Distributed training with Data Parallelism method in TensorFlow 2

Hoang-Quan Dang[1,2], Duc-Duy-Anh Nguyen[1,2], and Trong-Hop Do[1,2]

[1] University of Information Technology, Ho Chi Minh City, Vietnam
[2] Vietnam National University, Ho Chi Minh City, Vietnam
{18520339,18520455}@gm.uit.edu.vn, hopdt@uit.edu.vn

Abstract. Deep learning is becoming more and more popular, especially well suited for large data sets. Besides, deep learning network training also requires vast computing power. Taking advantage of the power of GPU or TPU can partly solve the massive computing of deep learning. However, training an extensive neural network like Resnet 152 on an ImageNet database of about 14 million images is not easy. That's why in this article, we are talking about not only leveraging the power of one GPU but also leveraging the power of multiple GPUs to reduce the training time of complex models by data parallelism method with two approaches Multi-worker Training and Parameter Server Training on two datasets flower and 30VNFoods.

Keywords: Distributed computing · Data parallelism · deep learning

1 Introduction

Deep Neural Networks (DNNs) have been the main force behind the most recent advances in Machine Learning. From progress in image processing, speech recognition, and forecasting, DNNs solve challenges in a diverse set of use cases. Over the past few years, advances in deep learning have driven tremendous progress in image processing, speech recognition, and forecasting. Recent advances are mostly due to the amount of data at our disposal, the fuel that keeps the Deep Learning engine running. Thus, we need to scale out model training to more computational resources is higher than ever. Training these neural network models is computationally demanding. Although, in recent years, significant advances have been made in GPU hardware, network architectures, and training methods, the fact remains that network training can take an impractically long time on a single machine. Fortunately, we are not restricted to a single machine: a significant amount of work and research has been conducted on enabling the efficient distributed training of neural networks. Distributed optimization and inference are becoming famous for solving large-scale machine learning problems. There are two ways to distribute training: data parallelization or model parallelization. In data parallelization, there are asynchronous and synchronous aggregations based on the execution timing of the operations. In this work, we will execute two methods of data-parallelization is, multi-worker and parameter-severs, by TensorFlow.

2 Dataset

In this article, we used the 30VNFoods [1] dataset that includes collected and labeled images of 30 famous Vietnamese dishes. This dataset is divided into training (17,581 images), validation (2,515 images), and testing (5,040 images) sets. In addition, we also used a small dataset with about 3700 images of flowers, which includes 5 folders corresponding to 5 types of flowers (daisy, dandelion, roses, sunflowers, tulips).

3 Method

Distributed training deep learning models saves a lot of training time, bringing many benefits to research and applications helping researchers to new ideas easily and quickly. There are two main methods of distributed training today, model parallelism and data parallelism.

The model parallelism is a distributed training strategy where the model divides parts among many processing devices. It helps a complex model have many hidden layers and weights, making it impossible to fit into a device's memory. Each device carries a set of model classes through which data flows and computations are shared and compiled. The performance of the model parallelism, in terms of GPU utilization and training time, is highly dependent on how the partitioned model is used to perform forward and backward passes.

The data parallelism method is a distributed training strategy that divides the training dataset into parts for the processors, each containing a model replica. Each processing device receives batches of different data, performs forward and backward passes, and shares the update weights with other devices, which may be *synchronous* (the training steps are synced across the workers and replicas) or *asynchronous* (the training steps are not strictly synced).

3.1 Multi-worker Training

This is a synchronous data parallelism training method where we send different data batches to each Worker. Each device has a full replica of the model, and it is trained only on a part of the data. The forward pass begins at the same time in all of them. They all compute different output, gradients, and synchronous update weights.

All the devices communicate with each other, and they aggregate the gradients using the All-Reduce algorithm. When the gradients are combined, they are sent back to all devices. And each device continues with the backward pass, usually updating the local copy of the weights. The following forward pass does not begin until all the variables are updated. And that's why it is synchronous. All the devices have the same weights at each point in time, although they produced different gradients because they trained on other data but updated from all the data.

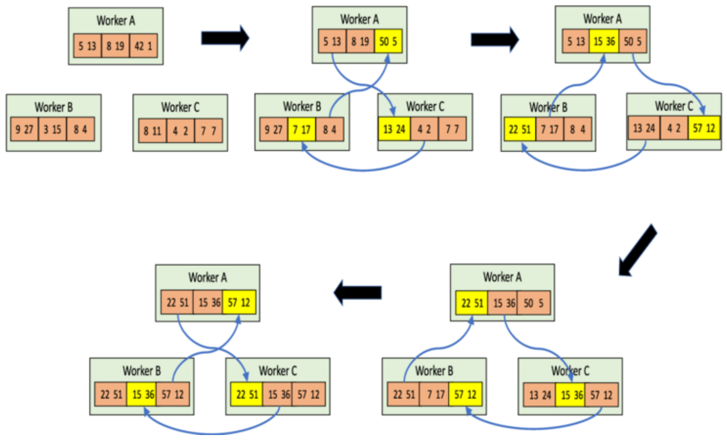


Fig. 1. The ring all-reduce method

All-Reduce is a parallel algorithm that aggregates the target arrays from all processes independently into a single array. Aggregation can be either concatenation, summation, or any other operation that allows for independent parallel processing of arrays. When applied to the gradient of deep learning, the average is the most common operation. To tweak performance of multi-worker training, TensorFlow provides multiple collective communication implementations:

- RING implements ring-based collectives using gRPC as the cross-host communication layer.
- NCCL uses the NVIDIA Collective Communication Library to implement collectives.
- AUTO defers the choice to the runtime.

The best choice of collective implementation depends upon the number of GPUs, the type of GPUs, and the network interconnect in the cluster. For example, we used the Ring All-Reduce 1 in our specific setup case. The gradient is divided into consecutive blocks at each device. Simultaneously, each block is updated using the previous device and updates the next, making a ring pattern.

In synchronous training, the cluster would fail if one of the Workers fails and no failure-recovery mechanism exists. Distribute strategy in TensorFlow comes with the advantage of fault tolerance in cases where Workers die or are otherwise unstable. We can do this by preserving the training state in the distributed file system of your choice, such that upon a restart of the instance that previously failed or preempted, the training state is recovered. Other Workers will fail when a Worker becomes unavailable (possibly after a timeout). In such cases, the unavailable Worker needs to be restarted, and other Workers that have failed.

3.2 Parameter Server Training

Data parallelism does not need to be synchronous. In asynchronous data parallelism, each worker also computes the gradients from a slice of the input data but makes updates to the parameter in an asynchronous fashion. Compared to synchronous strategies, asynchronous training has the benefit of fault tolerance because the workers are not dependent on one another. Parameter Server Training is a common asynchronous data parallelism method to scale up model training on multiple machines. A Parameter Server Training cluster consists of Workers and Parameter Servers. And as the name suggests, the model's parameters are stored on the Parameter Servers. On the other hand, Workers read and update these variables independently without synchronizing with each other. This is why sometimes parameter server-style training called *Asynchronous training*.

The way these asynchronous updates work is that each Worker independently fetches the latest parameter values from the Parameter Servers and computes the gradients based on a subset of training samples. Workers then send gradients back to the Parameter Servers, which update the parameters with those gradients. Each Worker does this independently. Because the Workers are not waiting for each other like they are when we do synchronous data parallelism, *Asynchronous training* scales well to a larger number of Workers where the training Workers might be preempted by high priority production jobs or maybe a machine goes down for maintenance.

The downside of this approach is that Workers can get out of sync as they could be computing parameter updates based on stale values which might delay convergence. In Parameter Server Training, the variables are stored on the Parameter Servers only. Workers do not store variables. They only perform the computation. For example, a cluster with one Parameter Server and two Workers. If the steps per epoch were 10, then one epoch would be complete when 10 gradient updates have occurred. And these updates do not need to be balanced across the two Workers. Because this is asynchronous, it's entirely possible that each Worker computes 5 gradient updates in the epoch or that one of them does 6 while the other only does 4, ...

There are two setup types for the Parameter Server Training: Multi-Client Setup and Single-Client Setup. Multi-Client Setup includes client machines controlling one worker machine. There are some disadvantages for this Setup: it is difficult to coordinate workers, implement, and get worker consensus.

Single-Client Setup is a solution. Distributed training In TensorFlow 2 involves a *cluster* with several *jobs*, and each of the *jobs* may have one or more *tasks*. When using Parameter Server Training, it is recommended to have: one *coordinator* job (which has the job name *chief*), multiple *worker* jobs (job name *worker*), multiple *parameter server* jobs (job name *ps*). The missions of the *coordinator* are creating resources, dispatching training tasks, writing checkpoints, and dealing with task failures. The *workers* and *parameter servers* are to listen for requests from the *coordinator*.

Another essential function is Variable Sharding, which refers to splitting a variable into many smaller variables, called *shards*, and storing them on different

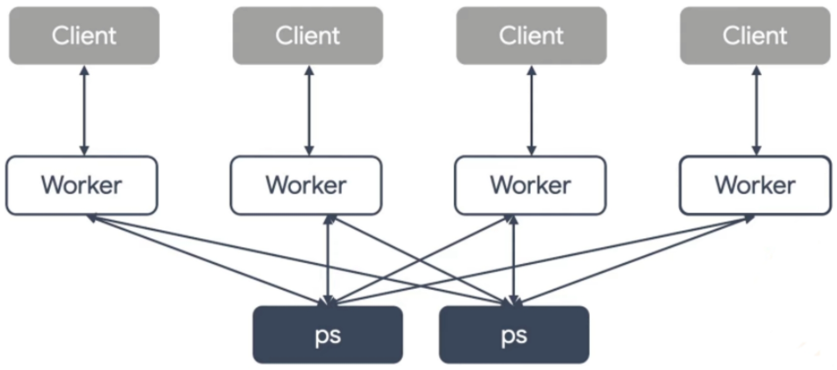


Fig. 2. Multi-Client Setup

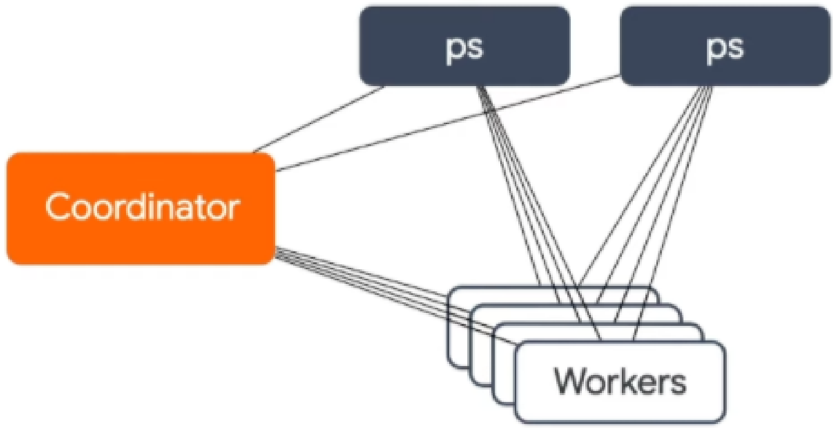


Fig. 3. Single-Client Setup

Parameter Servers. Variable Sharding plays an important role in balancing the load among Workers to achieve greater bandwidth efficiency for the Cluster. So it is very useful in distributing the computation and storage of a single variable across multiple Parameter Servers.

4 Experimental Setup

4.1 Multi-worker Training

We implemented Multi-worker Training on two machines with one GPU for each, which are NVIDIA RTX 2080 Super with 8GB VRAM and NVIDIA RTX 2060 6GB VRAM. To connect the two machines containing the above 2 GPUs, we have set up two connection ways.

The first is via a VPN tunnel to go through the Internet to connect machines in 2 different networks. Specifically, for Worker 1 with the RTX 2080 Super GPU, we set the static IP as 10.10.10.1, and 10.10.10.5 for Worker 2 with the RTX 2060 GPU. We also set up Port Forwarding on the routers of each Worker to forward the port from one network to the other. In addition, we also have to set up Firewall rules on both routers for communication purposes 4.

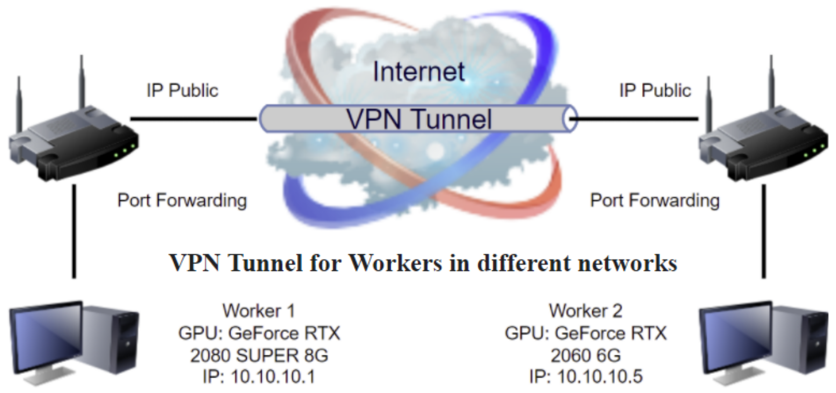


Fig. 4. Multi-worker via VPN Tunnel

The second way is via LAN with bandwidth up to 1 Gbps between the two Workers. Specifically, for Worker 1 with the RTX 2080 Super GPU, we set the static IP as 192.168.1.1, and 192.168.1.2 for Worker 2 with the RTX 2060 GPU. Because of connecting in a LAN environment, we don't need to set up Port Forwarding on the router 6.

As for model training, we used the MobileNetV2 model to train from scratch without using transfer learning. With the flowers dataset, we set the model parameters with the 224x224 image size with 10 epochs, then the global batch size is 64 corresponding to the batch size for each worker is 32. The 30VNFoods dataset is similar to the settings of the flowers dataset but only with 2 epochs because this dataset is much larger than the flower dataset and overflows the VRAM of the RTX 2060 GPU

4.2 Parameter Server Training

In this work, we set up a Cluster with five machines in a LAN network. Specifically, we used two laptops for the Parameter Server connecting to the Cluster via Wifi by 5GHz radio with the static IPs are 192.168.1.3 and 192.168.1.4, respectively. And two machines with one GPU for each as Workers, same as the Multi-worker Training above with the static IPs are 192.168.1.1 and 192.168.1.2, respectively. Finally is a machine as coordinator with 192.168.1.5 for the IP. For the model training, we used the same configuration as the Multi-worker Training above

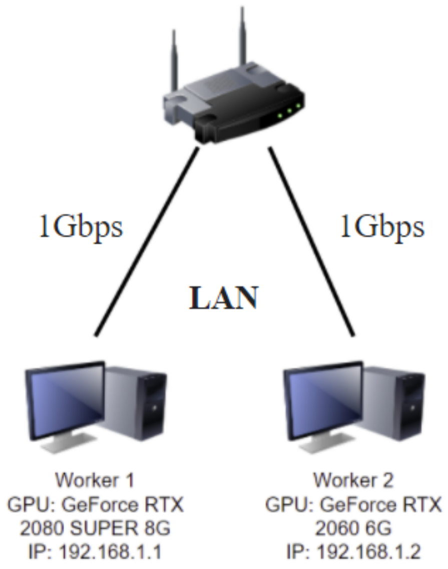
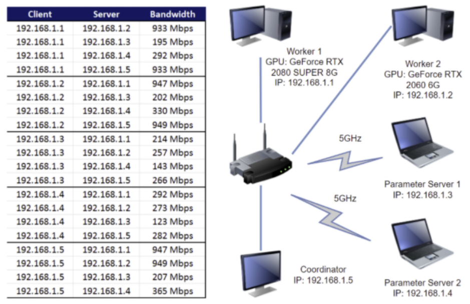


Fig. 5. Multi-worker via LAN



Client	Server	Bandwidth
192.168.1.1	192.168.1.2	933 Mbps
192.168.1.1	192.168.1.3	195 Mbps
192.168.1.1	192.168.1.4	292 Mbps
192.168.1.1	192.168.1.5	933 Mbps
192.168.1.2	192.168.1.1	947 Mbps
192.168.1.2	192.168.1.3	202 Mbps
192.168.1.2	192.168.1.4	330 Mbps
192.168.1.2	192.168.1.5	949 Mbps
192.168.1.3	192.168.1.1	214 Mbps
192.168.1.3	192.168.1.2	257 Mbps
192.168.1.3	192.168.1.4	143 Mbps
192.168.1.3	192.168.1.5	266 Mbps
192.168.1.4	192.168.1.1	292 Mbps
192.168.1.4	192.168.1.2	273 Mbps
192.168.1.4	192.168.1.3	123 Mbps
192.168.1.4	192.168.1.5	282 Mbps
192.168.1.5	192.168.1.1	947 Mbps
192.168.1.5	192.168.1.2	949 Mbps
192.168.1.5	192.168.1.3	207 Mbps
192.168.1.5	192.168.1.4	365 Mbps

Fig. 6. Parameter Server Training

5 Result and Discussion

Table 1. The results of Data Parallelism models

Training method	**Dataset**	**Connection**	**Avg. s/epoch**
Single-worker	**flowers**	**LAN**	**14**
Multi-worker	flowers	LAN	18
Multi-worker	**flowers**	**VPN Tunnel**	**635**
Multi-worker	30VNFoods	LAN	184
Parameter Server	30VNFoods	LAN	115

As the table 1, we can see that Single-worker Training is faster than Multi-worker with flowers dataset, specifically 14s/epoch with 18s/epoch and Multi-worker via a VPN tunnel environment trains very slowly with 635s/epoch. Besides, training with Parameter Server approach is 37.5% faster than Multi-worker, specifically 115s/epoch compared to 184s/epoch.

Many or most models will run faster on a single GPU than they will when split across multiple GPUs if it's possible to fit on a single GPU. Because the latency per step is lower bounded by the most expensive path through a DAG where each node is a kernel execution, whose cost is the time that kernel takes, and each edge is a data dependency, whose cost is the time it takes to transfer the data from the computation that produces it to the computation where it is consumed. If the model runs completely on 1 GPU, the data transfer times are all zero, and the overall execution time is just the sum of the kernel execution times. GPU computations can be fast, from a few microseconds to a few milliseconds. By contrast, data transfers by RPC (Remote Procedure Call) and even via PCIe bus are relatively slow.

Consider the following example. We want to do a dot product of two large vectors of $10\hat{9}$ float32 values, all of which reside on GPU0 to begin with. We also have GPU1 available on the same PCIe bus with a bandwidth of 6GB/s. Naively it seems like partitioning the computation between devices might be a good idea. Suppose both GPUs have an internal memory bandwidth of 200GB/s. Based on that limitation, doing the dot product just inside GPU0 should take $(10^9 * 4\text{Bytes})/(200 * 10^9) = 0.02s$. Half of the computation would take 0.01s. But copying 1/2 of the data between GPUs at 6GB/s would take $(0.5 * 10^9 * 4\text{Bytes})/(6 * 10^9) = 0.33s$, much longer than the on-GPU computation. The disadvantage is even greater if the data transfer time is even slower, say 10Gbps over a typical RPC network link.

Back to our specific case: We are trying a separate ps/worker architecture. For the reasons just cited, it should be clear that when using a single worker, it will be slower; this sort of architecture is only advantageous if you use a lot of workers so that the wall-clock training time is reduced.

But in our case, the time is slower, and by our measurement, the network is only 40% busy. There could be lots of reasons. 1Gbps is a slow network by contemporary standards. It's usually hard to get more than 80% of the theoretical capacity of a network, and maybe there's other traffic going through it competing with our workload. There may be other jobs timesharing with our TF jobs on both machines competing for kernel resources that interfere with RPC latency. Suppose we are getting 80% of the theoretical capacity going in each direction when there's something to send and measuring overall capacity as bidirectional. In that case, we could see 40% bandwidth consumption when our job is completely network bound.

On the other hand, in a distributed system, it is possible that the bottleneck could be CPU / GPU, Disk, or Network. Nowadays, networks are really fast, and in some cases faster than disk. Depending on the Workers configuration, CPU / GPU could be the bottle neck. So it really depends on the configuration of our hardware and network.

6 Conclusion and Future Work

This article has established data parallelism distributed training method through Synchronous Data Parallelism, and Parameter Server Training approaches. For the Synchronous Data Parallelism approach, we tested with 2 GPUs and compared it with just training with one GPU on the flowers dataset. The result is that training on one GPU is not much faster (14s/epoch vs. 18s/epoch) and connecting the GPU through VPN Tunnel dramatically slows down the model's training with 635s/epoch. This we explained in part 1, mainly because the network slowed down exchanging gradient results between GPUs. Besides, we have experimented with Parameter Server Training and compared it with the Multi-worker Training approach. We tested both approaches on the 30VNFoods dataset and performed on the same two workers (GPU). The result is that the Parameter Server Training approach is about 40% faster than the other approach. Thus, we can see that the approaches in data parallelism are significantly affected in connecting GPUs. The number of GPUs also dramatically affects the training speed of the approaches. This close. With device and network limitations, we have not been able to exploit the full power of GPUs with only taking advantage of 14-23% of the power of each GPU.

In the future, we plan to establish a model parallelism approach and compare it with the data parallelism approach. In addition, we also implemented data parallelism on more GPUs and placed them in a better network environment to get the most out of distributed training.

References

1. Trong-Hop Do, Duc-Duy-Anh Nguyen, Hoang-Quan Dang, Hoang-Nhan Nguyen, Phu-Phuoc Pham, and Duc-Tri Nguyen. 30vnfoods: A dataset for vietnamese foods recognition. pages 311–315, 07 2021.