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*.

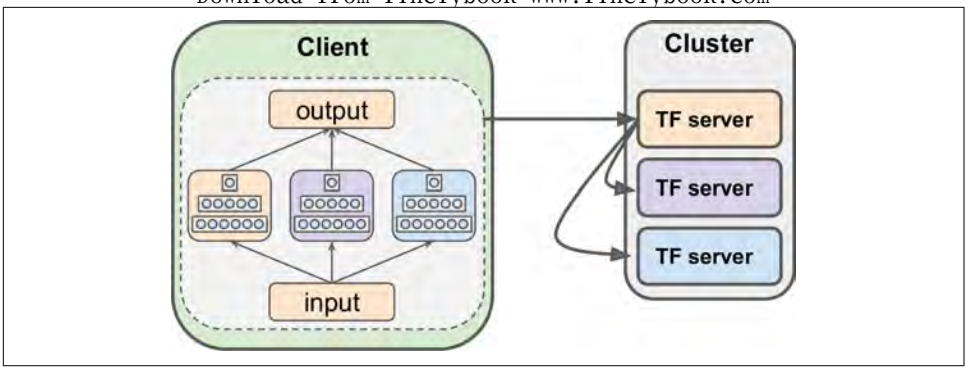

Figure 12-12. In-graph replication

- 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.

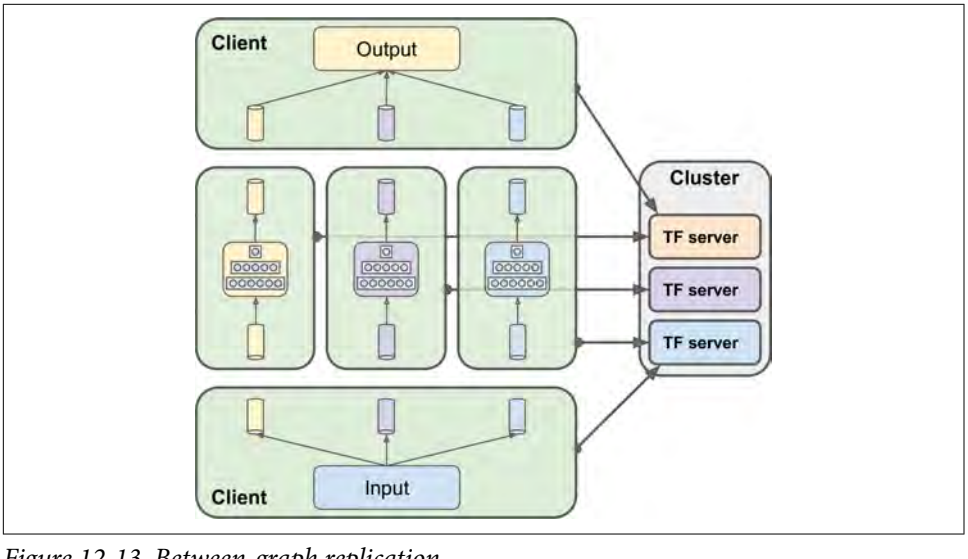

Figure 12-13. Between-graph replication

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-