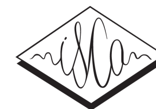

Distributed Asynchronous Optimization of Convolutional Neural Networks

William Chan[1], *Ian Lane*[1,2]

Carnegie Mellon University
[1]Electrical and Computer Engineering, [2] Language Technologies Institute
williamchan@cmu.edu, lane@cmu.edu

Abstract

Recently, deep Convolutional Neural Networks have been shown to outperform Deep Neural Networks for acoustic modelling, producing state-of-the-art accuracy in speech recognition tasks. Convolutional models provide increased model robustness through the usage of pooling invariance and weight sharing across spectrum and time. However, training convolutional models is a very computationally expensive optimization procedure, especially when combined with large training corpora. In this paper, we present a novel algorithm for scalable training of deep Convolutional Neural Networks across multiple GPUs. Our distributed asynchronous stochastic gradient descent algorithm incorporates sparse gradients, momentum and gradient decay to accelerate the training of these networks. Our approach is stable, neither requiring warm-starting or excessively large minibatches. Our proposed approach enables convolutional models to be efficiently trained across multiple GPUs, enabling a model to be scaled asynchronously across 5 GPU workers with $\approx 68\%$ efficiency.

Index Terms: deep neural network, distributed optimization

1. Introduction

Deep Neural Networks (DNNs) have been vital for state-of-the-art models in computer vision [1] and speech recognition [2]. DNNs are typically trained with backpropagation and stochastic gradient descent [3], which is sequential in nature. Training these DNN is a computationally expensive task, and training times only increase as we move to deeper, wider models and larger corpora which have been shown to increase model performance [4]. Fortunately, much of DNN computation can be easily decomposed into a series of general matrix-matrix multiplications (GEMM). This has naturally led to the usage of GPUs for most researchers.

Convolutional Neural Networks (CNNs) demonstrate even more promise in both computer vision [1] and speech recognition [5]. CNNs introduce weight sharing across spectrum and time, providing translational robustness to small shifts in the model. CNNs also typically incorporate pooling which gives additional translational and rotational invariance. However, typical implementations of CNNs only add to the training time costs, convolutional kernels can not match the GFLOP efficiency of GEMM kernels seen in fully connected networks as seen from [6].

Training these DNN and CNN models can be prohibitively computationally expensive, requiring large clusters of CPUs in some cases [7]. The usage of single GPUs in training have helped immensely in the field of deep learning research, however, even with a single GPU, training times with large datasets can take days to weeks [8]. Such large datasets are most likely intractable to run on a single machine or single GPU, and the desire for wider, deeper and convolution models with ever increasing corpus sizes only magnifies the need for distributed learning. Recently, there has been much interest among researchers and practitioners in distributed gradient based methods of which are suitable for deep networks [9, 7].

Our proposed method is an asynchronous stochastic gradient descent algorithm distributed across multiple GPUs. At the high level, our method distribute gradient computation to independent workers, each operating on a subset of the corpus and learning an independent gradient. The independent workers learns and relays back the gradient information to the master, which accumulates all the learnt gradients from the different workers. The master accepts the gradients with a penalty factor, the penalty factor is computed based on the time distance between the master state and the gradient information. The worker also updates its local weights from the global master copy to ensure the validity of the gradients in the overall optimization problem. Our implementation is based on efficient CUDA kernels and low latency communication between GPUs on the PCIE bus, communicating weights and gradients between master and workers.

2. Asynchronous Optimization

2.1. Stochastic Gradient Descent

Stochastic Gradient Descent is perhaps the most commonly used optimization procedure for deep learning. DistBelief [7] implemented Sandblaster L-BFGS and Downpour SGD, the later of which is a form of distributed asynchronous stochastic descent for deep learning. DistBelief's asynchronous implementation has been applied successfully in many fields including computer vision [9] and acoustic models [4, 10].

Our work differs from the previous work in several different ways. First, our platform is based on distributed GPUs rather than distributed CPUs. DistBelief relies on a large proprietary cluster of CPUs (e.g., 16 000 CPU cores [7]) which is not readily accessible to most researchers. However, 4 GPUs connected together via the PCIE bus on the same server machine are much more easily available and affordable. GPUs are an order of magnitude faster than CPUs in computing minibatch gradients [11], consequently the learning algorithm updates much more often and much more sensitive to communication latencies. In a distributed CPU architecture, the network latency (while large) may be easily hidden due to the relatively longer times to compute a single gradient.

With few exceptions, training DNNs is a highly non-convex problem, unlike convex optimizations where we have the luxury of convergence guarantees [12, 13] even in high latency setting. For example, suppose the master parameter network is at

Copyright © 2014 ISCA

time i and weight state Θ_i, we have a worker copy the weights $\theta_i = \Theta_i$ and compute a gradient $\nabla f(\theta_i)$, however by the time the worker finish computing the gradient, the master may have moved on to Θ_j where $j = i + n$ and n may be large. The update $\Theta_{j+1} = \Theta_j - \eta \nabla f(\theta_i)$ may no longer improve the loss, and potentially harmful to the overall optimization problem! This "stale gradient" problem presents stability issues. [7] tackled this problem via two techniques, (1) warmstarting the network with sequential SGD and (2) an adaptive learning rate AdaGrad [14].

Another approach to this stale gradient problem is to use larger minibatches [15] which produces more stable gradients. However such an approach is undesirable as smaller minibatches tend to converge faster in wall clock time. Convergence theory for non-strongly convex functions gives gradient descent a convergence rate of $O(t^{-1})$, compared to stochastic gradient's descent of $O(t^{-2})$, it is important to remember in gradient descent we must compute the gradient for the entire dataset which is $O(n)$ while SGD only requires a single datapoint at $O(1)$. Minibatch gradient descent sits somewhere in between, it is important to remember the main motivation for using minibatch gradient descent because of the availability of highly efficient GPU computing architecture, the cost of computing a minibatch of 256 is only marginally more expensive than computing a minibatch of say 16.

3. Architecture and Algorithm

3.1. Max Pooling

Max pooling emits the maximum neuron from a region of neurons in the same kernel map. Pooling provides us to additional robustness in translation and rotational invariance complementing our convolution layers [16]. Max pooling have been shown to outperform average pooling and perform very closely to stochastic pooling [5]. We utilize max pooling in our network architecture. Since max pooling only propagates the neurons with the maximum value within a region (typically only one neuron), the backpropagation algorithm will only backpropagate through this neuron, with the $\frac{dE}{dx}$ zero for the other neurons, and thus resulting in sparse gradients.

3.2. ReLU

Rectified Linear Units (ReLU) neurons [17] use the activation function $\max(0, x)$. ReLU have been shown to train much faster compared to sigmoid nonlinearities (such as sigmoid or $\tanh$) in computer vision [1] and acoustic modelling [18, 19]. The hard non-linearity will result in many sparse activations in the network [20]. The sparse exact zero activations will also block any backpropagation gradient paths backwards, resulting in exact zero gradients along those paths.

3.3. Dropout

Dropout [21] is a simple yet effective means of regularization especially in large deep networks. Dropout randomly selects a subset of neurons to be turned off (set to zero) in each minibatch gradient computation; this prevents complex co-adaptations in the feature detectors encouraging robustness. This technique has yield state of the art performance in computer vision [1] and speech [19], however at the cost of additional training time.

Our distributed algorithm incorporates dropout and works with it to produce robust models. Similar to ReLU neurons, dropout will leave many neurons to be exact zero activations, the backpropagation gradient paths will be blocked and produce exact zero gradients along those paths. In our experiments, we use a 50% dropout rate for the fully connected neurons, the convolution layers have no dropout.

3.4. Sparse Gradients

Our choice of max pooling, ReLU and dropout has naturally led to a model emitting sparse gradients. The cosine angle between sparse gradients $\cos(\theta) = \frac{\langle g_i, g_j \rangle}{\|g_i\| \|g_j\|}$ are more likely to be small and thus less likely to counteract each other (e.g., the independent workers are given a chance to learn unique information in the overall optimization problem). In our experiments, we found each gradient minibatch (averaged over 256 samples) to have an average sparsity around 30%.

3.5. Master Momentum

Momentum [22] is an inexpensive technique often applied in deep learning to accelerate and stabilize the optimization procedure. Given an objective function $f(\theta)$, classical momentum is given by:

$$v_{t+1} = \mu v_t - \eta \nabla f(\theta) \tag{1}$$

$$\theta_{t+1} = \theta_t + v_{t+1} \tag{2}$$

where $\nabla f(\theta)$ is the gradient, $\eta > 0$ is the learning rate and $\mu \in [0, 1]$ is the momentum coefficient.

We follow a technique similar to [10], on the master server side, we compute the gradient direction v on the master GPU with Θ as the master parameters while ∇_i is the gradient returned by the i-th worker, our update rule becomes:

$$v_{t+1} = \mu v_t - \eta \nabla_i \tag{3}$$

$$\Theta_{t+1} = \Theta_t + v_{t+1} \tag{4}$$

We find momentum accelerates and more importantly stabilizes the learning procedure in the distributed optimization.

3.6. Gradient Decay

Stale gradients become an issue when the worker returned gradient ∇_j, was computed from θ_i, and $n = j - i$ is large. These stale gradients can cause stability issues and oscillations in the learning procedure.

We introduce a simple and yet effective technique to fix this stale gradient problem. We simply penalize the stale gradients, using only a fraction of the information returned. The gradient is decayed based on the time distance between the current master state and the weights the worker used to compute its gradient. We found using a exponential decay function to work quite effectively, we define the gradient decay factor α:

$$\alpha = \beta^{(j-i)} \tag{5}$$

where $j - i \geq 0$ and β is our exponential decay parameter. Algorithm 1 describes the entire master process in our optimization procedure.

Algorithm 1 Master Momentum and Gradient Decay

```
function MASTERPROCESS()
    Θ ← INITRAND()
    i ← 0
    v ← 0

    for worker ∈ workers do
        PUSHWEIGHTANDINDEX(worker, [Θ, i])
    end for

    for parallel worker ∈ workers do
        while not CONVERGED(w) do
            [∇, j] ← GETGRADIENTANDINDEX(worker)

            α ← GRADIENTDECAY(i, j)
            μ ← MOMENTUM(i)
            η ← LEARNINGRATE(i)
            v ← μv − αη∇
            Θ ← Θ + v
            i ← i + 1

            PUSHWEIGHTANDINDEX(worker, [Θ, i])
        end while
    end for
end function
```

4. Implementation

4.1. Master and Workers

Our platform consists of NVIDIA K20s connected together via the PCIE bus. We dedicate one of the GPUs to be the "master", responsible for holding the latest copy of the model weights (in GPU memory) and accumulation of gradients sent by the workers through the PCIE bus. The master GPU is also responsible for the momentum and gradient decay computation. We found using a GPU master to scale much better than dedicating CPU(s) and RAM towards this task. We wanted to minimize the latencies between gradient updates. The CUDA architecture allow us to use DMA PCIE transfers bypassing the CPU completely in the communications between the master and workers with higher bandwidth and lower latencies. In large models, we found the CPU unable to keep up with the GPUs in just gradient accumulation, and found this to limit the number of rate of GPU synchronizations between the workers and master, which hurt overall model convergence performance, and thus negating some of the benefits of distributed optimization.

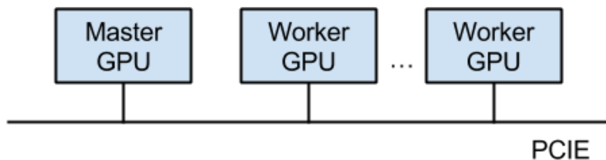

Figure 1: GPU Master and GPU workers: we bypass the CPU completely allowing DMA transfers of weights and gradients between workers and master.

Each GPU is an independent worker computing gradients running a highly efficient implementation of CNNs using CUDA, data is stream asynchronously from disk to CPU RAM to GPU RAM and our convolution kernels are based on [6]. The worker begins by updating to the latest model weights from the master GPU $\theta_i = \Theta_i$.

The worker will then independently sample from the dataset and compute SGD for n minibatches moving to θ_{i+n}, to be explicit, the worker updates its own weights after each minibatch (and thus can diverge quite significantly from the master Θ). Once a minimum of n minibatches have been completed, the worker does not immediately return the gradient to the master, but rather it will keep computing new gradients until the master GPU is free of computation and ready to accept new gradients. The worker will then compute $\nabla = \theta_j - \theta_i$, and return this ∇ gradient to the master GPU. The worker GPU then obtain the latest model weights Θ_k before proceeding with new gradient computation.

5. Experiments

Our model is a CNN architecture incorporating convolution filters, max pooling, fully connected layers and finally a softmax output layer. Figure 2 gives a description of our overall architecture. The weights were randomly initialized with Gaussians and biases set to a small positive value to accelerate the learning. All hidden layers use the ReLU activation, and the fully connected layers use 50% dropout. We train with cross entropy loss. We found using an exponential gradient decay of 0.9 for the distributed learning to work well with this model. Our objective is to demonstrate the scaling ability of our algorithm, thus we use constant learning rate of $\eta = 0.001$ rather than an adaptive learning rate scheduler. We also utilize a constant minibatch size of 256 and momentum (classical or distributed master) of $\mu = 0.9$. We do not apply warmstarting to any of our experiments.

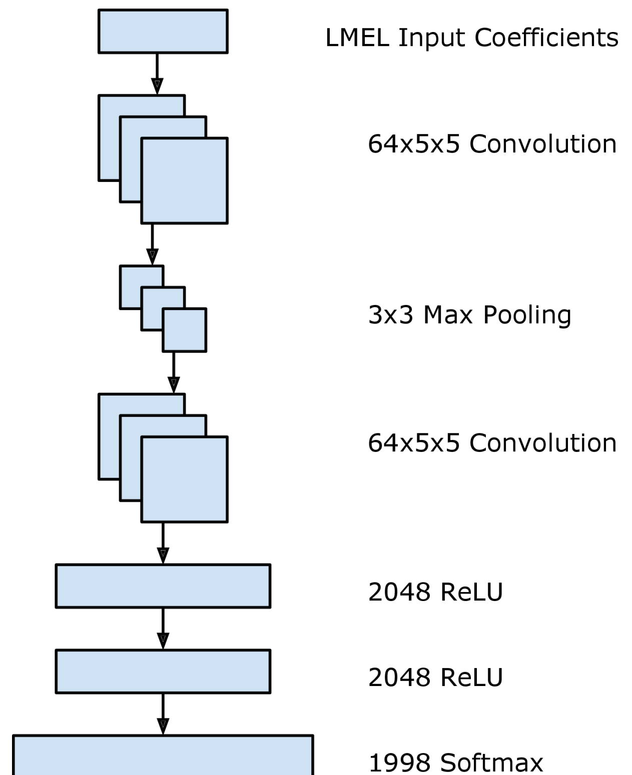

Figure 2: Our convolutional neural network accepts log-mel (LMEL) feature coefficients modelled with convolutional filters, max pooling, ReLU neurons and softmax outputs.

Table 1: Final convergence of Frame Accuracy (FA) and Cross Entropy Error (CSE).

Model	Workers	FA	CSE
Fully Connected ReLU	1	39.5%	2.85
Convolution	1	45.9%	2.53
Convolution	2	45.9%	2.52
Convolution	3	45.7%	2.55
Convolution	4	45.8%	2.53
Convolution	5	45.2%	2.58

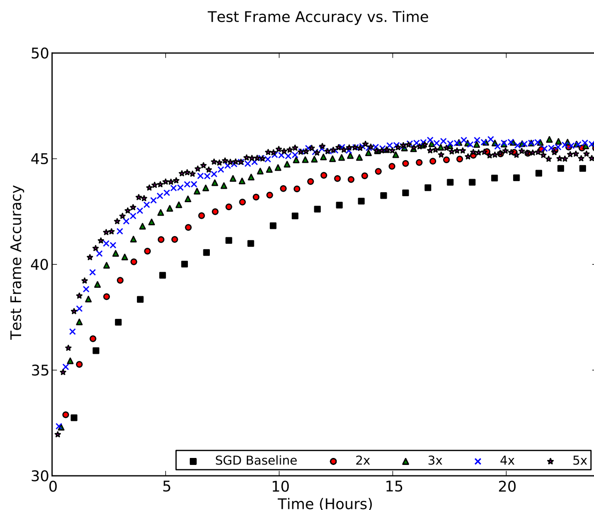

Figure 3: Distributed asynchronous stochastic gradient descent scaling comparison by varying the numbers of workers; baseline is standard SGD.

We experiment with the BABEL Bengali (IARPA-babel103b-v0.4b) corpus which has been collected and released under the IARPA BABEL research program. The optimization problem is to classify speech frames, our input features are 23-dimension log-mel (LMEL) feature coefficients padded with 11 left and right frame contexts, and our targets are 1998 softmax outputs.

We take a snapshot of the network after each epoch and compute the held out test frame accuracy. Table 1 gives the converged frame accuracy and cross entropy loss. With the exception of the case of 5 distributed workers, the held out loss was within $\pm 1.0\%$ relative of the baseline single GPU. For reference, we included a five layer fully connected ReLU model.

Figure 3 gives a plot of the observable speedup in frame accuracy against wallclock time. Our model does not require any warmstarting. Even with random initializations and 5 distributed workers, there is no observable oscillations seen during the early stages of the optimization procedure.

Table 2 gives the exact training time and scaling efficiency of our algorithm at the 40%, 43% and 44% frame accuracy mark. At the 44% frame accuracy point, we realize a speedup of $\times 3.4$ using $\times 5$ GPUs, saving us over 13 hours of training time. We observe an overall scaling efficiency between $\approx 68\%$ to $\approx 80\%$ depending on the number of workers and convergence point. As expected, we find distributed optimization algorithm to be more efficient with fewer workers due to fewer stale gradients and fewer gradient decay impact. Additionally,

Table 2: Time (hh:mm) and scaling efficiency (in brackets) comparison for convergence to 40%, 43% and 44% Frame Accuracy (FA).

Workers	40% FA	43% FA	44% FA
1	5:50 (100%)	14:36 (100%)	19:29 (100%)
2	3:36 (81.0%)	8:59 (81.3%)	11:58 (81.4%)
3	2:48 (69.4%)	5:59 (81.3%)	7:58 (81.5%)
4	2:05 (70.0%)	4:28 (81.7%)	6:32 (74.6%)
5	1:40 (70.0%)	3:49 (76.5%)	5:43 (68.2%)

as we scale to more workers, the master GPU becomes more occupied which increases the delay in which a worker can synchronize back with the master and thus more likely to produce stale gradients as well. The scaling efficiency also decreases as we get closer to the minima, this is especially true for the 5 distributed workers compared to 2 distributed workers. As we approach the minima, the model is much more sensitive to stale gradients and consequently our scaling efficiency suffers.

6. Conclusion

In this paper, we have presented a distributed asynchronous algorithm to train deep convolutional neural networks. Our algorithm incorporates sparse gradients, master momentum and gradient decay. Our approach is stable, neither requiring warmstarting or excessively large minibatches. Our model combining convolutional filters, max pooling and ReLU neurons demonstrated $\approx$ 68% scaling efficiency across 5 GPU workers in an asynchronous fashion. Our work will permit researchers and practitioners to train and study deeper and wider convolution models combined with large corpora.

7. Acknowledgements

This work was supported by the Intelligence Advanced Research Projects Activity (IARPA) via Department of Defense U.S. Army Research Laboratory (DoD / ARL) contract number W911NF-12-C-0015. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, DoD/ARL, or the U.S. Government. We thank Jungsuk Kim and Won Kyum Lee for comments and discussions.

8. References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Neural Information Processing Systems*, 2012.

[2] G. Hinton, L. Deng, D. Yu, G. Dahl, A. rahman Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury, "Deep Neural Networks for Acoustic Modeling in Speech Recognition," November 2012.

[3] Y. LeCun, L. Bottou, G. B. Orr, and K. R. Mller, "Efficient BackProp," pp. 5–50, 1998.

[4] G. Heigold, V. Vanhoucke, A. Senior, P. Nguyen, M. Ranzato, M. Devin, and J. Dean, "Multilingual acoustic models using distributed deep neural networks," in *IEEE Inter-*

national Conference on Acoustic Speech and Signal Processing, 2013.

[5] T. Sainath, A. rahman Mohamed, B. Kingsbury, and B. Ramabhadran, "Deep Convolutional Neural Networks for LVCSR," in *IEEE International Conference on Acoustic Speech and Signal Processing*, 2013.

[6] A. Krizhevsky, "cuda-convnet: High-performance C++/CUDA implementation of convolutional neural networks," 2012.

[7] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large Scale Distributed Deep Networks," in *Neural Information Processing Systems*, 2012.

[8] N. Jaitly, P. Nguyen, A. W. Senior, and V. Vanhoucke, "Application of Pretrained Deep Neural Networks to Large Vocabulary Speech Recognition," in *Interspeech*, 2012.

[9] Q. V. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G. S. Corrado, J. Dean, and A. Y. Ng, "Building high-level features using large scale unsupervised learning," in *International Conference on Machine Learning*, 2012.

[10] M. Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q. Le, P. Nguyen, V. V. A. Senior, J. Dean, and G. Hinton, "On Rectified Linear Units for Speech Processing," in *IEEE International Conference on Acoustic Speech and Signal Processing*, 2013.

[11] V. Vanhoucke and A. Senior, "Improving the speed of neural networks on CPUs," in *Neural Information Processing Systems: Deep Learning and Unsupervised Feature Learning Workshop*, 2011.

[12] J. Langford, A. J. Smola, and M. Zinkevich, "Slow Learners are Fast," in *Neural Information Processing Systems*, 2009.

[13] M. A. Zinkevich, M. Weimer, A. Smola, and L. Li, "Parallelized Stochastic Gradient Descent," in *Neural Information Processing Systems*, 2010.

[14] J. Duchi, E. Hazan, and Y. Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," *Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, July 2011.

[15] S. Zhang, C. Zhang, Z. You, R. Zheng, and B. Xu, "Asynchronous Stochastic Gradient Descent for DNN Training," in *IEEE International Conference on Acoustic Speech and Signal Processing*, 2013.

[16] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun, "What is the Best Multi-Stage Architecture for Object Recognition," in *International Conference on Computer Vision*, 2009.

[17] V. Nair and G. E. Hinton, "Rectified Linear Units Improve Restricted Boltzmann Machines," in *International Conference on Machine Learning*, 2010.

[18] A. Maas, A. Hannun, and A. Ng, "Rectifier Nonlinearities Improve Neural Network Acoustic Models," in *International Conference on Machine Learning: Workshop on Deep Learning for Audio, Speech, and Language Processing*, 2013.

[19] G. E. Dahl, T. N. Sainath, and G. E. Hinton, "Improving Deep Neural Networks for LVCSR Using Rectified Linear Units and Dropout," in *IEEE International Conference on Acoustic Speech and Signal Processing*, 2013.

[20] X. Glorot, A. Bordes, and Y. Bengio, "Deep Sparse Rectifier Neural Networks," in *International Conference on Artificial Intelligence and Statistics*, 2011.

[21] G. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," in *arXiv*, 2012.

[22] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the Importance of Initialization and Momentum in Deep Learning," in *International Conference on Machine Learning*, 2013.