CHAPTER 12
Distributing TensorFlow Across Devices and Servers

In Chapter 11 we discussed several techniques that can considerably speed up training: better weight initialization, Batch Normalization, sophisticated optimizers, and so on. However, even with all of these techniques, training a large neural network on a single machine with a single CPU can take days or even weeks.

In this chapter we will see how to use TensorFlow to distribute computations across multiple devices (CPUs and GPUs) and run them in parallel (see Figure 12-1). First we will distribute computations across multiple devices on just one machine, then on multiple devices across multiple machines.

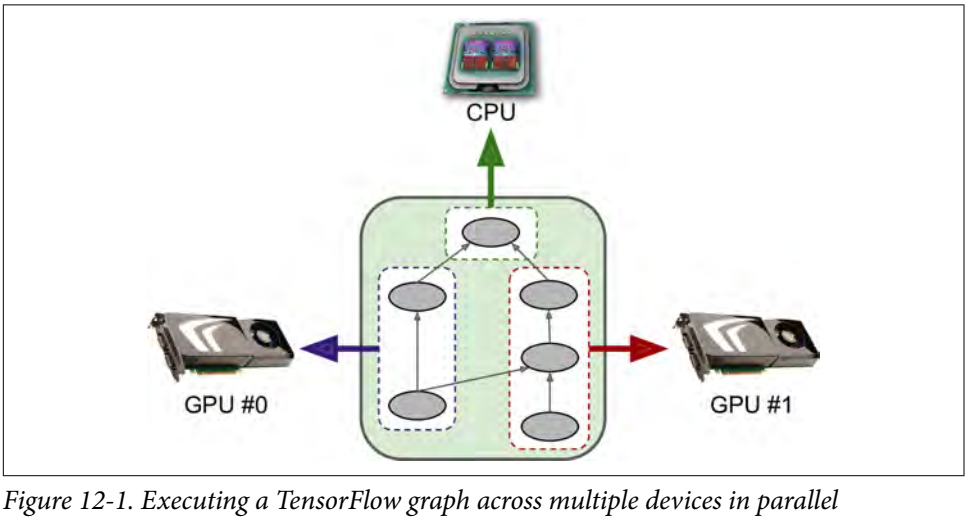

Figure 12-1. Executing a TensorFlow graph across multiple devices in parallel

TensorFlow's support of distributed computing is one of its main highlights compared to other neural network frameworks. It gives you full control over how to split (or replicate) your computation graph across devices and servers, and it lets you parallelize and synchronize operations in flexible ways so you can choose between all sorts of parallelization approaches.

We will look at some of the most popular approaches to parallelizing the execution and training of a neural network. Instead of waiting for weeks for a training algorithm to complete, you may end up waiting for just a few hours. Not only does this save an enormous amount of time, it also means that you can experiment with various models much more easily, and frequently retrain your models on fresh data.

Other great use cases of parallelization include exploring a much larger hyperparameter space when fine-tuning your model, and running large ensembles of neural networks efficiently.

But we must learn to walk before we can run. Let's start by parallelizing simple graphs across several GPUs on a single machine.

Multiple Devices on a Single Machine

You can often get a major performance boost simply by adding GPU cards to a single machine. In fact, in many cases this will suffice; you won't need to use multiple machines at all. For example, you can typically train a neural network just as fast using 8 GPUs on a single machine rather than 16 GPUs across multiple machines (due to the extra delay imposed by network communications in a multimachine setup).

In this section we will look at how to set up your environment so that TensorFlow can use multiple GPU cards on one machine. Then we will look at how you can distribute operations across available devices and execute them in parallel.

Installation

In order to run TensorFlow on multiple GPU cards, you first need to make sure your GPU cards have NVidia Compute Capability (greater or equal to 3.0). This includes Nvidia's Titan, Titan X, K20, and K40 cards (if you own another card, you can check its compatibility at *https://developer.nvidia.com/cuda-gpus*).

If you don't own any GPU cards, you can use a hosting service with GPU capability such as Amazon AWS. Detailed instructions to set up TensorFlow 0.9 with Python 3.5 on an Amazon AWS GPU instance are available in Žiga Avsec's helpful blog post. It should not be too hard to update it to the latest version of TensorFlow. Google also released a cloud service called *Cloud Machine Learning* to run TensorFlow graphs. In May 2016, they announced that their platform now includes servers equipped with *tensor processing units* (TPUs), processors specialized for Machine Learning that are much faster than GPUs for many ML tasks. Of course, another option is simply to buy your own GPU card. Tim Dettmers wrote a great blog post to help you choose, and he updates it fairly regularly.

You must then download and install the appropriate version of the CUDA and cuDNN libraries (CUDA 8.0 and cuDNN 5.1 if you are using the binary installation of TensorFlow 1.0.0), and set a few environment variables so TensorFlow knows where to find CUDA and cuDNN. The detailed installation instructions are likely to change fairly quickly, so it is best that you follow the instructions on TensorFlow's website.

Nvidia's *Compute Unified Device Architecture* library (CUDA) allows developers to use CUDA-enabled GPUs for all sorts of computations (not just graphics acceleration). Nvidia's *CUDA Deep Neural Network* library (cuDNN) is a GPU-accelerated library of primitives for DNNs. It provides optimized implementations of common DNN computations such as activation layers, normalization, forward and backward convolutions, and pooling (see Chapter 13). It is part of Nvidia's Deep Learning SDK (note that it requires creating an Nvidia developer account in order to download it). TensorFlow uses CUDA and cuDNN to control the GPU cards and accelerate computations (see Figure 12-2).

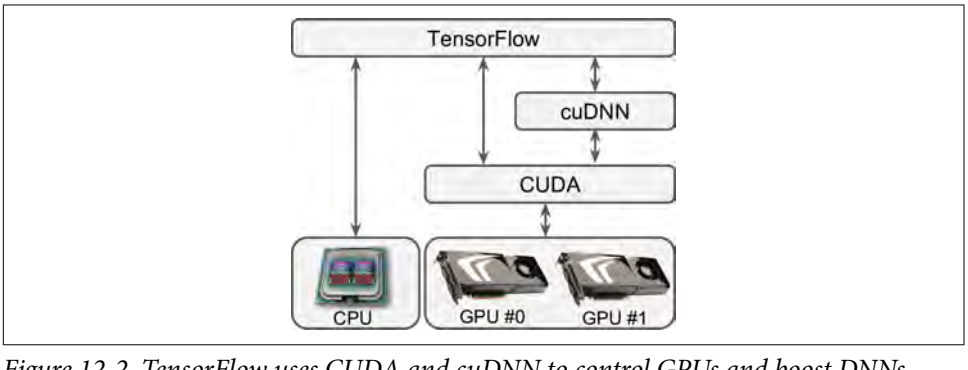

Figure 12-2. TensorFlow uses CUDA and cuDNN to control GPUs and boost DNNs

You can use the `nvidia-smi` command to check that CUDA is properly installed. It lists the available GPU cards, as well as processes running on each card:

```
$ nvidia-smi
Wed Sep 16 09:50:03 2016
+-----------------------------------------------------+
| NVIDIA-SMI 352.63     Driver Version: 352.63        |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GRID K520           Off  | 0000:00:03.0     Off |                  N/A |
| N/A   27C    P8    17W / 125W |     11MiB /  4095MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID  Type  Process name                               Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Finally, you must install TensorFlow with GPU support. If you created an isolated environment using virtualenv, you first need to activate it:

```
$ cd $ML_PATH               # Your ML working directory (e.g., $HOME/ml)
$ source env/bin/activate
```

Then install the appropriate GPU-enabled version of TensorFlow:

```
$ pip3 install --upgrade tensorflow-gpu
```

Now you can open up a Python shell and check that TensorFlow detects and uses CUDA and cuDNN properly by importing TensorFlow and creating a session:

```
>>> import tensorflow as tf
I [...]/dso_loader.cc:108] successfully opened CUDA library libcublas.so locally
I [...]/dso_loader.cc:108] successfully opened CUDA library libcudnn.so locally
I [...]/dso_loader.cc:108] successfully opened CUDA library libcufft.so locally
I [...]/dso_loader.cc:108] successfully opened CUDA library libcuda.so.1 locally
I [...]/dso_loader.cc:108] successfully opened CUDA library libcurand.so locally
>>> sess = tf.Session()
[...]
I [...]/gpu_init.cc:102] Found device 0 with properties:
name: GRID K520
major: 3 minor: 0 memoryClockRate (GHz) 0.797
pciBusID 0000:00:03.0
Total memory: 4.00GiB
Free memory: 3.95GiB
I [...]/gpu_init.cc:126] DMA: 0
I [...]/gpu_init.cc:136] 0:   Y
I [...]/gpu_device.cc:839] Creating TensorFlow device
(/gpu:0) -> (device: 0, name: GRID K520, pci bus id: 0000:00:03.0)
```

Looks good! TensorFlow detected the CUDA and cuDNN libraries, and it used the CUDA library to detect the GPU card (in this case an Nvidia Grid K520 card).

Managing the GPU RAM

By default TensorFlow automatically grabs all the RAM in all available GPUs the first time you run a graph, so you will not be able to start a second TensorFlow program while the first one is still running. If you try, you will get the following error:

```
E [...]/cuda_driver.cc:965] failed to allocate 3.66G (3928915968 bytes) from
device: CUDA_ERROR_OUT_OF_MEMORY
```

One solution is to run each process on different GPU cards. To do this, the simplest option is to set the `CUDA_VISIBLE_DEVICES` environment variable so that each process only sees the appropriate GPU cards. For example, you could start two programs like this:

```
$ CUDA_VISIBLE_DEVICES=0,1 python3 program_1.py
# and in another terminal:
$ CUDA_VISIBLE_DEVICES=3,2 python3 program_2.py
```

Program #1 will only see GPU cards 0 and 1 (numbered 0 and 1, respectively), and program #2 will only see GPU cards 2 and 3 (numbered 1 and 0, respectively). Everything will work fine (see Figure 12-3).

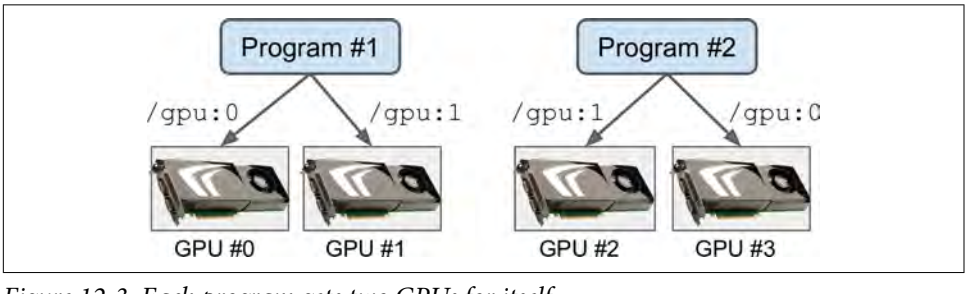

Figure 12-3. Each program gets two GPUs for itself

Another option is to tell TensorFlow to grab only a fraction of the memory. For example, to make TensorFlow grab only 40% of each GPU's memory, you must create a `ConfigProto` object, set its `gpu_options.per_process_gpu_memory_fraction` option to `0.4`, and create the session using this configuration:

```
config = tf.ConfigProto()
config.gpu_options.per_process_gpu_memory_fraction = 0.4
session = tf.Session(config=config)
```

Now two programs like this one can run in parallel using the same GPU cards (but not three, since $3 \times 0.4 > 1$). See Figure 12-4.

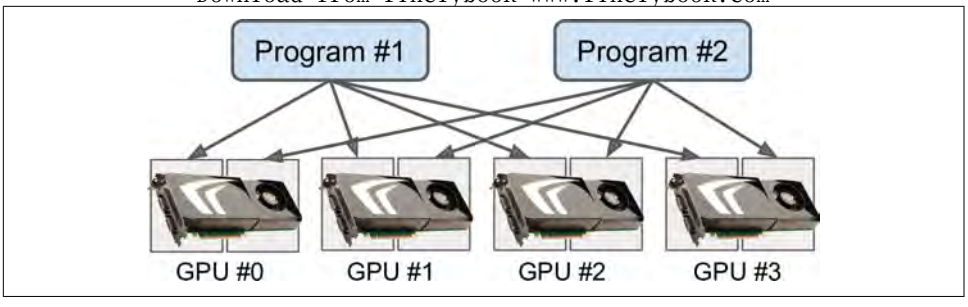

Figure 12-4. Each program gets all four GPUs, but with only 40% of the RAM each

If you run the `nvidia-smi` command while both programs are running, you should see that each process holds roughly 40% of the total RAM of each card:

```
$ nvidia-smi
[...]
+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID  Type  Process name                               Usage      |
|=============================================================================|
|    0      5231    C   python                                       1677MiB |
|    0      5262    C   python                                       1677MiB |
|    1      5231    C   python                                       1677MiB |
|    1      5262    C   python                                       1677MiB |
[...]
```

Yet another option is to tell TensorFlow to grab memory only when it needs it. To do this you must set `config.gpu_options.allow_growth` to `True`. However, TensorFlow never releases memory once it has grabbed it (to avoid memory fragmentation) so you may still run out of memory after a while. It may be harder to guarantee a deterministic behavior using this option, so in general you probably want to stick with one of the previous options.

Okay, now you have a working GPU-enabled TensorFlow installation. Let's see how to use it!

Placing Operations on Devices

The TensorFlow whitepaper[1] presents a friendly *dynamic placer* algorithm that automagically distributes operations across all available devices, taking into account things like the measured computation time in previous runs of the graph, estimations of the size of the input and output tensors to each operation, the amount of RAM available in each device, communication delay when transferring data in and out of

1 "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," Google Research (2015).

devices, hints and constraints from the user, and more. Unfortunately, this sophisticated algorithm is internal to Google; it was not released in the open source version of TensorFlow. The reason it was left out seems to be that in practice a small set of placement rules specified by the user actually results in more efficient placement than what the dynamic placer is capable of. However, the TensorFlow team is working on improving the dynamic placer, and perhaps it will eventually be good enough to be released.

Until then TensorFlow relies on the *simple placer*, which (as its name suggests) is very basic.

Simple placement

Whenever you run a graph, if TensorFlow needs to evaluate a node that is not placed on a device yet, it uses the simple placer to place it, along with all other nodes that are not placed yet. The simple placer respects the following rules:

- If a node was already placed on a device in a previous run of the graph, it is left on that device.
- Else, if the user *pinned* a node to a device (described next), the placer places it on that device.
- Else, it defaults to GPU #0, or the CPU if there is no GPU.

As you can see, placing operations on the appropriate device is mostly up to you. If you don't do anything, the whole graph will be placed on the default device. To pin nodes onto a device, you must create a device block using the `device()` function. For example, the following code pins the variable `a` and the constant `b` on the CPU, but the multiplication node `c` is not pinned on any device, so it will be placed on the default device:

```
with tf.device("/cpu:0"):
    a = tf.Variable(3.0)
    b = tf.constant(4.0)

c = a * b
```

The `"/cpu:0"` device aggregates all CPUs on a multi-CPU system. There is currently no way to pin nodes on specific CPUs or to use just a subset of all CPUs.

Logging placements

Let's check that the simple placer respects the placement constraints we have just defined. For this you can set the `log_device_placement` option to `True`; this tells the placer to log a message whenever it places a node. For example:

```
>>> config = tf.ConfigProto()
>>> config.log_device_placement = True
>>> sess = tf.Session(config=config)
I [...] Creating TensorFlow device (/gpu:0) -> (device: 0, name: GRID K520,
pci bus id: 0000:00:03.0)
[...]
>>> x.initializer.run(session=sess)
I [...] a: /job:localhost/replica:0/task:0/cpu:0
I [...] a/read: /job:localhost/replica:0/task:0/cpu:0
I [...] mul: /job:localhost/replica:0/task:0/gpu:0
I [...] a/Assign: /job:localhost/replica:0/task:0/cpu:0
I [...] b: /job:localhost/replica:0/task:0/cpu:0
I [...] a/initial_value: /job:localhost/replica:0/task:0/cpu:0
>>> sess.run(c)
12
```

The lines starting with `"I"` for Info are the log messages. When we create a session, TensorFlow logs a message to tell us that it has found a GPU card (in this case the Grid K520 card). Then the first time we run the graph (in this case when initializing the variable `a`), the simple placer is run and places each node on the device it was assigned to. As expected, the log messages show that all nodes are placed on `"/cpu:0"` except the multiplication node, which ends up on the default device `"/gpu:0"` (you can safely ignore the prefix `/job:localhost/replica:0/task:0` for now; we will talk about it in a moment). Notice that the second time we run the graph (to compute `c`), the placer is not used since all the nodes TensorFlow needs to compute `c` are already placed.

Dynamic placement function

When you create a device block, you can specify a function instead of a device name. TensorFlow will call this function for each operation it needs to place in the device block, and the function must return the name of the device to pin the operation on. For example, the following code pins all the variable nodes to `"/cpu:0"` (in this case just the variable `a`) and all other nodes to `"/gpu:0"`:

```
def variables_on_cpu(op):
    if op.type == "Variable":
        return "/cpu:0"
    else:
        return "/gpu:0"

with tf.device(variables_on_cpu):
    a = tf.Variable(3.0)
```

```
b = tf.constant(4.0)
c = a * b
```

You can easily implement more complex algorithms, such as pinning variables across GPUs in a round-robin fashion.

Operations and kernels

For a TensorFlow operation to run on a device, it needs to have an implementation for that device; this is called a *kernel*. Many operations have kernels for both CPUs and GPUs, but not all of them. For example, TensorFlow does not have a GPU kernel for integer variables, so the following code will fail when TensorFlow tries to place the variable `i` on GPU #0:

```
>>> with tf.device("/gpu:0"):
...     i = tf.Variable(3)
[...]
>>> sess.run(i.initializer)
Traceback (most recent call last):
[...]
tensorflow.python.framework.errors.InvalidArgumentError: Cannot assign a device
to node 'Variable': Could not satisfy explicit device specification
```

Note that TensorFlow infers that the variable must be of type `int32` since the initialization value is an integer. If you change the initialization value to `3.0` instead of `3`, or if you explicitly set `dtype=tf.float32` when creating the variable, everything will work fine.

Soft placement

By default, if you try to pin an operation on a device for which the operation has no kernel, you get the exception shown earlier when TensorFlow tries to place the operation on the device. If you prefer TensorFlow to fall back to the CPU instead, you can set the `allow_soft_placement` configuration option to `True`:

```
with tf.device("/gpu:0"):
    i = tf.Variable(3)

config = tf.ConfigProto()
config.allow_soft_placement = True
sess = tf.Session(config=config)
sess.run(i.initializer)  # the placer runs and falls back to /cpu:0
```

So far we have discussed how to place nodes on different devices. Now let's see how TensorFlow will run these nodes in parallel.

Parallel Execution

When TensorFlow runs a graph, it starts by finding out the list of nodes that need to be evaluated, and it counts how many dependencies each of them has. TensorFlow

then starts evaluating the nodes with zero dependencies (i.e., source nodes). If these nodes are placed on separate devices, they obviously get evaluated in parallel. If they are placed on the same device, they get evaluated in different threads, so they may run in parallel too (in separate GPU threads or CPU cores).

TensorFlow manages a thread pool on each device to parallelize operations (see Figure 12-5). These are called the *inter-op thread pools*. Some operations have multithreaded kernels: they can use other thread pools (one per device) called the *intra-op thread pools*.

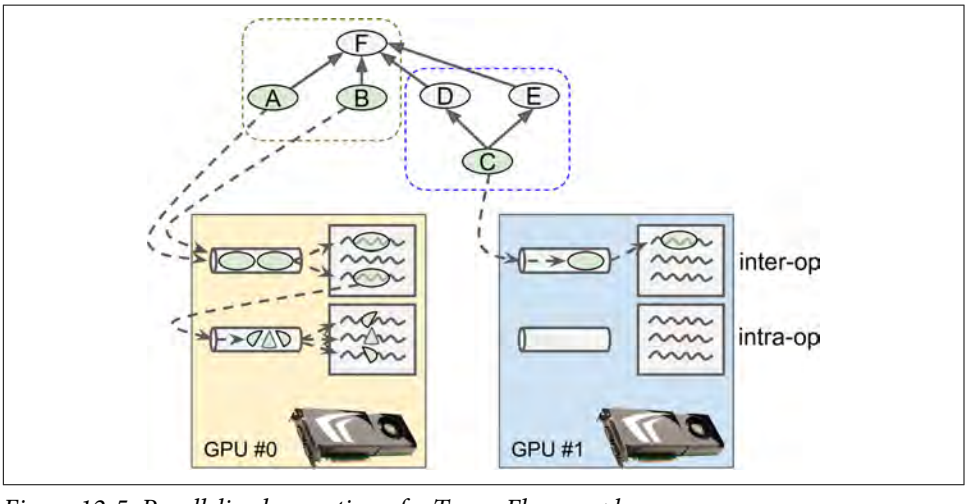

Figure 12-5. Parallelized execution of a TensorFlow graph

For example, in Figure 12-5, operations A, B, and C are source ops, so they can immediately be evaluated. Operations A and B are placed on GPU #0, so they are sent to this device's inter-op thread pool, and immediately evaluated in parallel. Operation A happens to have a multithreaded kernel; its computations are split in three parts, which are executed in parallel by the intra-op thread pool. Operation C goes to GPU #1's inter-op thread pool.

As soon as operation C finishes, the dependency counters of operations D and E will be decremented and will both reach 0, so both operations will be sent to the inter-op thread pool to be executed.

You can control the number of threads per inter-op pool by setting the `inter_op_parallelism_threads` option. Note that the first session you start creates the inter-op thread pools. All other sessions will just reuse them unless you set the `use_per_session_threads` option to `True`. You can control the number of threads per intra-op pool by setting the `intra_op_parallelism_threads` option.

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").

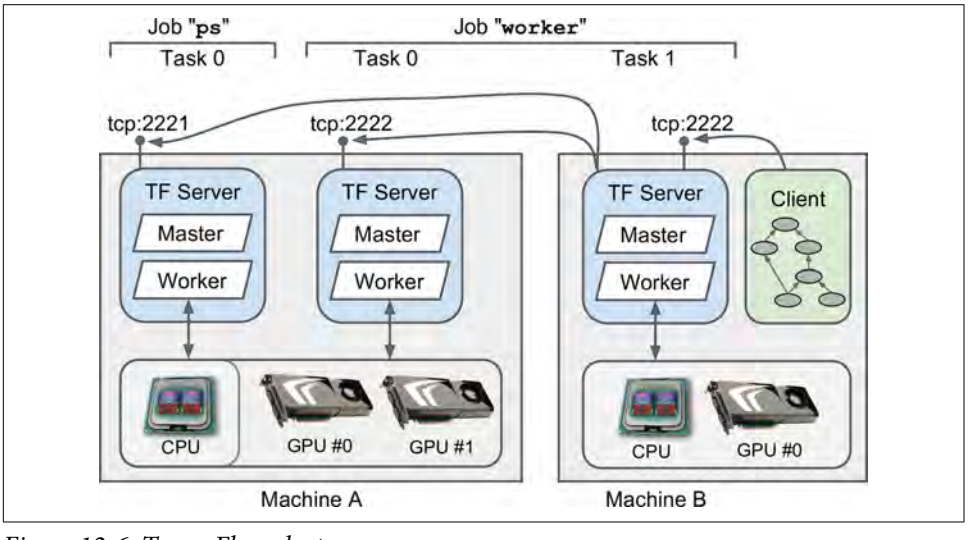

Figure 12-6. TensorFlow cluster

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.

Data is transmitted in the form of *protocol buffers*, another open source Google technology. This is a lightweight binary data interchange format.

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):
    v1 = tf.Variable(1.0)  # pinned to /job:ps/task:0
    v2 = tf.Variable(2.0)  # pinned to /job:ps/task:1
    v3 = tf.Variable(3.0)  # pinned to /job:ps/task:0
    v4 = tf.Variable(4.0)  # pinned to /job:ps/task:1
    v5 = tf.Variable(5.0)  # pinned to /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)):
    v1 = tf.Variable(1.0)  # pinned to /job:ps/task:0 (+ defaults to /cpu:0)
    v2 = tf.Variable(2.0)  # pinned to /job:ps/task:1 (+ defaults to /cpu:0)
    v3 = tf.Variable(3.0)  # pinned to /job:ps/task:0 (+ defaults to /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"):
            p2 = 3 * s     # pinned to /job:worker/task:1/gpu:1
```

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.

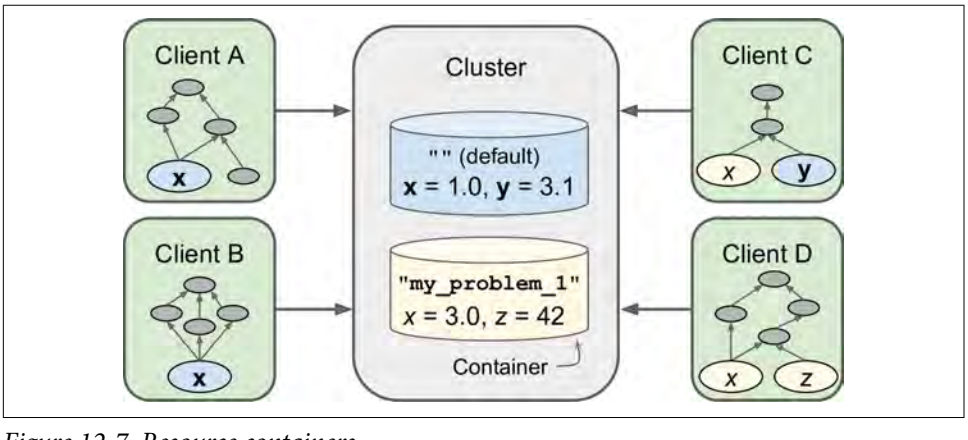

Figure 12-7. Resource containers

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 con-

siderably because the training operations don't have to wait for the next mini-batch at every step.

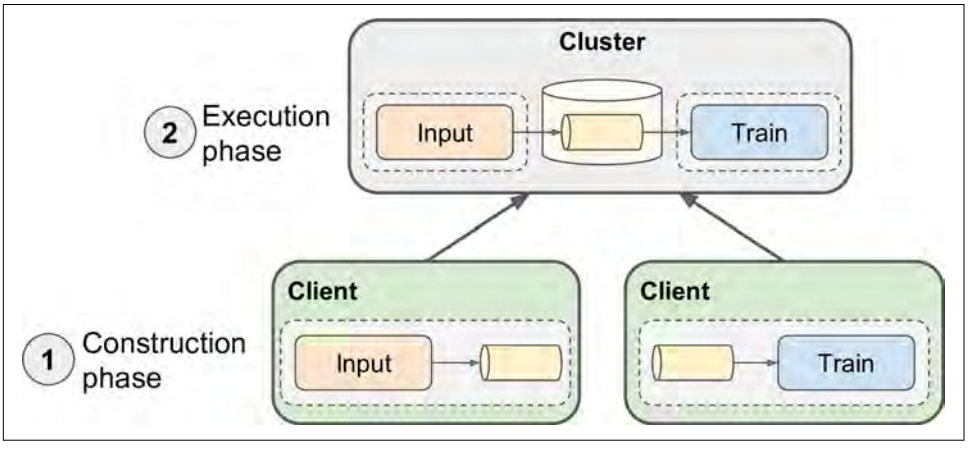

Figure 12-8. Using queues to load the training data asynchronously

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")
```

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:
    print(sess.run(dequeue))  # [1., 2.]
    print(sess.run(dequeue))  # [3., 4.]
    print(sess.run(dequeue))  # [5., 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:
    print(sess.run(dequeue_mini_batch))  # [[1., 2.], [4., 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
    print(b_val) # [[1., 2.], [3., 4.], [5., 6.]]
```

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])
    print(a) # [10, 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:
   print(sess.run(dequeue)) # [ 20.  15.  11.  12.   4.]   (17 items left)
   print(sess.run(dequeue)) # [  5.  13.   6.   0.  17.]   (12 items 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:
    data = [...]  # load the training data from the 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.

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.

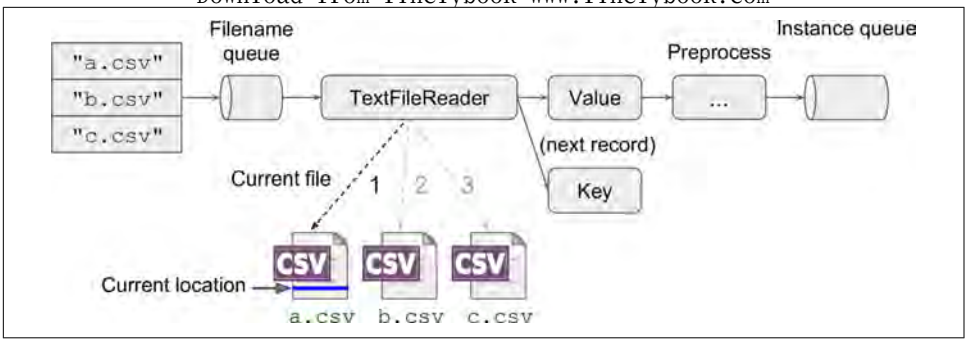

Figure 12-9. A graph dedicated to reading training instances from CSV files

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:
        pass # no more training 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.

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).

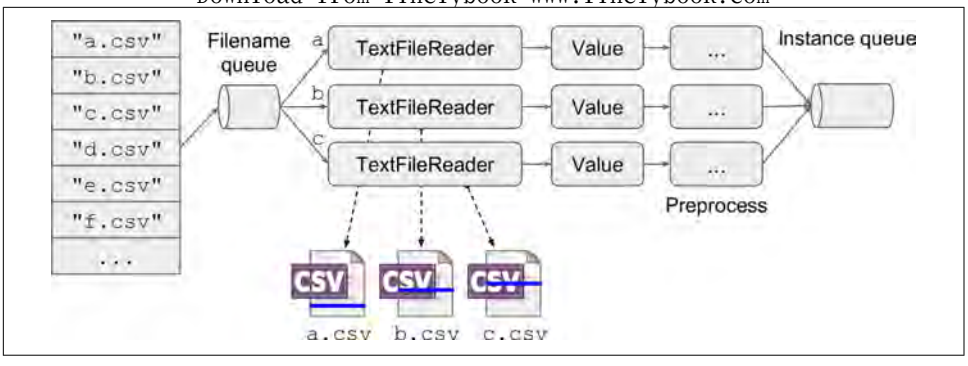

Figure 12-10. Reading simultaneously from multiple files

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!

Parallelizing Neural Networks on a TensorFlow Cluster

In this section, first we will look at how to parallelize several neural networks by simply placing each one on a different device. Then we will look at the much trickier problem of training a single neural network across multiple devices and servers.

One Neural Network per Device

The most trivial way to train and run neural networks on a TensorFlow cluster is to take the exact same code you would use for a single device on a single machine, and specify the master server's address when creating the session. That's it—you're done! Your code will be running on the server's default device. You can change the device that will run your graph simply by putting your code's construction phase within a device block.

By running several client sessions in parallel (in different threads or different processes), connecting them to different servers, and configuring them to use different devices, you can quite easily train or run many neural networks in parallel, across all devices and all machines in your cluster (see Figure 12-11). The speedup is almost linear.[4] Training 100 neural networks across 50 servers with 2 GPUs each will not take much longer than training just 1 neural network on 1 GPU.

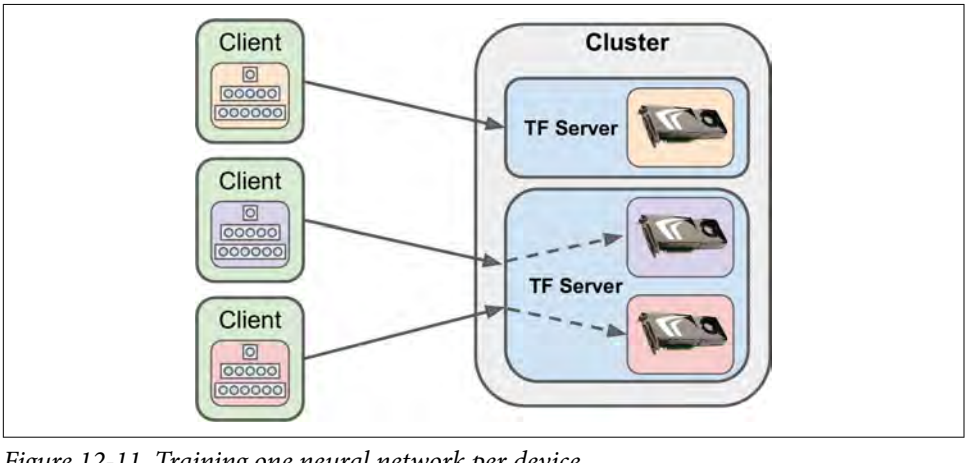

Figure 12-11. Training one neural network per device

4 Not 100% linear if you wait for all devices to finish, since the total time will be the time taken by the slowest device.

This solution is perfect for hyperparameter tuning: each device in the cluster will train a different model with its own set of hyperparameters. The more computing power you have, the larger the hyperparameter space you can explore.

It also works perfectly if you host a web service that receives a large number of *queries per second* (QPS) and you need your neural network to make a prediction for each query. Simply replicate the neural network across all devices on the cluster and dispatch queries across all devices. By adding more servers you can handle an unlimited number of QPS (however, this will not reduce the time it takes to process a single request since it will still have to wait for a neural network to make a prediction).

Another option is to serve your neural networks using *TensorFlow Serving*. It is an open source system, released by Google in February 2016, designed to serve a high volume of queries to Machine Learning models (typically built with TensorFlow). It handles model versioning, so you can easily deploy a new version of your network to production, or experiment with various algorithms without interrupting your service, and it can sustain a heavy load by adding more servers. For more details, check out *https://tensorflow.github.io/serving/*.

In-Graph Versus Between-Graph Replication

You can also parallelize the training of a large ensemble of neural networks by simply placing every neural network on a different device (ensembles were introduced in Chapter 7). However, once you want to *run* the ensemble, you will need to aggregate the individual predictions made by each neural network to produce the ensemble's prediction, and this requires a bit of coordination.

There are two major approaches to handling a neural network ensemble (or any other graph that contains large chunks of independent computations):

- You can create one big graph, containing every neural network, each pinned to a different device, plus the computations needed to aggregate the individual predictions from all the neural networks (see Figure 12-12). Then you just create one session to any server in the cluster and let it take care of everything (including waiting for all individual predictions to be available before aggregating them). This approach is called *in-graph replication*.

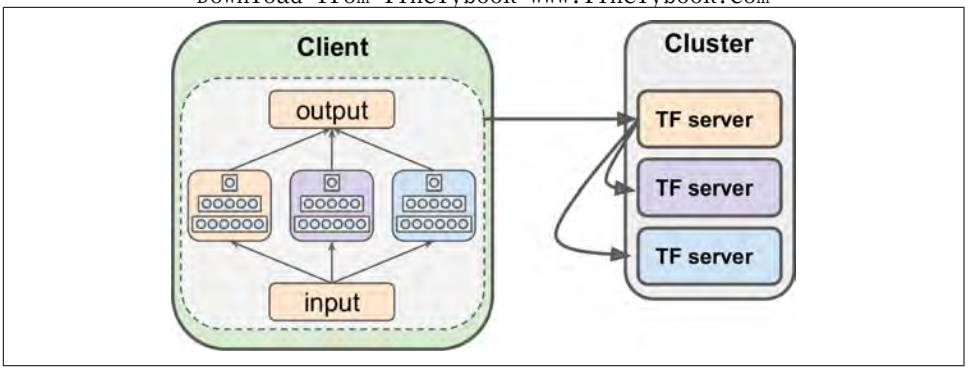

Figure 12-12. In-graph replication

- Alternatively, you can create one separate graph for each neural network and handle synchronization between these graphs yourself. This approach is called *between-graph replication*. One typical implementation is to coordinate the execution of these graphs using queues (see Figure 12-13). A set of clients handles one neural network each, reading from its dedicated input queue, and writing to its dedicated prediction queue. Another client is in charge of reading the inputs and pushing them to all the input queues (copying all inputs to every queue). Finally, one last client is in charge of reading one prediction from each prediction queue and aggregating them to produce the ensemble's prediction.

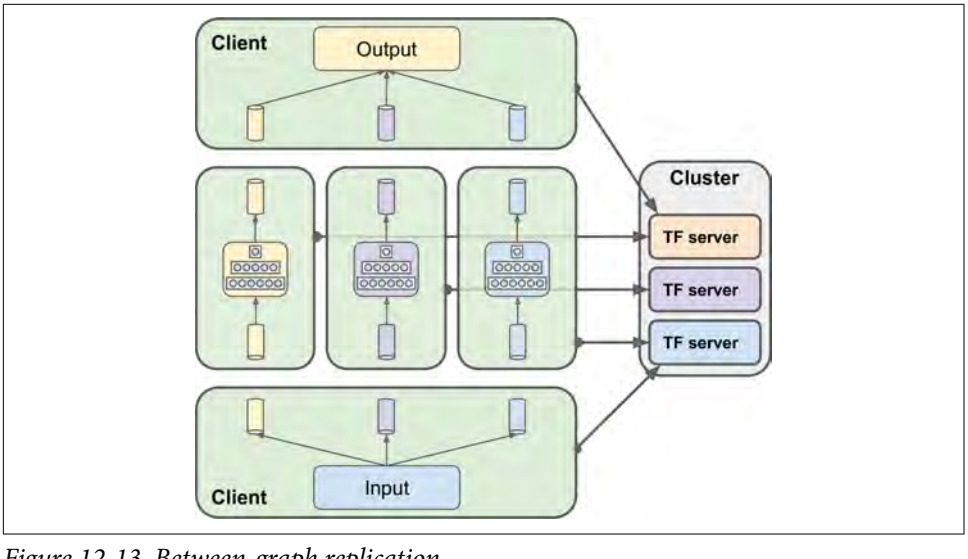

Figure 12-13. Between-graph replication

These solutions have their pros and cons. In-graph replication is somewhat simpler to implement since you don't have to manage multiple clients and multiple queues. However, between-graph replication is a bit easier to organize into well-bounded and easy-to-test modules. Moreover, it gives you more flexibility. For example, you could add a dequeue timeout in the aggregator client so that the ensemble would not fail even if one of the neural network clients crashes or if one neural network takes too long to produce its prediction. TensorFlow lets you specify a timeout when calling the `run()` function by passing a `RunOptions` with `timeout_in_ms`:

```
with tf.Session([...]) as sess:
    [...]
    run_options = tf.RunOptions()
    run_options.timeout_in_ms = 1000  # 1s timeout
    try:
        pred = sess.run(dequeue_prediction, options=run_options)
    except tf.errors.DeadlineExceededError as ex:
        [...] # the dequeue operation timed out after 1s
```

Another way you can specify a timeout is to set the session's `operation_timeout_in_ms` configuration option, but in this case the `run()` function times out if *any* operation takes longer than the timeout delay:

```
config = tf.ConfigProto()
config.operation_timeout_in_ms = 1000  # 1s timeout for every operation

with tf.Session([...], config=config) as sess:
    [...]
    try:
        pred = sess.run(dequeue_prediction)
    except tf.errors.DeadlineExceededError as ex:
        [...]  # the dequeue operation timed out after 1s
```

Model Parallelism

So far we have run each neural network on a single device. What if we want to run a single neural network across multiple devices? This requires chopping your model into separate chunks and running each chunk on a different device. This is called *model parallelism*. Unfortunately, model parallelism turns out to be pretty tricky, and it really depends on the architecture of your neural network. For fully connected networks, there is generally not much to be gained from this approach (see Figure 12-14). Intuitively, it may seem that an easy way to split the model is to place each layer on a different device, but this does not work since each layer needs to wait for the output of the previous layer before it can do anything. So perhaps you can slice it vertically—for example, with the left half of each layer on one device, and the right part on another device? This is slightly better, since both halves of each layer can indeed work in parallel, but the problem is that each half of the next layer requires the output of both halves, so there will be a lot of cross-device communication (repre-

sented by the dashed arrows). This is likely to completely cancel out the benefit of the parallel computation, since cross-device communication is slow (especially if it is across separate machines).

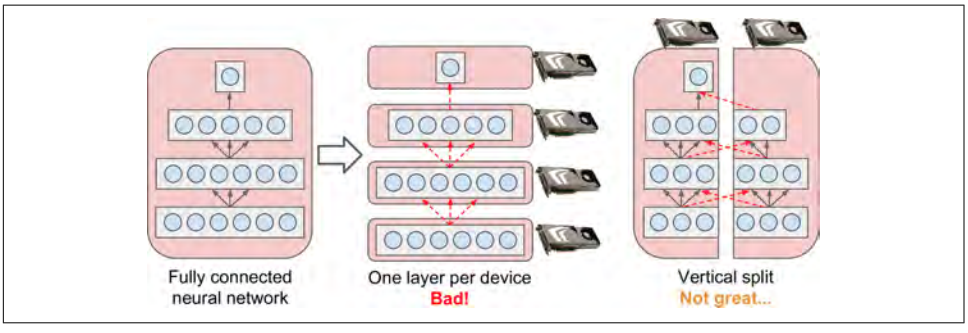

Figure 12-14. Splitting a fully connected neural network

However, as we will see in Chapter 13, some neural network architectures, such as convolutional neural networks, contain layers that are only partially connected to the lower layers, so it is much easier to distribute chunks across devices in an efficient way.

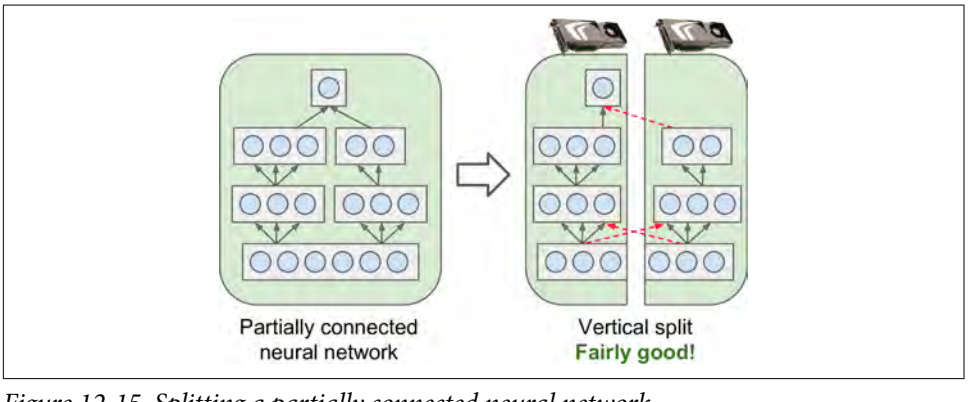

Figure 12-15. Splitting a partially connected neural network

Moreover, as we will see in Chapter 14, some deep recurrent neural networks are composed of several layers of *memory cells* (see the left side of Figure 12-16). A cell's output at time t is fed back to its input at time $t + 1$ (as you can see more clearly on the right side of Figure 12-16). If you split such a network horizontally, placing each layer on a different device, then at the first step only one device will be active, at the second step two will be active, and by the time the signal propagates to the output layer all devices will be active simultaneously. There is still a lot of cross-device communication going on, but since each cell may be fairly complex, the benefit of running multiple cells in parallel often outweighs the communication penalty.

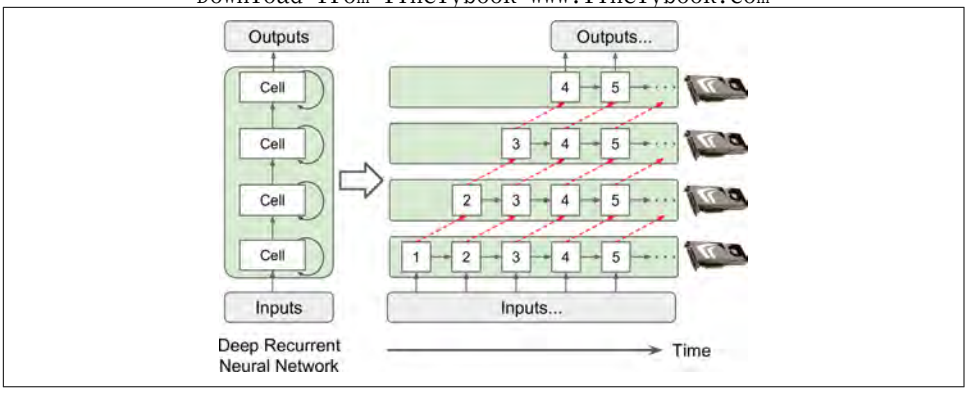

Figure 12-16. Splitting a deep recurrent neural network

In short, model parallelism can speed up running or training some types of neural networks, but not all, and it requires special care and tuning, such as making sure that devices that need to communicate the most run on the same machine.

Data Parallelism

Another way to parallelize the training of a neural network is to replicate it on each device, run a training step simultaneously on all replicas using a different mini-batch for each, and then aggregate the gradients to update the model parameters. This is called *data parallelism* (see Figure 12-17).

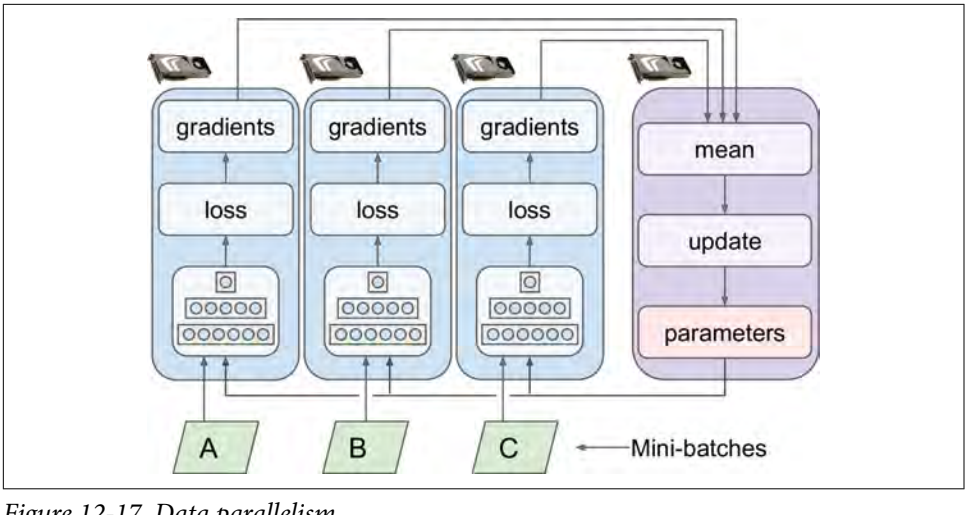

Figure 12-17. Data parallelism

There are two variants of this approach: *synchronous updates* and *asynchronous updates*.

Synchronous updates

With *synchronous updates*, the aggregator waits for all gradients to be available before computing the average and applying the result (i.e., using the aggregated gradients to update the model parameters). Once a replica has finished computing its gradients, it must wait for the parameters to be updated before it can proceed to the next mini-batch. The downside is that some devices may be slower than others, so all other devices will have to wait for them at every step. Moreover, the parameters will be copied to every device almost at the same time (immediately after the gradients are applied), which may saturate the parameter servers' bandwidth.

To reduce the waiting time at each step, you could ignore the gradients from the slowest few replicas (typically ~10%). For example, you could run 20 replicas, but only aggregate the gradients from the fastest 18 replicas at each step, and just ignore the gradients from the last 2. As soon as the parameters are updated, the first 18 replicas can start working again immediately, without having to wait for the 2 slowest replicas. This setup is generally described as having 18 replicas plus 2 *spare replicas*.[5]

Asynchronous updates

With asynchronous updates, whenever a replica has finished computing the gradients, it immediately uses them to update the model parameters. There is no aggregation (remove the "mean" step in Figure 12-17), and no synchronization. Replicas just work independently of the other replicas. Since there is no waiting for the other replicas, this approach runs more training steps per minute. Moreover, although the parameters still need to be copied to every device at every step, this happens at different times for each replica so the risk of bandwidth saturation is reduced.

Data parallelism with asynchronous updates is an attractive choice, because of its simplicity, the absence of synchronization delay, and a better use of the bandwidth. However, although it works reasonably well in practice, it is almost surprising that it works at all! Indeed, by the time a replica has finished computing the gradients based on some parameter values, these parameters will have been updated several times by other replicas (on average $N - 1$ times if there are N replicas) and there is no guarantee that the computed gradients will still be pointing in the right direction (see Figure 12-18). When gradients are severely out-of-date, they are called *stale gradients*: they can slow down convergence, introducing noise and wobble effects (the learning

5 This name is slightly confusing since it sounds like some replicas are special, doing nothing. In reality, all replicas are equivalent: they all work hard to be among the fastest at each training step, and the losers vary at every step (unless some devices are really slower than others).

curve may contain temporary oscillations), or they can even make the training algorithm diverge.

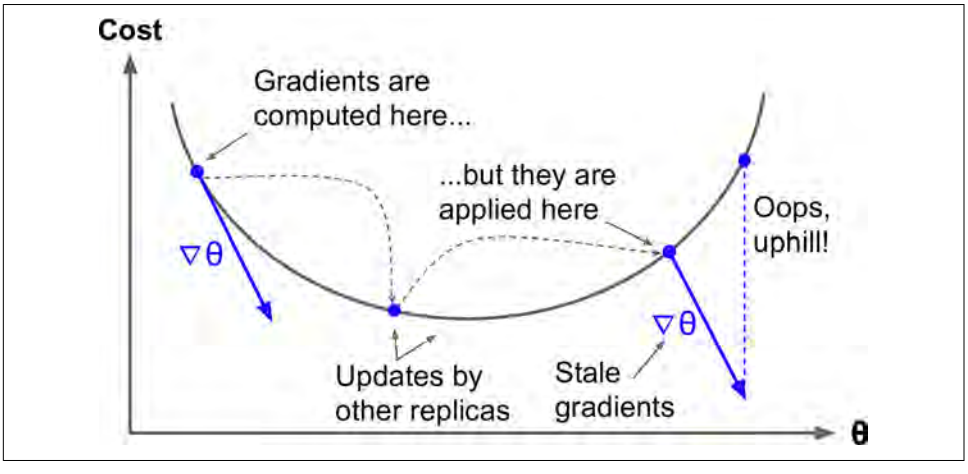

Figure 12-18. Stale gradients when using asynchronous updates

There are a few ways to reduce the effect of stale gradients:

- Reduce the learning rate.
- Drop stale gradients or scale them down.
- Adjust the mini-batch size.
- Start the first few epochs using just one replica (this is called the *warmup phase*). Stale gradients tend to be more damaging at the beginning of training, when gradients are typically large and the parameters have not settled into a valley of the cost function yet, so different replicas may push the parameters in quite different directions.

A paper published by the Google Brain team in April 2016 benchmarked various approaches and found that data parallelism with synchronous updates using a few spare replicas was the most efficient, not only converging faster but also producing a better model. However, this is still an active area of research, so you should not rule out asynchronous updates quite yet.

Bandwidth saturation

Whether you use synchronous or asynchronous updates, data parallelism still requires communicating the model parameters from the parameter servers to every replica at the beginning of every training step, and the gradients in the other direction at the end of each training step. Unfortunately, this means that there always comes a point where adding an extra GPU will not improve performance at all because the

time spent moving the data in and out of GPU RAM (and possibly across the network) will outweigh the speedup obtained by splitting the computation load. At that point, adding more GPUs will just increase saturation and slow down training.

For some models, typically relatively small and trained on a very large training set, you are often better off training the model on a single machine with a single GPU.

Saturation is more severe for large dense models, since they have a lot of parameters and gradients to transfer. It is less severe for small models (but the parallelization gain is small) and also for large sparse models since the gradients are typically mostly zeros, so they can be communicated efficiently. Jeff Dean, initiator and lead of the Google Brain project, reported typical speedups of 25–40x when distributing computations across 50 GPUs for dense models, and 300x speedup for sparser models trained across 500 GPUs. As you can see, sparse models really do scale better. Here are a few concrete examples:

- Neural Machine Translation: 6x speedup on 8 GPUs
- Inception/ImageNet: 32x speedup on 50 GPUs
- RankBrain: 300x speedup on 500 GPUs

These numbers represent the state of the art in Q1 2016. Beyond a few dozen GPUs for a dense model or few hundred GPUs for a sparse model, saturation kicks in and performance degrades. There is plenty of research going on to solve this problem (exploring peer-to-peer architectures rather than centralized parameter servers, using lossy model compression, optimizing when and what the replicas need to communicate, and so on), so there will likely be a lot of progress in parallelizing neural networks in the next few years.

In the meantime, here are a few simple steps you can take to reduce the saturation problem:

- Group your GPUs on a few servers rather than scattering them across many servers. This will avoid unnecessary network hops.
- Shard the parameters across multiple parameter servers (as discussed earlier).
- Drop the model parameters' float precision from 32 bits (`tf.float32`) to 16 bits (`tf.bfloat16`). This will cut in half the amount of data to transfer, without much impact on the convergence rate or the model's performance.

Although 16-bit precision is the minimum for training neural network, you can actually drop down to 8-bit precision after training to reduce the size of the model and speed up computations. This is called *quantizing* the neural network. It is particularly useful for deploying and running pretrained models on mobile phones. See Pete Warden's great post on the subject.

TensorFlow implementation

To implement data parallelism using TensorFlow, you first need to choose whether you want in-graph replication or between-graph replication, and whether you want synchronous updates or asynchronous updates. Let's look at how you would implement each combination (see the exercises and the Jupyter notebooks for complete code examples).

With in-graph replication + synchronous updates, you build one big graph containing all the model replicas (placed on different devices), and a few nodes to aggregate all their gradients and feed them to an optimizer. Your code opens a session to the cluster and simply runs the training operation repeatedly.

With in-graph replication + asynchronous updates, you also create one big graph, but with one optimizer per replica, and you run one thread per replica, repeatedly running the replica's optimizer.

With between-graph replication + asynchronous updates, you run multiple independent clients (typically in separate processes), each training the model replica as if it were alone in the world, but the parameters are actually shared with other replicas (using a resource container).

With between-graph replication + synchronous updates, once again you run multiple clients, each training a model replica based on shared parameters, but this time you wrap the optimizer (e.g., a `MomentumOptimizer`) within a `SyncReplicasOptimizer`. Each replica uses this optimizer as it would use any other optimizer, but under the hood this optimizer sends the gradients to a set of queues (one per variable), which is read by one of the replica's `SyncReplicasOptimizer`, called the *chief*. The chief aggregates the gradients and applies them, then writes a token to a *token queue* for each replica, signaling it that it can go ahead and compute the next gradients. This approach supports having *spare replicas*.

If you go through the exercises, you will implement each of these four solutions. You will easily be able to apply what you have learned to train large deep neural networks across dozens of servers and GPUs! In the following chapters we will go through a few more important neural network architectures before we tackle Reinforcement Learning.

Exercises

1. If you get a `CUDA_ERROR_OUT_OF_MEMORY` when starting your TensorFlow program, what is probably going on? What can you do about it?
2. What is the difference between pinning an operation on a device and placing an operation on a device?
3. If you are running on a GPU-enabled TensorFlow installation, and you just use the default placement, will all operations be placed on the first GPU?
4. If you pin a variable to `"/gpu:0"`, can it be used by operations placed on `/gpu:1`? Or by operations placed on `"/cpu:0"`? Or by operations pinned to devices located on other servers?
5. Can two operations placed on the same device run in parallel?
6. What is a control dependency and when would you want to use one?
7. Suppose you train a DNN for days on a TensorFlow cluster, and immediately after your training program ends you realize that you forgot to save the model using a `Saver`. Is your trained model lost?
8. Train several DNNs in parallel on a TensorFlow cluster, using different hyperparameter values. This could be DNNs for MNIST classification or any other task you are interested in. The simplest option is to write a single client program that trains only one DNN, then run this program in multiple processes in parallel, with different hyperparameter values for each client. The program should have command-line options to control what server and device the DNN should be placed on, and what resource container and hyperparameter values to use (make sure to use a different resource container for each DNN). Use a validation set or cross-validation to select the top three models.
9. Create an ensemble using the top three models from the previous exercise. Define it in a single graph, ensuring that each DNN runs on a different device. Evaluate it on the validation set: does the ensemble perform better than the individual DNNs?
10. Train a DNN using between-graph replication and data parallelism with asynchronous updates, timing how long it takes to reach a satisfying performance. Next, try again using synchronous updates. Do synchronous updates produce a better model? Is training faster? Split the DNN vertically and place each vertical slice on a different device, and train the model again. Is training any faster? Is the performance any different?

Solutions to these exercises are available in Appendix A.

CHAPTER 13
Convolutional Neural Networks

Although IBM's Deep Blue supercomputer beat the chess world champion Garry Kasparov back in 1996, until quite recently computers were unable to reliably perform seemingly trivial tasks such as detecting a puppy in a picture or recognizing spoken words. Why are these tasks so effortless to us humans? The answer lies in the fact that perception largely takes place outside the realm of our consciousness, within specialized visual, auditory, and other sensory modules in our brains. By the time sensory information reaches our consciousness, it is already adorned with high-level features; for example, when you look at a picture of a cute puppy, you cannot choose *not* to see the puppy, or *not* to notice its cuteness. Nor can you explain *how* you recognize a cute puppy; it's just obvious to you. Thus, we cannot trust our subjective experience: perception is not trivial at all, and to understand it we must look at how the sensory modules work.

Convolutional neural networks (CNNs) emerged from the study of the brain's visual cortex, and they have been used in image recognition since the 1980s. In the last few years, thanks to the increase in computational power, the amount of available training data, and the tricks presented in Chapter 11 for training deep nets, CNNs have managed to achieve superhuman performance on some complex visual tasks. They power image search services, self-driving cars, automatic video classification systems, and more. Moreover, CNNs are not restricted to visual perception: they are also successful at other tasks, such as *voice recognition* or *natural language processing* (NLP); however, we will focus on visual applications for now.

In this chapter we will present where CNNs came from, what their building blocks look like, and how to implement them using TensorFlow. Then we will present some of the best CNN architectures.