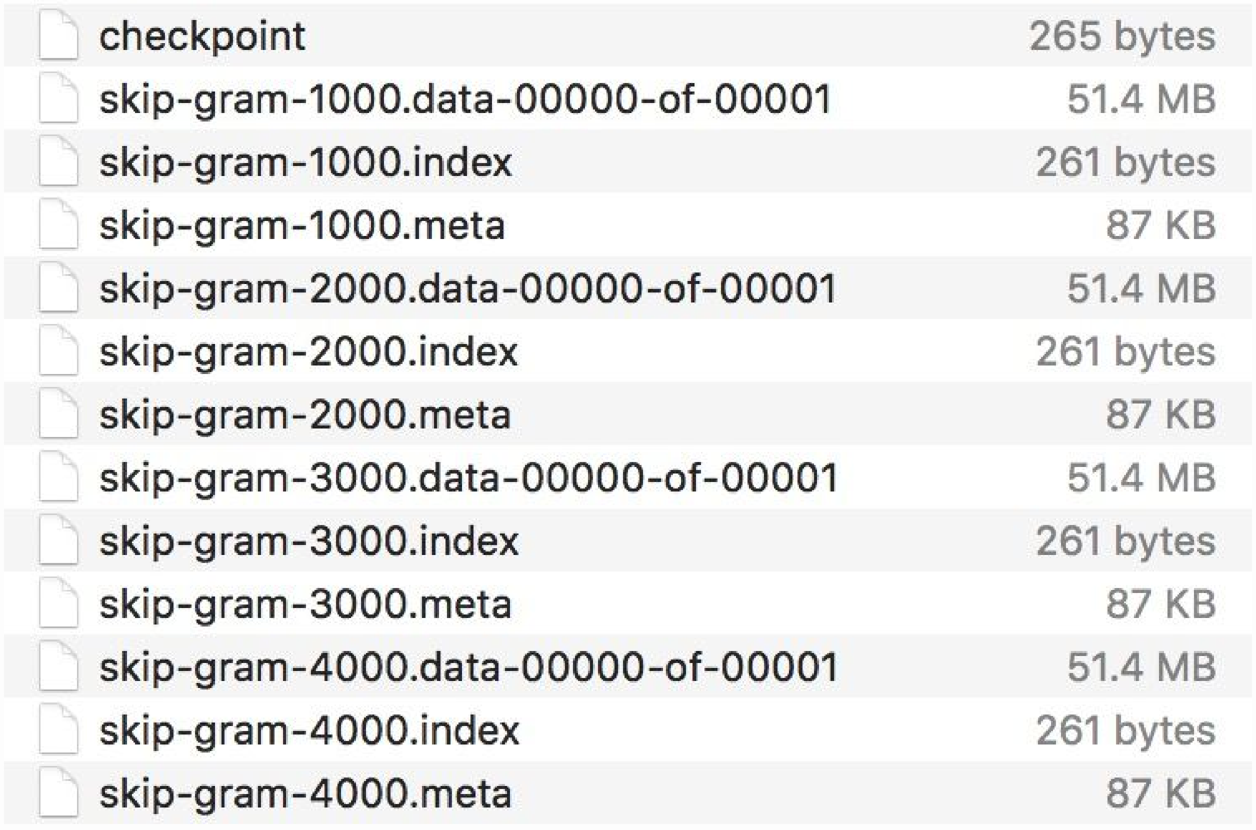
```
self.global_step = tf.Variable(0, dtype=tf.int32, trainable=False, name='global_step')

self.optimizer = tf.train.GradientDescentOptimizer(self.lr).minimize(self.loss,
global_step=self.global_step)

saver = tf.train.Saver() # defaults to saving all variables
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())

        average_loss = 0.0
        writer = tf.summary.FileWriter('./improved_graph', sess.graph)
        for index in xrange(num_train_steps):
            batch = batch_gen.next()
            loss_batch, _ = sess.run([model.loss, model.optimizer],
                                    feed_dict={model.center_words: batch[0],
                                               model.target_words: batch[1]})
            average_loss += loss_batch
            if (index + 1) % 1000 == 0:
                saver.save(sess, 'checkpoints/skip-gram', global_step=model.global_step)
```

If you go to the folder 'checkpoints', you will see files like the below:



```
checkpoint                                265 bytes
skip-gram-1000.data-00000-of-00001        51.4 MB
skip-gram-1000.index                      261 bytes
skip-gram-1000.meta                       87 KB
skip-gram-2000.data-00000-of-00001        51.4 MB
skip-gram-2000.index                      261 bytes
skip-gram-2000.meta                       87 KB
skip-gram-3000.data-00000-of-00001        51.4 MB
skip-gram-3000.index                      261 bytes
skip-gram-3000.meta                       87 KB
skip-gram-4000.data-00000-of-00001        51.4 MB
skip-gram-4000.index                      261 bytes
skip-gram-4000.meta                       87 KB
```

To restore the variables, we use tf.train.Saver.restore(sess, save_path). For example, you want to restore the checkpoint at 10,000th step.

```
saver.restore(sess, 'checkpoints/skip-gram-10000')
```

But of course, we can only load saved variables if there is a valid checkpoint. What you probably want to do is that if there is a checkpoint, restore it. If there isn't, train from the start. TensorFlow allows you to get checkpoint from a directory with tf.train.get_checkpoint_state('directory-name'). The code for checking looks something like this:

```
ckpt = tf.train.get_checkpoint_state(os.path.dirname('checkpoints/checkpoint'))
if ckpt and ckpt.model_checkpoint_path:
     saver.restore(sess, ckpt.model_checkpoint_path)
```

The file checkpoint automatically updates the path to the latest checkpoint.

```
model_checkpoint_path: "skip-gram-21999"
all_model_checkpoint_paths: "skip-gram-13999"
all_model_checkpoint_paths: "skip-gram-15999"
all_model_checkpoint_paths: "skip-gram-17999"
all_model_checkpoint_paths: "skip-gram-19999"
all_model_checkpoint_paths: "skip-gram-21999"
```

By default, saver.save() stores all variables of the graph, and this is recommended. However, you can also choose what variables to store by passing them in as a list or a dict when we create the saver object. Example from TensorFlow documentation.

```
v1 = tf.Variable(..., name='v1')
v2 = tf.Variable(..., name='v2')

# pass the variables as a dict:
saver = tf.train.Saver({'v1': v1, 'v2': v2})

# pass them as a list
saver = tf.train.Saver([v1, v2])

# passing a list is equivalent to passing a dict with the variable op names # as keys
saver = tf.train.Saver({v.op.name: v for v in [v1, v2]})
```

Note that savers only save variables, not the entire graph, so we still have to create the graph ourselves, and then load in variables. The checkpoints specify the way to map from variable names to tensors.

What people usually is not just save the parameters from the last iteration, but also save the parameters that give the best result so far so that you can evaluate your model on the best parameters so far.

tf.summary

We've been using matplotlib to visualize our losses and accuracy, which is cool but unnecessary because TensorBoard provides us with a great set of tools to visualize our summary statistics during our training. Some popular statistics to visualize is loss, average loss, accuracy. You can visualize them as scalar plots, histograms, or even images. So we have a new namescope in our graph to hold all the summary ops.

```
def _create_summaries(self):
    with tf.name_scope("summaries"):
            tf.summary.scalar("loss", self.loss
            tf.summary.scalar("accuracy", self.accuracy)
            tf.summary.histogram("histogram loss", self.loss)
            # because you have several summaries, we should merge them all
            # into one op to make it easier to manage
            self.summary_op = tf.summary.merge_all()
```

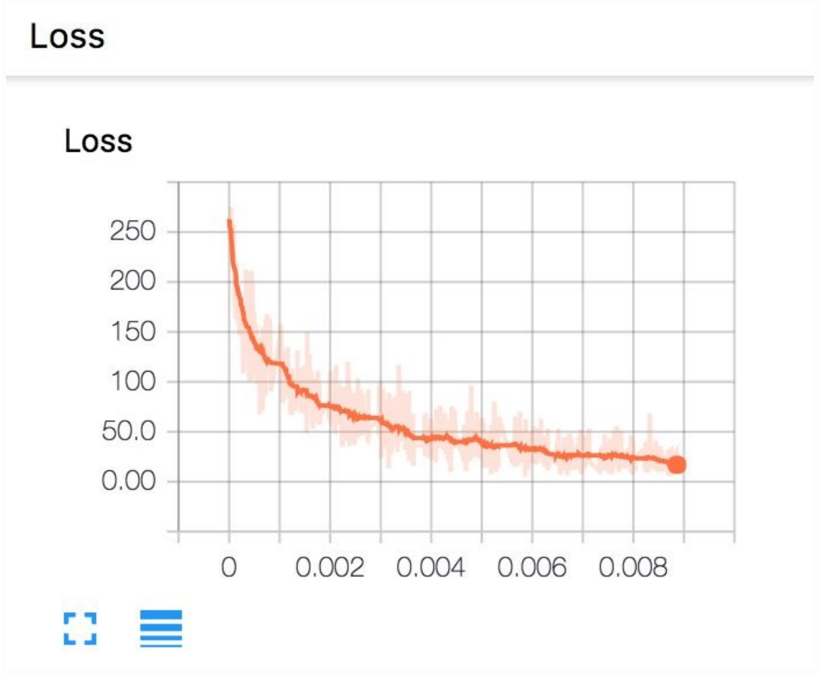
Because it's an op, you have to execute it with sess.run()

```
loss_batch, _, summary = sess.run([model.loss, model.optimizer, model.summary_op],
                                  feed_dict=feed_dict)
```

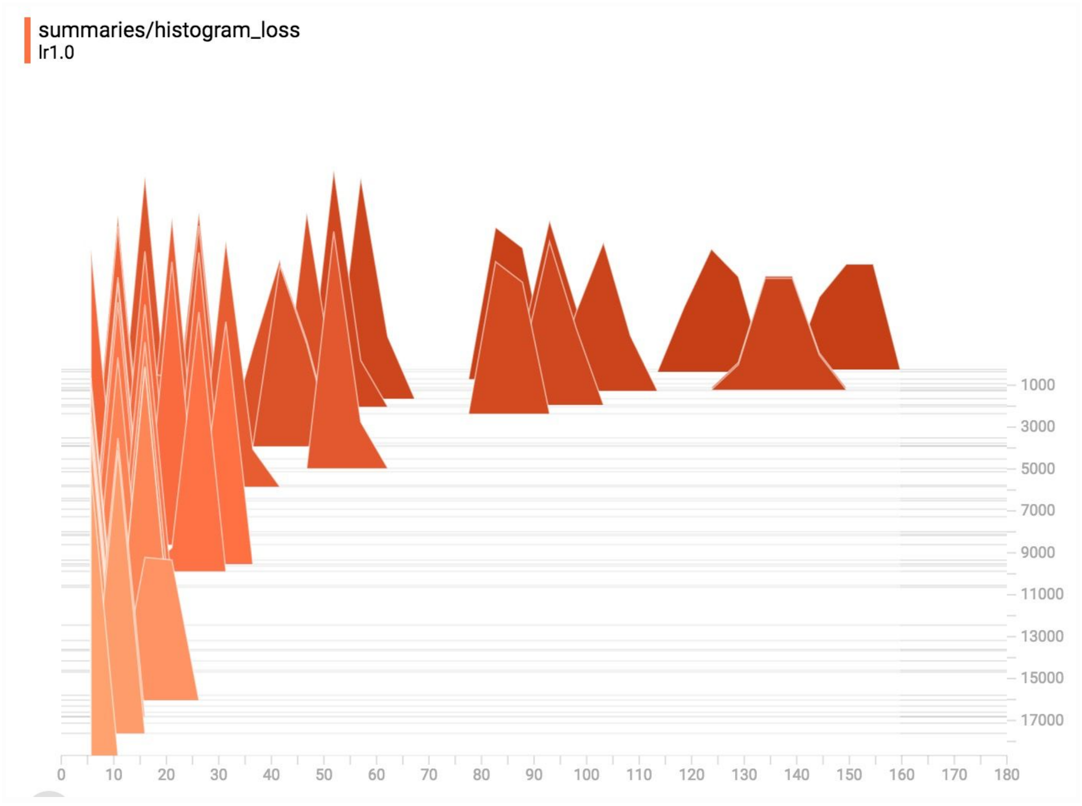
Now you've obtained the summary, you need to write the summary to file using the same FileWriter object we created to visual our graph.

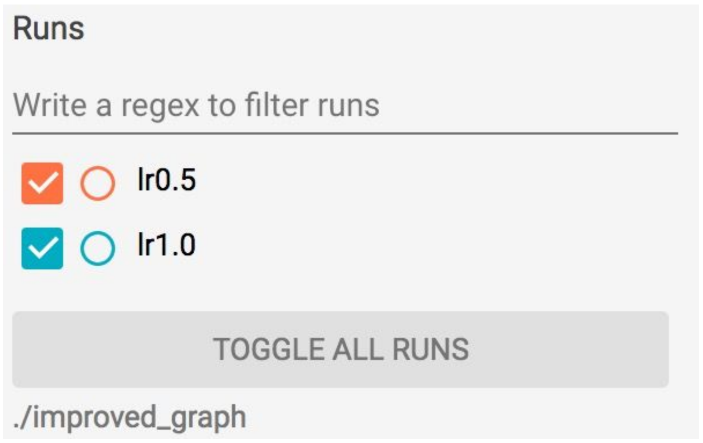
```
writer.add_summary(summary, global_step=step)
```

Now, if you go run tensorboard and go to http://localhost:6006/, in the Scalars page, you will see the plot of your scalar summaries. This is the summary of your loss in scalar plot.

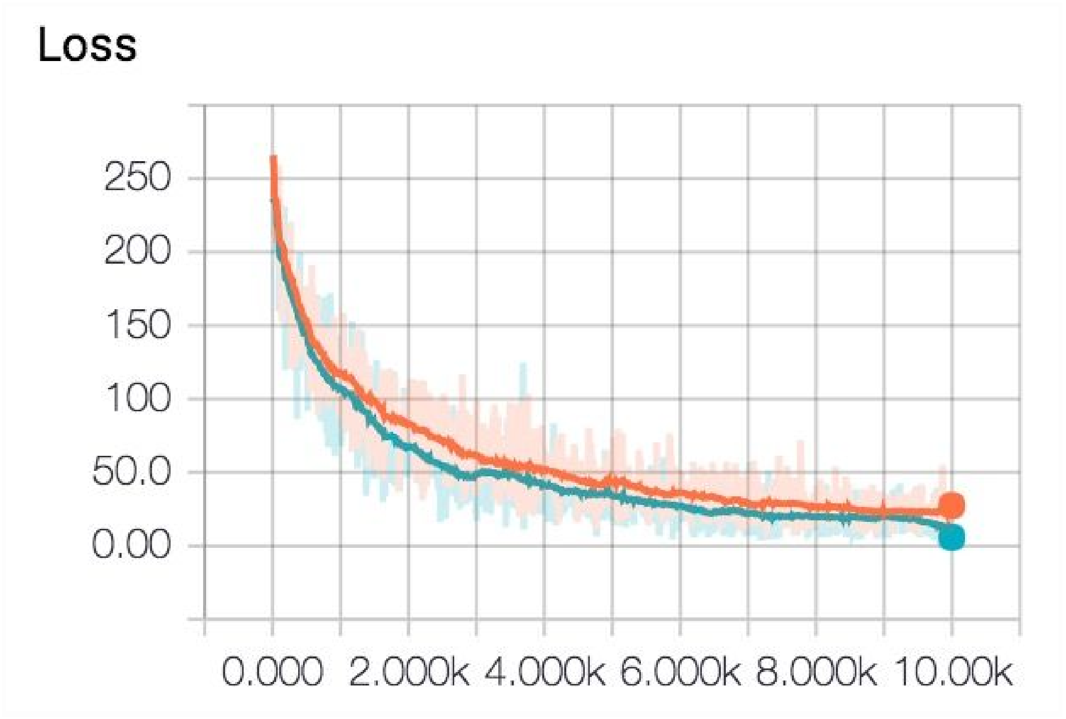


And the loss in histogram plot.





If you save your summaries into different sub-folder in your graph folder, you can compare your progresses. For example, the first time we run our model with learning rate 1.0, we save it in 'improved_graph/lr1.0' and the second time we run our model, we save it in 'improved_graph/lr0.5', on the left corner of the Scalars page, we can toggle the plots of these two runs to compare them. This can be really helpful when you want to compare the progress made with different optimizers or different parameters.



You can write a Python script to automate the naming of folders where you store the graphs/plots of each experiment.

You can visualize the statistics as images using tf.summary.image.

```
tf.summary.image(name, tensor, max_outputs=3, collections=None)
```

Control randomization

I never realized what an oxymoron this sounds like until I've written it down, but the truth is that you often have to control the randomization process to get stable results for your experiments. You're probably familiar with random seed and random state from NumPy. TensorFlow doesn't allow to you to get random state the way numpy does (at least not that I know of -- I will double check), but it does allow you to get stable results in randomization through two ways:

1. Set random seed at operation level. All random tensors allow you to pass in seed value in their initialization. For example:

```
my_var = tf.Variable(tf.truncated_normal((-1.0,1.0), stddev=0.1, seed=0))
```

Note that, session is the thing that keeps track of random state, so each new session will start the random state all over again.

```
c = tf.random_uniform([], -10, 10, seed=2)

with tf.Session() as sess:
	print sess.run(c) # >> 3.57493
	print sess.run(c) # >> -5.97319
```

```
c = tf.random_uniform([], -10, 10, seed=2)

with tf.Session() as sess:
	print sess.run(c) # >> 3.57493

with tf.Session() as sess:
	print sess.run(c) # >> 3.57493
```

With operation level random seed, each op keeps its own seed.

```
c = tf.random_uniform([], -10, 10, seed=2)
d = tf.random_uniform([], -10, 10, seed=2)

with tf.Session() as sess:
	print sess.run(c) # >> 3.57493
	print sess.run(d) # >> 3.57493
```

2. Set random seed at graph level with tf.Graph.seed

```
tf.set_random_seed(seed)
```

If you don't care about the randomization for each op inside the graph, but just want to be able to replicate result on another graph (so that other people can replicate your results on their own graph), you can use tf.set_random_seed instead. Setting the current TensorFlow random seed affects the current default graph only.

For example, you have two models a.py and b.py that have identical code:

```
import tensorflow as tf

tf.set_random_seed(2)
c = tf.random_uniform([], -10, 10)
d = tf.random_uniform([], -10, 10)

with tf.Session() as sess:
	print sess.run(c)
	print sess.run(d)
```

Without graph level seed, running python a.py and b.py will return 2 completely different results, but with tf.set_random_seed, you will get two identical results:

```
$ python a.py
>> -4.00752
>> -2.98339

$ python b.py
>> -4.00752
>> -2.98339
```

Reading Data in TensorFlow

There are two main ways to load data into a TensorFlow graph: one is through feed_dict that we are familiar with, and another is through readers that allow us to read tensors directly from file. There is, of course, the third way which is to load in your data using constants, but you should only use this if you want your graph to be seriously bloated and un-runnable (I made up another word but you know what I mean).

To see why we need something more than feed_dict, we need to look into how feed_dict works under the hood. Feed_dict will first send data from the storage system to the client, and then from client to the worker process. This will cause the data to slow down, especially if the client is on a different machine from the worker process. TensorFlow has readers that allow us to load data directly into the worker process.

The improvement will not be noticeable when we aren't on a distributed system or when our dataset is small, but it's still something worth looking into. TensorFlow has several built in readers to match your reading needs.

```
tf.TextLineReader
Outputs the lines of a file delimited by newlines
E.g. text files, CSV files

tf.FixedLengthRecordReader
Outputs the entire file when all files have same fixed lengths
E.g. each MNIST file has 28 x 28 pixels, CIFAR-10 32 x 32 x 3

tf.WholeFileReader
Outputs the entire file content

tf.TFRecordReader
Reads samples from TensorFlow's own binary format (TFRecord)

tf.ReaderBase
Allows you to create your own readers
```

Data can be read in as individual data examples or in batches of examples.

<not finished, will update soon>