# 12. Distributing TensorFlow Across Devices and Servers

## Multiple Devices Across Multiple Servers

To run a graph across multiple servers, you first need to define a *cluster*. A cluster is composed of one or more TensorFlow servers, called *tasks*, typically spread across several machines (see Figure 12-6). Each task belongs to a *job*. A job is just a named group of tasks that typically have a common role, such as keeping track of the model parameters (such a job is usually named `"ps"` for *parameter server*), or performing computations (such a job is usually named `"worker"`).

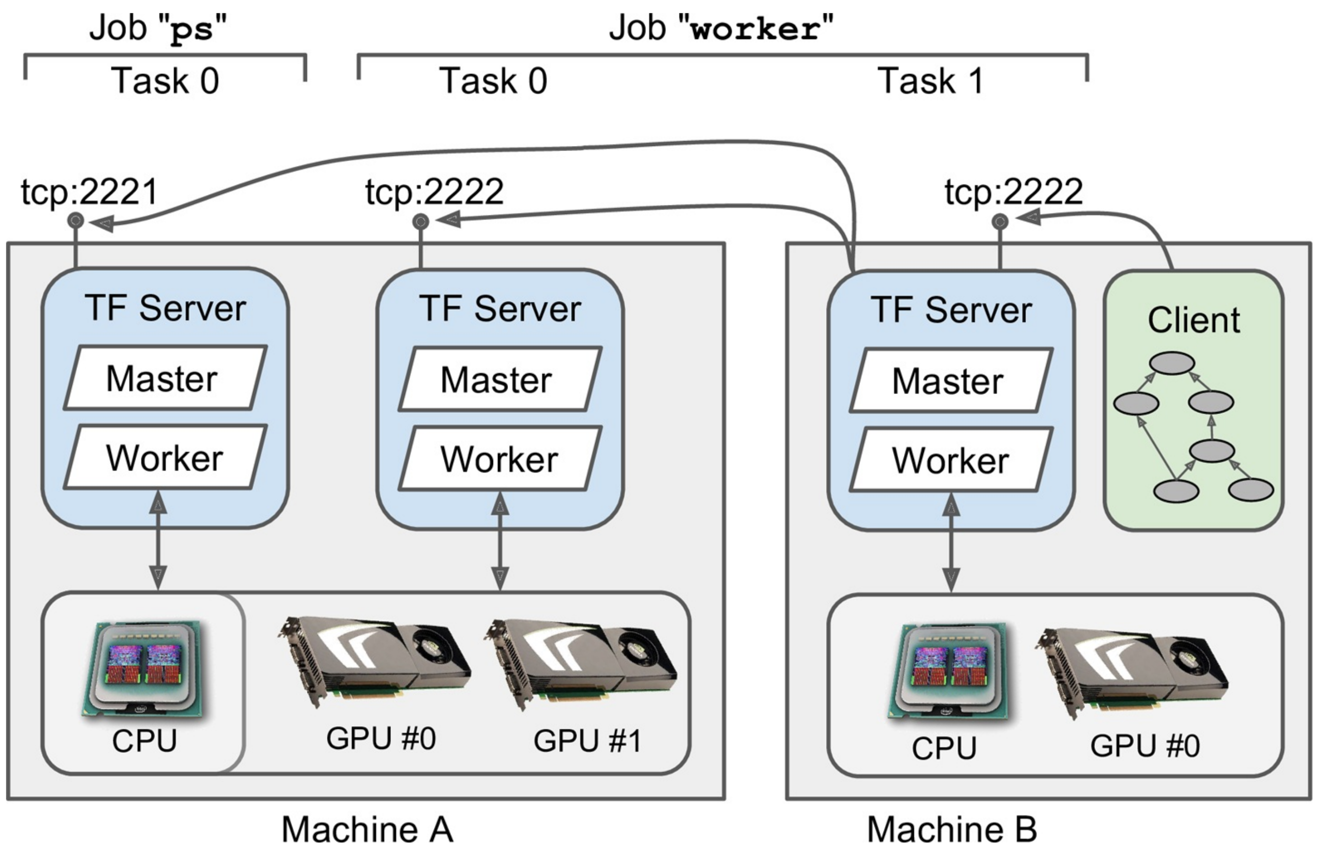

**Figure 12-6. TensorFlow cluster**

The following *cluster specification* defines two jobs, `"ps"` and `"worker"`, containing one task and two tasks, respectively. In this example, machine A hosts two TensorFlow servers (i.e., tasks), listening on different ports: one is part of the `"ps"` job, and the other is part of the `"worker"` job. Machine B just hosts one TensorFlow server, part of the `"worker"` job.

```
cluster_spec = tf.train.ClusterSpec({
    "ps": [
                                       #
        "machine-a.example.com:2221",  /job:ps/task:0
    ],
    "worker": [
        "machine-a.example.com:2222",
#
/job:worker/task:0
        "machine-b.example.com:2222",
#
/job:worker/task:1
    ]})
```

To start a TensorFlow server, you must create a `Server` object, passing it the cluster specification (so it can communicate with other servers) and its own job name and task number. For example, to start the first worker task, you would run the following code on machine A:

```
server = tf.train.Server(cluster_spec, job_name="worker", task_index=0)
```

It is usually simpler to just run one task per machine, but the previous example demonstrates that TensorFlow allows you to run multiple tasks on the same machine if you want. If you have several servers on one machine, you will need to ensure that they don't all try to grab all the RAM of every GPU, as explained earlier. For example, in Figure 12-6 the `"ps"` task does not see the GPU devices, since presumably its process was launched with `CUDA_VISIBLE_DEVICES=""`. Note that the CPU is shared by all tasks located on the same machine.

If you want the process to do nothing other than run the TensorFlow server, you can block the main thread by telling it to wait for the server to finish using the `join()` method (otherwise the server will be killed as soon as your main thread exits). Since there is currently no way to stop the server, this will actually block forever:

```
server.join()
# blocks until the server stops (i.e.,
never)
```

## Opening a Session

Once all the tasks are up and running (doing nothing yet), you can open a session on any of the servers, from a client located in any process on any machine (even from a process running one of the tasks), and use that session like a regular local session. For example:

```
a = tf.constant(1.0)
b = a + 2
c = a * 3

with tf.Session("grpc://machine-b.example.com:2222") as sess:
                         #
    print(c.eval())      9.0
```

This client code first creates a simple graph, then opens a session on the TensorFlow server located on machine B (which we will call the *master*), and instructs it to evaluate `c`. The master starts by placing the operations on the appropriate devices. In this example, since we did not pin any operation on any device, the master simply places them all on its own default device—in this case, machine B's GPU device. Then it just evaluates `c` as instructed by the client, and it returns the result.

## The Master and Worker Services

The client uses the *gRPC* protocol (*Google Remote Procedure Call*) to communicate with the server. This is an efficient open source framework to call remote functions and get their outputs across a variety of platforms and languages. It is based on HTTP2, which opens a connection and leaves it open during the whole session, allowing efficient bidirectional communication once the connection is established. Data is transmitted in the form of *protocol buffers*, another open source Google technology. This is a lightweight binary data interchange format.

**Warning**

All servers in a TensorFlow cluster may communicate with any other server in the cluster, so make sure to open the appropriate ports on your firewall.

Every TensorFlow server provides two services: the *master service* and the *worker service*. The master service allows clients to open sessions and use them to run graphs. It coordinates the computations across tasks, relying on the worker service to actually execute computations on other tasks and get their results.

This architecture gives you a lot of flexibility. One client can connect to multiple servers by opening multiple sessions in different threads. One server can handle multiple sessions simultaneously from one or more clients. You can run one client per task (typically within the same process), or just one client to control all tasks. All options are open.

## Pinning Operations Across Tasks

You can use device blocks to pin operations on any device managed by any task, by specifying the job name, task index, device type, and device index. For example, the following code pins `a` to the CPU of the first task in the `"ps"` job (that's the CPU on machine A), and it pins `b` to the second GPU managed by the first task of the `"worker"` job (that's GPU #1 on machine A). Finally, `c` is not pinned to any device, so the master places it on its own default device (machine B's GPU #0 device).

```
with tf.device("/job:ps/task:0/cpu:0")
    a = tf.constant(1.0)

with tf.device("/job:worker/task:0/gpu:1")
    b = a + 2

c = a + b
```

As earlier, if you omit the device type and index, TensorFlow will default to the task's default device; for example, pinning an operation to `"/job:ps/task:0"` will place it on the default device of the first task of the `"ps"` job (machine A's CPU). If you also omit the task index (e.g., `"/job:ps"`), TensorFlow defaults to `"/task:0"`. If you omit the job name and the task index, TensorFlow defaults to the session's master task.

## Sharding Variables Across Multiple Parameter Servers

As we will see shortly, a common pattern when training a neural network on a distributed setup is to store the model parameters on a set of parameter servers (i.e., the tasks in the `"ps"` job) while other tasks focus on computations (i.e., the tasks in the "worker" job). For large models with millions of parameters, it is useful to shard these parameters across multiple parameter servers, to reduce the risk of saturating a single parameter server's network card. If you were to manually pin every variable to a different parameter server, it would be quite tedious. Fortunately, TensorFlow provides the replica_device_setter() function, which distributes variables across all the "ps" tasks in a round-robin fashion. For example, the following code pins five variables to two parameter servers:

```
with tf.device(tf.train.replica_device_setter(ps_tasks=2):
                              # pinned to
    v1 = tf.Variable(1.0)     /job:ps/task:0
                              # pinned to
    v2 = tf.Variable(2.0)     /job:ps/task:1
                              # pinned to
    v3 = tf.Variable(3.0)     /job:ps/task:0
                              # pinned to
    v4 = tf.Variable(4.0)     /job:ps/task:1
                              # pinned to
    v5 = tf.Variable(5.0)     /job:ps/task:0
```

Instead of passing the number of ps_tasks, you can pass the cluster spec cluster=cluster_spec and TensorFlow will simply count the number of tasks in the "ps" job.

If you create other operations in the block, beyond just variables, TensorFlow automatically pins them to "/job:worker", which will default to the first device managed by the first task in the "worker" job. You can pin them to another device by setting the worker_device parameter, but a better approach is to use embedded device blocks. An inner device block can override the job, task, or device defined in an outer block. For example:

```
with tf.device(tf.train.replica_device_setter(ps_tasks=2)):
                              # pinned to /job:ps/task:0 (+ defaults to
    v1 = tf.Variable(1.0)     /cpu:0)
                              # pinned to /job:ps/task:1 (+ defaults to
    v2 = tf.Variable(2.0)     /cpu:0)
                              # pinned to /job:ps/task:0 (+ defaults to
    v3 = tf.Variable(3.0)     /cpu:0)
    [...]
    s = v1 + v2
# pinned to /job:worker (+ defaults to
task:0/gpu:0)
    with tf.device("/gpu:1"):
        p1 = 2 * s
# pinned to /job:worker/gpu:1 (+ defaults to
/task:0)
        with tf.device("/task:1"):
                              # pinned to
            p2 = 3 * s        /job:worker/task:1/gpu:1
```

**Note**

This example assumes that the parameter servers are CPU-only, which is typically the case since they only need to store and communicate parameters, not perform intensive computations.

## Sharing State Across Sessions Using Resource Containers

When you are using a plain *local session* (not the distributed kind), each variable's state is managed by the session itself; as soon as it ends, all variable values are lost. Moreover, multiple local sessions cannot share any state, even if they both run the same graph; each session has its own copy of every variable (as we discussed in Chapter 9). In contrast, when you are using *distributed sessions*, variable state is managed by *resource containers* located on the cluster itself, not by the sessions. So if you create a variable named `x` using one client session, it will automatically be available to any other session on the same cluster (even if both sessions are connected to a different server). For example, consider the following client code:

```
#
simple_client.py
import tensorflow as tf
import sys

x = tf.Variable(0.0, name="x")
increment_x = tf.assign(x, x + 1)

with tf.Session(sys.argv[1]) as sess:
    if sys.argv[2:]==["init"]:
        sess.run(x.initializer)
    sess.run(increment_x)
    print(x.eval())
```

Let's suppose you have a TensorFlow cluster up and running on machines A and B, port 2222. You could launch the client, have it open a session with the server on machine A, and tell it to initialize the variable, increment it, and print its value by launching the following command:

```
$ python3 simple_client.py grpc://machine-a.example.com:2222
init
1.0
```

Now if you launch the client with the following command, it will connect to the server on machine B and magically reuse the same variable `x` (this time we don't ask the server to initialize the variable):

```
$ python3 simple_client.py grpc://machine-
b.example.com:2222
2.0
```

This feature cuts both ways: it's great if you want to share variables across multiple sessions, but if you want to run completely independent computations on the same cluster you will have to be careful not to use the same variable names by accident. One way to ensure that you won't have name clashes is to wrap all of your construction phase inside a variable scope with a unique name for each computation, for example:

```
with tf.variable_scope("my_problem_1"):
    [...]
# Construction phase of problem
1
```

A better option is to use a container block:

```
with tf.container("my_problem_1"):
    [...]
# Construction phase of problem
1
```

This will use a container dedicated to problem #1, instead of the default one (whose name is an empty string `""`). One advantage is that variable names remain nice and short. Another advantage is that you can easily reset a named container. For example, the following command will connect to the server on machine A and ask it to reset the container named `"my_problem_1"`, which will free all the resources this container used (and also close all sessions open on the server). Any variable managed by this container must be initialized before you can use it again:

```
tf.Session.reset("grpc://machine-a.example.com:2222", ["my_problem_1"])
```

Resource containers make it easy to share variables across sessions in flexible ways. For example, Figure 12-7 shows four clients running different graphs on the same cluster, but sharing some variables. Clients A and B share the same variable `x` managed by the default container, while clients C and D share another variable named `x` managed by the container named `"my_problem_1"`. Note that client C even uses variables from both containers.

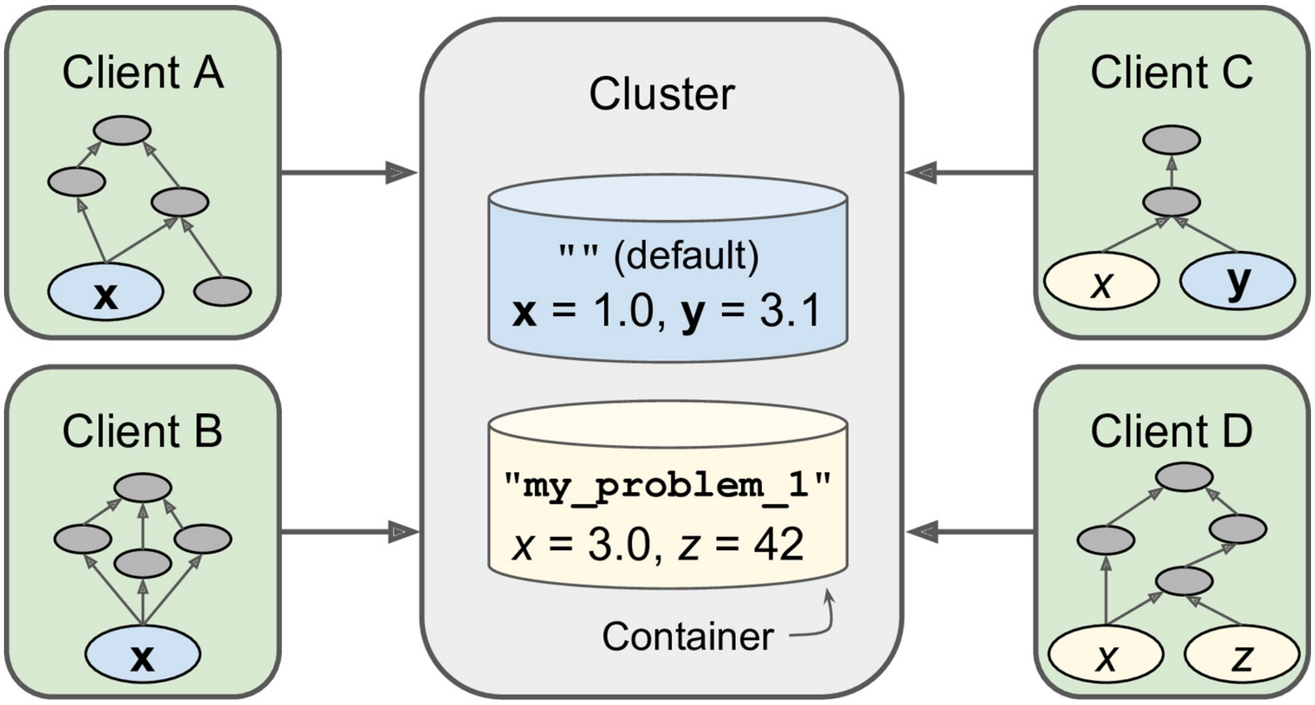

**Figure 12-7. Resource containers**

Resource containers also take care of preserving the state of other stateful operations, namely queues and readers. Let's take a look at queues first.

## Asynchronous Communication Using TensorFlow Queues

Queues are another great way to exchange data between multiple sessions; for example, one common use case is to have a client create a graph that loads the training data and pushes it into a queue, while another client creates a graph that pulls the data from the queue and trains a model (see Figure 12-8). This can speed up training considerably because the training operations don't have to wait for the next mini-batch at every step.

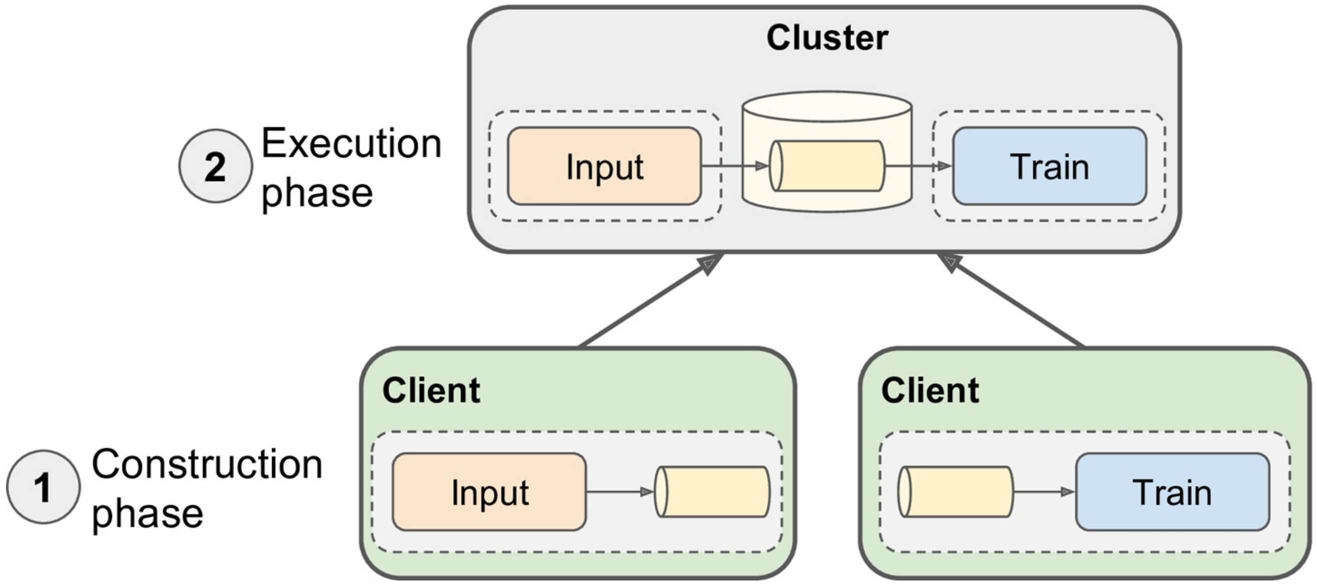

**Figure 12-8. Using queues to load the training data asynchronously**

TensorFlow provides various kinds of queues. The simplest kind is the *first-in first-out* (*FIFO*) queue. For example, the following code creates a FIFO queue that can store up to 10 tensors containing two float values each:

```
q = tf.FIFOQueue(capacity=10, dtypes=[tf.float32], shapes=[[2]],
                 name="q", shared_name="shared_q")
```

**Warning**

To share variables across sessions, all you had to do was to specify the same name and container on both ends. With queues TensorFlow does not use the `name` attribute but instead uses `shared_name`, so it is important to specify it (even if it is the same as the `name`). And, of course, use the same container.

### Enqueuing data

To push data to a queue, you must create an `enqueue` operation. For example, the following code pushes three training instances to the queue:

```
#
training_data_loader.py
import tensorflow as tf

q = [...]
training_instance = tf.placeholder(tf.float32, shape=(2))
enqueue = q.enqueue([training_instance])

with tf.Session("grpc://machine-a.example.com:2222") as sess:
   sess.run(enqueue, feed_dict={training_instance: [1., 2.]})
   sess.run(enqueue, feed_dict={training_instance: [3., 4.]})
   sess.run(enqueue, feed_dict={training_instance: [5., 6.]})
```

Instead of enqueuing instances one by one, you can enqueue several at a time using an `enqueue_many` operation:

```
[...]
training_instances = tf.placeholder(tf.float32, shape=(None, 2))
enqueue_many = q.enqueue([training_instances])

with tf.Session("grpc://machine-a.example.com:2222") as sess:
   sess.run(enqueue_many,
            feed_dict={training_instances: [[1., 2.], [3., 4.], [5., 6.
]]})
```

Both examples enqueue the same three tensors to the queue.

## Dequeuing data

To pull the instances out of the queue, on the other end, you need to use a `dequeue` operation:

```
#
trainer.py
import tensorflow as tf

q = [...]
dequeue = q.dequeue()

with tf.Session("grpc://machine-a.example.com:2222") as sess:
                              # [1.,
   print(sess.run(dequeue))   2.]
                              # [3.,
   print(sess.run(dequeue))   4.]
                              # [5.,
   print(sess.run(dequeue))   6.]
```

In general you will want to pull a whole mini-batch at once, instead of pulling just one instance at a time. To do so, you must use a `dequeue_many` operation, specifying the mini-batch size:

```
[...]
batch_size = 2
dequeue_mini_batch= q.dequeue_many(batch_size)

with tf.Session("grpc://machine-a.example.com:2222") as sess:
                                          # [[1., 2.], [4.,
   print(sess.run(dequeue_mini_batch))   5.]]
   print(sess.run(dequeue_mini_batch))
# blocked waiting for another
instance
```

When a queue is full, the enqueue operation will block until items are pulled out by a dequeue operation. Similarly, when a queue is empty (or you are using `dequeue_many()` and there are fewer items than the mini-batch size), the dequeue operation will block until enough items are pushed into the queue using an enqueue operation.

## Queues of tuples

Each item in a queue can be a tuple of tensors (of various types and shapes) instead of just a single tensor. For example, the following queue stores pairs of tensors, one of type `int32` and shape `()`, and the other of type `float32` and shape `[3,2]`:

```
q = tf.FIFOQueue(capacity=10, dtypes=[tf.int32, tf.float32], shapes=[[],[3,2]],
                 name="q", shared_name="shared_q")
```

The enqueue operation must be given pairs of tensors (note that each pair represents only one item in the queue):

```
a = tf.placeholder(tf.int32, shape=())
b = tf.placeholder(tf.float32, shape=(3, 2))
enqueue = q.enqueue((a, b))

with tf.Session([...]) as sess:
    sess.run(enqueue, feed_dict={a: 10, b:[[1., 2.], [3., 4.], [5., 6.
]]})
    sess.run(enqueue, feed_dict={a: 11, b:[[2., 4.], [6., 8.], [0., 2.
]]})
    sess.run(enqueue, feed_dict={a: 12, b:[[3., 6.], [9., 2.], [5., 8.
]]})
```

On the other end, the `dequeue()` function now creates a pair of dequeue operations:

```
dequeue_a, dequeue_b = q.dequeue()
```

In general, you should run these operations together:

```
with tf.Session([...]) as sess:
    a_val, b_val = sess.run([dequeue_a, dequeue_b
])
                 #
    print(a_val) 10
                 # [[1., 2.], [3., 4.], [5.,
    print(b_val) 6.]]
```

**Warning**

If you run `dequeue_a` on its own, it will dequeue a pair and return only the first element; the second element will be lost (and similarly, if you run `dequeue_b` on its own, the first element will be lost).

The `dequeue_many()` function also returns a pair of operations:

```
batch_size = 2
dequeue_as, dequeue_bs = q.dequeue_many(batch_size)
```

You can use it as you would expect:

```
with tf.Session([...]) as sess:
    a, b = sess.run([dequeue_a, dequeue_b])
             # [10,
    print(a) 11]
    print(b)
# [[[1., 2.], [3., 4.], [5., 6.]], [[2., 4.], [6., 8.], [0.,
2.]]]
    a, b = sess.run([dequeue_a, dequeue_b])
# blocked waiting for another
pair
```

### Closing a queue

It is possible to close a queue to signal to the other sessions that no more data will be enqueued:

```
close_q = q.close()

with tf.Session([...]) as sess:
    [...]
    sess.run(close_q)
```

Subsequent executions of `enqueue` or `enqueue_many` operations will raise an exception. By default, any pending enqueue request will be honored, unless you call `q.close(cancel_pending_enqueues=True)`.

Subsequent executions of `dequeue` or `dequeue_many` operations will continue to succeed as long as there are items in the queue, but they will fail when there are not enough items left in the queue. If you are using a `dequeue_many` operation and there are a few instances left in the queue, but fewer than the mini-batch size, they will be lost. You may prefer to use a `dequeue_up_to` operation instead; it behaves exactly like `dequeue_many` except when a queue is closed and there are fewer than `batch_size` instances left in the queue, in which case it just returns them.

### RandomShuffleQueue

TensorFlow also supports a couple more types of queues, including `RandomShuffleQueue`, which can be used just like a `FIFOQueue` except that items are dequeued in a random order. This can be useful to shuffle training instances at each epoch during training. First, let's create the queue:

```
q = tf.RandomShuffleQueue(capacity=50, min_after_dequeue=10,
                           dtypes=[tf.float32], shapes=
[()],
                           name="q", shared_name="shared_q"
)
```

The `min_after_dequeue` specifies the minimum number of items that must remain in the queue after a dequeue operation. This ensures that there will be enough instances in the queue to have enough randomness (once the queue is closed, the `min_after_dequeue` limit is ignored). Now suppose that you enqueued 22 items in this queue (floats `1.` to `22.`). Here is how you could dequeue them:

```
dequeue = q.dequeue_many(5)

with tf.Session([...]) as sess:
                             # [ 20.  15.  11.  12.   4.]   (17 items
   print(sess.run(dequeue)) left)
                             # [  5.  13.   6.   0.  17.]   (12 items
   print(sess.run(dequeue)) left)
   print(sess.run(dequeue))
# 12 - 5 < 10: blocked waiting for 3 more
instances
```

### PaddingFifoQueue

A `PaddingFIFOQueue` can also be used just like a `FIFOQueue` except that it accepts tensors of variable sizes along any dimension (but with a fixed rank). When you are dequeuing them with a `dequeue_many` or `dequeue_up_to` operation, each tensor is padded with zeros along every variable dimension to make it the same size as the largest tensor in the mini-batch. For example, you could enqueue 2D tensors (matrices) of arbitrary sizes:

```
q = tf.PaddingFIFOQueue(capacity=50, dtypes=[tf.float32], shapes=[(None, None)]
                        name="q", shared_name="shared_q")
v = tf.placeholder(tf.float32, shape=(None, None))
enqueue = q.enqueue([v])

with tf.Session([...]) as sess:
   sess.run(enqueue, feed_dict={v: [[1., 2.], [3., 4.], [5., 6.]]})
#
3x2
   sess.run(enqueue, feed_dict={v: [[1.]]})
#
1x1
                                                                           #
   sess.run(enqueue, feed_dict={v: [[7., 8., 9., 5.], [6., 7., 8., 9.]]}) 2x4
```

If we just dequeue one item at a time, we get the exact same tensors that were enqueued. But if we dequeue several items at a time (using `dequeue_many()` or `dequeue_up_to()`), the queue automatically pads the tensors appropriately. For example, if we dequeue all three items at once, all tensors will be padded with zeros to become 3 × 4 tensors, since the maximum size for the first dimension is 3 (first item) and the maximum size for the second dimension is 4 (third item):

```
>>> q = [...]
>>> dequeue = q.dequeue_many(3)
>>> with tf.Session([...]) as sess:
...     print(sess.run(dequeue))
[[[ 1.  2.  0.
0.]
  [ 3.  4.  0.
0.]
  [ 5.  6.  0.
0.]]

 [[ 1.  0.  0.
0.]
  [ 0.  0.  0.
0.]
  [ 0.  0.  0.
0.]]

 [[ 7.  8.  9.
5.]
  [ 6.  7.  8.
9.]
  [ 0.  0.  0.
0.]]]
```

This type of queue can be useful when you are dealing with variable length inputs, such as sequences of words (see Chapter 14).

Okay, now let's pause for a second: so far you have learned to distribute computations across multiple devices and servers, share variables across sessions, and communicate asynchronously using queues. Before you start training neural networks, though, there's one last topic we need to discuss: how to efficiently load training data.

## Loading Data Directly from the Graph

So far we have assumed that the clients would load the training data and feed it to the cluster using placeholders. This is simple and works quite well for simple setups, but it is rather inefficient since it transfers the training data several times:

1. From the filesystem to the client
2. From the client to the master task
3. Possibly from the master task to other tasks where the data is needed

It gets worse if you have several clients training various neural networks using the same training data (for example, for hyperparameter tuning): if every client loads the data simultaneously, you may end up even saturating your file server or the network's bandwidth.

### Preload the data into a variable

For datasets that can fit in memory, a better option is to load the training data once and assign it to a variable, then just use that variable in your graph. This is called *preloading* the training set. This way the data will be transferred only once from the client to the cluster (but it may still need to be moved around from task to task depending on which operations need it). The following code shows how to load the full training set into a variable:

```
training_set_init = tf.placeholder(tf.float32, shape=(None, n_features))
training_set = tf.Variable(training_set_init, trainable=False, collections=[],
                           name="training_set")

with tf.Session([...]) as sess:
                  # load the training data from the
    data = [...]  datastore
    sess.run(training_set.initializer, feed_dict={training_set_init: data})
```

You must set `trainable=False` so the optimizers don’t try to tweak this variable. You should also set `collections=[]` to ensure that this variable won’t get added to the `GraphKeys.GLOBAL_VARIABLES` collection, which is used for saving and restoring checkpoints.

**Note**

This example assumes that all of your training set (including the labels) consists only of `float32` values. If that’s not the case, you will need one variable per type.

## Reading the training data directly from the graph

If the training set does not fit in memory, a good solution is to use *reader operations*: these are operations capable of reading data directly from the filesystem. This way the training data never needs to flow through the clients at all. TensorFlow provides readers for various file formats:

- CSV
- Fixed-length binary records
- TensorFlow’s own `TFRecords` format, based on protocol buffers

Let’s look at a simple example reading from a CSV file (for other formats, please check out the API documentation). Suppose you have file named *my_test.csv* that contains training instances, and you want to create operations to read it. Suppose it has the following content, with two float features `x1` and `x2` and one integer `target` representing a binary class:

```
x1,  x2,
target
1. , 2. , 0
4. , 5  , 1
7. ,    , 0
```

First, let’s create a `TextLineReader` to read this file. A `TextLineReader` opens a file (once we tell it which one to open) and reads lines one by one. It is a stateful operation, like variables and queues: it preserves its state across multiple runs of the graph, keeping track of which file it is currently reading and what its current position is in this file.

```
reader = tf.TextLineReader(skip_header_lines=1)
```

Next, we create a queue that the reader will pull from to know which file to read next. We also create an enqueue operation and a placeholder to push any filename we want to the queue, and we create an operation to close the queue once we have no more files to read:

```
filename_queue = tf.FIFOQueue(capacity=10, dtypes=[tf.string], shapes=[()])
filename = tf.placeholder(tf.string)
enqueue_filename = filename_queue.enqueue([filename])
close_filename_queue = filename_queue.close()
```

Now we are ready to create a `read` operation that will read one record (i.e., a line) at a time and return a key/value pair. The key is the record's unique identifier—a string composed of the filename, a colon (`:`), and the line number—and the value is simply a string containing the content of the line:

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

We have all we need to read the file line by line! But we are not quite done yet—we need to parse this string to get the features and target:

```
x1, x2, target = tf.decode_csv(value, record_defaults=[[-1.], [-1.], [-1]])
features = tf.stack([x1, x2])
```

The first line uses TensorFlow's CSV parser to extract the values from the current line. The default values are used when a field is missing (in this example the third training instance's `x2` feature), and they are also used to determine the type of each field (in this case two floats and one integer).

Finally, we can push this training instance and its target to a `RandomShuffleQueue` that we will share with the training graph (so it can pull mini-batches from it), and we create an operation to close that queue when we are done pushing instances to it:

```
instance_queue = tf.RandomShuffleQueue(
    capacity=10, min_after_dequeue=2,
    dtypes=[tf.float32, tf.int32], shapes=[[2],[]],
    name="instance_q", shared_name="shared_instance_q")
enqueue_instance = instance_queue.enqueue([features, target])
close_instance_queue = instance_queue.close()
```

Wow! That was a lot of work just to read a file. Plus we only created the graph, so now we need to run it:

```
with tf.Session([...]) as sess:
    sess.run(enqueue_filename, feed_dict={filename: "my_test.csv"})
    sess.run(close_filename_queue)
    try:
        while True:
            sess.run(enqueue_instance)
    except tf.errors.OutOfRangeError as ex:
        pass
# no more records in the current file and no more files to
read
    sess.run(close_instance_queue)
```

First we open the session, and then we enqueue the filename `"my_test.csv"` and immediately close that queue since we will not enqueue any more filenames. Then we run an infinite loop to enqueue instances one by one. The `enqueue_instance` depends on the reader reading the next line, so at every iteration a new record is read until it reaches the end of the file. At that point it tries to read the filename queue to know which file to read next, and since the queue is closed it throws an `OutOfRangeError` exception (if we did not close the queue, it would just remain blocked until we pushed another filename or closed the queue). Lastly, we close the instance queue so that the training operations pulling from it won't get blocked forever. Figure 12-9 summarizes what we have learned; it represents a typical graph for reading training instances from a set of CSV files.

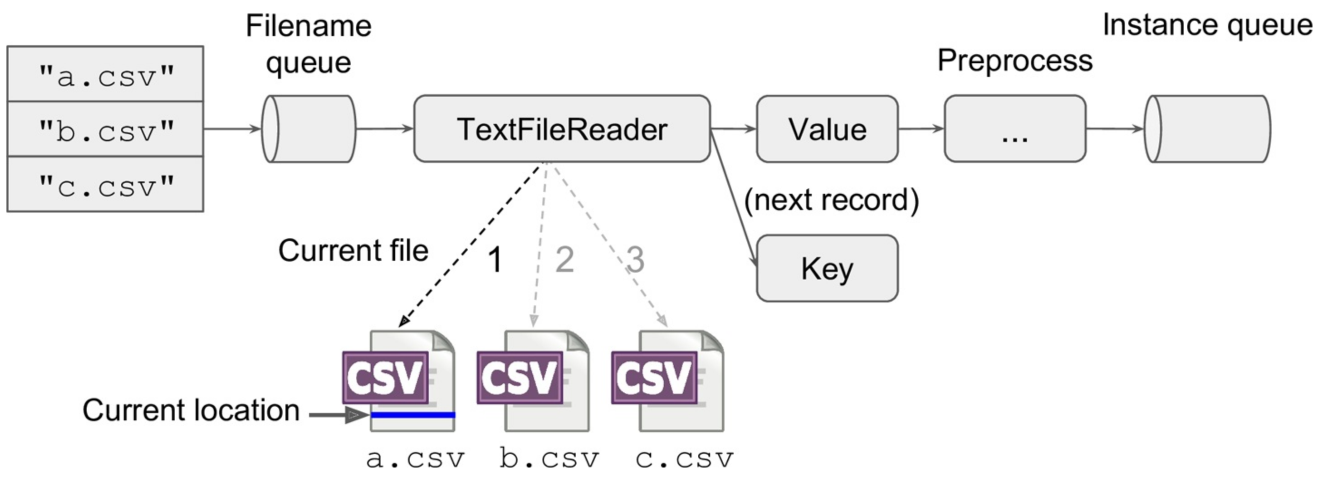

**Figure 12-9. A graph dedicated to reading training instances from CSV files**

In the training graph, you need to create the shared instance queue and simply dequeue mini-batches from it:

```
instance_queue = tf.RandomShuffleQueue([...], shared_name="shared_instance_q")
mini_batch_instances, mini_batch_targets = instance_queue.dequeue_up_to(2)
      # use the mini_batch instances and targets to build the training
[...] graph
training_op = [...]

with tf.Session([...]) as sess:
    try:
        for step in range(max_steps):
            sess.run(training_op)
    except tf.errors.OutOfRangeError as ex:
             # no more training
        pass instances
```

In this example, the first mini-batch will contain the first two instances of the CSV file, and the second mini-batch will contain the last instance.

**Warning**

TensorFlow queues don't handle sparse tensors well, so if your training instances are sparse you should parse the records after the instance queue.

This architecture will only use one thread to read records and push them to the instance queue. You can get a much higher throughput by having multiple threads read simultaneously from multiple files using multiple readers. Let's see how.

## Multithreaded readers using a Coordinator and a QueueRunner

To have multiple threads read instances simultaneously, you could create Python threads (using the `threading` module) and manage them yourself. However, TensorFlow provides some tools to make this simpler: the `Coordinator` class and the `QueueRunner` class.

A `Coordinator` is a very simple object whose sole purpose is to coordinate stopping multiple threads. First you create a `Coordinator`:

```
coord = tf.train.Coordinator()
```

Then you give it to all threads that need to stop jointly, and their main loop looks like this:

```
while not coord.should_stop():
            # do
     [...] something
```

Any thread can request that every thread stop by calling the `Coordinator`'s `request_stop()` method:

```
coord.request_stop()
```

Every thread will stop as soon as it finishes its current iteration. You can wait for all of the threads to finish by calling the `Coordinator`'s `join()` method, passing it the list of threads:

```
coord.join(list_of_threads)
```

A `QueueRunner` runs multiple threads that each run an enqueue operation repeatedly, filling up a queue as fast as possible. As soon as the queue is closed, the next thread that tries to push an item to the queue will get an `OutOfRangeError`; this thread catches the error and immediately tells other threads to stop using a `Coordinator` . The following code shows how you can use a `QueueRunner` to have five threads reading instances simultaneously and pushing them to an instance queue:

```
    # same construction phase as
[...] earlier
queue_runner = tf.train.QueueRunner(instance_queue, [enqueue_instance] * 5)

with tf.Session() as sess:
    sess.run(enqueue_filename, feed_dict={filename: "my_test.csv"})
    sess.run(close_filename_queue)
    coord = tf.train.Coordinator()
    enqueue_threads = queue_runner.create_threads(sess, coord=coord, start=True
)
```

The first line creates the `QueueRunner` and tells it to run five threads, all running the same `enqueue_instance` operation repeatedly. Then we start a session and we enqueue the name of the files to read (in this case just `"my_test.csv"`). Next we create a `Coordinator` that the `QueueRunner` will use to stop gracefully, as just explained. Finally, we tell the `QueueRunner` to create the threads and start them. The threads will read all training instances and push them to the instance queue, and then they will all stop gracefully.

This will be a bit more efficient than earlier, but we can do better. Currently all threads are reading from the same file. We can make them read simultaneously from separate files instead (assuming the training data is sharded across multiple CSV files) by creating multiple readers (see Figure 12-10).

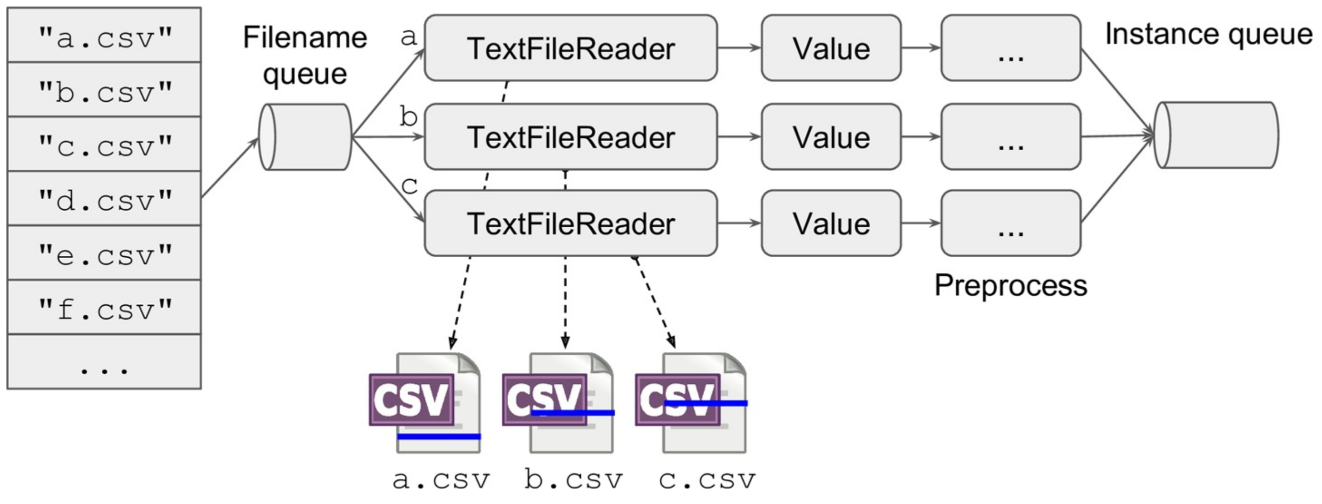

**Figure 12-10. Reading simultaneously from multiple files**

For this we need to write a small function to create a reader and the nodes that will read and push one instance to the instance queue:

```
def read_and_push_instance(filename_queue, instance_queue):
    reader = tf.TextLineReader(skip_header_lines=1)
    key, value = reader.read(filename_queue)
    x1, x2, target = tf.decode_csv(value, record_defaults=[[-1.], [-1.], [-1
]])
    features = tf.stack([x1, x2])
    enqueue_instance = instance_queue.enqueue([features, target])
    return enqueue_instance
```

Next we define the queues:

```
filename_queue = tf.FIFOQueue(capacity=10, dtypes=[tf.string], shapes=[()])
filename = tf.placeholder(tf.string)
enqueue_filename = filename_queue.enqueue([filename])
close_filename_queue = filename_queue.close()

instance_queue = tf.RandomShuffleQueue([...])
```

And finally we create the `QueueRunner`, but this time we give it a list of different enqueue operations. Each operation will use a different reader, so the threads will simultaneously read from different files:

```
read_and_enqueue_ops = [
    read_and_push_instance(filename_queue, instance_queue)
    for i in range(5)]
queue_runner = tf.train.QueueRunner(instance_queue, read_and_enqueue_ops)
```

The execution phase is then the same as before: first push the names of the files to read, then create a `Coordinator` and create and start the `QueueRunner` threads. This time all threads will read from different files simultaneously until all files are read entirely, and then the `QueueRunner` will close the instance queue so that other ops pulling from it don't get blocked.

## Other convenience functions

TensorFlow also offers a few convenience functions to simplify some common tasks when reading training instances. We will go over just a few (see the API documentation for the full list).

The `string_input_producer()` takes a 1D tensor containing a list of filenames, creates a thread that pushes one filename at a time to the filename queue, and then closes the queue. If you specify a number of epochs, it will cycle through the filenames once per epoch before closing the queue. By default, it shuffles the filenames at each epoch. It creates a `QueueRunner` to manage its thread, and adds it to the `GraphKeys.QUEUE_RUNNERS` collection. To start every `QueueRunner` in that collection, you can call the `tf.train.start_queue_runners()` function. Note that if you forget to start the `QueueRunner`, the filename queue will be open and empty, and your readers will be blocked forever.

There are a few other *producer* functions that similarly create a queue and a corresponding `QueueRunner` for running an enqueue operation (e.g., `input_producer()`, `range_input_producer()`, and `slice_input_producer()`).

The `shuffle_batch()` function takes a list of tensors (e.g., `[features, target]`) and creates:

- A `RandomShuffleQueue`
- A `QueueRunner` to enqueue the tensors to the queue (added to the `GraphKeys.QUEUE_RUNNERS` collection)
- A `dequeue_many` operation to extract a mini-batch from the queue

This makes it easy to manage in a single process a multithreaded input pipeline feeding a queue and a training pipeline reading mini-batches from that queue. Also check out the `batch()`, `batch_join()`, and `shuffle_batch_join()` functions that provide similar functionality.

Okay! You now have all the tools you need to start training and running neural networks efficiently across multiple devices and servers on a TensorFlow cluster. Let's review what you have learned:

- Using multiple GPU devices
- Setting up and starting a TensorFlow cluster
- Distributing computations across multiple devices and servers
- Sharing variables (and other stateful ops such as queues and readers) across sessions using containers
- Coordinating multiple graphs working asynchronously using queues
- Reading inputs efficiently using readers, queue runners, and coordinators

Now let’s use all of this to parallelize neural networks!