在Python中，线程和队列的使用范围很广，例如可以将运行时间较长的代码放在线程中执行，避免页面阻塞问题，例如利用多线程和队列快速读取数据等。然而在Tensorflow中，线程与队列的使用场景基本上是读取数据。并且**在1.2之后的版本中使用了 `tf.contrib.data` 模块代替了多线程与队列读取数据的方法（为了兼容旧程序，多线程与队列仍然保留）。**

1. 队列

Tensorflow中拥有队列的概念，用于操作数据，队列这种数据结构如在银行办理业务排队一样，队伍前面的人先办理，新进来的人排到队尾。队列本身也是图中的一个节点，与其相关的还有入队（enqueue）节点和出队（dequeue）节点。入队节点可以把新元素插入到队列末尾，出队节点可以把队列前面的元素返回并在队列中将其删除。

在Tensorflow中有两种队列，即FIFOQueue和RandomShuffleQueue，前者是标准的先进先出队列，后者是随机队列。

1.1 FIFOQueue

FIFOQueue创建一个先进先出队列，这是很有用的，当我们需要加载一些有序数据，例如按字加载一段话，这时候我们不能打乱样本顺序，就可以使用队列。

创建先进先出队列使用 `ft.FIFOQueue()` ，具体如下：

```
tf.FIFOQueue(
    capacity,  # 容量 元素数量上限
    dtypes,
    shapes=None,
    names=None,
    shared_name=None,
    name='fifo_queue')
```

如下，我们定义一个容量为3的，元素类型为整型的队列：

```
queue = tf.FIFOQueue(3, tf.int32)
```

注意： `ft.FIFOQueue()` 的参数 `dtypes` 、 `shapes` 、 `names` 均是列表（当列表的长度为1时，可以使用元素代替列表），且为对应关系，如果dtypes中包含两个dtype，则shapes中也包含两个shape，names也包含两个name。其作用是同时入队或出队多个有关联的元素。

如下，定义一个容量为3，每个队列值包含一个整数和一个浮点数的队列：

```
queue = tf.FIFOQueue(
    3,
    dtypes=[tf.int32, tf.float32],
    shapes=[[], []],
    names=['my_int', 'my_float'])
```

队列创建完毕之后，就可以进行出队与入队操作了。

入队

入队即给队列添加元素。使用 `quene.enqueue()` 或 `queue.enqueue_many()` 方法，前者用来入队一个元素，后者用来入队0个或多个元素。如下：

```
queue = tf.FIFOQueue(3, dtypes=tf.float32)
# 入队一个元素 简写
eq1 = queue.enqueue(1.)
# 入队一个元素 完整写法
eq2 = queue.enqueue([1.])
# 入队多个元素 完整写法
eq3 = queue.enqueue_many([[1.], [2.]])

with tf.Session() as sess:
    sess.run([eq1, eq2, eq3])
```

如果入队操作会导致元素数量大于队列容量，则入队操作会阻塞，直到出队操作使元素数量减少到容量值。

当我们指定了队列中元素的names时，我们在入队时需要使用使用字典来指定入队元素，如下：

```
queue = tf.FIFOQueue(
    3,
    dtypes=[tf.float32, tf.int32],
    shapes=[[], []],
    names=['my_float', 'my_int'])
queue.enqueue({'my_float': 1., 'my_int': 1})
```

出队

出队即给从队列中拿出元素。出队操作类似于入队操作 `queue.dequeue()` 、 `queue.dequeue_many()` 分别出队一个或多个元素，使用。如下：

```
queue = tf.FIFOQueue(3, dtypes=tf.float32)
queue.enqueue_many([[1.], [2.], [3.]])

val = queue.dequeue()
val2 = queue.dequeue_many(2)

with tf.Session() as sess:
    sess.run([val, val2])
```

如果队列中元素的数量不够出队操作所需的数量，则出队操作会阻塞，直到入队操作加入了足够多的元素。

1.2 RandomShuffleQueue

RandomShuffleQueue创建一个随机队列，在执行出队操作时，将以随机的方式拿出元素。当我们需要打乱样本时，就可以使用这种方法。例如对小批量样本执行训练时，我们希望每次取到的小批量的样本都是随机组成的，这样训练的算法更准确，此时使用随机队列就很合适。

使用 `tf.RandomShuffleQueue()` 来创建随机队列，具体如下：

```
tf.RandomShuffleQueue(
    capacity,  # 容量
    min_after_dequeue,  # 指定在出队操作之后最少的元素数量，来保证出队元素的随机性
    dtypes,
    shapes=None,
    names=None,
    seed=None,
    shared_name=None,
    name='random_shuffle_queue')
```

RandomShuffleQueue的出队与入队操作与FIFOQueue一样。

举例：

```
queue = tf.RandomShuffleQueue(10, 2, dtypes=tf.int16, seed=1)

with tf.Session() as sess:
    for i in range(10):
        sess.run(queue.enqueue([i]))
    print('queue size is : %d ' % sess.run(queue.size()))

    for i in range(8):
        print(sess.run(queue.dequeue()))
    print('queue size is : %d ' % sess.run(queue.size()))
    queue.close()
```

执行结果如下：

```
queue size is : 10
7
8
1
3
9
4
2
6
queue size is : 2
```

本例中用到了 `queue.size()` 来获取队列的长度，使用 `queue.close()` 来关闭队列。可以看到顺序入队，得到的是乱序的数据。

2. 线程

在Tensorflow1.2之前的版本中，数据的输入，需要使用队列配合多线程。在之后的版本中将不再推荐使用这个功能，而是使用 `tf.contrib.data` 模块代替。

Tensorflow中的线程调用的是Python的 `threading` 库。并增加了一些新的功能。

2.1 线程协调器

Python中使用多线程，会有一个问题，即多个线程运行时，一个线程只能在运行完毕之后关闭，如果我们在线程中执行了一个死循环，那么线程就不存在运行完毕的状态，那么我们就无法关闭线程。例如我们使用多个线程给队列循环入队数据，那么我们就无法结束这些线程。线程协调器可以用来管理线程，让所有执行的线程停止运行。

线程协调器使用类 `tf.train.Coordinator()` 来管理。

如下：

```
import tensorflow as tf
import threading

# 将要在线程中执行的函数
# 传入一个Coordinator对象
def myloop(coord):
    while not coord.should_stop():
        ...do something...
        if ...some condition...:
            coord.request_stop()
# 创建Coordinator对象
coord = tf.train.Coordinator()
# 创建10个线程
threads = [threading.Thread(target=myLoop, args=(coord,)) for _ in range(10)]
# 开启10个线程
for t in threads:
    t.start()
# 等待线程执行结束
coord.join(threads)
```

上述代码需要注意的是 `myloop` 函数中，根据 `coord.should_stop()` 的状态来决定是否运行循环体，默认情况下 `coord.should_stop()` 返回 `False` 。当某个线程中执行了 `coord.request_stop()` 之后，所有线程在执行循环时，都会因为 `coord.should_stop()` 返回 `True` 而终止执行，从而使所有线程结束。

2.2 队列管理器

上面队列的例子中，出队、入队均是在主线程中完成。这时候会有一个问题：当入队操作速度较慢，例如加载较大的数据时，入队操作不仅影响了主线程的总体速度，也会造成出队操作阻塞，更加拖累了线程的执行速度。我们希望入队操作能够在一些列新线程中执行，主进程仅仅执行出队和训练任务，这样就可以提高整体的运行速度。

如果直接使用Python中的线程进行管理Tensorflow的队列会出一些问题，例如子线程无法操作主线程的队列。在Tensorflow中，队列管理器可以用来创建、运行、管理队列线程。队列管理器可以管理一个或多个队列。

单队列管理

用法如下：

1. 使用 `tf.train.QueueRunner()` 创建一个队列管理器。队列管理器中需要传入队列以及队列将要执行的操作和执行的线程数量。
2. 在会话中开启所有的队列管理器中的线程。
3. 使用Coordinator通知线程结束。

例如：

```
g = tf.Graph()
with g.as_default():
    q = tf.FIFOQueue(100, tf.float32)
    enq_op = q.enqueue(tf.constant(1.))

    # 为队列q创建一个拥有两个线程的队列管理器
    num_threads = 2
    # 第一个参数必须是一个队列，第二个参数必须包含入队操作
    qr = tf.train.QueueRunner(q, [enq_op] * num_threads)

with tf.Session(graph=g) as sess:
    coord = tf.train.Coordinator()
    # 在会话中开启线程并使用coord协调线程
    threads = qr.create_threads(sess, start=True, coord=coord)

    deq_op = q.dequeue()
    for i in range(15):
        print(sess.run(deq_op))

    # 请求关闭线程
    coord.request_stop()
    coord.join(threads)
```

多队列管理

队列管理器还可以管理多个队列，例如可以创建多个单一的队列管理器分别进行操作。但为了统一管理多个队列，简化代码复杂度，Tensorflow可以使用多队列管理。即创建多个队列管理器，统一开启线程和关闭队列。

使用方法如下：

1. 与单队列意义，使用 `tf.train.QueueRunner()` 创建一个队列管理器（也可以使用 `tf.train.queue_runner.QueueRunner()` 创建队列管理器，两个方法一模一样）。一个队列管理器管理一个队列。队列管理器中需要传入将要执行的操作以及执行的线程数量。例如：

   ```
   # 队列1
   q1 = tf.FIFOQueue(100, dtypes=tf.float32)
   enq_op1 = q1.enqueue([1.])
   qr1 = tf.train.QueueRunner(q1, [enq_op1] * 4)

   # 队列2
   q2 = tf.FIFOQueue(50, dtypes=tf.float32)
   enq_op2 = q2.enqueue([2.])
   qr2 = tf.train.QueueRunner(q2, [enq_op2] * 3)
   ```

2. `tf.train.add_queue_runner()` 向图的队列管理器集合中添加一个队列管理器（也可以使用 `tf.train.queue_runner.add_queue_runner()` 添加队列管理器，两个方法一模一样）。一个图可以有多个队列管理器。例如：

```
tf.train.add_queue_runner(qr1)
tf.train.add_queue_runner(qr2)
```

3. `tf.train.start_queue_runners()` 在会话中开启所有的队列管理器（也可以使用 `tf.train.queue_runner.start_queue_runners()` 开启，两个方法一模一样），即开启线程。这时候，我们可以将线程协调器传递给它，用于完成线程执行。例如：

```
with tf.Session() as sess:
   coord = tf.train.Coordinator()
   threads = tf.train.start_queue_runners(sess, coord=coord)
```

4. 任务完成，结束线程。使用Coordinator来完成：

   注意：结束线程的这些操作还用到了Session，应该放在Session上下文中，否则可能出现Session关闭的错误。

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

完整案例：

在图中创建了一个队列，并且使用队列管理器进行管理。创建的队列

```
g = tf.Graph()
with g.as_default():
    # 创建队列q1
    q1 = tf.FIFOQueue(10, tf.int32)
    enq_op1 = q1.enqueue(tf.constant(1))
    qr1 = tf.train.QueueRunner(q1, [enq_op1] * 3)
    tf.train.add_queue_runner(qr1)

    # 创建队列q2
    q2 = tf.FIFOQueue(10, tf.int32)
    enq_op2 = q2.enqueue(tf.constant(2))
    qr2 = tf.train.QueueRunner(q2, [enq_op2] * 3)
    tf.train.add_queue_runner(qr2)

with tf.Session(graph=g) as sess:
    coord = tf.train.Coordinator()
    threads = tf.train.start_queue_runners(sess, coord=coord)

    my_data1 = q1.dequeue()
    my_data2 = q2.dequeue()

    for _ in range(15):
        print(sess.run(my_data1))
        print(sess.run(my_data2))

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

队列中op的执行顺序

队列管理器中会指定入队op，但有些时候入队op执行时，还需要同时执行一些别的操作，例如我们希望按顺序生成一个自然数队列，我们每次入队时，需要对入队的变量加1。然而如下写法输出的结果并没有按照顺序来：

```
q = tf.FIFOQueue(100, tf.float32)
counter = tf.Variable(tf.constant(0.))

assign_op = tf.assign_add(counter, tf.constant(1.))
enq_op = q.enqueue(counter)

# 使用一个线程入队
qr = tf.train.QueueRunner(q, [assign_op, enq_op] * 1)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    qr.create_threads(sess, start=True)

    for i in range(10):
        print(sess.run(q.dequeue()))
# 输出结果
# 1.0
# 3.0
# 4.0
# 4.0
# 5.0
# 7.0
# 9.0
# 10.0
# 12.0
# 12.0
```

可以看到 `assign_op` 与 `enq_op` 的执行是异步的，入队操作并没有严格的在赋值操作之后执行。