



(/blog/)

TensorFlow Data Input (Part 1): Placeholders, Protobufs & Queues

38
Shares

Luke Metz - April 25, 2016

28

TensorFlow is a great new deep learning framework provided by the team at Google Brain. It supports the symbolic construction of functions (similar to Theano) to perform some computation, generally a neural network based model. Unlike Theano, TensorFlow supports a number of ways to feed data into your machine learning model. The processes of getting data into a model can be rather annoying, with a lot of glue code. TensorFlow tries to fix this by providing a few ways to feed in data. The easiest of these is to use placeholders, which allow you to manually pass in numpy arrays of data.

The second method, my preferred, is to do as much as I possibly can on the graph and to make use of binary files and input queues. Not only does this lighten the amount of code I need to write, removing the need to do any data augmentation or file reading, but the interface is reasonably standard across different kinds of data. It is also conceptually cleaner. No longer is there an artificial divide between preprocessing and model computation.

Placeholders

Feed dicts are the simplest and easiest to understand. One copies in the data needed for each batch through placeholders, and TensorFlow does the work of training the model. This is great for small examples and easily interacting with data. Here's an example:

```

import tensorflow as tf

images_batch = tf.placeholder(dtype=tf.float32, shape=[None, 28*28,])
labels_batch = tf.placeholder(dtype=tf.int32, shape=[None, ])

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

sess = tf.Session()
init = tf.initialize_all_variables()
sess.run(init)

from skdata.mnist.view import OfficialVectorClassification
# or, if you have an old version: 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]

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]
        batch_size = 128
        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

iter_ = data_iterator()
while True:
    # get a batch of data
    images_batch_val, labels_batch_val = iter_.next()
    # pass it in as through feed_dict
    _, loss_val = sess.run([train_op, loss_mean], feed_dict={
        images_batch:images_batch_val,
        labels_batch:labels_batch_val
    })
    print loss_val

```

This is all fine and good for small toy problems like the one above. As tasks become more and more complicated, the problem of managing your own data becomes harder and harder. Take images, for instance. You need to have consistent image loading, and a scaling pipeline that is consistent at train and test time. In addition, you need to ensure that the input is fast enough – possibly multi-threaded.

TensorFlow makes use of a few concepts, such as protobuf and queues, to make this easier – but at the cost of a little bit more upfront learning.

Protobuf and binary formats

I used to hate binary files. To me they were opaque, hard to build, and annoying to work with. After a little bit of getting used to, they are actually quite nice and provide benefits that raw data could never hope to provide. They make better use of disk cache, are faster to copy and move around, don't need to keep separate files to denote labels or other information, don't have to fight with folders of millions of images messing with unix tools... the list goes on. In addition, these files will always be read off of disk in a standardized way and never all at once. In other words, your dataset size is only bounded by hard drive space. After working binary files a little bit (both in MXNet and TensorFlow), I am surprised I was able to make do without them. I would strongly recommend giving them a shot in your next project.

Much like many other deep learning frameworks, TensorFlow has its own binary format. Its format uses a mixture of its Records format and and Protocol Buffers to solve this problem. If you're unfamiliar with Protobuf, you can think about it as a way to serialize data structures, given some schema describing what the data is.

TFRecords are TensorFlow's default data format. A record is simply a binary file that contains serialized `tf.train.Example` Protobuf objects, and can be created from Python in a few lines of code. Below is an example to convert mnist to this format.

```

# Load up some dataset. Could be anything but skdata is convenient.
from skdata.mnist.views import OfficialVectorClassification
from tqdm import tqdm
import numpy as np
import tensorflow as tf

data = OfficialVectorClassification()
trIdx = data.sel_idxs[:]

# one MUST randomly shuffle data before putting it into one of these
# formats. Without this, one cannot make use of tensorflow's great
# out of core shuffling.
np.random.shuffle(trIdx)

writer = tf.python_io.TFRecordWriter("mnist.tfrecords")
# iterate over each example
# wrap with tqdm for a progress bar
for example_idx in tqdm(trIdx):
    features = data.all_vectors[example_idx]
    label = data.all_labels[example_idx]

    # construct the Example proto boject
    example = tf.train.Example(
        # Example contains a Features proto object
        features=tf.train.Features(
            # Features contains a map of string to Feature proto objects
            feature={
                # A Feature contains one of either a int64_list,
                # float_list, or bytes_list
                'label': tf.train.Feature(
                    int64_list=tf.train.Int64List(value=[label])),
                'image': tf.train.Feature(
                    int64_list=tf.train.Int64List(value=features.astype("int64"))),
            })
        # use the proto object to serialize the example to a string
        serialized = example.SerializeToString()
        # write the serialized object to disk
        writer.write(serialized)

```

To best understand these classes, I would recommend looking at the proto source:

example.proto

(<https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/example/example.proto>)

and feature.proto

(<https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/example/feature.proto>).

Basically, an `Example` always contains `Features`. `Features` contains a map of strings to `Feature`. And finally, a `Feature` contains one of a `FloatList`, a `ByteList` or a `Int64List`.

With this record, all needed information is stored in one place and is always aligned with all data needed to describe the example.

The data in these records can be read back sequentially in the same order they were written.

```
import tensorflow as tf

filename = "mnist.tfrecords"
for serialized_example in tf.python_io.tf_record_iterator(filename):
    example = tf.train.Example()
    example.ParseFromString(serialized_example)

    # traverse the Example format to get data
    image = example.features.feature['image'].int64_list.value
    label = example.features.feature['label'].int64_list.value[0]
    # do something
    print image, label
```

Reading into Queues

Once your data is in this format, you can make use of many tools TensorFlow provides to feed your machine learning model.

```
import tensorflow as tf

def read_and_decode_single_example(filename):
    # first construct a queue containing a list of filenames.
    # this lets a user split up there dataset in multiple files to keep
    # size down
    filename_queue = tf.train.string_input_producer([filename],
                                                    num_epochs=None)

    # Unlike the TFRecordWriter, the TFRecordReader is symbolic
    reader = tf.TFRecordReader()
    # One can read a single serialized example from a filename
    # serialized_example is a Tensor of type string.
    _, serialized_example = reader.read(filename_queue)
    # The serialized example is converted back to actual values.
    # One needs to describe the format of the objects to be returned
    features = tf.parse_single_example(
        serialized_example,
        features={
            # We know the Length of both fields. If not the
            # tf.VarLenFeature could be used
            'label': tf.FixedLenFeature([], tf.int64),
            'image': tf.FixedLenFeature([784], tf.int64)
        })
    # now return the converted data
    label = features['label']
    image = features['image']
    return label, image
```

Now to use this function we can do something like the following:

```
# returns symbolic label and image
label, image = read_and_decode_single_example("mnist.tfrecords")

sess = tf.Session()

# Required. See below for explanation
init = tf.initialize_all_variables()
sess.run(init)
tf.train.start_queue_runners(sess=sess)

# grab examples back.
# first example from file
label_val_1, image_val_1 = sess.run([label, image])
# second example from file
```

The fact that this works requires a fair bit of effort behind the scenes. First, it is important to remember that TensorFlow's graphs contain state. It is this state that allows the `TFRecordReader` to remember the location of the `tfrecord` it's reading and always return the next one. This is why for almost all TensorFlow work we need to initialize the graph. We can use the helper function `tf.initialize_all_variables()`, which constructs an op that initializes the state on the graph when you run it.

The next major piece involves Tensorflow's Queues and QueueRunners. A TensorFlow queue is quite similar to a regular queue, except all of its operations are symbolic and only performed on Tensorflow's graph when needed by a `sess.run`. See [here](https://www.tensorflow.org/versions/r0.7/how_tos/threading_and_queues/index.htm) for a comprehensive write up of how these things work and a great animation for queues (https://www.tensorflow.org/versions/r0.7/how_tos/threading_and_queues/index.htm).

As part of the API, a `TFRecordReader` always acts on a queue of filenames. It will pop a filename off the queue and use that filename until the `tfrecord` is empty. At this point it will grab the next filename off the filename queue.

To summarize thus far, we grab a filename off our queue of filenames and use it to get examples from a `TFRecordReader`. Both the queue and the `TFRecordReader` have some state to keep track of where they are. The only piece missing is how the filename queue is actually fed. On initialization it is empty. This is where the concept of `QueueRunners` comes in. There's nothing magical about it. At its core, it is simply a thread that uses a session and calls an enqueue op over and over again. TensorFlow has noticed this pattern and wrapped it in its `tf.train.QueueRunner` object. 99% of the time this can be ignored as it is used behind the scenes, (such as in the above example – `tf.train.string_input_producer` creates one).

We do, however, have to signal to TensorFlow to start these threads. Without this, the queue will be blocked indefinitely, waiting for data to be enqueued. To start the `QueueRunners` you can call `tf.train.start_queue_runners(sess=sess)`. This call is not symbolic. It goes and creates threads. It is important to note that this must be called **after** the initialization op is run. These threads try to enqueue data to queues. If the queues are not initialized, TensorFlow will rightly throw an error.

Great! Now we have two Tensors that represents a single example. We could train with this using gradient descent, but that is rather silly. It has been shown that training with batches of examples work far better than training with a single examples. Your estimates of gradients will have a lower variance which makes training faster. Luckily TensorFlow provides utilities to batch these examples and return batches

```

# get single examples
label, image = read_and_decode_single_example("mnist.tfrecords")
# groups examples into batches randomly
images_batch, labels_batch = tf.train.shuffle_batch(
    [image, label], batch_size=128,
    capacity=2000,
    min_after_dequeue=1000)

sess = tf.Session()
init = tf.initialize_all_variables()
sess.run(init)
tf.train.start_queue_runners(sess=sess)
labels, images= sess.run([labels_batch, images_batch])

```

At first glance, this seems like quite a natural API – but the implementation is rather subtle. As discussed above, if `label` and `image` represent a single example, how can we make a batch of different examples? The answer lies in another set of queues and `QueueRunners`. `shuffle_batch` constructs a `RandomShuffleQueue` and proceeds to fill it with individual `image` and `label` s. This filling is done on a separate thread with a `QueueRunner`. The `RandomShuffleQueue` accumulates examples sequentially until it contains `batch_size` + `min_after_dequeue` examples are present. It then selects `batch_size` random elements from the queue to return. The value actually returned by `shuffle_batch` is the result of a `dequeue_many` call on the `RandomShuffleQueue`.

With these batch variables, you can now go on your merry way and train a model.

```

# get single examples
label, image = read_and_decode_single_example("mnist.tfrecords")
image = tf.cast(image, tf.float32) / 255.
# groups examples into batches randomly
images_batch, labels_batch = tf.train.shuffle_batch(
    [image, label], batch_size=128,
    capacity=2000,
    min_after_dequeue=1000)

# 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()
init = tf.initialize_all_variables()
sess.run(init)
tf.train.start_queue_runners(sess=sess)

while True:
    # pass it in through the feed_dict
    _, loss_val = sess.run([train_op, loss_mean])
    print loss_val

```

To Conclude

In summary of the whole process, you need to:

1. Define a record reader function of some kind. This parses the record.
2. Construct a batcher of some kind.
3. Build the rest of your model from these ops.
4. Initialize all the ops.
5. Start queue runners.
6. Run your train loop.

You should now have several options at your disposal for feeding data into your machine learning model in a TensorFlow framework. As you may or may not have noticed, the example using queues while conceptually clean, is painfully slow. TensorFlow is constantly improving and hopefully this workflow speeds up. In Part 2, we discuss a hybrid approach to these methods (<https://indico.io/blog/tensorflow-data-input-part2-extensions/>) that allows for the niceties of queues while being as fast as the placeholder approach, as well as some extensions to the demo we looked at here. As always, feel free to reach out to us with questions and thoughts at contact@indico.io (<mailto:contact@indico.io>)!

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.

Suggested Posts

Machine Learning So Easy, Even Your Cat Could Do It (Part 2): Text Tags (<https://indico.io/blog/machine-learning-so-easy-even-your-cat-could-do-it-text-tags/>)

Auto-tagging Interior Design Styles (<https://indico.io/blog/auto-tagging-interior-design-styles/>)

New API Released: Twitter Engagement (<https://indico.io/blog/new-api-twitter-engagement/>)

(/)

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/>)



(<https://mixpanel.com/f/partner>)

