TensorFlow Data Input (Part 2): Extensions & Hacks

Luke Metz - May 2, 2016

In Part 1 of this mini series, we explored various methods of data input for machine learning models using TensorFlow (https://indico.io/blog/tensorflow-data-inputs-part1-placeholders-protobufs-queues/). In this article we'll discuss a hybrid approach of those methods that allows for faster training, as well as some extensions to the demo in Part 1.

A Hybrid Approach

While great in theory, the approach of reading into queues has its issues, namely being unacceptably slow. Sometimes this is fine. For large, image net sized models it's not unheard of to train at a few hundred examples a second. For many other cases though this will not cut it. This approach, while less elegant, is significantly faster.

The basic idea is to construct a custom worker that behaves similar to the TensorFlow `QueueRunner`. Instead of getting data from TensorFlow constructs, however, it pulls data from the same Numpy iterator we defined in the first section and enqueues that onto a queue. All this happens in a background thread so the main thread is free to train a model.

The one important thing here is the order in which you construct your graph. TensorFlow's sessions support running on separate threads. They do not support construction of the graph. This must happen on the main thread. In the example below I construct the `CustomRunner` on the main thread. Its constructor creates the necessary operations on the TensorFlow graph. These operations do not need a session at this point.

After these operations are constructed, I initialize all variables on the graph. Only after initializing is it okay to start the processing threads. If one of these steps is out of order you might encounter race conditions.

```python
import tensorflow as tf
import time
import threading
from skdata.mnist.views import OfficialVectorClassification
import numpy as np

# load data entirely into memory 😕
data = OfficialVectorClassification()
trIdx = data.sel_idxs[:]
features = data.all_vectors[trIdx]
labels = data.all_labels[trIdx]

batch_size = 128
def data_iterator():
    """ A simple data iterator """
    batch_idx = 0
    while True:

        # shuffle labels and features
        idxs = np.arange(0, len(features))
        np.random.shuffle(idxs)
        shuf_features = features[idxs]
        shuf_labels = labels[idxs]
        for batch_idx in range(0, len(features), batch_size):
            images_batch = shuf_features[batch_idx:batch_idx + batch_size] / 255.
            images_batch = images_batch.astype("float32")
            labels_batch = shuf_labels[batch_idx:batch_idx + batch_size]
            yield images_batch, labels_batch

class CustomRunner(object):
    """
    This class manages the the background threads needed to fill
        a queue full of data.
    """
    def __init__(self):
        self.dataX = tf.placeholder(dtype=tf.float32, shape=[None, 28*28])
        self.dataY = tf.placeholder(dtype=tf.int64, shape=[None, ])
        # The actual queue of data. The queue contains a vector for
        # the mnist features, and a scalar label.
        self.queue = tf.RandomShuffleQueue(shapes=[[28*28], []],
                                           dtypes=[tf.float32, tf.int64],
                                           capacity=2000,
                                           min_after_dequeue=1000)

        # The symbolic operation to add data to the queue
        # we could do some preprocessing here or do it in numpy. In this example
        # we do the scaling in numpy
        self.enqueue_op = self.queue.enqueue_many([self.dataX, self.dataY])

    def get_inputs(self):
        """
        Return's tensors containing a batch of images and labels
        """
        images_batch, labels_batch = self.queue.dequeue_many(128)
        return images_batch, labels_batch

    def thread_main(self, sess):
        """
        Function run on alternate thread. Basically, keep adding data to the queue.
        """
        for dataX, dataY in data_iterator():
            sess.run(self.enqueue_op, feed_dict={self.dataX:dataX, self.dataY:dataY})

    def start_threads(self, sess, n_threads=1):
        """ Start background threads to feed queue """
        threads = []
        for n in range(n_threads):
            t = threading.Thread(target=self.thread_main, args=(sess,))
            t.daemon = True # thread will close when parent quits
            t.start()
            t.start()
            threads.append(t)
        return threads

# Doing anything with data on the CPU is generally a good idea.
with tf.device("/cpu:0"):
    custom_runner = CustomRunner()
    images_batch, labels_batch = custom_runner.get_inputs()

# simple model
w = tf.get_variable("w1", [28*28, 10])
y_pred = tf.matmul(images_batch, w)
loss = tf.nn.sparse_softmax_cross_entropy_with_logits(y_pred, labels_batch)

# for monitoring
loss_mean = tf.reduce_mean(loss)
train_op = tf.train.AdamOptimizer().minimize(loss)


sess = tf.Session(config=tf.ConfigProto(intra_op_parallelism_threads=8))
init = tf.initialize_all_variables()
sess.run(init)

# start the tensorflow QueueRunner's
tf.train.start_queue_runners(sess=sess)
# start our custom queue runner's threads
custom_runner.start_threads(sess)

while True:
    _, loss_val = sess.run([train_op, loss_mean])
    print loss_val
```

Extensions and Thoughts

Now, the demo and walkthrough from Part 1 are just the basics. There are so many other things you can do with them.

Take working with large image data, for example. In practice nobody wants to store uncompressed images. They just take up too much space. If you are using the `tfrecords` approach you can store an arbitrarily sized jpeg encoded image inside a `tf.train.Feature` using a `byte_list`. You can then decode the jpeg symbolically with `tf.image.decode_jpeg`. With this image decoded, you can take a patch out with `tf.random_crop`. These crops can then be combined into a batch as they are now the same shape. See TensorFlow's image module (https://www.tensorflow.org/versions/r0.7/api_docs/python/image.html#image-adjustments) for more examples.

Another benefit to this workflow is that all of this decoding and augmentation is happening on a separate thread (that does not have the GIL). The results are placed in a queue that is read from the training thread. This means that the main thread is free to do your fast training or run other Python code.

The above example uses a single worker thread to fill the example queue.

At a base level, Queues let a user decouple the graph and the data. By default the loading and manipulation is done with `tfrecords`. You're also free to create a thread that does data manipulation and feeds a Queue by itself without using TensorFlow's `tfrecords` at all.

Finally, a note on performance. There are some tricks to make the `tfrecords` version work faster. It is hard to beat numpy indexes and ability to load the entire dataset into memory for small datasets though. It is important to always run data manipulation on the CPU (with a `tf.device("/cpu:0")` for example). In addition to that, I have found for large imagenet-like models, the augmentation pipeline is still slow. You can run multiple different copies of augmentation code in a session, then using `shuffle_batch_join` and that should speed things up a bit. In addition to this, the `intra_op_parallelism_threads` configuration in a session's `ConfigProto` affects performance (probably something to do with thread contention). In my tests, setting this to a low number like one or two helps a lot.

Working with large amounts of arbitrary sized images is no easy matter how you do it. Sadly, I have not found an approach that works very well in pure TensorFlow. My current approach to managing this kind of problem is to format my data into MXNet's record format and use something similar to the `CustomRunner` example to feed the resulting data into a TensorFlow queue.

If something goes wrong

Random non deterministic / changing errors about initialization and or node dependencies:
The error always changes but for me this signals I did not initialize all the variables in the session before trying to use them. This could be due to calling `tf.train.start_queue_runners` too early.

System hangs / doesn't read data:
This is most likely due to not calling `tf.train.start_queue_runners`. It is possible to construct some funky connections between queues that might also cause deadlocks as well.

Done reading and ready to build?

GET STARTED (HTTPS://INDICO.IO/PLANS)

(/)

Hack indico (/hack)
Gallery (/gallery/)
News (/news)
Blog (/blog/)
RSS Feed (https://indico.io/blog/feed/)

Careers (https://jobs.lever.co/indico)
Docs (/docs)
Team (/team)
Terms of Service (/terms)
Privacy (/terms#privacy)

Contact (/contact)

(https://github.com/IndicoDataSolutions)

(https://www.facebook.com/IndicoDataSolutions)

(https://twitter.com/indicodata)

(https://www.youtube.com/channel/UCGuUwm6PaPkeftGFmNOfTHw)

(https://instagram.com/indicodata/)

mixpanel MOBILE ANALYTICS (https://mixpanel.com/f/partner)