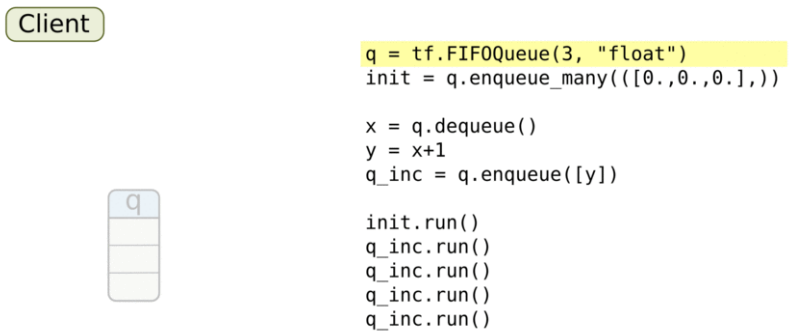
Threading and Queues

Queues are a powerful mechanism for asynchronous computation using TensorFlow.

Like everything in TensorFlow, a queue is a node in a TensorFlow graph. It's a stateful node, like a variable: other nodes can modify its content. In particular, nodes can enqueue new items in to the queue, or dequeue existing items from the queue.

To get a feel for queues, let's consider a simple example. We will create a "first in, first out" queue (`FIFOQueue`) and fill it with zeros. Then we'll construct a graph that takes an item off the queue, adds one to that item, and puts it back on the end of the queue. Slowly, the numbers on the queue increase.



`Enqueue`, `EnqueueMany`, and `Dequeue` are special nodes. They take a pointer to the queue instead of a normal value, allowing them to change it. We recommend you think of these as being like methods of the queue. In fact, in the Python API, they are methods of the queue object (e.g. `q.enqueue(...)`).

N.B. Queue methods (such as `q.enqueue(...)`) *must* run on the same device as the queue. Incompatible device placement directives will be ignored when creating these operations.

Now that you have a bit of a feel for queues, let's dive into the details...

Queue usage overview

Queues, such as `tf.FIFOQueue` (https://www.tensorflow.org/api_docs/python/tf/FIFOQueue) and `tf.RandomShuffleQueue` (https://www.tensorflow.org/api_docs/python/tf/RandomShuffleQueue), are important TensorFlow objects for computing tensors asynchronously in a graph.

For example, a typical input architecture is to use a `RandomShuffleQueue` to prepare inputs for training a model:

- 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

This architecture has many benefits, as highlighted in the Reading data how to (https://www.tensorflow.org/programmers_guide/reading_data), which also gives an overview of functions that simplify the construction of input pipelines.

The TensorFlow `Session` object is 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.

TensorFlow provides two classes to help: `tf.train.Coordinator` (https://www.tensorflow.org/api_docs/python/tf/train/Coordinator) and `tf.train.QueueRunner` (https://www.tensorflow.org/api_docs/python/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.

Coordinator

The `Coordinator` class helps multiple threads stop together.

Its key methods are:

- `tf.train.Coordinator.should_stop` (https://www.tensorflow.org/api_docs/python/tf/train/Coordinator#should_stop)
  : returns True if the threads should stop.
- `tf.train.Coordinator.request_stop` (https://www.tensorflow.org/api_docs/python/tf/train/Coordinator#request_stop)
  : requests that threads should stop.
- `tf.train.Coordinator.join` (https://www.tensorflow.org/api_docs/python/tf/train/Coordinator#join):
  waits until the specified threads have stopped.

You first create a `Coordinator` object, and then create a number of threads that use the coordinator. The threads typically run loops that stop

when **`should_stop()`** returns **`True`**.

Any thread can decide that the computation should stop. It only has to call **`request_stop()`** and the other threads will stop as **`should_stop()`** will then return **`True`**.

```
# Thread body: loop until the coordinator indicates a stop was
# If some condition becomes true, ask the coordinator to stop.
def MyLoop(coord):
  while not coord.should_stop():
    ...do something...
    if ...some condition...:
      coord.request_stop()

# Main thread: create a coordinator.
coord = tf.train.Coordinator()

# Create 10 threads that run 'MyLoop()'
threads = [threading.Thread(target=MyLoop, args=(coord,)) for

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

Obviously, the coordinator can manage threads doing very different things. They don't have to be all the same as in the example above. The coordinator also has support to capture and report exceptions. See the **`tf.train.Coordinator`** (https://www.tensorflow.org/api_docs/python/tf/train/Coordinator) documentation for more details.

QueueRunner

The **`QueueRunner`** class creates a number of threads that repeatedly run an enqueue op. These threads can use a coordinator to stop together. In addition, a queue runner runs a *closer thread* that automatically closes the queue if an exception is reported to the coordinator.

You can use a queue runner to implement the architecture described above.

First build a graph that uses a TensorFlow queue (e.g. a **`tf.RandomShuffleQueue`**) for input examples. Add ops that process examples and enqueue them in the queue. Add training ops that start by dequeueing from the queue.

```
example = ...ops to create one example...
# Create a queue, and an op that enqueues examples one at a ti
queue = tf.RandomShuffleQueue(...)
enqueue_op = queue.enqueue(example)
# Create a training graph that starts by dequeuing a batch of
inputs = queue.dequeue_many(batch_size)
train_op = ...use 'inputs' to build the training part of the g
```

In the Python training program, create a **`QueueRunner`** that will run a few threads to process and enqueue examples. Create a **`Coordinator`** and ask the queue runner to start its threads with the coordinator. Write a training loop that also uses the coordinator.

```
# Create a queue runner that will run 4 threads in parallel tc
# examples.
qr = tf.train.QueueRunner(queue, [enqueue_op] * 4)

# Launch the graph.
sess = tf.Session()
# Create a coordinator, launch the queue runner threads.
coord = tf.train.Coordinator()
enqueue_threads = qr.create_threads(sess, coord=coord, start=T
```

```
# Run the training loop, controlling termination with the coor
for step in xrange(1000000):
    if coord.should_stop():
        break
    sess.run(train_op)
# When done, ask the threads to stop.
coord.request_stop()
# And wait for them to actually do it.
coord.join(enqueue_threads)
```

Handling exceptions

Threads started by queue runners do more than just run the enqueue ops. They also catch and handle exceptions generated by queues, including the **`tf.errors.OutOfRangeError`** exception, which is used to report that a queue was closed.

A training program that uses a coordinator must similarly catch and report exceptions in its main loop.

Here is an improved version of the training loop above.

```
try:
    for step in xrange(1000000):
        if coord.should_stop():
            break
        sess.run(train_op)
except Exception, e:
    # Report exceptions to the coordinator.
    coord.request_stop(e)
finally:
    # Terminate as usual. It is safe to call `coord.request_st
    coord.request_stop()
    coord.join(threads)
```

上次更新日期：六月 19, 2017