Thresholded Gradient Synchronization in Distributed ConvNets

Asha Anoosheh
UC Berkeley
asha@berkeley.edu

Eugene Che
UC Berkeley
eyche@berkeley.edu

Michael Chen
UC Berkeley
themichaelchen@berkeley.edu

Yeung John Li
UC Berkeley
liyeungjohn@berkeley.edu

Edward Wang
UC Berkeley
ewang16@berkeley.edu

Abstract

Training deep neural networks has traditionally been done on single machines. However, recent developments have shown several methods to distribute the training process and the training data across a network of multiple machines. Previous works showed that the distributed implementations move the bottleneck of training time from the CPU/GPU performance of a single machine to the communication delay between machines. In this paper, we demonstrate multiple methods to alleviate these bottlenecks on distributed convolutional neural networks while retaining classification accuracy.

1 Introduction

In recent years, there have been multiple works on finding ways to reduce training time for neural networks by distributing the workload over multiple nodes, including methods that have been kept proprietary by some companies. However, past a certain number of nodes, the distribution of the training workload to many nodes no longer proves effective, as the bottleneck becomes the Inter-Process Communication (IPC) during synchronization. Here we explain multiple approaches to remedy this bottleneck by reducing IPC.

1.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) have proven to be popular for image recognition tasks in the past few years. Compared to traditional neural networks, they avoid fully connected layers closer to the input layer. Rather, layers consist of multiple small kernels that are convolved with their input. As a result, weight matrices can be much smaller for high resolution image inputs. This cuts the number of parameters greatly, relative to fully-connected networks, and allows for significantly faster training times. In this paper we focus on convolutional networks trained using stochastic gradient descent (SGD) only.

An important aspect of neural net training to note is its ability to train in mini-batches, moving stochastic gradient-descent closer to batch gradient-descent. Multiple inputs can be passed in simultaneously, and their resulting gradient matrices are simply added together element-wise to form a larger gradient that is to be used to change the network parameters, making the resulting descent steps smoother and hopefully in the direction of a local minimum more representative of the data as a whole. This idea is important for the discussion of distributed nets.

1.2 Distributed neural networks

The training times of deep CNNs on large datasets, such as the millions of images used for most image classification tasks, are still on the order of days, even using the highest-end GPUs. This makes performing various experiments on a network or tuning hyperparameters cumbersome, as one may have to wait days for each training configuration to complete.

Distributed neural networks aim to reduce training times by spreading the work over a multiple machines. There are multiple design architectures used for distributing networks, namely model-parallel and data-parallel. The former spreads a single network across multiple machines; in this paper we focus on the latter, which divvies different input data to multiple replicas of the network running in parallel. More specifically, we imagine the setup used many systems where each node in a cluster has an identical copy of the network on it [2]. Data is partitioned along the mini-batch dimension, so in the case of CNNs, that means distributing a network with batch size b among n nodes gives each node $\frac{b}{n}$ data points (images). Such architecture patterns are described in further detail in Dahl *et al.*[1].

Following each forward/backward pass, all gradients are calculated and summed up locally per node, yet no weight updates are performed until all nodes synchronize their gradients. To ensure the network stays consistent across all nodes at all times, the gradient matrices of each node must be added together to represent the update of an effective batch size of b. Only then the update step is performed. This means each node must block its operation until everyone has synchronized.

The actual communication is assumed to be done as is standard: by transferring data from the GPU to CPU (assuming we always use GPUs for our CNNs), encoding the data for the desired network transmission protocol, and sending it across network interconnects to another node. These synchronization processes are now known to be the bottlenecks holding distributed networks from efficiently scaling past a certain number of nodes, and thus the target of our experiment [2].

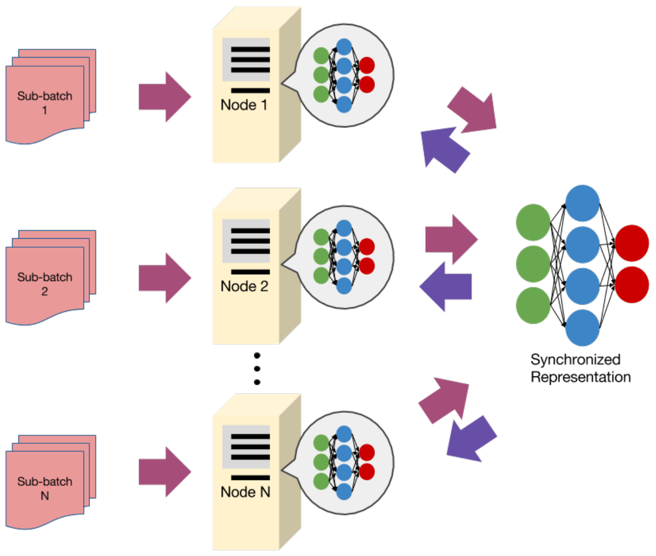

Figure 1: High-level view of data-parallel distributed net.

1.3 Goal

Given a cluster with a copy of the same convolutional network on each node, we aim to reduce the amount of data synchronized among worker nodes, thus reducing the total synchronization time for the entire cluster while maintaining acceptable accuracy. Currently, nodes in the distributed network described send entire gradient matrices; we abandon this in favor of lowering interprocess communication. In practice, most of the gradients propagated to a layer at each iteration are nonzero, but very close to zero [9]. Instead of inefficiently propagating gradient matrices that contain such small values, we can avoid unnecessary communication by undertaking a solution that delays weight updates by thresholding gradient values prior to inter-node communication.

A clear potential problem with doing this is that, while training time is reduced, the trained neural net will bear a different and likely lower accuracy. Furthermore, the decrease in accuracy also depends on the value of threshold chosen. Thus, it is important to find the best threshold that balances the tradeoff between accuracy and training time reduction. Our goal is to experiment and analyze with different threshold values and algorithms to explore this tradeoff.

1.4 Related works

Our inspiration came from the distributed neural net framework, FireCaffe, by Iandola *et al.*, whose paper mentions the potential use of a novel gradient synchronization scheme for distributed neural nets as devised by Amazon [4] [9]. Such a scheme was not explored with Firecaffe, and thus it is implemented in this experiment.

Amazon's scheme synchronized only gradients whose absolute value exceeded a threshold τ, and furthermore only communicated gradients as either τ or $-\tau$ (represented over the network as 1 or 0 for concision,) accumulating the residual values for future synchronization iterations. Their distributed network performed word recognition on a proprietary dataset achieved results, achieving a 54x speedup on a 80-node cluster with less than 2% relative error reduction. Yet having only relative error numbers for an unknown proprietary dataset is not entirely conclusive in our opinion. This is one of the methods we explore in this paper, with the aim to record both absolute and relative errors on a completely different type of dataset and network, namely image recognition with convolutional neural nets.

Two more similar techniques instead perform more standard quantization techniques to the gradients. The first, by Seide *et al.*, still maintains the residual-buildup property, yet computes multiple quantizations per iteration to achieve 1 bit communicated gradients [8]. The quantization function and scheme are not described in detail, and like the Amazon scheme, it is performed on a speech-recognition model rather than a CNN. The other, by Oland and Raj, computes entropy of the gradients each iteration and utilizes Huffman encoding before sending them, meaning each node communicates dynamic and uniquely compressed numbers [13].

Other methods, which will not be elaborated upon for sake of brevity, include Async-SGD, Elastic Averaging, Model Averaging, this *one weird trick*, and Downpour SGD, which was used by one of the first distributed machine learning systems, Google's DistBelief [11] [12] [10] [6] [2]. These various synchronization reduction schemes, although they strive to achieve the same goal, are not as closely related to ours as the above two are. Yet these, and all the aforementioned methods, achieved significant success in their targeted purpose, and helped guide our experimental choices.

2 Method

We target the aforementioned gradient matrices communicated in a distributed network in two ways: reducing the representation size of the numbers in each matrix and/or reducing the number of elements sent overall. While the former will always reduce the amount of communicated data, the latter will only decrease data should the number of elements sent be less than half the total size of its matrix (over half the matrix is zeros). If this is satisfied, the matrix is said to be sparse, and the gradients can be efficiently represented as a sparse-matrix data structure, reducing communicated data even further.

Not only would this decrease the number of data packets sent across the network interconnecting the compute nodes, it also means much less time spent encoding data for communication by each CPU, as well as reducing the amount of data transferred from the GPU to the CPU for the network communication to occur in the first place. We assume the cost of performing this newly-introduced thresholding operation on GPU is negligible compared to these other advantages, as it is a single-pass, parallelizable kernel call for each layer's gradient matrix.

Though there exist various implementations to lessen the number of total node-to-node communications that must occur to achieve full synchronization, each node sending out its gradients at least once is unavoidable. Thus our experiment is designed to be independent of the synchronization method used and only on this fact.

2.1 Procedure

We implement three different approaches, each making use of an unknown threshold value T, to our task: Whole gradient-transfer (WGT), Binarized gradient transfer (BGT), and Quantized gradient transfer (QGT). The first simply places a fixed threshold on gradients prior to synchronization: only gradients with a magnitude greater than T will be sent, while the remaining values are summed over time so they may be sent in future iterations. The second is the approach taken in Amazon's experiment: if a gradient has magnitude greater than T, then either T or $-T$ will be sent (depending on the original sign of the gradient,) and its difference will be stored as before. The last approach, inspired by that of Seide *et al.*, is quantization of the gradient values using a method similar to the second approach: if a gradient has magnitude greater than T, then the greatest multiple of T less than the gradient magnitude will be sent.

Of course since T is fixed for the duration of a training run, every node knows its value, and we can restrict the sizes of each gradient value sent. The gradients in WGT are the same size as the default, which we assume are 32-bit floats. In BGT, we can simply send any two binary indicators (i.e. $+1$ and -1) to represent T and $-T$, meaning only 1 bit required per gradient. Similarly for QGT we can just send the number of multiples of T instead of the actual value. We purposely restrict this multiple to be less than 256, so that it can fit in a single byte, or 8 bits. Reducing the number of bits required to represent the data will reduce the overall data bandwidth.

The three methods, denoted as *a, b,* and *c* respectively, are formalized as follows:
Given a layer's gradient array G and a threshold T, perform

```
for $i \leftarrow 1$ to $length(G)$ do
    if $|G_i| \geq T$ then

        (a)  send $G_i$
             $G_i \leftarrow 0$

        (b)  if $G_i > 0$ then
                 send $T$
                 $G_i \leftarrow G_i - T$
             else if $G_i < 0$ then
                 send $-T$
                 $G_i \leftarrow G_i + T$

        (c)  $m \leftarrow \lfloor \frac{G_i}{T} \rfloor \cdot T$
             $m = min(m, 255)$
             send $m$
             $G_i \leftarrow G_i - m$
```

after each backpropagation step and before each weight update of a single node.

It is very important to note that these all keep the residual of their thresholded values from G locally, summing them into the next iteration's gradients, so that they may eventually be communicated. Thus it doesn't avoid these updates, but rather delays them. And the actual threshold step takes $O(1)$ time within each layer, as it is a parallelizable filter-and-partition done via a single-pass CUDA kernel, visualized in Figure 2.

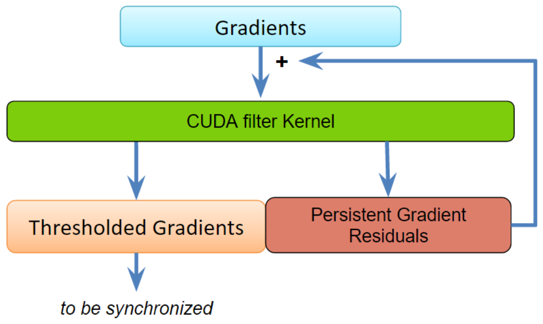

Figure 2: Thresholding procedure on GPU.

2.2 Setup

This experiment was performed running Caffe using an NVidia Tesla K20c GPU [5]. The CNN model chosen was the Network-in-Network (NiN) trained on the famous ImageNet dataset [7] [3]. We used a batch size of 128 and trained for 220,000 iterations, using a base learning rate of 0.01, momentum of 0.9, weight decay of 0.0005, and a factor-of-ten learning-rate reduction every 100,000 iterations.

The NiN achieves AlexNet accuracies with a model size of 29MB compared to AlexNet's 230MB; this is already one of the smallest model sizes on a competitive ImageNet-trained [7]. We use this compressed model to secure an already-suitable starting point for distributed communication before applying our further optimizations to it, which should render our results more meaningful.

We did not have access to a cluster to implement a truly distributed network. But since a distributed network with n nodes is identical to a network with n times the batch size, we can simulate this locally. Performing our threshold operations on n equal partitions (in the batch-size dimension) of each layer's gradients will achieve the same result. Here we arbitrarily chose $n = 8$ as our number of "nodes" to hold constant throughout the experiment, meaning our T values are being tuned for $\frac{128}{n} = 16$ images per node; though it's easy to see that the ideal T should, on average, increase linearly with this number as it is a summation of similar values. In practice n may be much higher, but that also means the relative speedup from communication reduction shall be even better, theoretically. Yet because of this simulated scenario, we cannot quantitatively measure the decrease in computation and communication that will result from a true distributed environment.

A downside is that thresholding in general will cause double the memory usage per layer, as partitioning the gradient matrix into the to-be-sent matrix and the residual matrix results in two matrices of the same size, as the former matrix is sparsified only after this step. This is why the batch size had to be sliced to 128, even though our GPU could handle 256. Though the resulting accuracies between these two sizes are approximately equal in the end.

3 Results

3.1 Baseline

Our baseline with no modifications achieved a top-1% accuracy of **57.9175%** on the validation set. The number of gradients communicated by all nodes per iteration (using a batch size of 128,) is 60,719,360. So over 220,000 iterations, 13,358,259,200,000 gradients are communicated total. Under the assumption that the gradients are, by default, sent as 32-bit floats, just over **53,433 GBs** must be sent over the course of the whole training, in the conventional implementation.

Note that this is merely the communication of each node outward once, serving as a lower bound. It does not include the potential extra communication that occurs in certain synchronization models nor the unavoidable overhead of the data structure used – most likely a csr-sparse-matrix in this case, whose overhead is linear in the number of data points anyway.

3.2 Whole gradient transfer

Table 1: Whole gradient transfer results

Threshold	Top 1% accuracy	Total GBs transferred	Ratio (%/GB)
5e-4	58.2681	15583.03	0.0037
7.5e-4	58.1941	11134.76	0.0052
8.75e-4	57.9464	9691.21	0.0060
1e-3	57.9424	6572.01	0.0088
2.5e-3	57.2790	2723.90	0.0210
5e-3	56.4698	933.35	0.0605

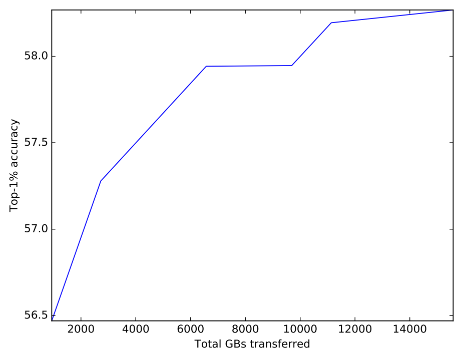

Figure 3: Accuracy vs data communicated.

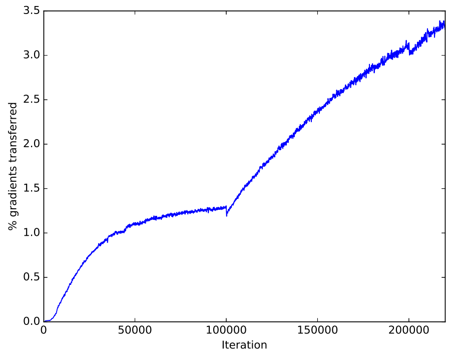

Figure 4: Communication fraction vs iteration for T = 5e-3.

3.3 Binarized gradient transfer

Table 2: Binarized gradient transfer results

Threshold	Top 1% accuracy	Total GBs transferred	Ratio (%/GB)
6.25e-4	51.6584	396.85	0.1302
8.75e-4	54.6196	316.63	0.1725
1e-3	55.0432	279.65	0.1968
5e-3	55.6646	31.22	1.7829
6.25e-3	55.9303	21.34	2.6215
8.75e-3	55.6945	12.15	4.5840

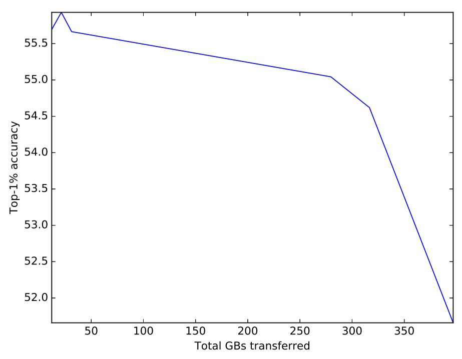

Figure 5: Accuracy vs data communicated.

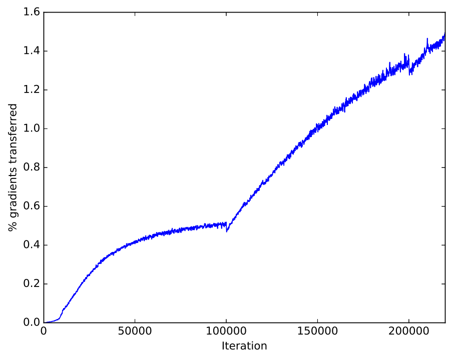

Figure 6: Communication fraction vs iteration for $T = 8.75\text{e-}3$.

3.4 Quantized gradient transfer

Table 3: Quantized gradient transfer results

Threshold	Top 1% accuracy	Total GBs transferred	Ratio (%/GB)
5e-6	57.1371	12055.74	0.0047
1e-5	57.8625	11632.94	0.0050
5e-5	58.3640	9879.56	0.0059
1e-4	58.1682	8525.72	0.0068
5e-4	57.9224	4204.91	0.0138
8.75e-3	56.2720	123.88	0.4542

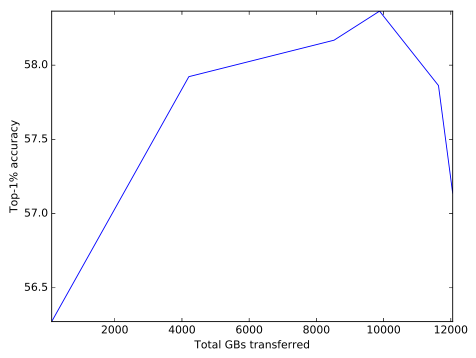

Figure 7: Accuracy vs data communicated.

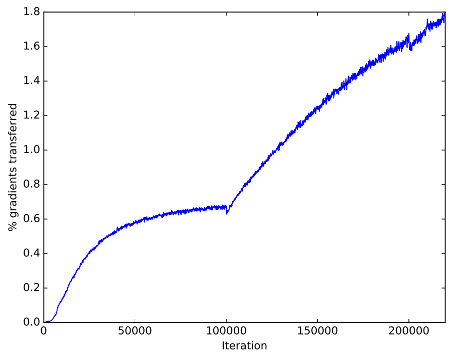

Figure 8: Communication fraction vs iteration for $T = 8.75\text{e-}3$.

4 Analysis

Naturally, we'd expect the resulting accuracies of thresholded experiments to be lower than that of the baseline. However, for some WGT and QGT trials, we notice the accuracies end up slightly higher than the baseline. It seems to indicate that these types of thresholding can be, to some extent, a form of regularization, as it removes small, noisy gradients and accumulates them for a later, "smoothed" update. Another explanation is fluctuations in the final accuracy due to random initializations and batches - that they might be only as good as the baseline, but appear so due to this occurrence.

Unlike with WGT group, the resulting BGT and QGT accuracies do not consistently decrease as the threshold increases, but rather rise and fall, as we expected. For BGT, increasing the threshold means more data is sent per gradient, but places a tighter restriction on what gradients can even participate – only those with magnitude greater than T. As $T \to 0$, more gradients are communicated, but are done so at a lower value, and at $T = 0$ all gradients are communicated as zero. As $T \to \infty$, fewer gradients are communicated, but are done so at a higher value, and at $T = \infty$ no gradients are communicated. QGT is similar as $T \to \infty$, but as its $T \to 0$, its accuracy should increase since it will decrease the remainder of floor-division – yet it becomes restricted by the artificial limit we placed for the result to fit within a single byte.

Comparing the methods, we see WGT can maintain baseline accuracy while transferring 12.3% of the total data, and can go 1.45% below the baseline while transferring 1.75% of the total data. BGT never quite reaches baseline accuracy, but can achieve an accuracy of 2.22% less than baseline while transferring a mere 0.023% of the total data. QGT can maintain baseline accuracy transferring 7.87% and 1.65% below baseline for 0.23% of the total data.

The current standard for interconnects among computers, Infiniband, has a maximum packet size of 4KB. Therefore, since packets are sent serially, any reductions in communications down to 4KB will result in reduced message transfer times, linear in the number of packets. The average data size sent per node per iteration in most of our trials ranges in the hundreds to thousands of kilobytes, and even in the case of least data communication (BGT with T =8.75e-3), it is 6.9KB; meanwhile the baseline sends just over 30MB per iteration. And since these numbers exclude the matrix formatting data that wraps them, the actual packet size in practice will be strictly greater (likely by a factor of 3 for sparse matrices), assuming no network compression. Additionally our reductions result in a linear reduction in the cost required to encode the data for network transfer, as well as a reduction in memory transfer time for GPU-to-CPU transfer, down until the PCI bandwidth.

5 Conclusions

Overall, it is feasible to reduce training time of a distributed convolutional neural network by reducing the interprocess communication between nodes. We've demonstrated three different methods of compressing the interprocess communication, some with a slight accuracy loss, and others with a slight accuracy bump over the baseline. We can reduce the $O(n)$ synchronization time by reducing the communication bandwidth via an $O(1)$ computation-time and memory thresholding solution, avoiding more complicated and/or expensive operations described by other communication-reducing techniques.

The most efficient method, when observing the accuracy-to-GB ratio, is the Binarized Gradient Transfer. But if lower accuracy is unsuitable for some application, Quantized Gradient Transfer and Whole Gradient Transfer provide alternatives, as they can both achieve baseline and above-baseline accuracies, but with varying efficiencies depending on the threshold chosen.

The results will likely show the same improvements with larger batch sizes, as exist in non-simulated environments, as the gradients of each image in the batch will follow similar distributions. Regardless, accuracy should be much higher when using a much larger batch size than possible on a single machine.

6 Further Work

We've shown that there exists threshold values for all the above experiments, though finding that value is a task in and of itself. By trying this same experiment on various network models and data inputs and performing statistical analysis, ideally we would find a consistent relationship between the best threshold value and the gradients in the network. Doing so would allow for automatic calculation of the optimal threshold for any situation. This is clearly the most natural progression of this experiment to explore next.

One extra observation to notice in the plots is that the percentage of gradients transferred increases over time, meaning the sizes of many of the gradients increase with the iterations. We plotted the percentiles of the gradient sizes (not shown here) to find that the gradients in almost every layer except the last do, in fact, increase over time. It would be interesting to investigate what this implies for the network as a whole, and whether there are ways to exploit this to improve CNN architectures.

Acknowledgments

We would like to thank Dr. Forrest Iandola and Professor Kurt Keutzer for their advice and hardware resources.

References

[1] George Dahl, Alan McAvinney, and Tia Newhall. Parallelizing neural network training for cluster systems. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks*, PDCN '08, pages 220–225, Anaheim, CA, USA, 2008. ACTA Press.

[2] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. Large scale distributed deep networks. In P. Bartlett, F.c.n. Pereira, C.j.c. Burges, L. Bottou, and K.q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1232–1240. 2012.

[3] J. Deng, W. Dong, R. Socher, L. J. Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255, June 2009.

[4] Forrest N. Iandola, Khalid Ashraf, Matthew W. Moskewicz, and Kurt Keutzer. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. *CoRR*, abs/1511.00175, 2015.

[5] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pages 675–678, New York, NY, USA, 2014. ACM.

[6] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014.

[7] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *CoRR*, abs/1312.4400, 2013.

[8] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and application to data-parallel distributed training of speech dnns. In *Interspeech 2014*, September 2014.

[9] Nikko Strom. Scalable distributed dnn training using commodity gpu cloud computing. In *INTERSPEECH*, pages 1488–1492. ISCA, 2015.

[10] Hang Su and Haoyu Chen. Experiments on parallel training of deep neural network using model averaging. *CoRR*, abs/1507.01239, 2015.

[11] S. Zhang, C. Zhang, Z. You, R. Zheng, and B. Xu. Asynchronous stochastic gradient descent for dnn training. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6660–6663, May 2013.

[12] Sixin Zhang, Anna Choromanska, and Yann LeCun. Deep learning with elastic averaging SGD. *CoRR*, abs/1412.6651, 2014.

[13] A. Øland and B. Raj. Reducing communication overhead in distributed learning by an order of magnitude (almost). In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2219–2223, April 2015.