Lecture note 9: Input Pipeline

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

I hope you guys enjoyed the last few guest lectures. Now we need to get back to doing some real work. Because of the guest lectures, we haven't been able to get to some important concepts that a lot of people seem to be confused about. I want to go over them in details just to make sure everyone is on the same page. A lot of what is written below is taken from TensorFlow documentation.

Queues and Coordinators

We briefly mentioned queues but never discussed them in details. In TensorFlow documentation, queues are described as "important TensorFlow objects for computing tensors asynchronously in a graph."

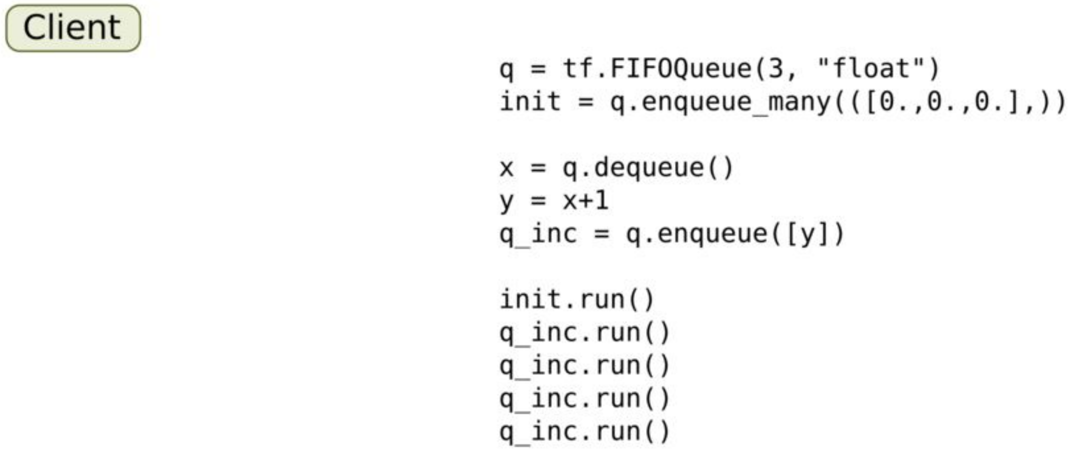
If you've done any project in deep learning, you probably don't need me to convince you why you need asynchronous programming. In input pipeline, multiple threats can help us reduce the bottleneck at the reading in data phase because reading in data is a lot of waiting. For example, in using queues to prepare inputs for training a model, we have:

- Multiple threads prepare training examples and push them in the queue.
- A training thread executes a training op that dequeues mini-batches from the queue.

The TensorFlow Session object is designed multithreaded, so multiple threads can easily use the same session and run ops in parallel. However, it is not always easy to implement a Python program that drives threads as described above. All threads must be able to stop together, exceptions must be caught and reported, and queues must be properly closed when stopping.

The documentation made it sound like threading is optional in running a queue, but actually without threading, it's very likely that your queue will run into a gridlock (one op waits for another) and crash your program. Fortunately, TensorFlow provides two classes to help with the threading: **tf.Coordinator** and **tf.train.QueueRunner**. These two classes are designed to be used together. The Coordinator class helps multiple threads stop together and report exceptions to a program that waits for them to stop. The QueueRunner class is used to create a number of threads cooperating to enqueue tensors in the same queue.

There are two main queue classes, tf.FIFOQueue and tf.RandomShuffleQueue. FIFOQueue creates a queue the dequeues elements in a first in first out order, while RandomShuffleQueue dequeues elements in, well, a random order. These two queues support the enqueue, enqueue_many, and dequeue (which do exactly what they sound). A common practice is that you enqueue many examples in when you read your data, but dequeue them one by one. dequeue_many is not allowed. If you want to get multiple elements at once for your batch training, you'll have to use tf.train.batch or tf.train.shuffle_batch if you want to your batch to be shuffled.



```
q = tf.FIFOQueue(3, "float")
init = q.enqueue_many(([0.,0.,0.],))

x = q.dequeue()
y = x+1
q_inc = q.enqueue([y])

init.run()
q_inc.run()
q_inc.run()
q_inc.run()
q_inc.run()
```

There is also tf.PaddingFIFOQueue which is a FIFOQueue that supports batching variable-sized tensors by padding. Sometimes you need to feed variable size batches in, for example, in sequence to sequence models for natural language processing, a lot of time you want each sentence to be a batch, but sentences don't have equal lengths. A PaddingFIFOQueue may contain components with dynamic shape, while also supporting dequeue_many. There is also the CS106-favorite tf.PriorityQueue, which is a FIFOQueue whose enqueues and dequeues take in another argument: the priority.

I don't know the exactly reason why dequeue_many is allowed in PaddingFIFOQueue but not in other queues. My wild guess through reading the reported issues on TensorFlow GitHub repository is that dequeue_many used to be supported in FIFOQueue and RandomShuffleQueue, but then people ran into a lot of problems with it, so TensorFlow just disallowed it.

You can create your queue independently with parameters such that min_after_dequeue (the minimum number of elements in the queue after you've dequeued), bounded capacity (the maximum elements in the queue at a time), shape of the elements in the queue (if shape is None then elements can be of any shape). However, in practice, you rarely use a queue by itself, but always with string_input_producer, so we'll go over this section briefly. We'll go cover string_input_producer in more details in a little bit.

```
tf.RandomShuffleQueue(capacity, min_after_dequeue, dtypes, shapes=None, names=None,
seed=None, shared_name=None, name='random_shuffle_queue')
```

An example is as below. You can see it on the GitHub repo under the name 09_queue_example.py

```
N_SAMPLES = 1000
NUM_THREADS = 4
# Generating some simple data
# create 1000 random samples, each is a 1D array from the normal distribution (10, 1)
data = 10 * np.random.randn(N_SAMPLES, 4) + 1
# create 1000 random labels of 0 and 1
target = np.random.randint(0, 2, size=N_SAMPLES)

queue = tf.FIFOQueue(capacity=50, dtypes=[tf.float32, tf.int32], shapes=[[4], []])

enqueue_op = queue.enqueue_many([data, target])
dequeue_op = queue.dequeue()

# create NUM_THREADS to do enqueue
qr = tf.train.QueueRunner(queue, [enqueue_op] * NUM_THREADS)
with tf.Session() as sess:
        # Create a coordinator, launch the queue runner threads.
        coord = tf.train.Coordinator()
        enqueue_threads = qr.create_threads(sess, coord=coord, start=True)
        for step in xrange(100): # do to 100 iterations
                if coord.should_stop():
                        break
                data_batch, label_batch = sess.run(dequeue_op)
        coord.request_stop()
        coord.join(enqueue_threads)
```

You also don’t need to use tf.Coordinator with TensorFlow queues, but can use it to manage threads of any thread you create. For example, you use the Python package threading to create threads to do some crazy job, you can still use tf.Coordinator to manage these threads too. The syntax of target and args are similar to the classic threadpool. For more details on threading, you should take CS 110. The example below is from TensorFlow documentation.

```
import threading

# thread body: loop until the coordinator indicates a stop was requested.
# if some condition becomes true, ask the coordinator to stop.

def my_loop(coord):
        while not coord.should_stop():
                ...do something...
        if ...some condition...:
                coord.request_stop()

# main code: create a coordinator.
coord = tf.Coordinator()

# create 10 threads that run 'my_loop()'
# you can also create threads using QueueRunner as the example above
threads = [threading.Thread(target=my_loop, args=(coord,)) for _ in xrange(10)]

# start the threads and wait for all of them to stop.
for t in threads:
        t.start()
coord.join(threads)
```

Data Readers

We went over data readers in lecture 5, and some students tried to implement it for assignment 1 but none got it to work. Data readers are a bit tricky to use, and the ambiguous documentation doesn't really help.

We've learned that there are 3 different ways to read in data for your TensorFlow. The first is through constants (which will seriously bloat your graph -- which you'll see in assignment 2). The second is through feed dict which has the drawback of first loading the data from storage to the client and then from the client to workers, which can be slow especially when the client and workers are on different machines. A common practice is to use data readers to load your data directly from storage to workers. In theory, this means that you can load in an amount of data limited only by your storage and not your device.

There are several built-in readers for several common data types. The most versatile one is TextLineReader, which will read in any file delimited by newlines and will just return a line in that with each call. There are also a reader to read in files of fixed length, a reader to read in entire files, and a reader to read in the file of the type TFRecord (which we will go into below).

```
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. This is useful when each file contains a sample

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

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

To use data reader, we first need to create a queue to hold the names of all the files you want to read in through tf.train.string_input_producer.

```
filename_queue = tf.train.string_input_producer(["heart.csv"])
reader = tf.TextLineReader(skip_header_lines=1)
# it means you choose to skip the first line for every file in the queue
```

My friend encouraged me to think of readers as ops that return a different value every time you call it -- similar to Python generators. So when you call reader.read(), it'll return you a pair key, value, in which key is a key to identify the file and record (useful for debugging if you have some weird records), and a scalar string value.

```
key, value = reader.read(filename_queue)
```

For each example, the call to read() above might return:

```
key = data/heart.csv:2
value = 144,0.01,4.41,28.61,Absent,55,28.87,2.06,63,1
```

Where the value “144,0.01,4.41,28.61,Absent,55,28.87,2.06,63,1” is the second line (excluding the header line) in the file heart.csv.

tf.train.string_input_producer creates a FIFOQueue under the hood, so to run the queue, we’ll need tf.Coordinator and tf.QueueRunner.

```
filename_queue = tf.train.string_input_producer(filenames)
reader = tf.TextLineReader(skip_header_lines=1) # skip the first line in the file
key, value = reader.read(filename_queue)

with tf.Session() as sess:
        coord = tf.train.Coordinator()
        threads = tf.train.start_queue_runners(coord=coord)
        print sess.run(key) # data/heart.csv:2
        print sess.run(value) # 144,0.01,4.41,28.61,Absent,55,28.87,2.06,63,1
        coord.request_stop()
        coord.join(threads)
```

The value returned is just a string tensor. If all you want is a string to feed into your model, that’s fine. But the majority of the time, you’d want to convert the string into a vector representation of features. For example, our heart.csv file has 10 columns, the first 9 columns correspond to 9 features, and the last corresponds to label (0/1). To do so, we need to use TensorFlow CSV decoder.

```
content = tf.decode_csv(value, record_defaults=record_defaults)
```

The line of code above will parse value into the tensor record defaults which we have to create ourselves. The record defaults serve two purposes:

- First, it tells the decoder what types of data to expect in each column.
- Second, if a space in a column happens to be empty, it’ll fill in that space with the default value of the data type that we specify.

For the record_defaults of this specific dataset, we’d like it to have 10 elements. All elements are either integers or floats, except for the fifth element that is a string. To make it easier, we assume that all feature integers are floats (we’ll still specify the 10th column to be integer, because we like our labels to be integer).

```
record_defaults = [[1.0] for _ in range(N_FEATURES)] # define all features to be floats
```

```
record_defaults[4] = [''] # make the fifth feature string
record_defaults.append([1])
content = tf.decode_csv(value, record_defaults=record_defaults)
```

You can also do all the kind of pre-processing you need for your data before feeding it in. For example, now we have our content is a list of 10 elements, 8 are floats, 1 is string, and 1 is integer. We'll have to convert the string to float (Absent as 0 and Present as 1), and then convert the first 9 features into a tensor that can be fed into the model.

```
# convert the 5th column (present/absent) to the binary value 0 and 1
condition = tf.equal(content[4], tf.constant('Present'))
content[4] = tf.select(condition, tf.constant(1.0), tf.constant(0.0))

# pack all 9 features into a tensor
features = tf.pack(content[:N_FEATURES])

# assign the last column to label
label = content[-1]
```

With that, every time the reader reads in a line from our CSV file, it'll convert that line into a feature tensor and a label!

But we often don't want to feed in a single sample into our model, but instead, we would want to batch 'em up. You can do so using tf.train.batch, or tf.train.shuffle_batch if you want to shuffle your batches.

```
# minimum number elements in the queue after a dequeue, used to ensure
# that the samples are sufficiently mixed
# I think 10 times the BATCH_SIZE is sufficient
min_after_dequeue = 10 * BATCH_SIZE

# the maximum number of elements in the queue
capacity = 20 * BATCH_SIZE

# shuffle the data to generate BATCH_SIZE sample pairs
data_batch, label_batch = tf.train.shuffle_batch([features, label], batch_size=BATCH_SIZE,
                                    capacity=capacity, min_after_dequeue=min_after_dequeue)
```

And with that we're done. You can simply use data_batch and label_batch the way you would have used input_placeholder and label_placeholder in our previous model, except you don't need to feed them in through the feed_dict parameters. The full code can be accessed on the GitHub repo under the name 05_csv_reader.py

TFRecord

Binary files are extremely useful, though I have met a lot of people who are somehow shy away from them because they think binary files are cumbersome. If you're one of those people, I hope that through this lecture will help you overcome your irrational fear of binary files. They make better use of disk cache. They are faster to move around. They can store data of different types (so you can put both images and labels in one place).

Like many machine learning frameworks, TensorFlow has its own binary data format which is called TFRecord. A TFRecord is a serialized tf.train.Example Protobuf object. They can be created in a few lines of code. Below is an example to convert an image into a TFRecord.

First, we need to read in the image and convert it to byte string.

```
def get_image_binary(filename):
    image = Image.open(filename)
    image = np.asarray(image, np.uint8)
    shape = np.array(image.shape, np.int32)
    return shape.tobytes(), image.tobytes() # convert image to raw data bytes in the array.
```

Next, you write these byte strings into a TFRecord file using tf.python_io.TFRecordWriter and tf.train.Features. You need the shape information so you can reconstruct the image from the binary format later.

```
def write_to_tfrecord(label, shape, binary_image, tfrecord_file):
    """ This example is to write a sample to TFRecord file. If you want to write
    more samples, just use a loop.
    """
    writer = tf.python_io.TFRecordWriter(tfrecord_file)
    # write label, shape, and image content to the TFRecord file
    example = tf.train.Example(features=tf.train.Features(feature={
                'label': tf.train.Feature(bytes_list=tf.train.BytesList(value=[label])),
                'shape': tf.train.Feature(bytes_list=tf.train.BytesList(value=[shape])),
                'image':tf.train.Feature(bytes_list=tf.train.BytesList(
                                                                 value=[binary_image]))
                }))
    writer.write(example.SerializeToString())
    writer.close()
```

To read a TFRecord file, you use TFRecordReader and tf.decode_raw.

```
def read_from_tfrecord(filenames):
    tfrecord_file_queue = tf.train.string_input_producer(filenames, name='queue')
    reader = tf.TFRecordReader()
    _, tfrecord_serialized = reader.read(tfrecord_file_queue)

    # label and image are stored as bytes but could be stored as
    # int64 or float64 values in a serialized tf.Example protobuf.
    tfrecord_features = tf.parse_single_example(tfrecord_serialized,
                        features={
                            'label': tf.FixedLenFeature([], tf.string),
                            'shape': tf.FixedLenFeature([], tf.string),
                            'image': tf.FixedLenFeature([], tf.string),
                        }, name='features')
    # image was saved as uint8, so we have to decode as uint8.
    image = tf.decode_raw(tfrecord_features['image'], tf.uint8)
    shape = tf.decode_raw(tfrecord_features['shape'], tf.int32)
    # the image tensor is flattened out, so we have to reconstruct the shape
    image = tf.reshape(image, shape)
    label = tf.cast(tfrecord_features['label'], tf.string)
    return label, shape, image
```

Keep in mind that label, shape, and image returned are tensor objects. To get their values, you'll have to eval them in tf.Session().

Style Transfer

(Discussion of Style Transfer -- see assignment 2 handout)