Communication Cost in Distributed Deep Learning: A Comprehensive Survey
Group 8

Vinayak Kothari
vkothar3@asu.edu

Anirudh Kaushik
akaush21@asu.edu

Hasan Ahmed Faisal
hfaisal@asu.edu

May 1, 2021

Abstract

The explosion of data and an increase in model size has led to great interest in training deep neural networks on distributed systems.However, communication cost is one of the major issues when it comes to distributed training workloads.Due to this reason, many compression techniques have been proposed to reduce volume of transferred data. Even though many compression techniques are available, there is a lack of comprehensive information comparing the compression techniques used. In this paper, we aim to survey some of the most common compression techniques used with the help of parameter server architecture.We basically use some of the common state of the art quantization and sparsification methods available.

Keywords: Distributed Deep Learning, Compression, Quantization, Sparsification, Parameter Server, QSGD, TopK

1 Introduction

Deep learning (DL) has attained a lot of advances in recent years and it is being used by the world's leading researchers in Artificial Intelligence, Computer Vision, NLP. The explosion of data and an increase in model size has led to great interest in training deep neural networks on distributed systems. In particular, the data parallel approach is a widely used technique in the field of distributed computing. Standard machine learning algorithms are not able to provide accurate results in field of Computer Vision, NLP, Audio Processing , so deep learning was introduced which provided better accuracy and concrete results to deploy in real world, but to achieve this kind of accuracy these models are data hungry and takes days to train. So, there are typically two ways to increase speed of training a) Scale Up - Increase processing speed and storage capacity of single system, but getting such huge processor is costly not readily available. b) Scale Out - Distribute training process to multiple systems, and communicate with them efficiently to train models. In latter, setting of system is more tedious but after that computation is really easy and it is more cheap to build than Scale Up. It is more powerful and comprises greater storage capacity than scaling up system.

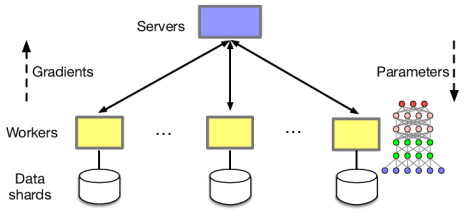

Figure 1: Parameter Server architecture

2 Problem Statement

Nowadays, it is quite common to use distributed systems in order to reduce training time of deep neural networks. Data parallelism is one such approach that speeds up the convergence of the model by distributing the training among the workers and utilizing each worker's compute power.Unfortunately, transfer of large data across different workers increasingly becomes communication bound due to which network becomes a bottleneck. In order to reduce the communication cost, many compression techniques are used which helps us speed up the training process by reducing the amount of data to be transferred. However, there is very less existing work which comprehensively compares and contrasts existing compression methods.Due to this reason, in this paper we provide a comprehensive survey of some of the compression techniques used to tackle communication cost using a parameter server(Data Parallelism).

3 Related Works

Optimizing communication cost is a well known problem in distributed deep learning and some state of the art models include Gradient Sparsification which was proposed by Aji et al [1]. In this algorithm they truncated the smallest gradients and transmitted remaining large gradients. Communication overhead was greatly reduced and 22% speed gain was achieved across 4 GPUs for neural machine translation without translation accuracy. Gradient Quantization method was proposed by DoReFa-Net [2] derived from AlexNet which reduced the bit length of weights, activation's and gradients to 1 ,2 6 respectively. It led to a 9.8% decrease in accuracy of model prediction.

4 System Architecture And Algorithms

4.1 Parameter Server Architecture

The Parameter Server consist of mainly two components servers and workers. In vanilla PS, server has specific Deep learning model and copies same model in all workers. Each worker has its own different dataset which is called data shards. It is similar to master slave model where server is master and workers

act as slave. As from **Figure 1** we can see that workers perform computation of model on their data shards and find loss generated by their model on that dataset, with help of backward propagation they find out gradients for model weights . On every iteration each worker push gradients to the servers. We have considered one iteration to be computation of 1 batch and weight updates of both worker and server. Server averages gradients received from workers and updates it's own model with help of these gradients. Now, server has updated model and workers pull this newly updated model weights, and update their own model,this processes continues till the model converges. There are mainly two ways in which gradient transfer can take place, synchronous PS and asynchronous PS.

4.1.1 Synchronous PS

In this type of gradient transfer, **every** worker pushes its gradients to the server and pulls newly calculated weights from server in each iteration. Server has to wait for all workers to push gradients and workers need to wait for the server to calculate weights and then pull new weights from the server. So, if some worker is slow it can slow down whole process. It has an advantage of better accuracy and less convergence time because each worker is training their model with most up to date weights.

4.1.2 Asynchronous PS

In this type of PS, each and every worker pushes their gradient to the server **independently** and pulls gradients from server **independently** in each training iterations. Server updates its model based on single set of gradients received from worker and the worker pulls gradients from the server. It has an advantage of lesser training time as workers push their gradients to the servers independently and never are blocked by other workers. But it suffers a disadvantage of low accuracy and more time to converge because only those workers receive new model weights who had pushed its gradients to the server, so some workers will update the server with stale gradients as they do not have up to date weights.

4.2 Compression Technique

We find out that in PS communication between worker and server is major bottleneck as for each vanilla PS iteration we need $2 * N$ data transfer where N is number of workers. From results, we find that for each iteration, we need to transfer total of 137MB data for an end to end VGG11 model. Hence, in case of slow network, system can slow down drastically increasing the training time from days to weeks. One thing to note here is that lossy compression techniques increase the number of epochs to converge. Total Convergence time in this case should be less than vanilla PS's convergence time. In order to reduce communication data we apply two compression techniques namely Quantized Stochastic Gradient Descent mentioned in [3] and Top-k sparsification mentioned in [4].

4.2.1 Quantized Stochastic Gradient Descent

QSGD is a family of lossy compression schemes which trade off low bit complexity and convergence time. It was proposed by Alistarh et al [3]. QSGD quantizes each component of the stochastic gradient via randomized rounding to a discrete set of values (i.e., code-words) that preserve the statistical properties of the original stochastic gradient. A gradient component g[i] is quantized to

$$f(x) = \begin{cases} ||g||_2.sign(g[i]).\frac{l}{s} & with \quad probability \quad p_i = \frac{s|g[i]|}{||g||_2} - l \\ ||g||_2.sign(g[i]).\frac{l+1}{s} & otherwise \end{cases}$$

where $||.||_2$ is Euclidean normalization of vector, $s \geq 1$ and $l \in N$ N are user-defined parameters, such that $0 \leq l \leq s$ and $\frac{||g[i]||}{||g||_2} \in [\frac{l}{s}, \frac{l+1}{s}]$

4.2.2 Top-k sparsification

Sparsification method selects less number of elements to be sent of the original stochastic gradient, resulting in a sparse vector. Top-k sparsification introduced by Shi, Shaohuai [4] select top K percentile elements with largest absolute value. These gradients are pushed to server along with their indexed and shape so that we are able to create matrix on server side. So if top k elements are N then we sent 2*N elements including their position.

5 Our work

Our work was based on data parallelism architecture in distributed deep learning and is divided into 6 parts.

1. **Method 1 - Vanilla Parameter Server**
2. **Method 2 - Gradient Compression in PS**
3. **Method 3 - Pushing and pulling gradient only**
4. **Method 4 - Gradient compression while pushing and pulling gradient**
5. **Method 5 - Compression and sparsification while pushing and pulling gradient**
6. **Method 6 - Reducing pulling and pushing rate in Method 5.**

.

Method 1: First we setup synchronous vanilla parameter server for which we used Pytorch Distributed and for communicating with worker and server Gloo back-end is used. We used two workers and one server initially, LeNet model is used with MNIST dataset and VGG11 was used with Cifar-10 dataset. Here, MNIST and Cifar-10 datasets were shuffled and divided into two parts.

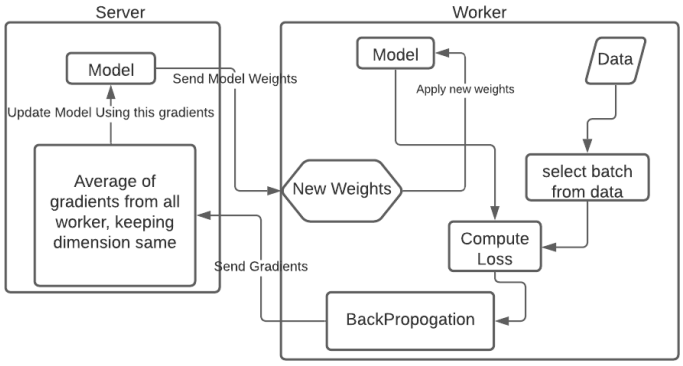

Figure 2: Parameter Server for single iteration. Every layer of gradient is sent separately and comparison is also applied layer wise

Each worker was provided one half of data that we divided i.e. worker 1 got half part of MNIST and half part of Cifar10. LeNet and VGG11 model was created for server and random weights had been applied to it, then this model with random weights were copied to both workers. This was done to make sure that our workers don't start from different model weights, as initial gradients calculated will be based on that initial model. On top of that, if initial model weights are not same, it will increase epochs needed to converge. After every 20 iterations, we saved our model so that even if one of our worker fails, we can restart the worker from the saved model and we don't have to start all over again.

Method 2: Next we tried to reduce communication data by applying the compression technique when we are sending gradients from worker to server. So, worker applies Quantized Stochasitc Gradient gradient descent algorithm on gradient and sends it to server, server then decompresses the gradient, averages gradient from all workers and then applies gradients to its own model and then sends model weights to the worker.

We even applied QSGD compression while sending model weights from Server to worker but found out that as QSGD is lossy compression , data loss in model weights is not acceptable which led to non convergence of Model.

Method 3: Moving ahead in reducing communication data , it would be great if we could apply two side compression i.e. while pushing data to server and while pulling data from server. So this can be achieved if we push and pull gradients only, pulling model weights is not required. Initially, server creates model with random weights, transfers the model and its weights to all workers. This is done so that all workers have same weights all the time as mentioned in [5]. So, if weights are not same in all iterations, gradients will be produced based on unequal weights, so averaging those gradients and applying them to

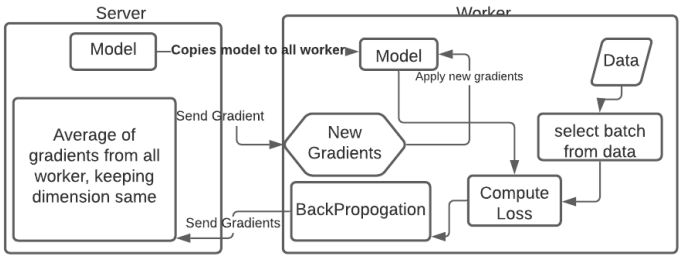

Figure 3: Parameter Server sending gradient both sides. Every layer of gradient is sent separately and compression is also applied layer wise

worker might not always result in the convergence of the model or even take more epochs to converge. This method is also good in the sense that computation from server side is very less, and workers do not have to wait for the server to finish its task. Besides, server can be used for other computations, while worker are computing on their data.

Method 4: Here, QSGD compression is applied on **Method 3** technique when only gradients are pushed and pulled from the server. Decompression is applied when we need to use those gradients like when averaging those gradients or updating of the models. And we got promising results with these methodology which are discussed in results section. Here, gradients of all layers are sent one after the other, so QSGD is applied on each and every layer separately.

Method 5: To further reduce the communication data, we apply Top-k sparsificaiton technique before QSGD. We select absolute highest top K percentile values and all other values are made 0. So this creates sparse matrix and to recreate actual matrix from this sparse matrix we need positions of all elements in the actual matrix, thus we need to send all positions also together with the shape of actual matrix. Gradient of every layer in model is sent one after the other and sparsificaiton is also applied layer wise. After applying sparsificaiton, we apply QSGD compression to further reduce the data sent and also send other information to recreate actual matrix with some loss.

Method 6: We find from our experiments that we do not need to push and pull gradients after every iteration. We take model in **Method 5** communicate with the server after every fixed iteration, till then workers is updating its model with their own gradients only. We found out that all of the workers still converge. It will take more epochs to converge but total data transmitted will be lesser compared to the previous technique. Here, dependency on server is highly reduced but as workers do not communicate always to update model, they will have different weights with them . So we will find workers with best accuracy on test data, and each worker's model weights will be replaced with model weights of worker with highest accuracy.

6 Model Used

For implementation and evaluation of our above work we decided to choose two CNN architecture models LeNet-5 and VGG11. LeNet CNN architecture is a very basic model introduced by LeCun Yann [6], it consists of total 7 layers in which 3 convolutional layers, 2 max pooling layers and 2 fully connected layers are present. Last layer is the softmax layer which predicts classification of input image. We have used MNIST dataset with this model. This model is selected because of its simple architecture and easy to converge ability. So that we could try our compression technique and understand whether they are working correctly or not.

Next we tested Cifar-10 dataset using model heavy model VGG11 [7]. It consists of total 16 layers with 8 convolution layers, 4 max pooling layers, and 2 fully connected layers and last softmax layer for classification of output. We selected this model as its computation time is reasonable for running in Google Colab. Accuracy on Cifar 10 is claimed to be 92.7 which is good enough for our compression technique comparison.

7 Datasets

To compare our mentioned approaches we use mainly two data sets to benchmarks our results MNIST and Cifar-10. MNIST consist of 28x28 grayscale handwritten images from number 0-9, each image consists of one handwritten number. Total 60000 images are there for training and 1000 images for testing purpose. Next, we have Cifar-10 dataset which comprises color images with 32x32 dimension of object divided into 10 categories.There are 50000 images for training and 10000 images for testing purpose. Each images consist of one object and categories are airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks.

8 Results

We have done our implementation in Intel(R) Xeon(R) CPU @ 2.20GHz and RAM: 12 GB in Google Colab. For both the models, we have selected batch size = 64, momentum = 0.9, worker = 2, server = 1, Optimizer = SGD, Framework - PyTorch , Communicator - Gloo, epoch =20 for LeNet and epoch =50 for VGG11. Our results are divided into four parts and can be seen in **Figure 5**.

1. **Top1 Accuracy.**
2. **Computation and Communication Time.**
3. **Average Communication Cost per iteration.**
4. **End to End training Time.**

(a) MNIST (b) Cifar 10

Figure 4: Dataset Mentioned

From accuracy **Table 5a** we can see that loss in accuracy is not much as compression is increased. In LeNet model, reduction of accuracy is less than $< 2\%$ even after applying **Method 6**. But in VGG11 model we find that when we apply method 5 there is significant accuracy drop from 86 % to 79% which can be attributed to the fact that too much loss in precision due to QSGD and Top-k sparsification but this accuracy drop is not observed in LeNet model as MNIST is quite simple and even in these type of conditions it easily gets converged. And when we look at method 6, as communication between server and worker is decreased, its accuracy is much better.

In **Table 5b** show us how much is computation cost and communication cost. Total cost here is the actual cost of running the program. Here we calculated cost for VGG11 model only, as LeNet model has very few gradients so improving communication cost in that is not much useful. Here workers are very near to each other so communication cost at max is just 20 minutes in method 1 and we were able to reduce it to 5 minutes and 10 minutes in method 6 and method 5 respectively. This is a good result and highly beneficial if any of the worker has poor network. Here computation cost is increased when compression scheme is applied because compression ,decompression, sparsification of gradients for each and every layer increase computation time.

In **Table 5c**, cost is calculated based on $\frac{TotalTrainingTime}{NumberofIteration}$, where single iteration is one batch computation of each worker and single push and pull of gradients between worker and server. Here we find that communication data is decreasing as we increase compression schemes. Although for method 6, epochs have increased still the average communication data is just 0.48 for VGG11 model and Cifar 10 dataset. Because here for all 50 epochs, communication happens after 20^{th} iteration due to which communication data is drastically reduced. Method 1 and Method 3 are sending same data because gradients and weights are of same size , so even if we replace gradients with weights it wont make difference in communication but compression can be applied on gradient

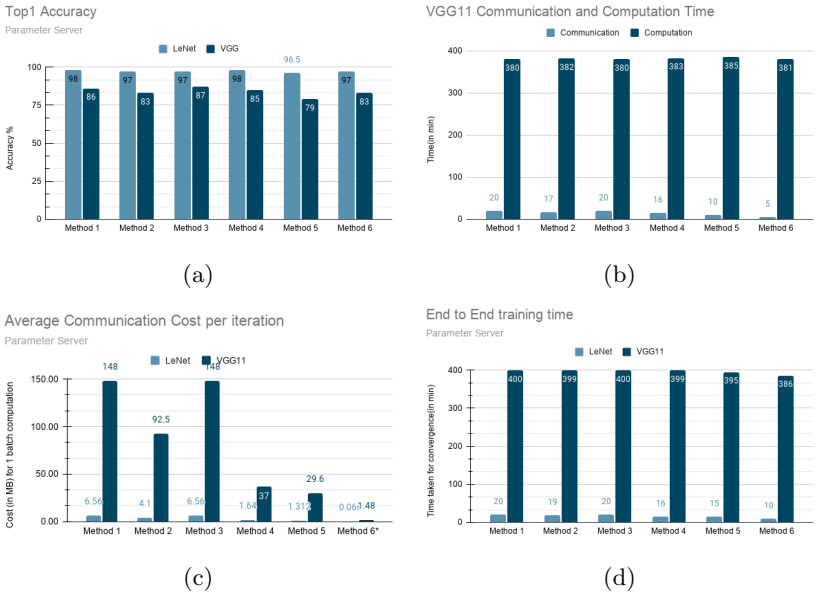

Figure 5: Results

push and pull.

In **Table 5d**, total training time of model is mentioned. Here training time is either constant or decreasing because communication is reduced. Total time difference between Method 1 and Method 6 is 15 minutes for VGG11 but remarkable improvement in LeNet is the reduction from 20 minutes to 10 minutes from method 1 to method 6 respectively. Basically, it increased training time. Here we find that our workers were very efficient in training and network lag was not observed. So here, communication did not play much efficient role proving usefulness of compression technique.

9 Conclusion and Future Work

In this paper we worked with different compression techniques including quantization and sparsification methods. We implemented the compression techniques like QSGD and Top-k sparsification and also tested using combined QSGD and Top-K compression scheme with the help of parameter server architecture. We found out that compression greatly reduces communication data , without much loss in accuracy. During slower network situations, this will have a better performance improvement. In future we would like to expand our compression approach to Pipeline parallelism architectures like GPipe [8] and PipeDream [9]. Communication in Pipeline parallelism is very important and we should be able to communicate well in poor network condition. In addition to this, we would also like to test our methods on a variety of other models such as Resnet-50

which we were not able to do, due to memory constraints on Google Colab. We hope to make use of Amazon Sagemaker for the aforementioned task.

References

[1] A. F. Aji and K. Heafield, "Sparse communication for distributed gradient descent," *CoRR*, vol. abs/1704.05021, 2017.

[2] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *CoRR*, vol. abs/1606.06160, 2016.

[3] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "Qsgd: Communication-efficient sgd via gradient quantization and encoding," *Advances in Neural Information Processing Systems*, vol. 30, pp. 1709–1720, 2017.

[4] S. Shi, X. Chu, K. C. Cheung, and S. See, "Understanding top-k sparsification in distributed deep learning," *arXiv preprint arXiv:1911.08772*, 2019.

[5] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li, "Terngrad: Ternary gradients to reduce communication in distributed deep learning," *arXiv preprint arXiv:1705.07878*, 2017.

[6] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, "Handwritten digit recognition with a back-propagation network," in *Advances in neural information processing systems*, pp. 396–404, 1990.

[7] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[8] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *arXiv preprint arXiv:1811.06965*, 2018.

[9] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 1–15, 2019.