

---

## TensorFlow Ecosystem

This repository contains examples for integrating TensorFlow with other open-source frameworks. The examples are minimal and intended for use as templates. Users can tailor the templates for their own use-cases.

If you have any additions or improvements, please create an issue or pull request.

### Contents

- docker - Docker configuration for running TensorFlow on cluster managers.
- kubeflow - A Kubernetes native platform for ML
  - A K8s custom resource for running distributed TensorFlow jobs
  - Jupyter images for different versions of TensorFlow
  - TFServing Docker images and K8s templates
- kubernetes - Templates for running distributed TensorFlow on Kubernetes.
- marathon - Templates for running distributed TensorFlow using Marathon, deployed on top of Mesos.
- hadoop - TFRecord file InputFormat/OutputFormat for Hadoop MapReduce and Spark.
- spark-tensorflow-connector - Spark TensorFlow Connector
- spark-tensorflow-distributor - Python package that helps users do distributed training with TensorFlow on their Spark clusters.

### Distributed TensorFlow

See the Distributed TensorFlow documentation for a description of how it works. The examples in this repository focus on the most common form of distributed training: between-graph replication with asynchronous updates.

### Common Setup for distributed training

Every distributed training program has some common setup. First, define flags so that the worker knows about other workers and knows what role it plays in distributed training:

```
1 # Flags for configuring the task
2 flags.DEFINE_integer("task_index", None,
3                     "Worker task index, should be >= 0. task_index=0
   is "
```

---

```
4         "the master worker task the performs the variable
5         "
6         "initialization.")
7 flags.DEFINE_string("ps_hosts", None,
8                    "Comma-separated list of hostname:port pairs")
9 flags.DEFINE_string("worker_hosts", None,
10                   "Comma-separated list of hostname:port pairs")
11 flags.DEFINE_string("job_name", None, "job name: worker or ps")
```

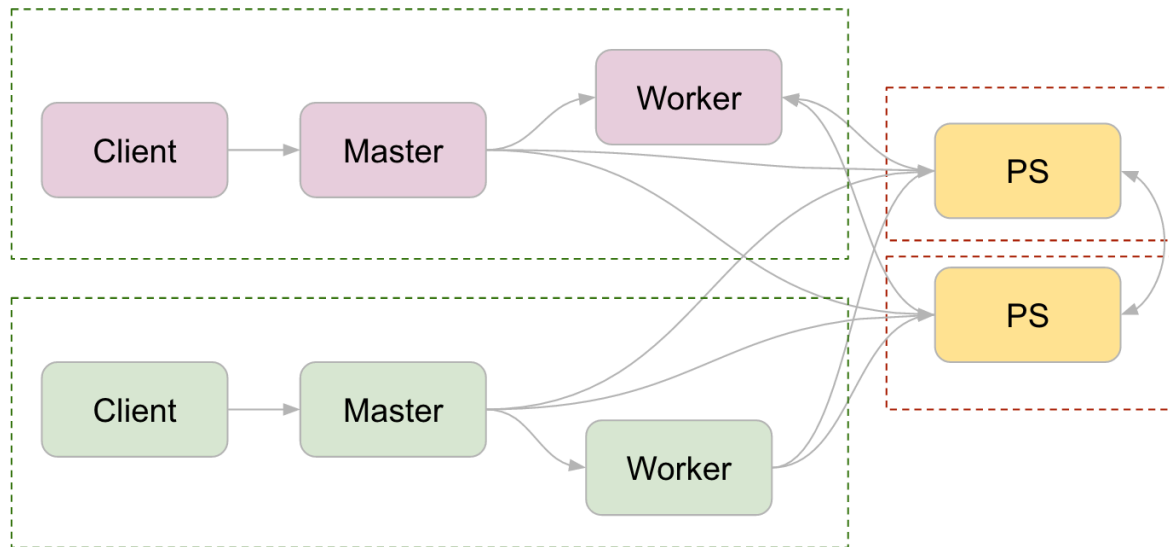
Then, start your server. Since worker and parameter servers (ps jobs) usually share a common program, parameter servers should stop at this point and so they are joined with the server.

```
1 # Construct the cluster and start the server
2 ps_spec = FLAGS.ps_hosts.split(",")
3 worker_spec = FLAGS.worker_hosts.split(",")
4
5 cluster = tf.train.ClusterSpec({
6     "ps": ps_spec,
7     "worker": worker_spec})
8
9 server = tf.train.Server(
10     cluster, job_name=FLAGS.job_name, task_index=FLAGS.task_index)
11
12 if FLAGS.job_name == "ps":
13     server.join()
```

Afterwards, your code varies depending on the form of distributed training you intend on doing. The most common form is between-graph replication.

### Between-graph Replication

In this mode, each worker separately constructs the exact same graph. Each worker then runs the graph in isolation, only sharing gradients with the parameter servers. This set up is illustrated by the following diagram. Please note that each dashed box indicates a task.



You must explicitly set the device before graph construction for this mode of training. The following code snippet from the Distributed TensorFlow tutorial demonstrates the setup:

```
1 with tf.device(tf.train.replica_device_setter(
2     worker_device="/job:worker/task:%d" % FLAGS.task_index,
3     cluster=cluster)):
4     # Construct the TensorFlow graph.
5
6     # Run the TensorFlow graph.
```

## Requirements To Run the Examples

To run our examples, Jinja templates must be installed:

```
1 # On Ubuntu
2 sudo apt-get install python-jinja2
3
4 # On most other platforms
5 sudo pip install Jinja2
```

Jinja is used for template expansion. There are other framework-specific requirements, please refer to the README page of each framework.