Scalable, Efficient Deep Learning by Means of Elastic Averaging

Kevin Peng

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2018-77
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-77.html

May 18, 2018

Acknowledgement

I would like to thank my research advisor, Professor John F. Canny, for the opportunity to work on BIDMach during these last two years, for the support,
insight, and advice he provided during this time, and for everything I have learned in his Special Topics in Deep Learning class.

I would also like to thank my colleagues Mick Jermsurawong, Hao Li, Yao Yuan,
Lincoln Lin, and Yiwei Zhang for their invaluable contributions to this project. You all were congenial colleagues to work with.

I would also like to thank Professor Kurt Keutzer for his help on my master's thesis committee.

Finally, I would like to thank my friends and family for all the support and encouragement they have provided me over these last five years during my undergraduate and graduate career at Berkeley.

Scalable, Efficient Deep Learning by Means of Elastic Averaging

by Kevin Peng

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor John F. Canny
Research Advisor

5/16/2018

(Date)

* * * * * * *

Professor Kurt Keutzer
Second Reader

5/16/2018

(Date)

Scalable, Efficient Deep Learning by Means of Elastic Averaging

Kevin Peng

May 14, 2018

Abstract

Much effort has been dedicated to optimizing the parallel performance of machine learning frameworks in the domains of training speed and model quality. Most current approaches make use of parameter servers or distributed allreduce operations. However, these approaches all suffer from various bottlenecks or must trade model quality for efficiency. We present BIDMach, a distributed machine learning framework that eliminates some of these bottlenecks without sacrificing model quality. We accomplish this by using a grid layout that enables allreduces with only $\sqrt[d]{n}$ nodes instead of the usual n, a modified allreduce operation resistant to lagging nodes, an architecture responsive to nodes leaving and joining, and elastic averaging to mitigate the effect of worker iteration divergence.

Acknowledgments

I would like to thank my research advisor, Professor John F. Canny, for the opportunity to work on BIDMach during these last two years, for the support, insight, and advice he provided during this time, and for everything I have learned in his Special Topics in Deep Learning class.

I would also like to thank my colleagues Mick Jermsurawong, Hao Li, Yao Yuan, Lincoln Lin, and Yiwei Zhang for their invaluable contributions to this project. You all were congenial colleagues to work with.

I would also like to thank Professor Kurt Keutzer for his help on my master's thesis committee.

Finally, I would like to thank my friends and family for all the support and encouragement they have provided me over these last five years during my undergraduate and graduate career at Berkeley.

Contents

Chapter 1

Introduction

In any data parallel machine learning system, the different nodes in the system need some means of synchronizing their model parameters. A simple way to implement this synchronization is to have a central server which holds an authoritative copy of all of the model parameters. During training, nodes compute gradients and upload those gradients to the parameter server. The parameter server, after having collected gradients from every node, then applies the gradients to the model parameters, and broadcasts the updated model parameters to every node.

One early parameter-server-based method was the DOWNPOUR algorithm, used by Distbelief [7]. DOWNPOUR makes some modifications to the basic parameter server model: to help with load balancing, instead of a single parameter server, there are multiple parameter servers, each holding a portion of the model parameters. DOWNPOUR also makes use of model parallelism, clustering nodes into *model replicas*, with the nodes in each model replica collectively holding a complete copy of the model.

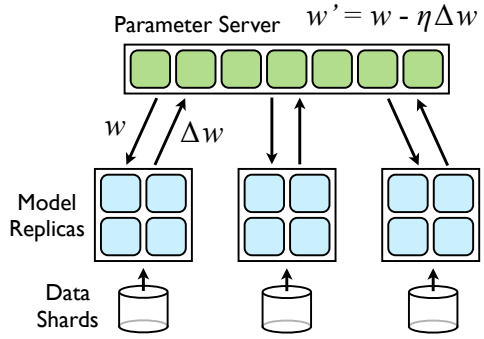

Figure 1.1: Architecture of DOWNPOUR SGD (from [7])

Later parameter-server-based methods made additional improvements to this architecture. MXNet [5], for instance, augments the parameter server with operations for avoiding extra transmissions in instances where the same data is sent twice, and sparsifying matrices in instances where the model weights contain a large number of zeros [14].

Parameter servers, however, suffer from a few issues. In a naïve system with a single parameter server, the parameter server quickly becomes a bottleneck. In a more sophisticated system with multiple parameter servers, communication between the different parameter servers turns out to be rather expensive [6, 15]. Parameter servers also require a warm-up phase where training is executed on a single node in order to facilitate convergence [6, 15].

Another common synchronization strategy has been to eschew the parameter server and instead exchange update gradients using an allreduce sum operation. Caffe2 [8] is an example of such a system. These systems do not have many of the pitfalls of parameter-server-based systems, but allreduce operations nevertheless can suffer from poor latency, as every node is required to wait for the slowest node to catch up in order for

the allreduce to finish.[1] Allreduce operations involving n nodes also require $O(\log n)$ time to run, compared to the $O(1)$ time needed to synchronize with a parameter server.

Data parallel distributed ML systems also draw a distinction between *synchronous training*, where all nodes are required to be on the same iteration at any given time, and *asynchronous training*, where each node is allowed to train at its own pace. Synchronous training has the same latency issue as allreduce, that all nodes must wait for the slowest node to catch up in order for each new iteration to start. Asynchronous training avoids this issue, but comes with a different problem: as training progresses, nodes increasingly diverge from each other in terms of progress. A node at iteration $i+10$ generally has a better quality model than a node at iteration $i-10$; however, when those two nodes send their gradients to the parameter server, the parameter server will blindly combine them. When the node at $i+10$ then pulls the updated model from the parameter server, it will be catapulted about 10 iterations back in time due to the outdated gradients from the other node. This fundamental tradeoff between latency and model quality has existed for a long time.

Some researchers have experimented with gossip-based protocols for model parameter synchronization. In these protocols, nodes directly exchange gradients with other nodes in a peer-to-peer fashion. There are two primary such protocols, published by Jin *et al.* [12] and Blot *et al.* [3] However, as those authors report, the protocols suffer from performance and convergence degradation at moderate scale.

BIDMach is able to address these problems using a combination of the allreduce and gossip approaches. In BIDMach, the nodes are arranged into a logical grid. Nodes exchange model parameters with other nodes in the same row and column, reducing the number of nodes in each allreduce operation from n to $\sqrt{n}$. The allreduce operations are further modified so that they can finish with only a quorum of servers. BIDMach utilizes asynchronous training; to address the iteration divergence problem, BIDMach utilizes *elastic averaging* [17] to update model parameters. This way, nodes which are ahead of the crowd do not get thrown back in time by importing model parameter contributions from other nodes, while nodes are still kept in sync with each other. We show that using these techniques, BIDMach achieves both very good scaling behavior and comparable model quality as achieved by single-node training.

[1] Parameter servers which synchronously wait for every node to push gradients before applying them also suffer from the same issue.

Chapter 2

Scaling Up Deep Learning

2.1 Grid-structured Allreduce

In BIDMach, nodes are arranged in a grid. The grid is organized by a *grid master*, which is implemented as an Akka actor. The grid master arranges the worker nodes into a $\lceil\sqrt{n}\rceil \times \lceil\sqrt{n}\rceil$ grid, using a placement algorithm that ensures that the number of nodes in each of the rows and columns differs by no more than 1 from that of every other row and column. The grid master also appoints one *line master* for each row and one line master for each column. (Each line master only manages one dimension.)

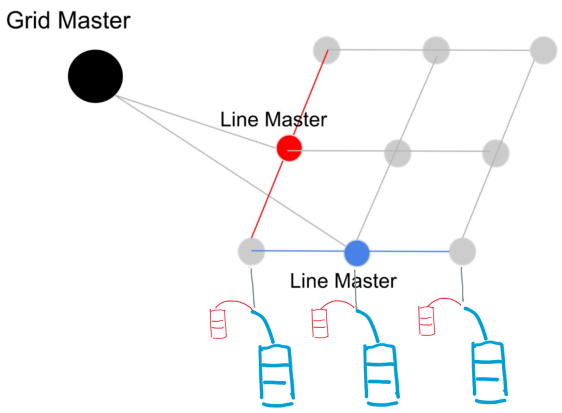

Figure 2.1: Nodes are arranged in a grid by the GridMaster. Each grid row and column has a LineMaster.

Each node is represented by an Akka actor of type Node. Each Node actor has two DimensionNode child actors, which are each responsible for coordinating allreduce options along a single dimension. Each DimensionNode further has several Worker nodes, which are each responsible for executing a single round of allreduce.

The allreduce proceeds as follows. At the beginning of each round, a line master sends a message to the DimensionNode of every node in its row or column ("its peers"). Each DimensionNode then initializes a Worker node to handle the allreduce. The Workers of each node then fetch the nodes' respective model weight matrices. Now suppose there are p peers. The Workers then partition the model weights into p slices, and each scatters the i^{th} slice of its model weights to the i^{th} peer. Then, each Worker, having collected the i^{th} slice of each peer's model weights, performs a reduction to calculate the i^{th} slice of the mean of the model weights. Then each Worker performs a gather operation to get the other slices of the mean model parameter. Finally, each Worker uses elastic averaging as will be explained in Chapter 3 to update its node's model weight matrices. The process is detailed in Algorithm 1.

Algorithm 1 Allreduce algorithm

input: model matrices m, list of peers of length p, thresholds θ_r, θ_c, elastic constant α
create buffers s, r and g
copy m to buffer s

partition s into p blocks. Let $s(i)$ denote the i^{th} block
for each peer i **do**
    send $s(i)$ to peer i
end for
repeat
    receive block from peer i and store it in $r(i)$
until received from θ_r of peers

reduce blocks in r to obtain elementwise mean $\tilde{x}$

for each peer i **do**
    send $\tilde{x}$ to peer i
end for
repeat
    receive computed mean from peer i and store it in $g(i)$
until received from θ_c of peers

$m \leftarrow (1 - \alpha) \cdot m + \alpha \cdot g$ ▷ elastic averaging update

The scatter and gather steps have a configurable quorum; a Worker begins the reduction step when it has received the scattered model parameters from a proportion θ_r of its peers, and a worker begins the elastic averaging step when it has received the reduced model parameters from a proportion θ_c of its peers. If a peer has not reported in, the previous values of the slice of the model parameters it was supposed to have sent are used.

Line masters are responsible for initiating each allreduce round. Line masters allow a certain configurable number of allreduce operations to execute simultaneously. As with the scatter and gather operations, only a quorum of θ_a of participating Workers need to report that they have completed a round of allreduce in order for the next round to begin.

Allreduces run independently per-dimension, so for any given node, the allreduces being performed along its row execute completely independently from the allreduces being performed along its column. Within even a single dimension, there can be multiple allreduces happening simultaneously. Each simultaneous allreduce is handled by a different Worker.

The allreduce operations run in parallel with and occur independently of the training. Other than coordinating access to the model weight matrices through mutexes, the tasks do not otherwise interact. This ensures efficient overlap of communication and computation between the learner and the communicators.

While most implementations of allreduce scale by utilizing a tree structure to ensure that only $O(\log n)$ messages are sent for an allreduce involving n nodes, in BIDMach's allreduce, each nodes sends data to every other peer. Scaling is instead handled by expanding the number of dimensions in the grid as described further in Subsection 6.1.1.

Since allreduce operations in BIDMach do not involve every node, parameters cannot travel from one node to every other node in a single allreduce operation. However, they can still travel to every other node in at most two iterations (one horizontal allreduce and one vertical allreduce). As empirically shown in Chapter 4, this does not have a major adverse impact on the training's ability to converge.

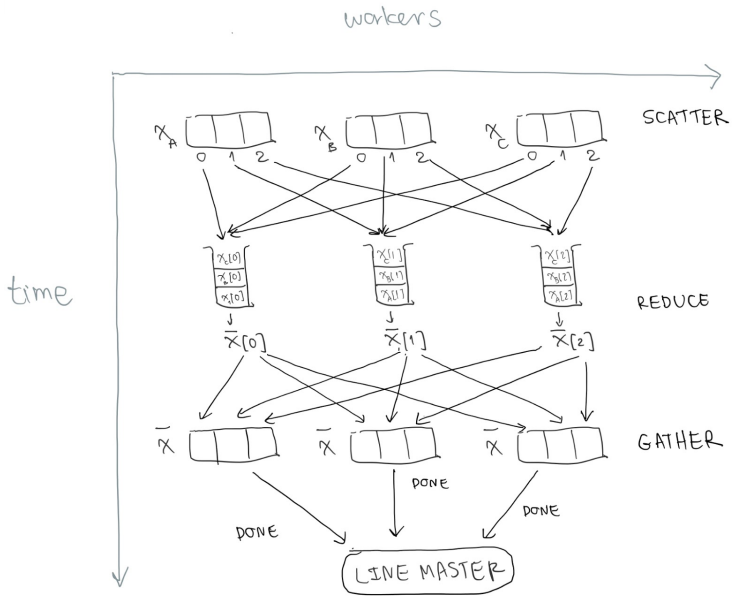

Figure 2.2: Illustration of the allreduce process.

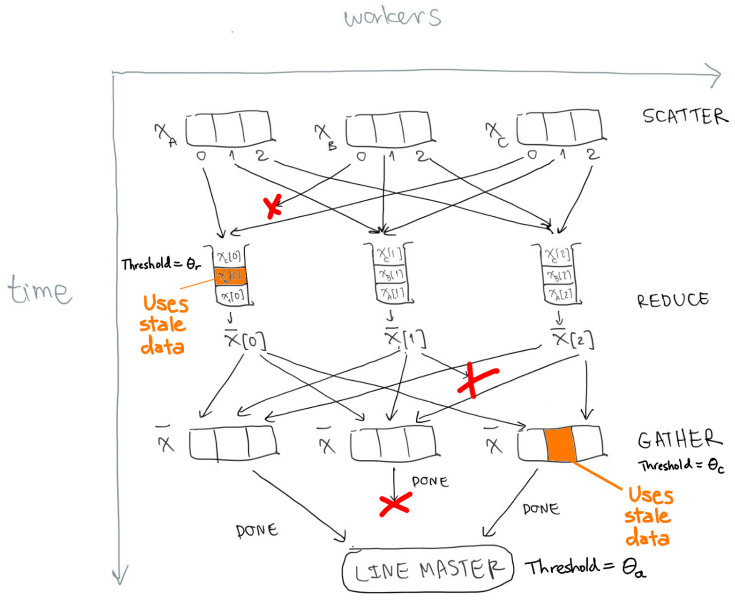

Figure 2.3: Thresholds allow the allreduce to progress even when some nodes fail to respond in a timely manner

2.2 The Dynamic Grid

As the number of nodes in a system increases, the likelihood of some node crashing increases. In a system with thousands of nodes, some amount of node failure is inevitable.

BIDMach has been constructed to handle node joins and departures gracefully. When a node joins, it sends a message to the GridMaster. The GridMaster then assigns that node a location in the grid, resizing the grid and moving other nodes if necessary, and informs that node's new LineMasters of the new addition. When a node quits, the GridMaster receives a Terminated message from the Akka framework, and it removes the node from the grid (downsizing the grid and moving other nodes if necessary), and informs the relevant LineMasters of the node's disappearance.

Each node has a line master actor; however most such actors are dormant. A line master only operates when it has been activated by a GridMaster. When a line master disappears from the network, the GridMaster activates another node's line master in the same row/column. When the grid is resized, the GridMaster deactivates line masters and reappoints new line masters in accordance to how the grid was resized. The GridMaster maintains the grid layout so that the difference in population between any row or column is at most one, to ensure load balancing across all rows and columns.

We have designed BIDMach to be as stateless as possible, to avoid having to maintain any shared state. Each node has its own local shard of the training and testing data, so no state needs to be maintained as to which nodes are using which part of the dataset.

The only component that is absolutely critical to the functioning of the cluster is the GridMaster. Currently, there is no remedy for the GridMaster going down. Remediating this is a future project.

Chapter 3

Elastic Averaging

Elastic Averaging SGD (EASGD), as published by Zhang, Choromanska, and LeCun [17], is an algorithm which reduces the amount of communication needed between nodes and allows for better exploration of the model parameter space.

In this algorithm, each node maintains its own copy of model parameters, and a parameter server also maintains a center copy of model parameters. In each iteration, the model parameters on each node are slightly moved toward the center via an elastic force, while the center model parameters are moved toward the mean of the node model parameters by another elastic force. The equations for these updates are:

$$x_{t+1}^i = x_t^i - \eta g_t^i(x_t^i) - \alpha(x_t^i - \tilde{x}_t) \tag{3.1}$$

$$\tilde{x}_{t+1} = (1-\beta)\tilde{x}_t + \beta\left(\frac{1}{p}\sum_{i=1}^{p} x_t^i\right) \tag{3.2}$$

In these equations, x^i is an arbitrary model parameter variable on some node i, and $\tilde{x}$ is the corresponding center variable. $g_t^i(x_t^i)$ is the stochastic gradient of the loss function with respect to x^i at iteration t. p is the number of nodes, η is the learning rate, and α and β are elastic constants.

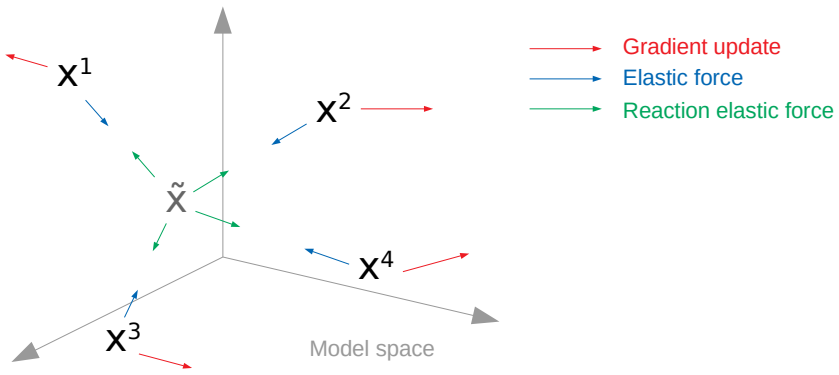

Figure 3.1: Forces experienced by model parameter variables in EASGD.

Note that the update equation for x^i is simply the standard SGD update equation with an extra term of $-\alpha(x_t^i - \tilde{x}_t)$ added. Thus, the averaging can be seen as an extra step added to SGD that maintains the node model parameter x^i's proximity to the center model parameter $\tilde{x}$. In vanilla EASGD, β is set to $p\alpha$, which makes the elastic force acting on the center model parameters $\tilde{x}$ equal and opposite to the sum of the elastic forces acting on the node parameters.

To implement momentum, Equation 3.1 can be modified analogously to normal momentum SGD, as follows:

$$v_{t+1}^i = \delta v_t^i + \eta g_t^i(x_t^i)$$
$$x_{t+1}^i = x_t^i + v_t^i - \alpha(x_t^i - \tilde{x}_t)$$

An asynchronous variant of EASGD also exists. In the asynchronous variant, each worker node queries the parameter server for the current value of $\tilde{x}$ every τ steps, calculates the elastic difference $\alpha(x^i - \tilde{x})$, and subtracts the elastic difference from its model parameters x^i. It then sends the elastic difference to the parameter server, which then applies the same but opposite difference to $\tilde{x}$. The algorithm is detailed in [17].

BIDMach makes several modifications to the EASGD algorithm. Firstly, BIDMach eliminates the parameter server in EASGD in favor of allreduce operations. In order to avoid having to keep center model parameters around between allreduce iterations, we set $\beta = 0$, reducing $\tilde{x}$ to a simple mean of every x^i that can be computed afresh in every allreduce operation. The allreduce is also decoupled from the training so that both run in parallel. Thus, the elastic averaging step effectively occurs once every τ iterations, where τ depends on the relative speed of the trainer versus the allreducer. On an EC2 p2.xlarge instance, τ is empirically equal to about 3. As detailed in Chapter 4, this modified EASGD has been empirically shown to converge.

Chapter 4

Results

We used BIDMach at commit ID 14026b0 and MXNet at commit ID 278c708[1], both compiled with CUDA 8.0, to train AlexNet on the ILSVRC12 dataset, using a batch size of 128, learning rate of 1×10^{-2} and weight decay factor of 0.0005. The learning rate decays by a factor of 0.1 after 40 epochs, and again after 80 epochs. These experiments were run on p2.xlarge instances on AWS.

4.1 Scaling

Table 4.1 shows the weak scaling results for BIDMach, MXNet in synchronous mode, and MXNet in asynchronous mode. BIDMach scales well, achieving 87% of ideal speedup, while MXNet's scaling is inferior, achieving less than 50% of ideal speedup.

BIDMach

# of nodes	Total samples/second	Total gflops	Speedup	% of ideal
1	193.4	1199.2	—	—
4	672.4	4171.3	3.48	87%
16	2662.5	16499.6	13.77	86%

MXNet synchronous

# of nodes	Total samples/second	Speedup	% of ideal
1	147.46	—	—
4	246.44	1.67	42%
16	738.56	5.01	31%

MXNet asynchronous

# of nodes	Total samples/second	Speedup	% of ideal
1	147.46	—	—
4	259.69	1.76	44%
16	856.68	5.80	36%

Table 4.1: BIDMach and MXNet speedup when training AlexNet on different numbers of nodes

4.2 Roofline Analysis

We can analyze the core allreduce in BIDMach (lines 3 to 16 in Algorithm 1) using the Roofline model. The allreduce runs on one core per node. The AWS instances we used have Intel Xeon E5-2686 CPUs, which run at 2.30 GHz, and can process up to 32 floating point numbers per instruction cycle. Thus our peak flop rate is $2.30 \times 32 = 73.6$ GFLOPS.

[1]As of this writing, commit 278c708 is part of a pull request fixing a bug in MXNet.

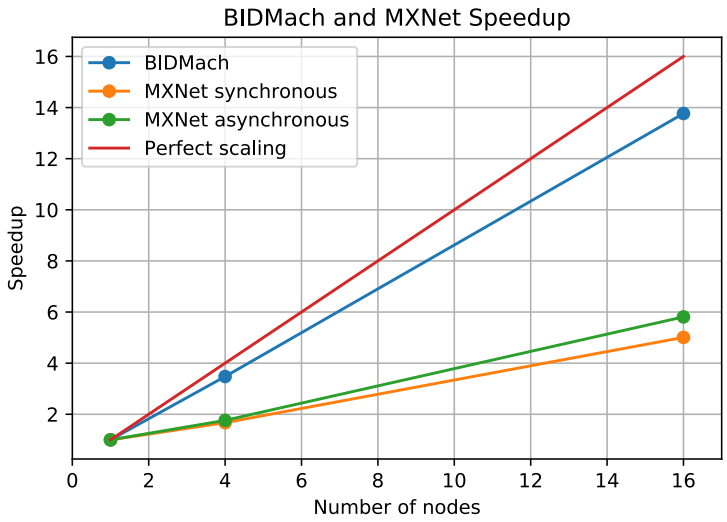

Figure 4.1: BIDMach and MXNet speedups

Now consider the data transfer rates. The most limiting data transfer rate is network bandwidth. The network speed between our EC2 nodes is approximately 1.24 Gbits/s in each direction, or 2.48 Gbits/s overall.

Now, we consider how fast an ideal system can do the allreduce. Suppose that the model parameters consist of N 32-bit floating point numbers (and therefore take up $4N$ bytes).

Let us analyze the computation costs. In the allreduce algorithm, the three buffers s, r, and g are all of size $4N$ bytes. Recall that there are $\sqrt{n}$ peers in each allreduce. The one step in Algorithm 1 involving floating point operations is line 10, where we perform $\sqrt{n}-1$ additions of blocks, each of $N/\sqrt{n}$ floats.

Let us now analyze the communication costs. In line 5, Workers send a block of size $4N/\sqrt{n}$ for $\sqrt{n}-1$ times. In line 8, Workers receive blocks of the same size $\sqrt{n}-1$ times. Then the same process repeats in lines 12 and 15 (albeit with a different buffer). Altogether, the number of bytes sent and received by one node is $4(\sqrt{n}-1)(4N/\sqrt{n})$. Thus, the ideal arithmetic intensity of the allreduce operation is

$$\frac{\frac{N}{\sqrt{n}}(\sqrt{n}-1)}{4(\sqrt{n}-1)\frac{4N}{\sqrt{n}}} = \frac{1}{16}.$$

Per the Roofline model, this allreduce will be compute-limited if its arithmetic intensity is greater than $73.6\text{G}/(2.48\text{G}/8) \approx 237$. Since $1/16 \ll 237$, the allreduce is very much bandwidth-limited. The ideal FLOPS is thus given by the product of arithmetic intensity and peak bandwidth, which is $1/16 \times 2.48\text{G}/8 = 19.375$ MFLOPS.

To analyze BIDMach's actual performance with respect to this ideal, we tested BIDMach's allreduce on the AlexNet model, which has $N = 62{,}378{,}344$, on a cluster of 4 nodes and a cluster of 16 nodes. On each cluster we measured the duration of each allreduce operation, starting immediately after model matrices had been copied and ending immediately before the elastic averaging step.

On the 4 node cluster, we profiled a total of 1004 iterations of allreduce among the 4 nodes, and they took 3374.439 seconds collectively. Since each allreduce operation takes $(N/\sqrt{n})(\sqrt{n}-1) = 62378344 \times 1/2$ floating point operations, there are $62378344 \times 1/2 \times 1004$ floating point operations in total. The FLOPS is therefore $62378344 \times 1/2 \times 1004/3374.439 = 9279743$, which is 47.9% of the ideal.

On the 16 node cluster, we profiled a total of 4274 iterations of allreduce among the 16 nodes, and they took 16061.724 seconds collectively. Since each allreduce operation takes $(N/\sqrt{n})(\sqrt{n}-1) = 62378344 \times 3/4$ floating point operations, there are $62378344 \times 3/4 \times 4274$ floating point operations in total. The FLOPS is therefore $62378344 \times 3/4 \times 4274/16061.724 = 12449086$, which is 64.3% of the ideal.

4.3 Bandwidth Usage

Having shown that BIDMach's allreduce is fundamentally limited by network bandwidth, we can also demonstrate that BIDMach is most effectively utilizing this bandwidth. When the allreduce step is run in isolation, it consumes nearly all available network bandwidth, as Table 4.2 shows. This means that BIDMach's core allreduce does overlap communication and computation to a great extent, and communication is seldom blocked on computation.

Cluster size \ Data size	80,000	8,000,000
2×2	2.33 Gbit/s	2.35 Gbit/s
4×4	2.34 Gbit/s	2.32 Gbit/s

Table 4.2: Bandwidth consumed by raw allreduce

4.4 Model Quality

Empirically, BIDMach is able to achieve high model quality under our distributed framework. BIDMach training AlexNet on a 2×2 cluster and 4×4 cluster, as shown in Figures 4.2 and 4.3, approaches the validation accuracy of 0.5767 achieved by the original AlexNet paper [13]. Changing batch size and elastic constant led to no significant difference in results.

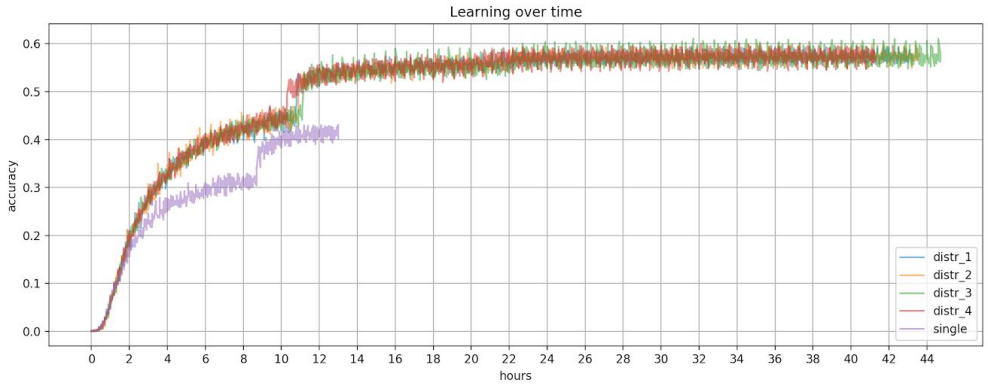

Figure 4.2: BIDMach validation accuracy on AlexNet on a 2×2 cluster

Notice the higher variance for the 4×4 cluster. We believe that it is likely due to the fact that, as the cluster size grows larger, each node in the cluster gets a smaller slice of the dataset to train on; as a result, the gradients that each node locally generates have higher variance.

On the other hand, MXNet shows an inferior ability to scale. We trained AlexNet using MXNet on clusters of 1, 4, and 16 nodes, in both synchronous and asynchronous modes, using the parameters previously described. Synchronous mode performance of MXNet on 4 nodes achieved an accuracy of 45% after 90 epochs and on 16 nodes, 35%, as shown in 4.4 and 4.5. However, in asynchronous mode, MXNet performs poorly: a cluster of 4 nodes is not able to gain beyond 11% validation accuracy, and a cluster of 16 nodes completely failed to train at all, as show in 4.6 and 4.7. Whereas BIDMach is able to achieve high training accuracy with asynchronous training due to elastic averaging, MXNet suffers from total convergence collapse due to the iteration divergence problem.

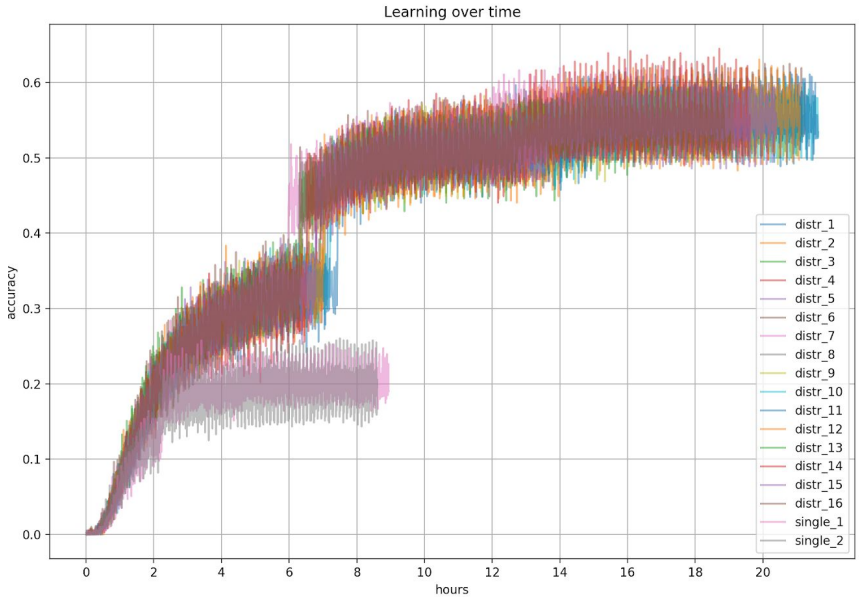

Figure 4.3: BIDMach validation accuracy on AlexNet on a 4×4 cluster

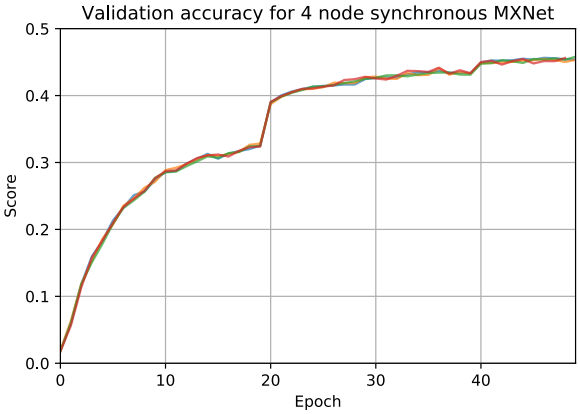

Figure 4.4: MXNet synchronous training performance on AlexNet with 4 nodes

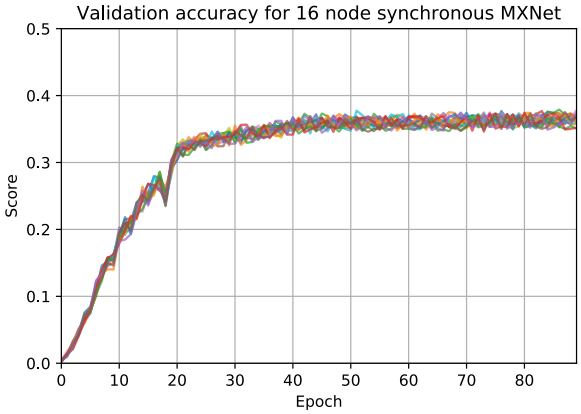

Figure 4.5: MXNet synchronous training performance on AlexNet with 16 nodes

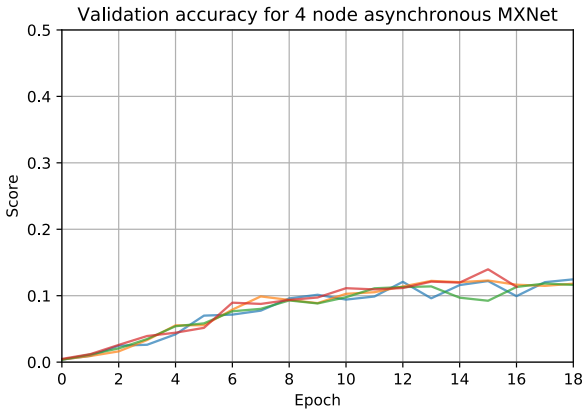

Figure 4.6: MXNet asynchronous training performance on AlexNet with 4 nodes

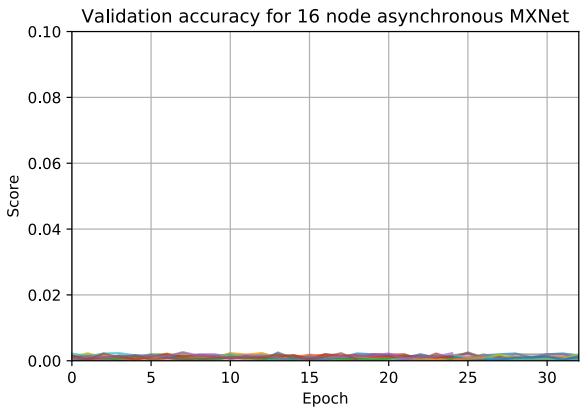

Figure 4.7: MXNet asynchronous training performance on AlexNet with 16 nodes

Chapter 5

Related Work

5.1 Caffe

Caffe has been a *de facto* standard for image-based machine learning for several years. Originally developed by Yangqing Jia and the Berkeley Vision and Learning Center (now BAIR), it has been forked and extended many times by different parties to meet different requirements. (For instance, FireCaffe [10] is aimed to minimize communication overhead on HPC clusters; S-Caffe [1] is optimized for multi-GPU training; Intel-Caffe [11] and NVIDIA-Caffe [16] are tuned to use efficient primitives on Intel Xeon processors and NVIDIA GPUs respectively). Many tools exist to allow other machine learning frameworks to interoperate with Caffe's model format; as one of my research tasks, I added functionality to BIDMach to load trained and untrained Caffe models.

Caffe does not natively support multi-machine distributed training. An early, formerly widely used derivative of Caffe, CaffeOnSpark [4], accomplishes distributed training by means of performing allreduce reductions on a Spark cluster. Other Caffe derivatives such as FireCaffe and S-Caffe achieve much greater communication efficiency through highly optimized, highly specific implementations.

More recently, Facebook has released Caffe2 [8], a successor to Caffe. Caffe2 supports distributed training natively through tree-structured allreduce operations, and coordinates the allreduces to occur in parallel with backpropagation.

5.2 MXNet

Apache MXNet is a deep learning framework, supported by Amazon, that uses a parameter server for distributed learning. It was originally designed and developed by Mu Li, then a PhD student at Carnegie Mellon. He has detailed its operation in several papers [15, 5, 14].

MXNet builds on the parameter server approach introduced by other deep learning systems by further optimizing the parameter server for GPU operations. MXNet also supports several data consistency models for simultaneous task execution (sequential consistency, eventual consistency and bounded delay) to control the tradeoff between system performance and algorithm convergence rate. MXNet achieves fault tolerance by replicating the parameter server and using redundancy between worker nodes, so when a server or worker node fails it can seamlessly switch to a redundant node. To safeguard consistent behavior after a failure, vectors communicated between nodes have a clock attached to them which records each node's timestamp for that vector.

BIDMach does not currently run multiple training operations in parallel within a single node; in MXNet terms, it only supports sequential consistency. BIDMach achieves fault tolerance through the dynamic grid, which reassigns nodes' responsibilities when a node joins or leaves. BIDMach also versions the configuration messages sent from GridMasters to LineMasters and from LineMasters to Workers with a monotonically increasing number, so after a failure, workers can still distinguish new messages from outdated ones.

5.3 GossipGrad

GossipGraD [6] is a recently published distributed SGD algorithm that aims to solve the same problems that our grid-structured elastic allreduce aims to solve. GossipGraD eschews allreduce operations in favor of a gossip-based protocol. Previous gossip-based protocols, in which each node chooses a random partner to send updates to, have suffered from performance and convergence degradation at scale, due to communication imbalance (as partners are randomly rather than systematically selected), poor latency due to synchronous communication, and ineffective diffusion of gradients [12, 3, 6]. In GossipGraD, nodes exchange gradients with other nodes in a systematic order. This approach, the authors show, provides similar accuracy and validation performance to SGD and asynchronous SGD.

While BIDMach arranges nodes in a d-dimensional cube, GossipGraD instead arranges them in a circle. In a repeating process lasting $\log n$ timesteps per cycle, each node sends its model parameters to the node 2^i steps to the left (where i is the current timestep in the cycle). After each cycle, the order of the nodes in the circle is shuffled, and the data subset that each node trains on is shuffled as well.

We have not had time to thoroughly evaluate GossipGraD due to its very recent publication date. However, it would be interesting to try combining GossipGraD with elastic updates, which may be able to increase the exploration of GossipGraD.

Chapter 6

Conclusion

By combining elastic averaging with a dynamic grid design and modified allreduce, we have successfully built a distributed machine learning system that achieves high model quality even while using asynchronous communication between nodes. Our system furthermore handles node failure and node arrival gracefully and resiliently, avoids excess latency, and makes maximum use of parallelism.

Grid-structured allreduce is not limited to use in BIDMach, and can be potentially adapted to other distributed applications which use allreduce operations to disseminate information among nodes (as we do with model parameters). This will reduce the number of nodes involved in every allreduce to $O(\sqrt{n})$, or more generally, $O(\sqrt[d]{n})$ for a d-dimensional grid. The robustness of this allreduce in the face of lagging or failing nodes makes it further useful in environments of thousands of nodes, where latency is inevitable and node failures and restarts are frequently encountered.

Likewise, elastic averaging can also be adapted to other machine learning systems using different distributed architectures and potentially help these systems to achieve better model quality, especially when doing asynchronous training. Depending on the architecture, the relative gains of using elastic averaging may vary; exploring its effect on various architectures is left as a future project.

6.1 Future Directions

There are plenty of improvements that can be explored for BIDMach:

6.1.1 d-dimensional grid

It would be straightforward to extend the two-dimensional grid that is currently used to d dimensions. This would reduce the number of nodes in each allreduce from $\sqrt{n}$ to $\sqrt[d]{n}$.

If d is chosen to be in $\Theta(\log n)$, then the number of nodes in each allreduce operation is $\Theta(\sqrt[\log n]{n}) = \Theta(1)$, the same order of growth as for communication with a parameter server.

It would take a greater number of iterations for model parameters to propagate over all axes (d iterations instead of 2 iterations). However, when the model parameters do propagate to all other cells, they might be applied with an increased weight compared to the two-dimensional case. Suppose that X and Y are two nodes positioned in a d-dimensional grid such that they do not lie on the same hyperface (e.g. X and Y are opposite corners). After d iterations, the values from X will have been discounted by a factor of $d!\left(\frac{\alpha}{\sqrt{n}}\right)^d$. This is because there are $d!$ paths between X and Y, one for each ordering of axes; along each path, the value is discounted by a factor of $\alpha/\sqrt{n}$ per time step, or $(\alpha/\sqrt{n})^d$ in total. If we again let $d \in \Theta(\log n)$, then this discount factor would be $d!(\alpha\Theta(1))^d$, which increases with d.

Thus, our particular EASGD algorithm would likely remain stable and converge correctly in the n-dimensional case, as it does for the 2-dimensional case. However, more work is needed to determine whether this actually is the case.

6.1.2 Reduce the amount of data communicated

Currently, BIDMach allreduce workers communicate the entire contents of model matrices to each other, without any form of compression. However, network bandwidth is one of the scarcest resources in a datacenter [2]. It would thus be greatly beneficial to reduce the amount of data that needs to be sent over the wire.

As it turns out, in some cases (e.g. logistic regression), the model matrices consist of many zeros, so these matrices can be encoded in CSR form, reducing the amount of data transmitted [14]. This compression technique is used by MXNet.

In [9], Han, Mao and Dally have gone even further, detailing a method for reducing the amount of memory needed by deep neural networks by applying pruning to remove unimportant neural connections, weight quantization to enforce weight sharing and thus compressibility, and Huffman coding to more efficiently encode the resulting distribution of quantized weights. Using their approach, they were able to reduce the storage required by AlexNet from 240 MB to 6.9 MB and the storage required by VGG-16 from 552 MB to 11.3 MB, without any loss of accuracy. The resulting models are small enough to be bundled in mobile apps and used on mobile devices without consuming voracious amounts of power.

Implementing these kinds of data compression techniques in BIDMach is a desirable future goal.

6.1.3 Improve coordination between parameter exchange and training

Currently, within each node, the learner thread and the allreduce Workers run in parallel, oblivious to each other's progress, and only coordinate through mutexes around the model matrices. The allreduce Workers are programmed to write model matrices in reverse order to reduce that chance that the learner and the worker will repeatedly try to lock the same matrices. However, if a Worker tries to update matrices during the backward pass, then repeated contention for mutexes between the Worker and the learner is a possibility.

It would be desirable for the allreduce Workers and the learner to have a greater degree of knowledge of each other's progress. Implementing this would allow them to avoid trying to simultaneously access the same matrices, and could allow the user to manually adjust the ratio of training iterations per allreduce iteration.

6.1.4 More sophisticated handling of thresholds

Suppose that the threshold for a particular allreduce step is 70%. Currently, that means BIDMach will move on to the next step as soon as 70% of nodes finish that allreduce step. However, a more ideal solution would be for the allreduce step to normally wait for 100% of nodes, and only move on with 70% of nodes reporting when some timeout is exceeded.

6.1.5 Greater tolerance for failure conditions

There is currently no failure tolerance for the grid master; if the grid master crashes during a computation, node joins and departures would cease to be processed, which can compromise the correctness of the computation. Tolerance to failure could be added by adding redundant grid masters and using a leadership election to choose one to be active.

6.1.6 Port to HPC

In order for BIDMach to be used effectively on an HPC system with thousands of nodes, it would likely be necessary for BIDMach to support inter-node communication via Infiniband. BIDMach currently uses TCP/IP as its only method of interconnection between different nodes. While IP over Infiniband could be used, adding a dedicated Akka transport for Infiniband would probably give higher performance.

6.1.7 Momentum exchange simulation

SGD gradients suffer from an imbalanced distribution of magnitude in different dimensions, as illustrated in Figure 6.1. This imbalance carries over to momentum as well.

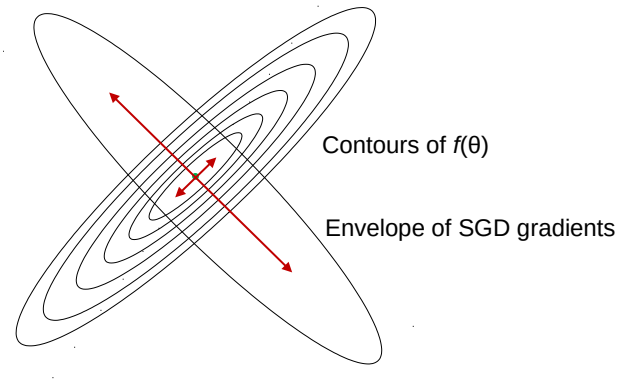

Figure 6.1: Uneven distribution across dimensions of SGD gradient magnitude

By the equipartition theorem, the average energy of particles in a gas is equally distributed across all dimensions. This property holds because, unlike in SGD, particles in a gas are constantly exchanging energy across different dimensions with one another. As suggested by Professor John Canny, it may be possible to correct the SGD momentum magnitude imbalance by treating the nodes as colliding particles in a particle simulation and allowing the nodes to exchange momentum with one another. This momentum exchange, in theory, should homogenize the "energies" (and thus momentum magnitudes) of each node across dimensions.

References

[1] Ammar Ahmad Awan et al. "S-Caffe: Co-designing MPI Runtimes and Caffe for Scalable Deep Learning on Modern GPU Clusters". In: *SIGPLAN Not.* 52.8 (Jan. 2017), pp. 193–205. ISSN: 0362-1340. DOI: 10.1145/3155284.3018769. URL: http://doi.acm.org/10.1145/3155284.3018769.

[2] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. 2013. URL: http://dx.doi.org/10.2200/S00516ED2V01Y201306CAC024.

[3] Michael Blot et al. "Gossip training for deep learning". In: *CoRR* abs/1611.09726 (2016). arXiv: 1611.09726. URL: http://arxiv.org/abs/1611.09726.

[4] *CaffeOnSpark*. Yahoo! URL: https://github.com/yahoo/CaffeOnSpark.

[5] Tianqi Chen et al. "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems". In: *CoRR* abs/1512.01274 (2015). arXiv: 1512.01274. URL: http://arxiv.org/abs/1512.01274.

[6] Jeff Daily et al. "GossipGraD: Scalable Deep Learning using Gossip Communication based Asynchronous Gradient Descent". In: *CoRR* abs/1803.05880 (2018). arXiv: 1803.05880. URL: http://arxiv.org/abs/1803.05880.

[7] Jeffrey Dean et al. "Large Scale Distributed Deep Networks". In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'12. Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 1223–1231. URL: http://dl.acm.org/citation.cfm?id=2999134.2999271.

[8] Priya Goyal et al. "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour". In: *CoRR* abs/1706.02677 (2017). arXiv: 1706.02677. URL: http://arxiv.org/abs/1706.02677.

[9] Song Han, Huizi Mao, and William J. Dally. "Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding". In: *CoRR* abs/1510.00149 (2015). arXiv: 1510.00149. URL: http://arxiv.org/abs/1510.00149.

[10] Forrest N. Iandola et al. "FireCaffe: near-linear acceleration of deep neural network training on compute clusters". In: *CoRR* abs/1511.00175 (2015). arXiv: 1511.00175. URL: http://arxiv.org/abs/1511.00175.

[11] *Intel-Caffe*. Intel. URL: https://github.com/intel/caffe.

[12] Peter H. Jin et al. "How to scale distributed deep learning?" In: *CoRR* abs/1611.04581 (2016). arXiv: 1611.04581. URL: http://arxiv.org/abs/1611.04581.

[13] Alex Krizhevsky. "One weird trick for parallelizing convolutional neural networks". In: *CoRR* abs/1404.5997 (2014). arXiv: 1404.5997. URL: http://arxiv.org/abs/1404.5997.

[14] Mu Li. "Scaling Distributed Machine Learning with System and Algorithm Co-design". PhD thesis. Pittsburgh, PA 15213: Carnegie Mellon University School of Computer Science, Feb. 2017.

[15] Mu Li et al. "Communication Efficient Distributed Machine Learning with the Parameter Server". In: *Advances in Neural Information Processing Systems 27*. Ed. by Z. Ghahramani et al. Curran Associates, Inc., 2014, pp. 19–27. URL: http://papers.nips.cc/paper/5597-communication-efficient-distributed-machine-learning-with-the-parameter-server.pdf.

[16] *NVIDIA-Caffe*. NVIDIA. URL: https://github.com/NVIDIA/caffe.

[17] Sixin Zhang, Anna Choromanska, and Yann LeCun. "Deep learning with Elastic Averaging SGD". In: *CoRR* abs/1412.6651 (2014). arXiv: 1412.6651. URL: http://arxiv.org/abs/1412.6651.