

## 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
```