Reading data

There are three main methods of getting data into a TensorFlow program:

- Feeding: Python code provides the data when running each step.
- Reading from files: an input pipeline reads the data from files at the beginning of a TensorFlow graph.
- Preloaded data: a constant or variable in the TensorFlow graph holds all the data (for small data sets).

Feeding

TensorFlow's feed mechanism lets you inject data into any Tensor in a computation graph. A python computation can thus feed data directly into the graph.

Supply feed data through the `feed_dict` argument to a run() or eval() call that initiates computation.

```
with tf.Session():
  input = tf.placeholder(tf.float32)
  classifier = ...
  print(classifier.eval(feed_dict={input: my_python_preprocessing_fn()}))
```

While you can replace any Tensor with feed data, including variables and constants, the best practice is to use a **`tf.placeholder`** (https://www.tensorflow.org/api_docs/python/tf/placeholder) node. A **`placeholder`** exists solely to serve as the target of feeds. It is not initialized and contains no data. A placeholder generates an error if it is executed without a feed, so you won't forget to feed it.

An example using **`placeholder`** and feeding to train on MNIST data can be found in **`tensorflow/examples/tutorials/mnist/fully_connected_feed.py`** (https://www.github.com/tensorflow/tensorflow/blob/r1.2/tensorflow/examples/tutorials/mnist/fully_connected_feed.py), and is described in the MNIST tutorial (https://www.tensorflow.org/get_started/mnist/mechanics).

Reading from files

A typical pipeline for reading records from files has the following stages:

1. The list of filenames
2. *Optional* filename shuffling
3. *Optional* epoch limit
4. Filename queue
5. A Reader for the file format
6. A decoder for a record read by the reader
7. *Optional* preprocessing
8. Example queue

Filenames, shuffling, and epoch limits

For the list of filenames, use either a constant string Tensor (like `["file0", "file1"]` or `[("file%d" % i) for i in range(2)]`) or the `tf.train.match_filenames_once` (https://www.tensorflow.org/api_docs/python/tf/train/match_filenames_once) function.

Pass the list of filenames to the `tf.train.string_input_producer` (https://www.tensorflow.org/api_docs/python/tf/train/string_input_producer) function. `string_input_producer` creates a FIFO queue for holding the filenames until the reader needs them.

`string_input_producer` has options for shuffling and setting a maximum number of epochs. A queue runner adds the whole list of filenames to the queue once for each epoch, shuffling the filenames within an epoch if `shuffle=True`. This procedure provides a uniform sampling of files, so that examples are not under- or over- sampled relative to each other.

The queue runner works in a thread separate from the reader that pulls filenames from the queue, so the shuffling and enqueuing process does not block the reader.

File formats

Select the reader that matches your input file format and pass the filename queue to the reader's read method. The read method outputs a key identifying the file and record (useful for debugging if you have some weird records), and a scalar string value. Use one (or more) of the decoder and conversion ops to decode this string into the tensors that make up an example.

CSV files

To read text files in comma-separated value (CSV) format (https://tools.ietf.org/html/rfc4180), use a `tf.TextLineReader` (https://www.tensorflow.org/api_docs/python/tf/TextLineReader) with the `tf.decode_csv` (https://www.tensorflow.org/api_docs/python/tf/decode_csv) operation. For example:

```
filename_queue = tf.train.string_input_producer(["file0.csv", "file1.csv"])

reader = tf.TextLineReader()
key, value = reader.read(filename_queue)

# Default values, in case of empty columns. Also specifies the type of the
# decoded result.
record_defaults = [[1], [1], [1], [1], [1]]
col1, col2, col3, col4, col5 = tf.decode_csv(
    value, record_defaults=record_defaults)
features = tf.stack([col1, col2, col3, col4])

with tf.Session() as sess:
  # Start populating the filename queue.
  coord = tf.train.Coordinator()
  threads = tf.train.start_queue_runners(coord=coord)

  for i in range(1200):
    # Retrieve a single instance:
    example, label = sess.run([features, col5])

  coord.request_stop()
  coord.join(threads)
```

Each execution of `read` reads a single line from the file. The `decode_csv` op then parses the result into a list of tensors. The `record_defaults` argument determines the type of the resulting tensors and sets the default value to use if a value is missing in the input string.

You must call `tf.train.start_queue_runners` to populate the queue before you call `run` or `eval` to execute the `read`. Otherwise `read` will block while it waits for filenames from the queue.

Fixed length records

To read binary files in which each record is a fixed number of bytes, use `tf.FixedLengthRecordReader` (https://www.tensorflow.org/api_docs/python/tf/FixedLengthRecordReader) with the `tf.decode_raw` (https://www.tensorflow.org/api_docs/python/tf/decode_raw) operation. The `decode_raw` op converts from a string to a uint8 tensor.

For example, the CIFAR-10 dataset (http://www.cs.toronto.edu/~kriz/cifar.html) uses a file format where each record is represented using a fixed number of bytes: 1 byte for the label followed by 3072 bytes of image data. Once you have a uint8 tensor, standard operations can slice out each piece and reformat as needed. For CIFAR-10, you can see how to do the reading and decoding in `tensorflow_models/tutorials/image/cifar10/cifar10_input.py` (https://github.com/tensorflow/models/tree/master/tutorials/image/cifar10/cifar10_input.py) and described in this tutorial (https://www.tensorflow.org/tutorials/deep_cnn#prepare_the_data).

Standard TensorFlow format

Another approach is to convert whatever data you have into a supported format. This approach makes it easier to mix and match data sets and network architectures. The recommended format for TensorFlow is a TFRecords file (https://www.tensorflow.org/api_guides/python/python_io#tfrecords_format_details) containing `tf.train.Example` protocol buffers (https://www.github.com/tensorflow/tensorflow/blob/r1.2/tensorflow/core/example/example.proto) (which contain `Features` (https://www.github.com/tensorflow/tensorflow/blob/r1.2/tensorflow/core/example/feature.proto) as a field). You write a little program that gets your data, stuffs it in an `Example` protocol buffer, serializes the protocol buffer to a string, and then writes the string to a TFRecords file using the `tf.python_io.TFRecordWriter` (https://www.tensorflow.org/api_docs/python/tf/python_io/TFRecordWriter). For example, `tensorflow/examples/how_tos/reading_data/convert_to_records.py` (https://www.github.com/tensorflow/tensorflow/blob/r1.2/tensorflow/examples/how_tos/reading_data/convert_to_records.py) converts MNIST data to this format.

To read a file of TFRecords, use `tf.TFRecordReader` (https://www.tensorflow.org/api_docs/python/tf/TFRecordReader) with the `tf.parse_single_example` (https://www.tensorflow.org/api_docs/python/tf/parse_single_example) decoder. The `parse_single_example` op decodes the example protocol buffers into tensors. An MNIST example using the data produced by `convert_to_records` can be found in `tensorflow/examples/how_tos/reading_data/fully_connected_reader.py` (https://www.github.com/tensorflow/tensorflow/blob/r1.2/tensorflow/examples/how_tos/reading_data/fully_connected_reader.py), which you can compare with the `fully_connected_feed` version.

Preprocessing

You can then do any preprocessing of these examples you want. This would be any processing that doesn't depend on trainable parameters. Examples include normalization of your data, picking a random slice, adding noise or distortions, etc. See `tensorflow_models/tutorials/image/cifar10/cifar10_input.py` (https://github.com/tensorflow/models/tree/master/tutorials/image/cifar10/cifar10_input.py) for an example.

Batching

At the end of the pipeline we use another queue to batch together examples for training, evaluation, or inference. For this we use a queue that randomizes the order of examples, using the `tf.train.shuffle_batch` (https://www.tensorflow.org/api_docs/python/tf/train/shuffle_batch).

Example:

```
def read_my_file_format(filename_queue):
  reader = tf.SomeReader()
  key, record_string = reader.read(filename_queue)
  example, label = tf.some_decoder(record_string)
  processed_example = some_processing(example)
  return processed_example, label

def input_pipeline(filenames, batch_size, num_epochs=None):
  filename_queue = tf.train.string_input_producer(
      filenames, num_epochs=num_epochs, shuffle=True)
  example, label = read_my_file_format(filename_queue)
  # min_after_dequeue defines how big a buffer we will randomly sample
  #   from -- bigger means better shuffling but slower start up and more
  #   memory used.
  # capacity must be larger than min_after_dequeue and the amount larger
  #   determines the maximum we will prefetch.  Recommendation:
  #   min_after_dequeue + (num_threads + a small safety margin) * batch_size
  min_after_dequeue = 10000
  capacity = min_after_dequeue + 3 * batch_size
  example_batch, label_batch = tf.train.shuffle_batch(
      [example, label], batch_size=batch_size, capacity=capacity,
      min_after_dequeue=min_after_dequeue)
  return example_batch, label_batch
```

If you need more parallelism or shuffling of examples between files, use multiple reader instances using the `tf.train.shuffle_batch_join` (https://www.tensorflow.org/api_docs/python/tf/train/shuffle_batch_join). For example:

```
def read_my_file_format(filename_queue):
  # Same as above

def input_pipeline(filenames, batch_size, read_threads, num_epochs=None):
  filename_queue = tf.train.string_input_producer(
      filenames, num_epochs=num_epochs, shuffle=True)
  example_list = [read_my_file_format(filename_queue)
                  for _ in range(read_threads)]
  min_after_dequeue = 10000
  capacity = min_after_dequeue + 3 * batch_size
  example_batch, label_batch = tf.train.shuffle_batch_join(
      example_list, batch_size=batch_size, capacity=capacity,
      min_after_dequeue=min_after_dequeue)
  return example_batch, label_batch
```

You still only use a single filename queue that is shared by all the readers. That way we ensure that the different readers use different files from the same epoch until all the files from the epoch have been started. (It is also usually sufficient to have a single thread filling the filename queue.)

An alternative is to use a single reader via the `tf.train.shuffle_batch` (https://www.tensorflow.org/api_docs/python/tf/train/shuffle_batch) with `num_threads` bigger than 1. This will make it read from a single file at the same time (but faster than with 1 thread), instead of N files at once. This can be important:

- If you have more reading threads than input files, to avoid the risk that you will have two threads reading the same example from the same file near each other.
- Or if reading N files in parallel causes too many disk seeks.

How many threads do you need? the `tf.train.shuffle_batch*` functions add a summary to the graph that indicates how full the example queue is. If you have enough reading threads, that summary will stay above zero. You can view your summaries as training progresses using TensorBoard (https://www.tensorflow.org/get_started/summaries_and_tensorboard).

Creating threads to prefetch using `QueueRunner` objects

The short version: many of the `tf.train` functions listed above add `tf.train.QueueRunner` (https://www.tensorflow.org/api_docs/python/tf/train/QueueRunner) objects to your graph. These require that you call `tf.train.start_queue_runners` (https://www.tensorflow.org/api_docs/python/tf/train/start_queue_runners) before running any training or inference steps, or it will hang forever. This will start threads that run the input pipeline, filling the example queue so that the dequeue to get the examples will succeed.

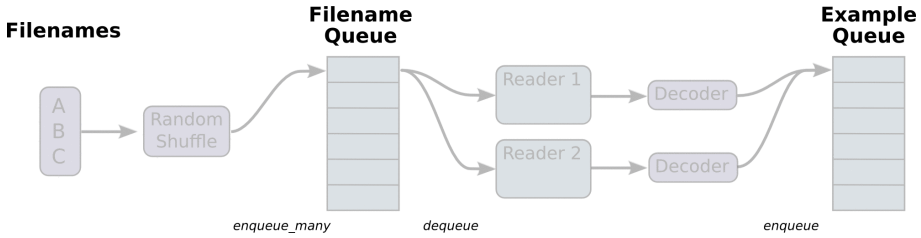
This is best combined with a `tf.train.Coordinator` (https://www.tensorflow.org/api_docs/python/tf/train/Coordinator) to cleanly shut down these threads when there are errors. If you set a limit on the number of epochs, that will use an epoch counter that will need to be initialized. The recommended code pattern combining these is:

```
# Create the graph, etc.
init_op = tf.global_variables_initializer()

# Create a session for running operations in the Graph.
sess = tf.Session()

# Initialize the variables (like the epoch counter).
sess.run(init_op)

# Start input enqueue threads.
coord = tf.train.Coordinator()
threads = tf.train.start_queue_runners(sess=sess, coord=coord)

try:
    while not coord.should_stop():
        # Run training steps or whatever
        sess.run(train_op)

except tf.errors.OutOfRangeError:
    print('Done training -- epoch limit reached')
finally:
    # When done, ask the threads to stop.
    coord.request_stop()

# Wait for threads to finish.
coord.join(threads)
sess.close()
```

Aside: What is happening here?

First we create the graph. It will have a few pipeline stages that are connected by queues. The first stage will generate filenames to read and enqueue them in the filename queue. The second stage consumes filenames (using a `Reader`), produces examples, and enqueues them in an example queue. Depending on how you have set things up, you may actually have a few independent copies of the second stage, so that you can read from multiple files in parallel. At the end of these stages is an enqueue operation, which enqueues into a queue that the next stage dequeues from. We want to start threads running

these enqueuing operations, so that our training loop can dequeue examples from the example queue.



The helpers in `tf.train` that create these queues and enqueuing operations add a `tf.train.QueueRunner` (https://www.tensorflow.org/api_docs/python/tf/train/QueueRunner) to the graph using the `tf.train.add_queue_runner` (https://www.tensorflow.org/api_docs/python/tf/train/add_queue_runner) function. Each `QueueRunner` is responsible for one stage, and holds the list of enqueue operations that need to be run in threads. Once the graph is constructed, the `tf.train.start_queue_runners` (https://www.tensorflow.org/api_docs/python/tf/train/start_queue_runners) function asks each QueueRunner in the graph to start its threads running the enqueuing operations.

If all goes well, you can now run your training steps and the queues will be filled by the background threads. If you have set an epoch limit, at some point an attempt to dequeue examples will get an `tf.errors.OutOfRangeError` (https://www.tensorflow.org/api_docs/python/tf/errors/OutOfRangeError). This is the TensorFlow equivalent of "end of file" (EOF) -- this means the epoch limit has been reached and no more examples are available.

The last ingredient is the `tf.train.Coordinator` (https://www.tensorflow.org/api_docs/python/tf/train/Coordinator). This is responsible for letting all the threads know if anything has signalled a shut down. Most commonly this would be because an exception was raised, for example one of the threads got an error when running some operation (or an ordinary Python exception).

For more about threading, queues, QueueRunners, and Coordinators see here (https://www.tensorflow.org/programmers_guide/threading_and_queues).

Aside: How clean shut-down when limiting epochs works

Imagine you have a model that has set a limit on the number of epochs to train on. That means that the thread generating filenames will only run that many times before generating an `OutOfRange` error. The QueueRunner will catch that error, close the filename queue, and exit the thread. Closing the queue does two things:

- Any future attempt to enqueue in the filename queue will generate an error. At this point there shouldn't be any threads trying to do that, but this is helpful when queues are closed due to other errors.
- Any current or future dequeue will either succeed (if there are enough elements left) or fail (with an `OutOfRange` error) immediately. They won't block waiting for more elements to be enqueued, since by the previous point that can't happen.

The point is that when the filename queue is closed, there will likely still be many filenames in that queue, so the next stage of the pipeline (with the reader and other preprocessing) may continue running for some time. Once the filename queue is exhausted, though, the next attempt to dequeue a filename (e.g. from a reader that has finished with the file it was working on) will trigger an `OutOfRange` error. In this case, though, you might have multiple threads associated with a single QueueRunner. If this isn't the last thread in the QueueRunner, the `OutOfRange` error just causes the one thread to exit. This allows the other threads, which are still finishing up their last file, to proceed until they finish as well. (Assuming you are using a `tf.train.Coordinator` (https://www.tensorflow.org/api_docs/python/tf/train/Coordinator), other types of errors will cause all the threads to stop.) Once all the reader threads hit the `OutOfRange` error, only then does the next queue, the example queue, gets closed.

Again, the example queue will have some elements queued, so training will continue until those are exhausted. If the example queue is a `tf.RandomShuffleQueue` (https://www.tensorflow.org/api_docs/python/tf/RandomShuffleQueue), say because you are using `shuffle_batch` or `shuffle_batch_join`, it normally will avoid ever having fewer than its `min_after_dequeue` attr elements buffered. However, once the queue is closed that restriction will be lifted and the queue will eventually empty. At that point the actual training threads, when they try and dequeue from example queue, will start getting `OutOfRange` errors and exiting. Once all the training threads are done, `tf.train.Coordinator.join` (https://www.tensorflow.org/api_docs/python/tf/train/Coordinator#join) will return and you can exit cleanly.

Filtering records or producing multiple examples per record

Instead of examples with shapes `[x, y, z]`, you will produce a batch of examples with shape `[batch, x, y, z]`. The batch size can be 0 if you want to filter this record out (maybe it is in a hold-out set?), or bigger than 1 if you are producing multiple examples per record. Then simply set `enqueue_many=True` when calling one of the batching functions (such as `shuffle_batch` or `shuffle_batch_join`).

Sparse input data

SparseTensors don't play well with queues. If you use SparseTensors you have to decode the string records using **`tf.parse_example`** (https://www.tensorflow.org/api_docs/python/tf/parse_example) **after** batching (instead of using **`tf.parse_single_example`** before batching).

Preloaded data

This is only used for small data sets that can be loaded entirely in memory. There are two approaches:

- Store the data in a constant.
- Store the data in a variable, that you initialize and then never change.

Using a constant is a bit simpler, but uses more memory (since the constant is stored inline in the graph data structure, which may be duplicated a few times).

```
training_data = ...
training_labels = ...
with tf.Session():
  input_data = tf.constant(training_data)
  input_labels = tf.constant(training_labels)
  ...
```

To instead use a variable, you need to also initialize it after the graph has been built.

```
training_data = ...
training_labels = ...
with tf.Session() as sess:
  data_initializer = tf.placeholder(dtype=training_data.dtype,
                                    shape=training_data.shape)
  label_initializer = tf.placeholder(dtype=training_labels.dtype,
                                     shape=training_labels.shape)
  input_data = tf.Variable(data_initializer, trainable=False, collections=[])
  input_labels = tf.Variable(label_initializer, trainable=False, collections=
  ...
  sess.run(input_data.initializer,
           feed_dict={data_initializer: training_data})
  sess.run(input_labels.initializer,
           feed_dict={label_initializer: training_labels})
```

Setting **`trainable=False`** keeps the variable out of the **`GraphKeys.TRAINABLE_VARIABLES`** collection in the graph, so we won't try and update it when training. Setting **`collections=[]`**

keeps the variable out of the `GraphKeys.GLOBAL_VARIABLES` collection used for saving and restoring checkpoints.

Either way, `tf.train.slice_input_producer` (https://www.tensorflow.org/api_docs/python/tf/train/slice_input_producer) can be used to produce a slice at a time. This shuffles the examples across an entire epoch, so further shuffling when batching is undesirable. So instead of using the `shuffle_batch` functions, we use the plain `tf.train.batch` (https://www.tensorflow.org/api_docs/python/tf/train/batch) function. To use multiple preprocessing threads, set the `num_threads` parameter to a number bigger than 1.

An MNIST example that preloads the data using constants can be found in `tensorflow/examples/how_tos/reading_data/fully_connected_preloaded.py` (https://www.github.com/tensorflow/tensorflow/blob/r1.2/tensorflow/examples/how_tos/reading_data/fully_connected_preloaded.py), and one that preloads the data using variables can be found in `tensorflow/examples/how_tos/reading_data/fully_connected_preloaded_var.py` (https://www.github.com/tensorflow/tensorflow/blob/r1.2/tensorflow/examples/how_tos/reading_data/fully_connected_preloaded_var.py), You can compare these with the `fully_connected_feed` and `fully_connected_reader` versions above.

Multiple input pipelines

Commonly you will want to train on one dataset and evaluate (or "eval") on another. One way to do this is to actually have two separate processes:

- The training process reads training input data and periodically writes checkpoint files with all the trained variables.
- The evaluation process restores the checkpoint files into an inference model that reads validation input data.

This is what is done in the example CIFAR-10 model (https://www.tensorflow.org/tutorials/deep_cnn#save_and_restore_checkpoints). This has a couple of benefits:

- The eval is performed on a single snapshot of the trained variables.
- You can perform the eval even after training has completed and exited.

You can have the train and eval in the same graph in the same process, and share their trained variables. See the shared variables tutorial

(https://www.tensorflow.org/programmers_guide/variable_scope).

上次更新日期：六月 19, 2017