Control Dependencies

In some cases, it may be wise to postpone the evaluation of an operation even though all the operations it depends on have been executed. For example, if it uses a lot of memory but its value is needed only much further in the graph, it would be best to evaluate it at the last moment to avoid needlessly occupying RAM that other operations may need. Another example is a set of operations that depend on data located outside of the device. If they all run at the same time, they may saturate the device's communication bandwidth, and they will end up all waiting on I/O. Other operations that need to communicate data will also be blocked. It would be preferable to execute these communication-heavy operations sequentially, allowing the device to perform other operations in parallel.

To postpone evaluation of some nodes, a simple solution is to add *control dependencies*. For example, the following code tells TensorFlow to evaluate `x` and `y` only after `a` and `b` have been evaluated:

```
a = tf.constant(1.0)
b = a + 2.0

with tf.control_dependencies([a, b]):
    x = tf.constant(3.0)
    y = tf.constant(4.0)

z = x + y
```

Obviously, since `z` depends on `x` and `y`, evaluating `z` also implies waiting for `a` and `b` to be evaluated, even though it is not explicitly in the `control_dependencies()` block. Also, since `b` depends on `a`, we could simplify the preceding code by just creating a control dependency on `[b]` instead of `[a, b]`, but in some cases "explicit is better than implicit."

Great! Now you know:

- How to place operations on multiple devices in any way you please
- How these operations get executed in parallel
- How to create control dependencies to optimize parallel execution

It's time to distribute computations across multiple 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").

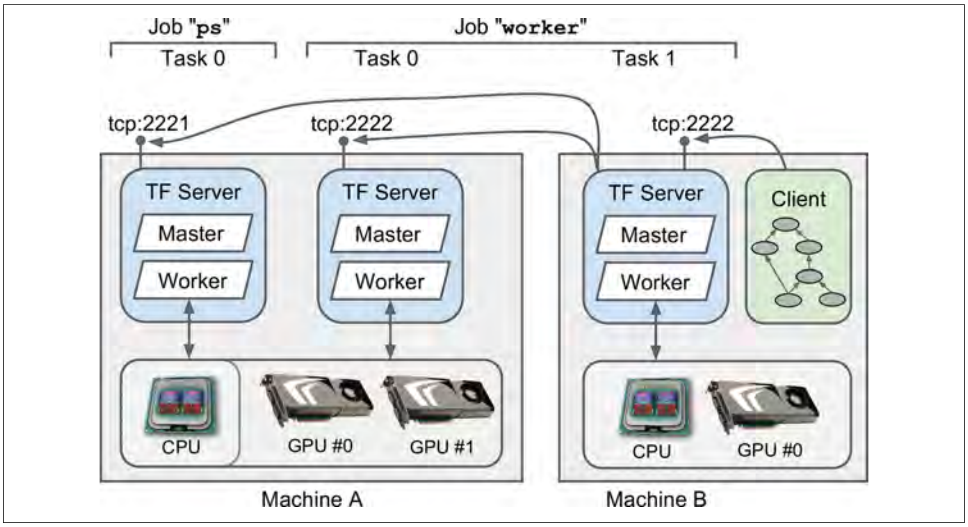

Figure 12-6. TensorFlow cluster

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.[2] 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.[3] 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.

2 You can even start multiple tasks in the same process. It may be useful for tests, but it is not recommended in production.

3 It is the next version of Google's internal *Stubby* service, which Google has used successfully for over a decade. See *http://grpc.io/* for more details.