Scalable Distributed DNN Training Using Commodity GPU Cloud Computing

Nikko Strom

Amazon.com

nikko@amazon.com

Abstract

We introduce a new method for scaling up distributed Stochastic Gradient Descent (SGD) training of Deep Neural Networks (DNN). The method solves the well-known communication bottleneck problem that arises for data-parallel SGD because compute nodes frequently need to synchronize a replica of the model. We solve it by purposefully controlling the rate of weight-update per individual weight, which is in contrast to the uniform update-rate customarily imposed by the size of a mini-batch. It is shown empirically that the method can reduce the amount of communication by three orders of magnitude while training a typical DNN for acoustic modelling. This reduction in communication bandwidth enables efficient scaling to more parallel GPU nodes than any other method that we are aware of, and it can be achieved with neither loss in convergence rate nor accuracy in the resulting DNN. Furthermore, the training can be performed on commodity cloud infrastructure and networking.

Index Terms: Speech recognition, deep neural networks, distributed stochastic gradient descent

1. Introduction

Stochastic Gradient Descent (SGD) is the work-horse of Deep Neural Network (DNN) training. Error Back-Propagation (BP) combined with SGD is the most popular training technique across most applications of DNN (cf. [1], [2]). Even as other techniques are used, such as RBM (e.g. [1]), or 2nd order methods (e.g. Hessian-Free optimization [3], [4]), SGD is typically still an important part of the recipe and consumes a large fraction of the training time. Consequently, much effort has been expended to improve its efficiency and speed. Convergence has been shown to improve by weight conditioning [5], as well as adaptive update formulae such as AdaGrad [6] and AdaDelta [7]. More fundamentally, it is common practice to gain significant speed-up from computing sub-gradients of mini-batches of samples rather than updating weights after each training sample.

Another source of great reduction in training time was the introduction of general computation on Graphics Processing Units (GPU). However, the massive local computational density afforded by GPUs magnifies the fundamental problem of data-parallel training on many compute nodes. Consider that any DNN architecture has a characteristic compute-per-trainable-weight ratio. For example, the computation required in the forward and backward pass of a fully connected layer is proportional to the number of weights. Further, the amount of communication bandwidth required in a data-parallel training scheme (e.g. [8]) is nominally proportional to the number of trainable weights. Therefore, as the speed of computation per node increases, so does the required bandwidth. This paper is addressing that fundamental limitation.

It is worth pointing out that the parallel training problem can also be cast in a model-parallel framework. For sparsely structured connected layers, such as in Convolutional Neural Networks (CNN) [2], it is often possible to relatively efficiently distribute the computation of each layer [8], and for fully-connected layers, each layer may be processed on a different node [9]. Model parallelism is particularly attractive for very large models that do not completely fit in the working memory of a single GPU [8] [10], and has been most successful for small, tightly connected clusters of GPU devices [10] [11] [12]. However, for densely connected networks, model parallelism is in practice often bounded by the number of layers in the DNN [9]. Since in this paper we are concerned with larger scale distributed training on commodity compute nodes, we focus on data-parallel methods. However, as has been shown [8] [13], data parallelism can also be combined with model parallelism.

There have been many notable attempts at scaling up distributed training across compute nodes [13] [8] [11] [4]. The results are mixed, with greater success for use cases with higher compute-per-weight ratio, such as Convolutional Neural Networks (CNN) where parameter tying reduces the number of free weights, as well as CPU-based systems which have a lower compute density per node. Arguably the most challenging cases are fully connected architectures trained on GPU hardware [13]. State-of-the-art speech recognition acoustic models have exactly that profile, and this is the task we use to demonstrate our method in section 3 after first giving the details of the general method.

2. Method

2.1. Data-parallel distributed SGD

In data-parallel distributed SGD, each compute node has a local replica of the DNN and computes sub-gradients based on different partitions of the training data. Sub-gradients are computed in parallel for different mini-batches of data at each node (e.g. [8]). The sub-gradients are the basis for updates to the weights of the DNN and the weight-updates must be synchronized across all compute nodes. Nominally, weight updates are the same memory size as the model (i.e., the trainable weights), which makes it challenging to synchronize weights after each mini-batch. This has led to exploration of methods that synchronize less frequently [5] and other methods for compressing the gradients [13]. At a high level, our method can be viewed as a generalization of those ideas.

2.2. Two observations

Our first observation, as was also pointed out in [9] (see also [14]), is that many techniques for speeding up SGD can be formulated as variants of delaying weight updates. Mini-batching is an obvious case where the updates of sub-gradients for individual samples are delayed until the end of the mini-batch. Momentum and Nesterov accelerated gradients [15] are other, less obvious, examples of delays. It is also common to introduce additional delay to mask the communication latency (e.g., double buffering [13] or asynchronous SGD [16]).

Our second, empirical, observation is that sub-gradients are very sparse. It is well-known that typically the weights of a fully connected DNN are sparsely distributed with most weights close to zero [17] [18], so it is not surprising that sub-gradients are also sparse. Furthermore, since sub-gradients are based on a small amount of the training data they are normally even sparser than the weight distribution. This leads to the core idea of the method; only a small fraction of the weights are required to be updated after each mini-batch. Elements of the gradient that are near zero can safely be delayed much longer than the typical mini-batch size.

2.3. Compaction and dead reckoning

We can significantly compact sub-gradients by considering only gradient elements whose absolute values exceed a threshold. To code the resulting sparse gradient we construct key-value maps where keys are the indices into the full gradient and values are the corresponding individual gradient element. These maps constitute the messages exchanged in the distributed SGD. Because sub-gradients are very sparse, this leads to a great bandwidth reduction compared to transmitting the full gradient matrices.

To reduce memory copy bandwidth between the host and the GPU device, the compaction of the gradient is most efficiently done on the device. The operation is not trivially parallelized on the GPU, but it belongs to a class of problems, *string compaction*, that has been studied extensively and well-known fast implementations exist (e.g. [19]).

It is worth noting that there is no explicit synchronization of weights. Instead we apply "dead reckoning", i.e., senders communicate deltas of individual weights to all other nodes and receivers apply the deltas to the local replica of the DNN. For all copies of the DNN to stay synchronized, all nodes must receive exactly the same deltas and apply them with the same logic. The deltas may be received in different orders as long as the update logic is commutative.

2.4. Gradient residual

It is not sufficient to simply forget all gradient element values below the threshold because different gradient elements have different dynamic range. Instead, on each compute node, after each mini-batch, gradient values are aggregated in what we call the *gradient residual*. This ensures that weights with small but biased sub-gradient values are eventually updated, albeit at a lower update frequency. Only elements of the residual that exceed the threshold τ are encoded and communicated to peers and then subtracted from the residual. Effectively, this logic generalizes the concept of delayed updates and implements a variable length delay where smaller gradient elements are delayed more (see pseudo code in Section 2.6).

2.5. Quantization and compression

In a naïve implementation, we would encode each gradient element as two numbers; an integer element index and the floating point gradient element. That goes a long way; however, we can do better by quantizing the gradient and packing both the quantized gradient and the index in a single 32-bit integer field. A key detail is that the quantization error is not discarded but added back to the gradient residual (cf. "error feedback" [13]). Empirically we have found that 1-bit quantization is sufficient and carries no significant degradation in neither accuracy nor convergence speed. This somewhat surprising finding is consistent with [13]. Thus, nodes simply communicate weight deltas of $+\tau$ or $-\tau$, and we are left with 31 bits of address space for the index, which is more than enough for all our use cases.

Peer-to-peer messages can be further compressed by entropy coding. The quantized update messages are sorted sets of integers. It is easy to see that gaps between the integers have lower entropy than the absolute values. We have found empirically that Golomb-Rice [20] coding the gaps reduces the average size per weight update to 10-11 bits, representing an additional 3x compression. However, entropy compression incurs a small transmission delay due to the computation required, and we are not using it in the experiments below.

2.6. Pseudo code

The following pseudo code describes a mini-batch cycle in a single compute node participating in distributed SGD. This is the special case of 1-bit quantization.

1. Receive and uncompress any weight update messages from other compute nodes and apply them to the local replica of the DNN
2. Load feature vectors and supervision targets for a mini-batch
3. Compute a sub-gradient $\mathbf{G}^{(s)}$ by Back-Propagation
4. Aggregate the sub-gradient in the gradient residual $\mathbf{G}^{(r)} = \mathbf{G}^{(r)} + \mathbf{G}^{(s)}$
5. Reset the message map $\mathbf{M}$
6. For each element $\mathbf{g_i}^{(r)}$ of $\mathbf{G}^{(r)}$:
7. If $\mathbf{g_i}^{(r)} > \boldsymbol{\tau}$ then
   push the pair $\{i, +\tau\}$ to the message $\mathbf{M}$
   Subtract $\boldsymbol{\tau}$ from residual: $\mathbf{g_i}^{(r)} = \mathbf{g_i}^{(r)} - \boldsymbol{\tau}$
   Else if $\mathbf{g_i}^{(r)} < -\boldsymbol{\tau}$ then
   push the pair $\{i, -\tau\}$ to the message $\mathbf{M}$
   Add $\boldsymbol{\tau}$ to the residual: $\mathbf{g_i}^{(r)} = \mathbf{g_i}^{(r)} + \boldsymbol{\tau}$
8. Compress $\mathbf{M}$ and send to all other compute nodes
9. Apply $\mathbf{M}$ to the local replica of the DNN

Asynchronous SGD (in the sense of "Hogwild" [16]) is simply achieved if messages are communicated asynchronously. Empirically we find that asynchronous operation is significantly faster for scales above 10 compute nodes. Furthermore, in some cases, such as for sequence-based BMMI training [21], the batch-size is variable, which makes synchronous training particularly inefficient. Thus, we have chosen to focus on asynchronous operation.

3. Experiments

3.1. Hardware and infrastructure

All experiments were carried out on commodity Amazon Web Services (AWS) cloud infrastructure. The compute nodes are Elastic Compute Cloud (EC2) G2 instances with an NVIDIA GRID K520 GPU with 1,536 CUDA cores and 4GB of device memory. Communication between the compute nodes is served by standard AWS networking.

External data, i.e., feature vectors and supervision targets, are stored on AWS Simple Storage Service (S3) and fetched on-demand by compute nodes.

3.2. The task and data

The general method can be applied to other areas, but here we demonstrate by the task of acoustic modelling for automatic speech recognition (ASR). Contemporary state-of-the-art ASR systems use context dependent hybrid DNN-HMM where the DNN models the posterior probabilities of phonetic states given an observation vector (e.g. [1] for details). We use such a system to evaluate the end-to-end performance in terms of Word Error Rate (WER).

For training data and supervision targets we use a 1,000 hour subset of transcribed in-house Amazon speech data. Feature vectors consist of 20 log Mel frequency filter bank coefficients (for more detail, see for example [1]). The features of the training set were force-aligned using an existing ASR system. The speech signal is segmented in frames with 25 ms analysis window and 10 ms step size, which yields a total of 368 million training feature vectors with supervision targets.

We use a separate 185,000 word test set. For the purpose of this study we are not concerned with the absolute WER, but primarily use the test set to verify that the distributed algorithm does not degrade speech recognition accuracy relative to a single-node baseline.

3.3. DNN architecture and training recipe

To initialize the DNN, pre-training is performed by supervised layer-by-layer back-propagation training using a small subset of the training data [22]. This is very fast and is done by single-threaded SGD training.

The DNN structure after pre-training is a stack of fully connected layers. The input layer is a window covering the center frame and eight frames to the left and right. Each hidden layer is an affine transform with trainable weights followed by a sigmoid non-linearity. The output layer is a trainable affine transform followed by a softmax layer with outputs for the phonetic classes. There are five hidden layers resulting in a total of 14.6 million trainable parameters in the network.

After pre-training the network undergoes 10 epochs of distributed SGD training. Here we only report results for cross entropy training, but the method has successfully been applied to other loss objectives such as segment based BMMI [21]. For learning-rate control, the rate is simply halved after each epoch. As has been observed in [9], convergence of SGD training can be sensitive to long combined effective mini-batch sizes, in particular early in the training. Therefore, after some initial experimentation, to avoid poor convergence, we use a smaller mini-batch size and fewer compute nodes in the first $1/6^{th}$ of the first epoch (see Table 1*)*. Without this, the training diverges in the first epoch.

This recipe can be refined in many ways, but we choose this simple form for the purpose of demonstrating the distributed SGD method.

Table 1. *Training recipe for all experiments. The first epoch is split with a short first period of 166 hours with a smaller effective batch size.*

Epoch	1		2	3	...	10
Hours of training data	166	834	1000	1000	...	1000
Learning rate	0.008	0.008	0.004	0.002	...	1.5e-5
Local mini-batch size	256	512	1024	1024	...	1024
Number of compute nodes	10	10	5-80	5-80	...	5-80

3.4. Results

3.4.1. Baseline

To produce a baseline result we trained a DNN using only single-threaded SGD with otherwise the same dimensions, hyper-parameters, and on the same hardware and infrastructure. The ASR accuracy achieved by this DNN serves as the baseline by which we measure relative WER reduction in all other results below. The total elapsed time for epoch 1–10 was 102 hours, equivalent to a rate of around 10,000 frames / second. Note that in the following, we always only report on the last nine epochs 2-10.

3.4.2. Scaling the number of compute nodes

To study the scaling properties of the method, the number of compute nodes was varied from 5 to 80 in epochs 2–10 (see Table 2). All other hyper-parameters were constant. The number of compute nodes in the first epoch was held constant at 10 (see Table 1). For reference, the elapsed training time of the first epoch was 84 minutes. As can be seen Table 2 and Figure 1, the method scales very well up to at least 80 GPU compute nodes. And perhaps surprisingly, the convergence per epoch is faster and the asymptotic accuracy is slightly better for the distributed training. We don't have a definitive explanation for this, but more importantly there is no evidence that the method is detrimental for either convergence speed or accuracy.

Table 2. *Training speed for varying number of compute nodes in epoch 2-10. The first row shows the baseline result. The relative WER reduction is measured relative to the baseline on the test set. The small variations in WER are not significant.*

Nodes	Frame rate/sec	Elapsed [min.]	Relative speed	Relative WER reduction
1	10,000	5,470	1	0%
5	43,000	1,287	4.3	1.6%
10	88,000	626	8.7	1.5%
20	171,000	323	17	1.6%
40	331,000	167	33	1.6%
80	547,000	101	54	1.8%

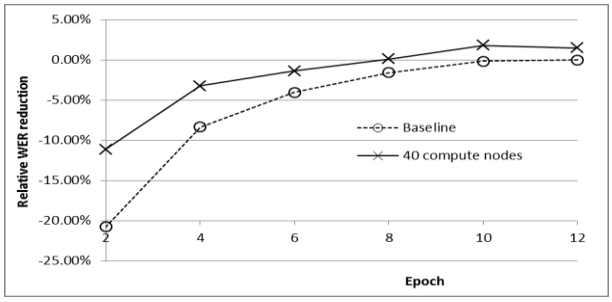

Figure 1. *Convergence of the baseline and the DNN trained with 40 parallel hosts. Note that for these two conditions, epoch 11 and 12 were also trained.*

3.4.3. Effect of varying the threshold τ

The results in Table 2 were achieved with $\tau = 4$. To study the effect of the threshold τ we reran epochs 2-10 with 40 compute nodes for different values of τ. When the threshold is too high convergence is affected and thus the accuracy (WER) is degraded. On the other hand, if the threshold is too low, the weight update messages become too large and slow down training due to a communication bottleneck. However, as can be seen in Table 3, there exists a wide interval of τ that yields both fast convergence and produces small enough update messages to ensure optimal training speed.

Table 3. *Performance for varying thresholds τ. Compression ratio is the size of the full gradient divided by the size of our sparse weight updates.*

τ	Frames/second (epoch 2-10)	Update size/mini-batch [KB]	Compression ratio	Elapsed [minutes]	Relative WER reduction
2.0	303,000	210	278	182	1.8%
4.0	331,000	69	846	180	1.6%
6.0	334,000	33	1,770	179	1.1%
9.0	335,000	15	3,893	179	0.2%
12.0	335,000	7.8	7,487	165	-0.5%
15.0	341,000	4.5	12,978	162	-1.9%

3.4.4. Scaling the model size

Scaling up the model size increases both computation and the size of weight updates, but larger models tend to produce even sparser weight update distributions resulting in a lower ratio of communication per computation. To demonstrate, we trained DNNs with double, and triple the hidden layer size. We also trained a DNN with eight hidden layers. Results are shown in Table 4. As expected the WER is significantly better than the baseline, but more importantly the weight update size does not grow with the size of hidden layers and it grows slower than the number of hidden layers. Thus the method is becoming more efficient as the model size is increased.

Table 4. *Performance for different model sizes.*

Layer dimensions	Weights [million]	Frames per sec.	Compress ratio	Elapsed [minutes]	Relative WER reduction
5 hidden layers	14.6	331,000	846	180	1.6%
8 hidden layers	21.7	186,000	998	247	4.2%
5 hidden layers of 2x size	23.7	213,000	1,436	259	2.8%
5 hidden layers of 3x size	48.8	109,000	2,871	508	4.1%

3.5. Comparison with previous results

We are not aware of previous results for fully-connected DNN architectures competitive with the results in Table 2. An early CPU-based example is [8], where for a 42 million weight ASR DNN, the speed-up was a modest 2.2x and bottomed out for 8 compute nodes with up to 20 CPU cores each. More recently, [5] used a different approach to scale up to at least four GPU nodes, but scaling beyond 16 nodes is not yet demonstrated.

There are several published results for configurations that limit the degree of parallelism to the number of GPU devices that can be installed in a single server. Low single digit speed-ups are reported in [11], [9] and [12].

The closest comparable results in terms of scaling are for clusters with Infiniband networking. For example, [10] demonstrates impressive speed-ups, using up to 64 GPU nodes for model-parallel CNN training. However, for fully-connected DNNs, the closest result is [13] where a 10-fold speedup was demonstrated by a combination of model- and data-parallelism across 20 GPU nodes.

4. Discussion

For contemporary state-of-the-art model sizes and data sets, our method practically solves the distributed DNN training problem; very large models can now be trained in days or hours rather than months and the method becomes more efficient as model sizes are increased, which allows it to scale to larger models in the future. On the other hand, the success in scaling the number of compute nodes is not completely limitless.

First, as the number of compute nodes grow, so does the effective batch size, i.e., the number of samples that are aggregated in weight updates. Early in the training of a DNN, longer batch sizes tend to degrade the convergence rate. As mitigation, hyper-parameters may be tuned, or automatic mini-batch size selection [13] can be utilized, but ultimately this is a fundamental trade-off of SGD.

The second limiting factor is the asymptotic peer-to-peer gradient communication. Above a certain task- and system-dependent number of compute nodes, training will again become communication-bound. Potential mitigations might include hierarchical aggregation of messages [8]. However, since messages consist of sparse quantized sets, aggregation does not necessarily substantially reduce the total size.

In these experiments we have chosen the most challenging configuration with respect to the two limitations. We also used moderately sized DNNs and we know that the method scales better for larger models. Therefore, while we are aware of theoretical limitations, in practice the method may scale to considerably larger numbers of compute nodes than evidenced in this report.

5. Conclusions

We have introduced a novel method for data-parallel distributed SGD training of DNNs. It was shown empirically that the method scales well to at least 80 GPU instances on commodity cloud infrastructure with no loss of accuracy or rate of convergence. This was made possible by reducing the required communication bandwidth by three orders of magnitude. In addition, it was shown that the method becomes increasingly efficient as the model size increases. To our knowledge, the scale and training speed achieved is the highest ever reported for fully connected DNN training.

6. References

[1] G. Hinton, L. Deng, D. Yu, G. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath and B. Kingsbury, "Deep neural networks for acoustic modeling in speech recognition," *IEEE Signal Processing Magazine*, 2012.

[2] Y. LeCun, F.-J. Huang and L. Bottou, "Learning methods for generic object recognition with invariance to pose and lighting," in *Proc. of the Computer Vision and Pattern Recognition Conference*, Los Alamitos, CA, 2004.

[3] J. Martens, "Deep learning via Hessian-free optimization," in *Proc. of ICML*, 2010.

[4] T. N. Sainath, I.-h. Chung, B. Ramabhadran, M. Picheny, J. Gunnels, B. Kingsbury, G. Saon, V. Austel and U. Chaudhari, "Parallel deep neural network training for LVCSR tasks using Blue Gene/Q," in *Proc. of INTERSPEECH*, Singapore, 2014.

[5] D. Povey, X. Zhang and S. Khudanpur, "Parallel training of DNNs with natural gradient and parameter averaging," in *Proc. of ICLR 2015*, San Diego, 2015.

[6] J. Duchi, E. Hazan and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, p. 2121–2159, 2011.

[7] M. D. Zeiler, "ADADELTA: An adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.

[8] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. A. Ranzato, A. Senior, P. Tucker, K. Yang and A. Y. Ng., "Large scale distributed deep networks," in *Proc. of Advances in Neural Information Processing Systems*, Lake Tahoe, NV, 2012.

[9] F. Seide, F. Hao, J. Droppo, G. Li and D. Yu, "On parallelizability of stochastic gradient descent for speech DNNs," in *Proc. of ICASSP*, Florence, Italy, 2014.

[10] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro and N. A, "Deep learning with COTS HPC systems," in *Proc. of the 30th International Conference on Machine Learning*, 2013.

[11] Z. Shanshan, C. Zhang, Z. You, Z. Rong and B. Xu, "Asynchronous stochastic gradient descent for DNN training," in *Proc. of ICASSP*, Vancouver, Canada, 2013.

[12] C. Xie, A. Eversole, G. Li, D. Yu and F. Seide, "Pipelined Back-Propagation for Context-Dependent Deep Neural Networks," in *Proc. of INTERSPEECH*, Portland, OR, 2012.

[13] F. Seide, H. Fu, J. Droppo, G. Li and D. Yu, "1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs," in *Proc. of INTERSPEECH*, Singapore, 2014.

[14] A. Agarwal and J. J. Duchi, "Distributed delayed stochastic optimization," in *Proc. of Advances in Neural Information Processing Systems*, Granada, Spain, 2011.

[15] Y. Nesterov, "Gradient methods for minimizing composite objective function," in *CORE Discussion Paper 2007/76, Catholic University of Louvain*, 2007.

[16] B. Recht, C. Re, S. Wright and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Proc. of Advances in Neural Information Processing Systems*, Granada, Spain, 2011.

[17] N. Ström, "A tonotopic artificial neural network architechture for phoneme probability estimation," in *Proc. of IEEE Workshop on Speech Recognition and Understanding*, Santa Barbara, CA, 1997.

[18] N. Ström, "Sparse connection and pruning in large dynamic artificial neural networks," in *Proc. of EUROSPEECH*, Rhodes, Greece, 1997.

[19] N. Wilt, The cuda handbook: A comprehensive guide to gpu programming, Pearson Education, 2013.

[20] R. F. Rice and R. Plaunt, "Adaptive Variable-Length Coding for Efficient Compression of Spacecraft Television Data," *IEEE Transactions on Communication Technology*, vol. 16, no. 9, 1971.

[21] K. Veselý, A. Ghoshal, L. Burget and D. Povey, "Sequence-discriminative training of deep neural networks," in *Proc. of INTERSPEECH*, Lyon, France, 2013.

[22] F. Seide, G. Li, X. Chen and D. Yu, "Feature engineering in context-dependent deep neural networks for conversational speech transcription," in *Proc. of IEEE ASRU*, Waikoloa, HI, 2011.