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).

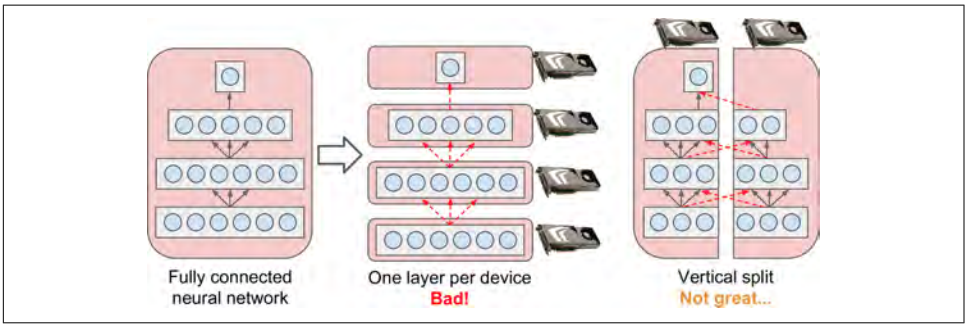

Figure 12-14. Splitting a fully connected neural network

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.

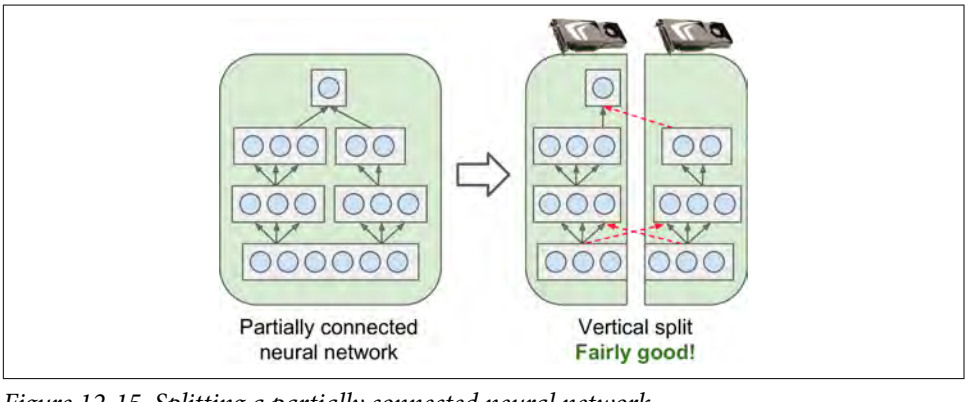

Figure 12-15. Splitting a partially connected neural network

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.

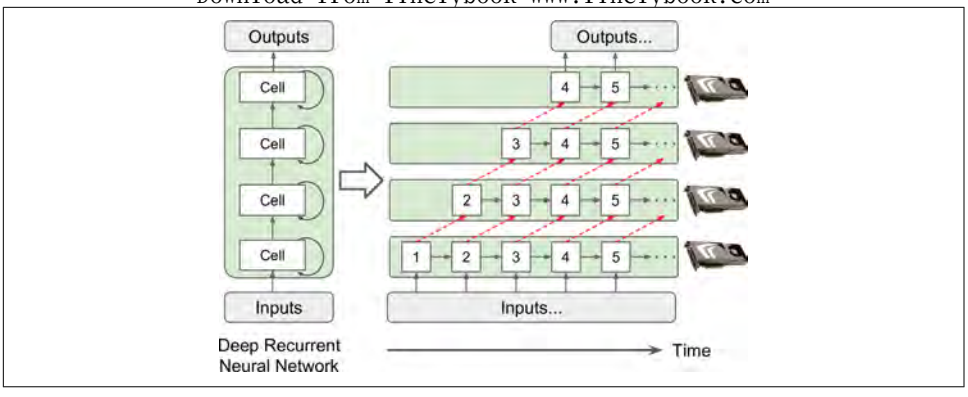

Figure 12-16. Splitting a deep recurrent neural network

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).

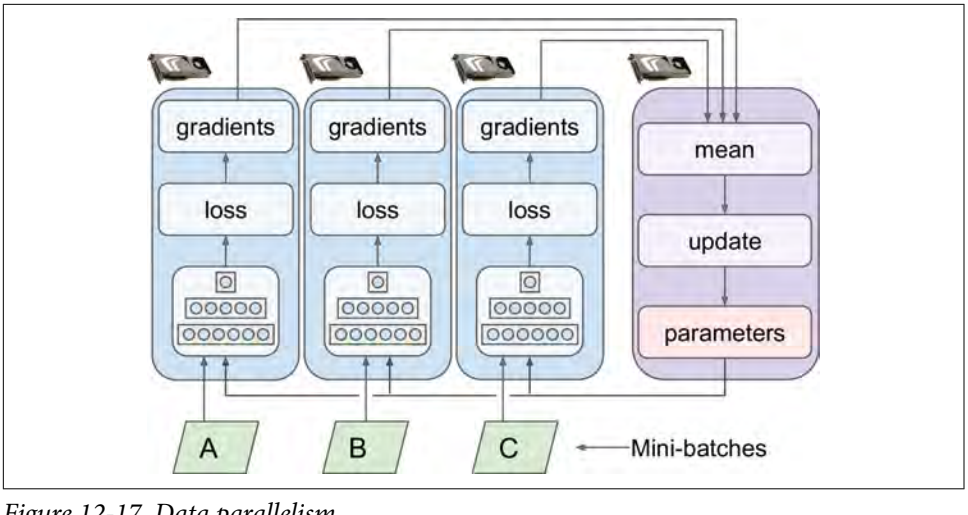

Figure 12-17. Data parallelism

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