Distributed TensorFlow

David Simmons

November 8, 2017

Abstract

This document summarizes the fundamentals of distributed TensorFlow training in section 1 and discusses the code and its design in section 2.

1 Background

In the following subsections, we overview the fundamentals of distributed TensorFlow training.

1.1 Workers and Parameter Servers

The typical approach to implementing a distributed TensorFlow [1] solution is to split the problem across two process types:

1. a stateful process, the parameter server (PS);
2. a stateless process, the worker.

The PSs generally deal with low complexity aspects of the problem (e.g., storing, updating, and distributing model parameters), while the workers deal with the computationally intensive aspects of the problem (e.g., performing forward and back propagation).

1.2 Model Distribution Paradigms

There are two standard approaches that may be taken when distributing the computationally intensive parts of the graph to the workers [1]:

1. model replication (also called data parallelism), where the entire computational model is copied to each GPU/worker and each GPU/worker operates on a subset of the data;
2. model parallelism, used when the computational model cannot be stored by each GPU/worker, so the model is partitioned into submodels and each GPU/worker operates on a submodel.

Owing to the relative simplicity of model replication, it tends to be the implementation that is most commonly focused on.

1.3 Single or Multiple Clients

Each worker defined by the client will be able to interact with each parameter server, and vice versa. The PSs and workers are coordinated by the client they have been instantiated by. In the context of model replication, the number of clients present determines whether our system is one of two types:

1. in-graph replication, where the system is coordinated by a single master client;
2. between-graph replication, where the system is coordinated by multiple clients - often one for each worker task.

In-graph replication tends to work well when the number of graph replicas (i.e., workers) is small. However, as this number grows, the work load on the client grows too. This can result in the single master client getting bogged down by the coordination of lots of worker tasks, resulting in slower training times. The solution is to increase the number of clients and employ between-graph replication [1].

1.4 Synchronous vs Asynchronous Workers

For multi GPU architectures, the computations can be one of two types [1]:

1. synchronous;
2. asynchronous.

Synchronous operations occur when the calculations from each GPU of a current training loop are accumulated and used to update the parameter server before the next training loop begins. See [2] for an example implementation. The issue with this approach is that GPUs are necessarily blocked from operation until the slowest GPU has completed its computation. To resolve this issue, asynchronous GPU computations can be performed. One approach to asynchronous operations is to assign each worker to a subset of the GPUs, and once each worker has finished its computation it independently updates the parameter server's state. The issue with this approach is that slow workers tend to update using stale data. With that being said, asynchronous computations tend to outperform synchronous ones [1].

2 Designing the Code

In this section, we overview the design/development of a distributed TensorFlow solution. The author's computer had a GEFORCE 610M, which has a Cuda compute capability of 2.1. TensorFlow requires a Cuda capability of 3. For the solution, we provide an option to utilize the Xception convolutional neural network (CNN) model [3, 4] or the standard TensorFlow tutorial CNN solution [2] (this is because the Xception model would not run on the author's machine). For our data set, we use the the CIFR-10 data set [5, 6].

The trainer defaults to a three server model: a PS operating on `localhost:2222` and two workers operating on `localhost:2223` and `localhost:2224`. To activate the default model, run the scripts

```
python main.py \
--job_name=ps --task_index=0
```

```
python main.py \
--job_name=worker --task_index=0
```

and (note, only the first worker will operate if `data_loader.maybe_download_and_extract()` (discussed below) has not fully executed)

```
python main.py \
--job_name=worker --task_index=1
```

After parsing the inputs, if the server is the chief worker (i.e., the worker operating on task 0) the script will call `data_loader.maybe_download_and_extract()` (line 276 of `model_inputs.py`) at the beginning of `main()`, which will determine whether a local directory to the CIFR-10 data set exists in `./trainer_functions_data` of the local machine. If it does not, `data_loader.maybe_dow` will download the binary from `http://www.cs.toronto.edu/~kriz/cifar-10-binary.tar.gz`. Once this has downloaded, all other workers can be run. A cluster specification dictionary will then be built from the parsed data, and used to set up a local server (for the default setup, three servers in total will be created: one PS and two workers).

```
  # Create a cluster from the parameter server and worker hosts.
cluster = tf.train.ClusterSpec({"ps": ps_hosts, "worker": worker_hosts})
  # Create and start a server for the local task.
server = tf.train.Server(cluster,job_name=settings.FLAGS.job_name,
                                                    task_index=settings.FLAGS.task_index)
```

If the server is a PS task, the script will be blocked at `server.join()` (line 27 of `main.py`) sitting idly waiting for the workers to forward parameter updates to it; otherwise, the worker will begin operation.

Before the workers construct the graph, the script will ask whether the Xception model is to be used. After this, `model_inputs.distorted_inputs()` (line 148 of `model_inputs.py`) will be called, which will construct a TensorFlow queue of images. It then randomly distorts the example images in numerous ways (this allows for a more diverse sample set). If Xception was chosen, it will also re-size (make larger) the images so that they are compatible with Xception. `model_inputs.distorted_inputs()` will then call `model_inputs._generate_image_and_label_batch()` (line 109 of `model_inputs.py`), which will generate a corresponding queue of batches that can be dequeued by each of the workers. The queue allows for multithreaded data retrieval by making sure that a batch is always available to any of the threads that are running. The workers then construct the computational graph and train the model using `tf.train.MonitoredTrainingSession()` in the following script.

```
with tf.train.MonitoredTrainingSession(master=server.target,
                                       is_chief=(settings.FLAGS.task_index == 0),
                                       checkpoint_dir="./tmp/train_logs",
                                       hooks=hooks) as mon_sess:
        prev_time = time.time()
        while not mon_sess.should_stop():
        # Run a training step asynchronously.
        # See `tf.train.SyncReplicasOptimizer` for additional details on how to
        # perform *synchronous* training.
        # mon_sess.run handles AbortedError in case of preempted PS.
                mon_sess.run(train_op)
                if mon_sess.run(global_step)%20 == 0:
                        duration = time.time() - prev_time
                        prev_time = time.time()
                        examples_per_sec = settings.FLAGS.log_frequency *
                                                        settings.FLAGS.batch_size / duration
                        print ("examples/sec: %d" % examples_per_sec + ",
                                                        loss: %f" % mon_sess.run(loss))
```

The TensorFlow method `tf.train.MonitoredTrainingSession()` creates a `MonitoredSession()` object, which handles the multithreaded queuing. When the worker is the chief, it initializes/restores the session from the most recent checkpoint, and also stores the checkpoints in `./tmp/train_logs`. When the worker is not a chief, it allows the workers to wait for the chief to begin operation.

References

[1] Distributed TensorFlow. `https://www.tensorflow.org/deploy/distributed`.

[2] Convolutional Neural Networks. `https://github.com/tensorflow/models/blob/master/tutorials/image/cifar10/cifar10_multi_gpu_train.py`.

[3] TensorFlow implementation of the Xception Model by Francois Chollet. `https://github.com/kwotsin/TensorFlow-Xception`.

[4] Xception: Deep Learning with Depthwise Separable Convolutions. `https://arxiv.org/abs/1610.02357`.

[5] The CIFAR-10 dataset. `http://www.cs.toronto.edu/~kriz/cifar.html`.

[6] Learning Multiple Layers of Features from Tiny Images. `https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf`.