Distributed Learning using Iterative Pair-wise Averaging

Resource-/Time-Dependent Learning (CSC591)

Srikar Chundury
North Carolina State University
Raleigh, USA
schundu3@ncsu.edu

Abstract

This project explores the trade-off between network activity and accuracy of models trained in a distributed fashion, particularly in Ring algorithms. Using Horovod [10], a simple neural network is trained on the MNIST [7] dataset, and the network I/O activity between Spark master and Spark drivers is captured and analyzed. Further, a variant of the ring-reduce algorithm is used to average the learned weights only of a subset of workers (using the number of backward passes) avoiding the network-costly all-reduce averaging step. As one would expect, averaging within a group of workers involves lesser network I/O, but it affects the accuracy of the model trained. But, choosing an appropriate number of passes (averaging within a preset number of subsets) and increasing number of iterations, reduces the accuracy loss. Therefore, this approach reduces network I/O while achieving an accuracy that is comparable to that of it's all-reduce version.

1 Introduction

Deep neural network training can be performed faster using high-performance systems that utilize GPUs. GPUs are costly and are often limited due to their memory involved. For instance, Nvidia's most powerful GPU A100 has a shared memory of 80 GB. Since these systems utilize multiple cores and threads to execute a task, they are often referred to as parallel systems. Hence, the need to be able to span a task across machines comes into play.

In distributed systems, a complex task is executed using multiple machines (workers) usually in a master-slave fashion to aggregate results from each individual result. Some of the famous distributed system frameworks are Hadoop [6], Spark [11] and Ray [8]. We use Apache Spark for the distributed needs of this project.

1.1 Parallelism

Parallel systems leverage multiple threads within a system, when combined with the power of distributed systems allow us to utilize threads across machines (workers). It is known that distributed systems utilize map-reduce algorithms. In brief, if a complex task can be divided into smaller tasks, each task can be run in parallel (map step), and then the result can be agglomerated (reduce step). In the realm of neural networks, parallel training can be done in two ways: data parallelism and model parallelism.

1.1.1 Data Parallelism: Involves dividing training data into multiple parts. The same model is then trained on each part in parallel (map phase). At the end of each iteration, after each part shares the learned weights, a global averaging is performed (reduce phase). This type of parallelism is typically used when training data is too big and takes a long time to train in finite number of iterations.

1.1.2 Model Parallelism: Model Parallelism: Involves dividing the machine learning model itself into multiple parts. Each part is trained using the entire training data. This type of parallelism is used when the model in question is too large to fit into a single machine.

This project focuses on data parallelism. It involves a synchronous step (reduce phase) where weights learned by different parts (workers) are averaged before the next iteration. There exist multiple reduce algorithms in the deep learning realm. We explore a variant of ring algorithms.

My contributions:

1. A custom multi-platform (works with ARM and x86 machines) docker image with Horovod, PyTorch and Spark for use in distributed training. It has been made publically available on docker hub [1].
2. Custom docker compose files to raise a high performance spark cluster up easily.
3. A technique to collect and compare docker stats. We focus on network I/O in the project.
4. Changed the default all-reduce algorithm to take into account local gradient averaging (when number of passes is 1, it becomes iterative pair-wise, as the title describes).
5. Show a trade-off between network I/O and number of backward_passes (of workers), and how it impacts the trained model's accuracy.

6. Code and results with Jupyter notebooks are uploaded to NCSU GitHub. https://github.ncsu.edu/schundu3/RTDL

2 Related Work

2.1 Parameter Server

Historically, the averaging step happened at a central server called the parameter server. This server gets the weights from all workers at different times. Then all weights are averaged here and sent back to all workers for subsequent iterations. Pictorially, the same can be seen in figure 1.

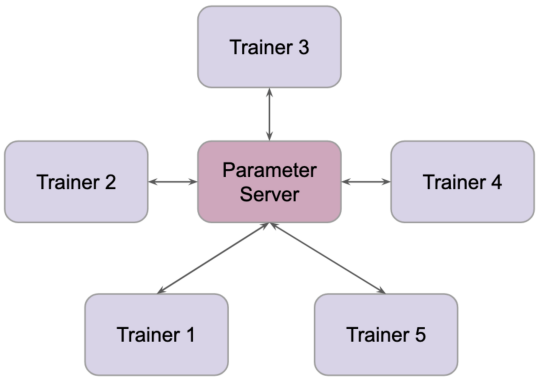

Figure 1. Central server that aggregates weights after every iteration. [2]

But, this technique quickly went out of style as it suffers from the server's bottleneck problem as it becomes the central point of failure.

2.2 Ring Algorithms

To overcome the drawbacks that parameter server approach suffers from, Huawei [5] first came up with an approach where there's no central server anymore. The workers are arranged in a circular ring fashion. A random worker is chosen as the leader where the averaging step is performed. The weight updates are then propagated to all workers, one after the other, in a cyclic fashion, as seen in figure 2. Uber later implemented this approach in a framework called Horovod [10]. In an event where the chosen leader dies (killed or becomes unreachable), another random worker is chosen as the leader using Big Data algorithms like the bully algorithm.

2.2.1 Ring all-reduce algorithm: Today's Horovod, uses this variant of ring reduce algorithm where the averaging step uses weight updates from all the workers. For instance, if the first worker is chosen as the leader, the weight is propagated as seen in figure 2.

Weight update propagation occurs in two cycles. In the first cycle, each worker sends its updates to the next

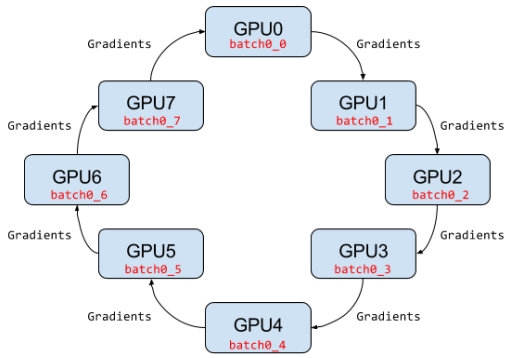

Figure 2. Ring model with all-reduce weight updates after each iteration. [4]

and so on until the first one gets all of them. Then, the first worker averages it out and starts the second cycle. In the second cycle, the updated gradients are sent to all the workers. Therefore, this approach is very network activity heavy.

This project is an attempt to reduce this network I/O while still achieving competitive accuracies.

Uber previously explored this technique using local gradient aggregation, but quickly moved away from it because of issues with lowered accuracies [3].

3 Proposed Method

Setting Horovod's number of backward passes to one during the distributed training forces, weight averaging between every two workers. This way, as expected, the network activity is reduced along with the accuracy of the resultant model. Increasing the number of iterations effectively recovers the lost accuracy while not increasing the network I/O too much. I also test out multiple combinations of number of passes for different number of workers to estimate in an attempt to find the ideal number of passes so that network I/O is reduced significantly while not severely affecting the accuracy.

4 Implementation

4.0.1 Setup: We use Apache Spark as the base for performing distributed compute. We use docker citemerkel2014docker as the resource manager, but statically setup for different number of workers for different experiments. The base image used for spark master, worker and the driver is a multi-platform compatible one. We deploy the spark cluster on an Apple Macbook pro M1 (ARM based) that has 32 GB of RAM and 16 compute cores. All workers together share 6 GB of RAM (example- 2 workers setup have 3 GB each, 6 workers setup have 1GB each) via docker using docker-compose. And, we use PyTorch [9] with Horovod to submit tasks

to the spark cluster. This section has taken significant amounts of time and efforts, mainly because there was no open-sourced version available online. Hence, this can be something others can readily use for their distributed training setup with Spark and Horovod.

5 Testing

Once the git repo is pulled, with docker installed on your machine, it is as simple as running "run_cluster.sh" script to bring up the Spark cluster. We can then run "run_driver.sh" to run the Spark driver that is setup to submit a distributed training task in Spark's own docker virtual network.
Then we use "run_program_in_driver.sh" with any program that is written using Horovod and PyTorch to submit and start the training process. It is important to note that the number of partitions (shuffles) while submitting this distributed task relies on the number of workers present in the Spark environment; the number of data partitions should be equal to the number of statically-setup workers. All these scripts have been collated to run in one-shot using "measure.sh" that captures the model training output and the docker network stats into separate files.

6 Experiments

As a proof of concept, the MNIST dataset is used to train a rather simple neural network with 3 convolution layers. Since our major focus is on network statistics and not the machine learning model itself, this model is taken right out of Horovod's git source code and then altered to work for a different number of workers.

#Workers	#Passes	#Iterations
2	1	12
3	1,2	12,15
4	1,2,3	12,15,18
5	1,2,3,4	12,15,18,21
6	1,2,3,4,5	12,15,18,24

Table 1. All configurations of experiments.

We performed distributed training for various configurations (as seen in table 1) of number of workers and number of passes, and then measured and compared network statistics (measured at 1 second interval using docker). Due to the limitations of our test setup size and compute power, we present this project as a proof-of-concept and train the model using a subset of the MNIST dataset only.

7 Results

The overall network activity along with the accuracy achieved is shown in the table 2. As the accuracy usually goes down with reducing number of backward passes for the same number of iterations. We gradually increase the number of iterations in an attempt to recover the accuracy lost. The overall network activity is the sum of data (collected at 1 second intervals) sent and received across all the workers, the driver and the master. Hence, it's in the order of few TBs. Since, it's only a comparison metric, only relative sizes are of relevance to us. **??** lists results of the experiments performed. As expected, reducing the number of backward passes reduces the network overhead (accuracy gets worse with the same number of iterations). Increasing number of iterations till similar (to all-reduce) accuracies are seen still reduces the overall network I/O.

#Workers	#Passes,#Iterations	Overall network activity (TB)	Accuracy
2	2,12	105.59545628547289	95.81%
2	**1,15**	**105.5264167976342**	**96.83%**
3	3,12	105.37346571922211	91.43%
3	2,15	105.29663928031844	91.85%
3	**1,18**	**105.03285879135152**	**95.41%**
4	4,12	104.32535861968955	78.49%
4	3,15	104.25853464126584	87.27%
4	2,18	104.02306207656859	89.02%
4	**1,21**	**103.29913792610171**	**90.38%**
5	5,12	125.59344117164736	80.334%
5	4,15	101.99513598442127	90.70%
5	3,18	101.59431415558072	89.10%
5	2,21	101.17704673767324	89.72%
5	**1,24**	**100.21186826706173**	**89.23%**
6	6,12	107.1364613914505	66.87%
6	5,15	98.37390358925066	90.09%
6	4,18	98.01313048362785	90.40%
6	3,21	97.18158767700298	91.97%
6	2,24	96.22509217262373	86.45%
6	**1,27**	**94.74648439407457**	**94.43%**

Table 2. Results for different configurations. The rows in bold are the ones with minimum network overhead.

We see that as number of passes is reduces the network activity also reduces significantly. But it also negatively affects the accuracy of the model. This accuracy loss can be regained by increasing the number of iterations. The intent is find the right number of passes and the number of iterations such that overall network activity is reduced when compared to all-reduce while not significantly impacting the accuracy.

The core of the proposed technique can be summarized in the figure 3 where experiments are for a 5 worker Spark cluster setup. The left most bar is the all-reduce version that achieves an accuracy of 91%, but when we reduce the number of backward passes to 1 (iterative pairwise averaging) we see that the network activity is reduced significantly, but it also affects the accuracy (down to 31%). We can regain this lost accuracy by increasing

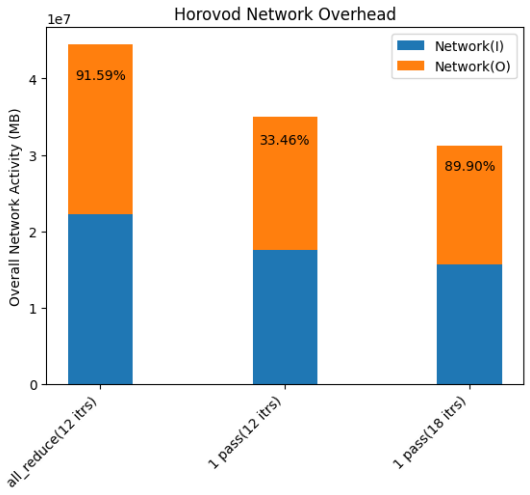

Figure 3. Network activity comparison between 3 scenarios (all-reduce, iterative averaging with 12 iterations and iterative averaging with 18 iterations). The percentages printed on each bar are the accuracies of the trained model.

the number of iterations, as the last bar (rightmost) bar shows. With 18 iterations, we get 89% while reducing the network activity.

8 Conclusion

This project demonstrates one way to minimize network I/O activity while minimizing the accuracy loss in ring reduce algorithms during distributed training. I think it is worth pursuing further research in this direction, as it could be particularly useful for applications where network activity is of primary concern.

9 Acknowledgement

I thank Dr. Jung-Eun Kim for her guidance and support throughout the semester.

References

[1] Docker hub image. https://hub.docker.com/layers/chundury/horovod-pytorch-spark-multi-platform/latest/images/sha256-18a077617dc0669adf238a919973db03c684d96836290a4bb5a679431847c5f7?context=repo.

[2] Flyte - image source. https://docs.flyte.org/projects/cookbook/en/stable/auto/case_studies/ml_training/spark_horovod/index.html. Accessed: 2022-11-21.

[3] Local gradient aggregation. https://www.uber.com/blog/horovod-v0-21/. Accessed: 2022-12-5.

[4] Orielly - image source. https://www.oreilly.com/content/distributed-tensorflow/. Accessed: 2022-11-21.

[5] Ring reduce algorithm - huawei. http://learningsys.org/papers/LearningSys_2015_paper_14.pdf. Accessed: 2022-11-21.

[6] Apache Software Foundation. Hadoop.

[7] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[8] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging ai applications, 2017.

[9] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[10] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow, 2018.

[11] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, oct 2016.