Parallel and Distributed Deep Learning

Vishakh Hegde
Stanford University
vishakh@stanford.edu

Sheema Usmani
Stanford University
sheema@stanford.edu

Abstract

The goal of this report is to explore ways to parallelize/distribute deep learning in multi-core and distributed setting. We have analyzed (empirically) the speedup in training a CNN using conventional single core CPU and GPU and provide practical suggestions to improve training times. In the distributed setting, we study and analyze synchronous and asynchronous weight update algorithms (like Parallel SGD, ADMM and Downpour SGD) and come up with worst case asymptotic communication cost and computation time for each of the these algorithms.

1. Introduction

In this report, we introduce deep learning in 1.1 and explain the need for parallel and distributed algorithms for deep learning in 1.2. We then go on to give a brief overview of ways in which we can parallelize this problem in section 2. We then perform an empirical analysis on CPU and GPU times in section 3. We then explain stochastic gradient descent briefly in section 4 and provide the pseudo-code for, and analyze a few distributed gradient update algorithms in section 5. We conclude in section 6 and give some ideas for future work.

1.1. Deep Learning

Deep neural networks are good at discovering correlation structures in data in an unsupervised fashion. Therefore it is widely used in speech analysis, natural language processing and in computer vision. This information of the structure of the data is stored in a distributed fashion. i.e. Information about the model is distributed across different layers in a neural network and in each layer, model information (weights) are distributed in different neurons. There are a lot of ways to combine the information in a layer spread across different neurons and there are lot of ways to combine layers in order to minimize a loss function (which is a proxy for how well the neural network is doing in terms of achieving its goals). In our project, we use a deep network for classifying greyscale images of size 224×224 pixels into one of 40 possible classes to which we know it should belong.

1.2. Need for Parallel and Distributed Algorithms in Deep Learning

In typical neural networks, there are a million parameters which define the model and requires large amounts of data to learn these parameters. This is a computationally intensive process which takes a lot of time. Typically, it takes order of days to train a deep neural network (like VGG network [13] on a single core CPU and about $\frac{1}{q}$ on a single machine with q cores in the CPU, which is still in the order of several hours (assuming we have 8 cores, it still takes up to 10 hours (roughly) to train a model like VGGNet on a single machine). Sometimes the data-set is too large to be stored on a single machine.

Therefore it is important to come up with parallel and distributed algorithms which can run much faster and which can drastically reduce training times.

2. Parallel and Distributed Methods

One can think of several methods to parallelize and/or distribute computation across multiple machines and multiple cores. We list some of the methods used to achieve faster training times:

- Local training: The model and data is stored on a single machine.
  - Multi-core processing: Here, we assume that the whole model and the data can be fit into the memory of a single machine with multiple cores. These multiple cores share the memory (PRAM model). There are two ways to use multiple cores to speed up the training process.
    - Use the cores to process multiple images at once, in each layer. This is an embarrassingly parallel process.
    - Use multiple cores to perform SGD of multiple mini-batches in parallel.

  - Use GPU for computationally intensive subroutines like matrix multiplication.
  - Use both multi-core processing and GPU where all cores share the GPU and computationally intensive subroutines are pushed to the GPU.

- Distributed training: [1] When it is not possible to store the whole data-set or a model on a single machine, it becomes necessary to store the data or model across multiple machines.
  - Data parallelism: Data is distributed across multiple machines. This can be used in case data is too large to be stored on a single machine or to achieve faster training.
  - Model parallelism: If the model is too big to be fit into a single machine, it can be split across multiple machines. For example, a single layer can be fit into the memory of a single machine and forward and backward propagation involves communication of output from one machine to another in a serial fashion. We resort to model parallelism only if the model cannot be fit into a single machine and not so much to fasten the training process.

3. Empirical analysis: CPU versus GPU time

We used Amazon AWS EC2 instances with GPU. Here is the configuration for the machine we used to perform this analysis:

- 8 High Frequency Intel Xeon E5-2670 Processors.
- NVIDIA GRID K520 GPU with 1,536 CUDA cores and 4GB of video memory.

In this section, we present the results from tests we've conducted on computational times for various operations on a CPU and a GPU and also talk about steps to be taken in order to speed up the learning process. We do not provide any theoretical analysis for this section.

In a Convolutional Neural Network (CNN), the first layer is usually a convolution. A convolution is a sliding kernel (of a fixed size, and is usually square in shape) which does an element-wise multiplication on each of the pixels it has an overlap with, and sums all the elements together. In a modern CNN, there are several different kernels applied to a single image and the results are stacked in the output. Each kernel slides through the whole image (in predetermined steps with a fixed step size). Backpropagation is used to update the parameters of these kernels (also called weights). So both forward and backward propagation is computationally intensive. In a deep network, there are several layers of convolution and therefore adds a lot to total compute time on a CPU. Therefore, an important way to improve the performance of the whole network is to reduce the run-time of convolution.

3.1. Parallel Implementation of Convolution in Caffe

Since the same kernel slides over the whole image, Caffe [8] uses the function `im2colgpu` to unwrap the parts of the image that the kernel slides over, into vectors. These vectors are stacked to form a matrix. For example, in our network, the first convolution layer is made up of filters of size 11×11 and has a stride of 4 (i.e. it slides in steps of 4 pixels). Therefore, there are 54 parts in the image where a single kernel overlaps with the image. For each of the 54 parts which have a size 11×11, we unroll it into vectors of size 121. Since there are 54 such vectors, they are stacked together to form a matrix of size 54×121.

In order to perform convolution for a single kernel, we need to perform a matrix-vector multiplication, where the vector is the unrolled form of the kernel. There are multiple kernels used in a single convolution layer. Therefore, the unrolled vectors of kernels are stacked together to form another matrix. In our case, there are 96 kernels used in the convolution layer. So we form a matrix of size 121×54. Therefore in order to perform convolution for the whole convolution layer, we now need to perform a matrix-matrix multiplication. In our case is a multiplication between matrix of size 54×121 and 121×96. All this is done for a single image. It can be noted here that matrix-matrix multiplication is the most computationally intensive part of the whole process.

The CPU handles all the complicated logic part of this process, while `im2colgpu` is called for unrolling the image into a matrix (in parallel) and for performing the matrix-matrix product (this is also computed in parallel). This happens serially for all the images in a mini-batch. Once all the images have been processed, the CPU moves to the next layer in the model. [12]

3.2. Results

We timed forward propagation times for convolution layer and fully connected layer 10 times and found the average computational times for both CPU and GPU for different batch sizes (for the convolution layer) and for different matrix sizes (for the fully connected layer).

3.2.1 Convolution Layer

This layer consists of outputs from 96 different filters applied at a stride (or sliding step-size) of 4. Each kernel has

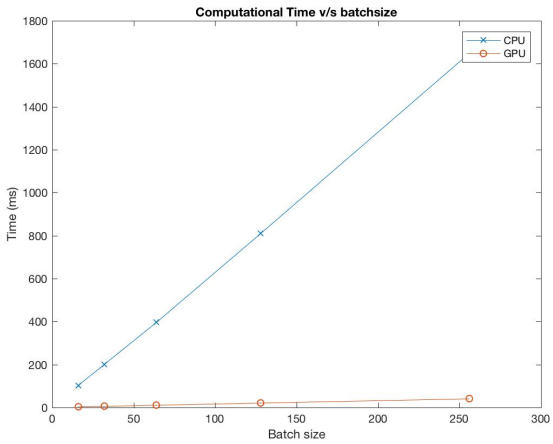

Figure 1. Comparison between CPU and GPU for time taken to forward-propagate through a convolution layer as a function of batch-size

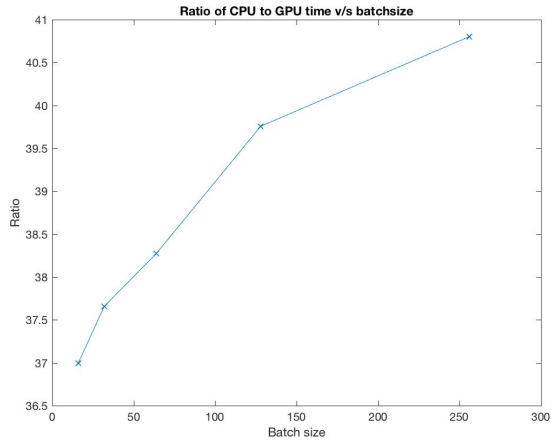

Figure 2. CPU to GPU speedup as a function of batch-size for the convolutional layer

size 11×11. The size of the image is 224×224.

The slope of the CPU line in figure 1 is 6.45 and the slope of the GPU line is 0.16. This means that on an average, GPU is about 40 times faster than a CPU when computing the convolution of an image. It is important to note that both lines are linear, which means that it is $\mathcal{O}(n)$. It is not surprising that this is the case for a CPU (it processes images sequentially). For a GPU though, the program control still rests with the CPU while the GPU takes care of computationally intensive subroutines (like convolution and matrix multiplication). Therefore, even for a GPU, it is $\mathcal{O}(n)$. However, the constants multiplying n for CPU is 40 times that of a GPU. We also plot the ratio of CPU to GPU time to forward propagate the convolution layer summarized in figure 2:

This indicates that the speedup is more if the batch-size is

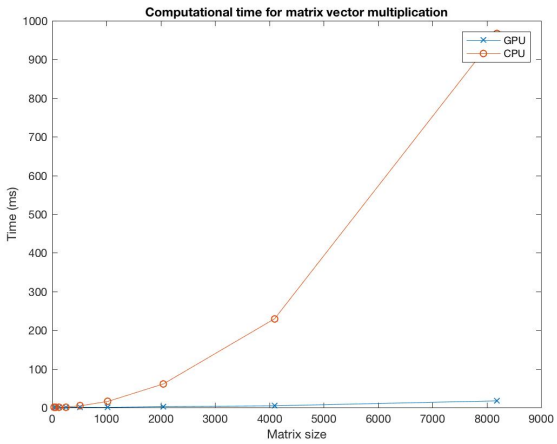

Figure 3. Comparison between CPU and GPU for time taken to perform matrix-matrix multiplication as a function of matrix size

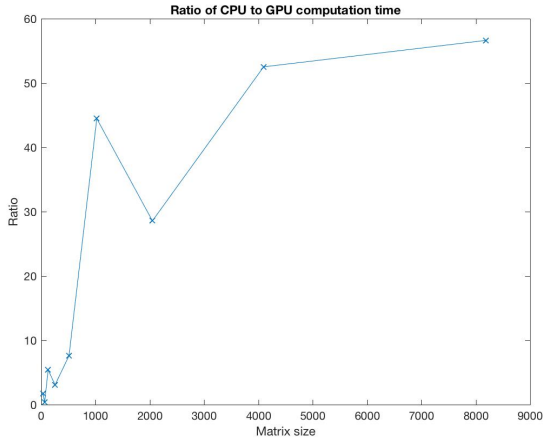

Figure 4. CPU to GPU speedup as a function of batch-size for matrix-vector multiplication, as a function of matrix size

higher. The conclusion of this experiment is that it is almost always better to have a higher batch-size while training a CNN since it gives a higher speedup as seen in the figure.

3.2.2 Fully Connected Layer

The fully connected layer is implemented as a matrix-vector multiplication.

In figure 3, it can be seen that for both CPU and GPU, the computation time is $\mathcal{O}(n^2)$, which is as expected. However, the constant multiplying n^2 are very different (again, by about a factor of 40). Figure 4 is a graph of the ratio of CPU to GPU times for different matrix sizes.

Like before, it is apparent that higher matrix sizes imply higher speedup. However, this information cannot be used for our advantage because the dimensionality of the fully connected layer is usually fixed in a model.

4. Stochastic Gradient Descent

The goal of a learning algorithm is to minimize the loss function in a systematic manner. In the case of neural networks, the total-loss function is a separable and differentiable function of the model parameters. We need to come up with a way to iteratively update these parameters so that the value of the total-loss function reduces. One can visualize the total-loss function as consisting of a bunch of peaks and valleys and the goal is to get to the deepest valley [3].

One of the most popular ways to achieve this is to use a greedy approach: by following a direction opposite to the gradient of the loss function, since this is the direction which is most promising, locally (so to speak). The loss function in the case of neural networks is normally a separable function (i.e. it is average of loss functions for individual data points). So, in order to make the most optimal decision, we need to compute the gradient of the loss for all the images in the data-set with respect to all the paramaters of the model. However, doing this is computationally expensive because of the sheer number of images on which we train these neural networks [4].

Therefore, it is necessary to use stochastic gradient descent, which computes the gradient of loss functions of a representative subset of the original data-set. This is repeated for many subsets of the original data-set until all images have been used up. This is called an epoch and the subset of data used for parameter update is called a mini-batch. Let the weights of the model be w. Here is the gradient descent update:

$$w \leftarrow w - \alpha \nabla_w L_{total} \tag{1}$$

Where $L_{total} = \frac{1}{n} \sum_{i=1}^{n} L_i$ and $\nabla_w L_{total}$ is the gradient of the total loss function with respect to the weights. In the neural network that we trained, we used the logistic loss function for each image, given by:

$$L_i = -f_{y_i} + \log \sum_j e^{f_j} \tag{2}$$

where f_j means the j^{th} element of the vector of class scores f.

In stochastic gradient descent, we have the following weight update rule:

$$w \leftarrow w - \alpha \nabla_w L_{minibatch} \tag{3}$$

Here,

$$L_{minibatch} = \frac{1}{m} \sum_{i \in \mathbb{M}} L_i \tag{4}$$

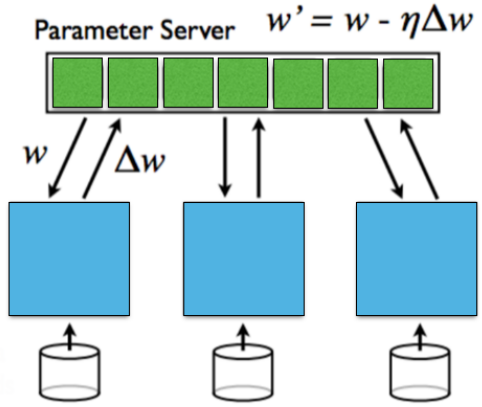

Figure 5. Data parallelism for updating parameters

where $\mathbb{M}$ is the set of all images in a mini-batch and $m = |\mathbb{M}|$, the size of the minibatch[2]. Performing gradient descent on multiple mini-batches is computationally intensive and time consuming. Therefore, there is a need to parallelize the weight update procedure to achieve faster learning.

5. Data Parallelism

Data Parallelism is a way to distribute computing across different machines in a way that data is split across different machines and some computation is performed locally in each machine using the whole model, but only for part of the full data-set. For the purposes of analysis, we will assume that we can fit all the parameters of the model on a single machine. This will be the driver. Stochastic Gradient Descent (and some flavors of it) is the most commonly used update rules in neural networks.

Depending on how these parameters are updated, we have two paradigms for parameter update:

- Synchronous update
- Asynchronous update

5.1. Synchronous update

For synchronous update, all loss gradients in a given mini-batch are computed using the same weights and full information of the average loss in a given mini-batch is used to update weights. The synchronization part comes because we wait till loss-gradients for all images in the mini-batch are computed.

5.1.1 Parallel SGD

Assume that the images are distributed across several machines and that they are not stored in a random manner. We

can also assume that the data-set is sorted according to the label. This happens in many realistic scenarios. Assume further that the total loss function we are trying to minimize is strongly convex (this is not a very reasonable assumption to make. It has been shown that the loss function in typical deep learning scenarios is non-convex, with multiple valleys). However, we can get around such an assumption performing multiple iterations. Here is the pseudo-code for the parallel SGD algorithm. [9]

Algorithm 1 ParallelSGD

procedure PARALLELSGD($(parameters, data, k)$)
Shuffle the data on all machines so that each machine has a representative subset of the global data-set
  for *each machine* $i \in \{1, ...k\}$ *in parallel* **do**
    $v_i \leftarrow SGD(parameters, data)$
  Aggregate from all machines $v \leftarrow (1/k)\sum_{i=1}^{k}[v_i]$ *and* **return** v

The first step in the above algorithm is shuffling so as to have a representative subset of the full data-set in each of the machines. Below is the pseudo-code to achieve this. `L` is the label of the images, `ImInd` is the index of the image (location of the image in the entire data-set), `CLoc` is the current location of the image (the machine on which it sits prior to shuffling).

Algorithm 2 ShuffleSGD

procedure SHUFFLESGD(`L`, `ImInd`, `CLoc`)
Obtain uniform sample of data from each machine - perform k-way merge.
Create k bins to have equal number of images in each bin.
Decide the destination machine for each image based on the bins
Communicate this decision to each machine
Perform all-to-all communication to shuffle the data

Analysis of `ShuffleSGD` algorithm

Assume that the total size of the data-set is D and that there are k machines at our disposal. Let each machine have a network bandwidth of B and latency of L. We can assume that data is sorted in each machine. Also assume that there are q processors on each machine. Assume that we sample d images (and send a data structure which only contains label, index and machine on which the sampled images currently reside) from each machine and send it to the driver for it to put together a distribution.

Work-depth analysis: k- way merge in the driver machine is $\mathcal{O}(kd)$ work and $\mathcal{O}(\frac{kd}{q} + log(kd))$ depth for q processors. Deciding the bin size and start/end points of each bin is $W(n) = \mathcal{O}(kd)$ in the work case.

Communication Cost: For establishing the distribution of data, each machine passes a sample data of size d to master machine to find the distribution of the data. To achieve this, we do a bit-torrent aggregate communication where in the first round, $\frac{k}{2}$ machines talk to $\frac{k}{2}$ other machines and pass d message from one machine to another. In the next $\frac{k}{4}$ machines communicated $2d$ data between each other and so on. There are $log(k)$ rounds of communication happening.

Here is the Communication cost of this procedure:

$$= \frac{k}{2}(L + \frac{d}{B}) + \frac{k}{4}(L + \frac{2d}{B}) + \ldots (\texttt{log(k)}\ terms)$$
$$= L(\frac{k}{2} + \frac{k}{4} + \ldots) + \frac{dk}{B}(\frac{1}{2} + \frac{1}{2} + \ldots)$$
$$= \mathcal{O}(Lk) + \mathcal{O}(\frac{dk}{B} logk)$$

Once the distribution has been established, we perform a One-to-All communication to send this information to each of the k machines. This is again a bit-torrent aggregate pattern. Therefore, the communication cost is similar to All-to-One communication explained about. Communication cost is:

$$= \mathcal{O}(Lk) + \mathcal{O}(\frac{dk}{B} logk)$$

Since k, d and L are very small in comparison to D (the total size of the data-set), we can easily ignore all the costs associated with establishing the distribution.

Now, each machine must transfer some if its data (in the worst case, all its data) during the shuffle procedure. Let total data of size D be split across k machines i.e. $N = \frac{D}{k}$ data for each machine.

Communication cost of all to all communication (to shuffle data based on the distribution) will be $\mathcal{O}(kN) = \mathcal{O}(D)$ because the total size of the data-set is D.

Analysis of `ParallelSGD` algorithm

In this section, we analyze the communication costs of everything else apart from `ShuffleSGD` step. The assumption made here is that of strong convexity.

Computation Time: Once each machine has data which mimics the distribution of the whole data-set, we can run SGD on each machine locally. If we want to achieve an error less than ϵ, the computation time of SGD on each machine is $O(p \log \frac{1}{\epsilon})$ where p is size of the parameters. (with $O(\log(\frac{1}{\epsilon}))$ iterations for convergence).

Computation time for All-to-One step when aggregating the gradients (using BitTorrent Aggregate): There are $\log k$ rounds of communication and in each round we sum up the parameters. Each summation is $W = \mathcal{O}(p)$ work on a single processor. With q processors, we can do this summation in $\mathcal{O}(\frac{p}{q} + \log\ p)$ depth. Therefore computation time is $O(\frac{p}{q}\log k) + O(p \log \frac{1}{\epsilon})$.

Communication Cost: SGD is computed on each machine locally. Therefore we will not have any communication cost for SGD. Once all the parameters are updated for each machine locally, we need to perform an All-to-One communication to send it to the driver machine where it will be averaged. For this, we will do a BitTorrent aggregate communication. The communication cost for this will be:

$$= L(\frac{k}{2} + \frac{k}{4} + \ldots) + \frac{kp}{B}(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots)$$
$$= \mathcal{O}(Lk) + \mathcal{O}(\frac{kp}{B})$$

Communication cost for broadcasting parameters (One-to-All) computing average (All to one) in the last step is O(kp) as $pk(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots)$. Therefore the total communication cost is $O(Nk) + O(pk)$.

Communication Time: Communication time for All to one step when aggregating the gradients (using BitTorrent Aggregate): There are $\log k$ rounds of parallel communication over the network and each require $\mathcal{O}(p)$ communication time (as communication time depends linearly on the size of the message). Thus Communication Time is $\mathcal{O}(p \log k)$.

`ParallelSGD` assumes the loss function to be strongly convex, which gives it a unique minima. However, this is not the case in deep neural networks. One way to improve the accuracy is to systematically shuffle data between machines and carry out another round of SGD on each machine locally and updating the parameters. Systematic shuffling can be done by having two machines exchange part of the data in a single round of communication (say half of the data in each machine). The total communication cost will be $\mathcal{O}(kN)$. However, the communication time will be $\mathcal{O}(N)$, where $N = \frac{D}{k}$. Since SGD happens in parallel, one round of SGD will be equivalent to one epoch of training. Typically, images are trained on order of 10 epochs. Therefore, this procedure looks like a practical approach to training deep networks in a distributed fashion.

5.1.2 Alternating Direction Method of Multipliers SGD [10]

Algorithm 3 ADMM.SGD

procedure STARTSYNCHRONOUSLYFETCHINGPARAMETERS(*parameters*)
  parameters $\leftarrow$ $GETPARAMETERSFROMPARAMSERVER()$
procedure STARTSYNCHRONOUSLYPUSHINGGRADIENTS(gradients)
  SENDGRADIENTSTOPARAMSERVER(gradients)
procedure PARAMSERVER($(p_1, p_2, ...p_k, k)$)
  Aggregate from all k machines p $\leftarrow$ $(1/k)\sum_{i=1}^{k}[p_i]$ *and* **update** p
procedure ADMM.SGD(*parameters*)
  $step \leftarrow 0$
  while *true* **do** STARTSYNCHRONOUSLYFETCHINGPARAMETERS(parameters)
    $data \leftarrow GETNEXTMINIBATCH()$
    gradient $\leftarrow$ $COMPUTEGRADIENT(parameters, data)$
    $parameters \leftarrow parameters - \alpha gradient$
    STARTSYNCHRONOUSLYPUSHINGGRADIENTS(gradients)
    $step \leftarrow step + 1$

Analysis: In the ADMM procedure, we communicate the parameters to each machine, where SGD is used to update weights of the model. Once this is done, the weights are sent over to the driver machine for aggregation. Once the parameters have been aggregated and updated using data from all machine (this is why it is synchronous), parameters are broadcasted to all machines for the whole procedure to be repeated. This avoids the need for shuffling data between machines, which is the main bottleneck for `ParallelSGD` algorithm.

Computation Time: It is to be noted that same amount of computation is done in both ADMM and ParallelSGD procedure while performing SGD. In ADMM, computation is interspersed with communication of parameters to the driver machine. Computation time of SGD on each machine is $O(p \log \frac{1}{\epsilon})$, where p is size of the parameter. (with $O(\log(\frac{1}{\epsilon}))$ iterations required for convergence).

The difference is that in each iteration, we summed up weights while sending it to the driver machine. There are $\log k$ stages and each require O(p) computation when aggregating the gradients between two machines. For $\log(\frac{1}{\epsilon})$ iterations the computation time is,

$$O(p \log k \log \frac{1}{\epsilon}) + O(p \log \frac{1}{\epsilon})$$

Communication Cost: Each iteration requires the gradient to be sent over the network. Communication cost for broadcasting the parameters(One to all) and computation of average (All to one) in the last step is O(kp) as $pk(\frac{1}{2}+\frac{1}{4}+\frac{1}{8}...)$. We do this for $\log\frac{1}{\epsilon}$ iterations. Thus the total communication cost is O(kplog $\frac{1}{\epsilon}$).

Communication Time: Communication time for All to one step when aggregating the gradients (using BitTorrent Aggregate) there are $\log k$ layers and each require O(p) communication time (as communication time depends linearly on the size of the message). This is the communication time for each iteration. For $\log\frac{1}{\epsilon}$ iterations, the Communication Time is $O(p\log\frac{1}{\epsilon}\log k)$.

5.2. Asynchronous Update Methods[11]

So far, in synchronous update methods, we have used loss gradients from all the machines before we updated the parameters in the parameter server. Because we have a limited bandwidth, we might have to queue these updates. Apart from that, we will have to wait till the last machine performs and sends its parameter update before we perform another iteration. These clearly are communication bottlenecks. One way to avoid such bottlenecks is to resort to asynchronous update methods. In these methods, as soon as a machine finishes computing updates, the parameters in the driver get updated [6][5]. Any machine using the parameters will fetch the updated parameters from the server.

Algorithm 4 DOWNPOURSGD

1: **procedure** STARTASYNCHRONOUSLYFETCHINGPARAMETERS($(parameters)$)
2: $parameters \leftarrow GETPARAMETERSFROMPARAMSERVER()$
3: **procedure** STARTASYNCHRONOUSLYPUSHINGGRADIENTS($(accruedgradients)$)
4: SENDGRADIENTSTOPARAMSERVER(gradients)
5: **procedure** DOWNPOURSGD
6: $step \leftarrow 0$
7: **while** $true$ **do** STARTASYNCHRONOUSLYFETCHINGPARAMETERS(parameters)
8: $data \leftarrow GETNEXTMINIBATCH()$
9: $gradient \leftarrow COMPUTEGRADIENT(parameters, data)$
10: $parameters \leftarrow parameters - \alpha gradient$
11: STARTASYNCHRONOUSLYPUSHINGGRADIENTS(gradients)
12: $step \leftarrow step + 1$

5.2.1 Downpour SGD [7]

Computation Time: This is similar to the computation time of ADMM procedure.

Communication Cost: Each iteration requires the gradient to be sent over the network. This is $\mathcal{O}(p)$ data. We do this for θ iterations (say). The total communication cost is O(θp).

Communication Time: Let T be the time taken to compute the gradient and τ be the time the server takes to receive the gradient message and apply parameter update. When there are few machines, the speed of training is bound by the gradient computation time T. Thus, even with an infinite pool of workers, we cannot perform more than $1/\tau$ updates per second. A pool of k workers will, on average, serve up a new gradient every $\frac{T}{k}$ seconds. Thus, once we have P = $\frac{T}{\tau}$ workers, we will no longer see any improvement by adding more workers.

The run-time for a single parameter update with single machine is T + τ. Both T and τ are $\mathcal{O}(p)$, but the constant factor differs substantially. Thus if $k < \frac{T}{\tau}$, communication cost is governed by gradient computation time i.e. $\mathcal{O}(\theta\frac{p}{k})$, otherwise communication cost is governed by time taken by servers to update i.e. $\mathcal{O}(\tau)$.

Currently asynchronous update of parameters is done for each layer. When we use back-propagation to update weights of each layer, we move to the previous layer to update its paramaters, only after completely updating parameters of the current layer. Therefore, parameter update is still in many ways, synchronous. Truly asynchronous weight update can be achieved if we do the back-propagation also in an asynchronous manner.

6. Conclusion and Future Work

Neural networks typically have millions of parameter and requires large amounts of data to tune these parameters to achieve a goal (for example, classification of images into one of several possible classes). With growing size of the network and larger the data-set, we are able to extract more complex representations, but at the cost of insanely high computation time. Some Neural Networks take weeks to train on a single core CPU. Also the size of the neural network may not fit in a single machine. This made exploring ways to parallelize convolutional neural network an interesting problem.

We have done an empirical analysis on the speedup of convolutional layer (as a function of batch-size) and matrix-vector multiplication (as a function of matrix size) and have

mentioned practical suggestions to improve training times. We have talked about ways to achieve distributed training and have analyzed most commonly used algorithms (like `parallelSGD`), `ADMM` and `DownpourSGD`) used to update weights in a distributed fashion. In particular, we have analyzed `ParallelSGD` in depth and provide practical suggestions to get around the strong convexity assumption problem, since the loss function in typical deep neural networks are non-convex in nature.

In a distributed setting we may implement Model Parallelism i.e. split the model across different machines or Data Parallelism and analyze the cost of computation and communication cost.

We sincerely thank Reza for providing us with the opportunity to work on such an interesting project and for providing computational resources required for the project.

References

[1] S. W. W. W. Q. G. C. J. G. Z. L. A. K. H. T. Y. W. Z. X. M. Z. K. Z. B. Chin Ooi, K. Tan. Singa: A distributed deep learning platform. *ACM Multimedia*, 2015.

[2] L. Bottou. Large-scale machine learning with stochastic gradient descent. *COMPSTAT*, 2010.

[3] L. Bottou. Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade*, pages 421–436, 2012.

[4] K. O. Christopher De Sa and C. Re. Global convergence of stochastic gradient descent for some nonconvex matrix problems. *ICML*, 2015.

[5] S. O. Cyprien Noel. Dogwild!distributed hogwild for cpu and gpu. 2014.

[6] C. R. F. Niu, B. Retcht and S. J. Wright. Hogwild! a lock free approach to parallelizing stochastic gradient descent. *Neural Information Processing Systems*, 2011.

[7] R. M. K. C. M. D. Q. L. M. Z. M. M. R. A. S. P. T. K. Y. A. Y. N. J. Dean, G. Corrado. Large scale distributed deep networks. *Neural Information Processing Systems*, 2012.

[8] Y. Jia. Caffe: An open source convolutional architecture for fast feature embedding. http://caffe. berkeleyvision.org/.

[9] A. S. L. L. Martin A. Zinkevich, Markus Weimer. Parallelized stochastic gradient descent. *Neural Information Processing Systems*, 2010.

[10] E. C. B. P. Stephen Boyd, Neal Parikh and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 2011.

[11] J. Y. Z. L. T. Paine, H. Jin and T. S. Huang. Gpu asynchronous stochastic gradient descent to speed up neural network training. *CoRR, abs/1312.6186*, 2013.

[12] K. L. Xiaqing Li, Guangyan Zhang and W. Zheng. Deep learning and its parallelization.

[13] K. S. . A. Zisserman. Very deep convolutional networks for large-scale image recognition. 2015.