Multi-Layer Perceptron on the Blue Gene/Q

Peter Horak, Dylan Elliott, Andrew Showers

Rensselaer Polytechnic Institute

May 2, 2017

Abstract

In this paper we develop a parallel program to train a simple neural network model and observe the performance of the training program on a high-performance computing system. The algorithms for our parallel training program are derived through graph computation and initially validated to achieve good classification accuracy on the MNIST dataset. We perform a strong scaling study on several different neural network architectures on a Blue Gene/Q. We observe that the large amount of synchronization and communication that is required within our parallel program introduces certain bottlenecks that limit the overall performance. The sources of these bottlenecks are analyzed and we discuss results that reveal insight into methods that could decrease the time needed for our parallel program to train the network.

1 Introduction

A large amount of research has focused on using high performance computing (HPC) to decrease the training time of artificial neural networks (ANN) [3]. In this paper, we investigate the challenges faced when training a fully-connected ANN on a Blue Gene/Q (BG/Q) supercomputer, which has a total of 5120 16-core compute nodes. Specifically, we formulate a multi-layer perceptron (MLP) and train it to perform classification on the MNIST dataset of handwritten digits. The MLP training algorithm is constructed as a parallel program using the MPI library to assign subsets of neurons in the MLP to MPI ranks. For the bulk of this study, we analyze the performance of our parallel MLP training program for several different MLP architectures and at varying levels of parallelization in order to investigate the strong scaling of our program on an HPC system.

We begin the paper by giving a brief overview of the MLP network and the learning algorithm used to train the MLP. We then derive the local learning algorithms from the perspective of each neuron in the MLP and translate this computation to a parallel program using the MPI library. The MLP model is then tested on the MNIST dataset using our parallel program run on the RPI Kratos cluster. Having demonstrated the correctness of the learning algorithm in the MLP parallel program, we perform several scaling experiments with varying MLP architectures on the BG/Q. Finally, we conclude with an analysis of our results and a discussion of future work.

2 Background

2.1 Multi-Layer Perceptron

The multi-layer perceptron (MLP) is an artificial neural network consisting of fully connected layers of artificial neurons. Each neuron receives the outputs from all neurons in a previous layer, computes its own output value from these outputs and sends its output value to all neurons in the next layer. The computation each neuron performs is usually a non-linear transformation applied to a linear combination of unique weights with the outputs from all of the neurons in a previous layer. The MLP therefore transforms data from its original representation to a new representation in which similar classes of input data become linearly separable. An MLP is trained to produce desired outputs through supervised training, in which correct pairs of inputs and outputs are presented iteratively to the network until the

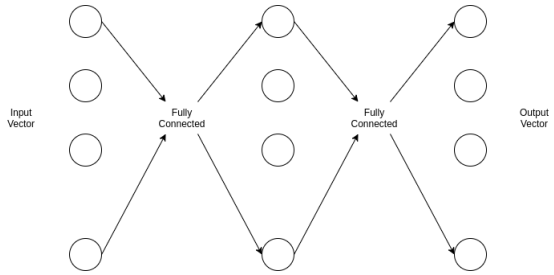

Figure 1: *Basic structure of an MLP.*

network learns to correctly reproduce these mappings. Usually, the outputs of an MLP are one-hot encoded classification vectors.

An approach widely used to train MLP's is stochastic gradient decent (SGD), in which the parameter space of the MLP is iteratively traversed, given one training sample at a time, with the goal of minimizing some objective function that is indirectly related to the performance of the MLP. The direction of traversal through this parameter space is controlled by the gradient of the output error (objective function) of the MLP with respect to its weight and bias parameters. This gradient can be computed either analytically by using error back propagation [11], or by estimation through perturbation of the MLP's parameters to produce small changes in the objective function at the MLP's output [2]. In our implementation, we use error back propagation, in which an error signal at the output of the MLP is propagated backwards through the network using the chain rule in order to derive the partial gradients with respect to each parameter in the MLP. As the MLP is presented with training data pairs, its parameters are iteratively updated until an error minima is reached and it is at this point that the MLP is trained on the presented data.

Fig. 1 shows the basic structure of an MLP. Each layer of the MLP is a vertical column of neurons (circles) in Fig. 1. The input vector in Fig. 1 is a sample from the training dataset which is then propagated and transformed at each layer until a vector at the output layer is produced. The process of propagating data throughout the network can be expressed locally at each neuron in terms of the input values it receives. The computation at each neuron can therefore be done independently among neurons of the same layer, but the computation from layer to layer must be synchronized, as the outputs from one layer depend on the outputs of the previous layer.

Usually, MLP's are trained using batches, in which the gradient of the objective function with respect to the parameter space of the model is computed after the propagation of multiple input samples is performed. Since each input sample presented to the model belongs to a single output class, implementing batch updates results in a gradient computation that represents multiple classes, which is more accurate and speeds up training. Parameter updates can also also occur after the propagation of a single sample in the training data. This training is usually slower, as parameter updates are not necessarily in the direction that decreases overall error the most [1], but the local computation at each neuron is generally simpler and therefore we design the learning algorithm of our parallel MLP program using this approach.

2.2 Parallel Neural Networks

Much research exists on the topic of speeding up artificial neural networks (ANNs) through parallelism. Pethick et al. discuss four approaches: training session, training sample (e.g. [4], neuron, and weight parallelism [10]. This list is not exhaustive; there are various other approaches, such as treating the neuron weights as a matrix multiplication and parallelizing it, which can work well for GPUs [9].

The best parallelization method will depend on the target architecture and application. For example, Sainath et al. specifically focus on network training on a BG/Q supercomputer [12]. Meanwhile, Yan et al. investigate optimal combinations of neuron, matrix multiplication, and "service" (similar to session) parallelism for quickly running pre-trained networks [14]. We choose to study the training of neural networks in this report, specifically using gradient decent with back propagation.

Pethick et al. analytically and empirically compare sample parallelism and neuron parallelism for training neural networks with back propagation [10]. For the first approach, multiple ANNs process different training samples in parallel and then share their weight updates

in a communication phase. For the second, the neurons are divided up over processes and their computations performed in parallel. Pethick et al. report that sample parallelism exhibits speedup with more processes and larger numbers of training samples while neuron parallelism exhibits speedup with more processes and larger network sizes.

Sample parallelism may be preferable for compute clusters with relatively slow communications because it requires sending a few large messages unlike neuron parallelism, which requires sending many small messages often [4, 10]. In contrast, since the BG/Q's torus network has very high bandwidth (2 GB/s) and low latency (90 ns), we choose to study neuron parallelism and its scaling limits due to dense, frequent communications.

3 Methods

For this study, we construct several MLP models containing one or two hidden layers in which the number of neurons in each hidden layer (network width) is the same for each model. We initially construct an MLP with two hidden layers, each with a width of 100 neurons, to be trained on the MNIST dataset. A replica of this model is constructed in Google's TensorFlow in order to rapidly tune hyper-parameters and verify that the model can achieve at least 90% accuracy. For the parallel scaling studies, we test several MLP's that each have one hidden layer but variable network widths.

3.1 Algorithm

The computation performed at each neuron of an MLP can be described in two steps; one step for the forward propagation of input signals to an output signal and one step for the backward propagation of a gradient signal at the output of a neuron to gradient signals at the inputs of a neuron.

For forward propagation, each neuron computes a linear combination between its weights and the inputs it receives from other neurons and adds a bias value to this computation. This value is then fed to a sigmoid activation function that squashes the output to a value between 0 and 1. The sigmoid activation function adds non-linearity to each neuron and therefore gives the overall MLP more flexibility to learn non-linear relationships. Usually, a different activation function is used for the output layer of an MLP but in our case we will use the same sigmoid activation function for every neuron including the neurons in the output layer.

For backward propagation, each neuron is presented with an accumulation of error gradient signals at its input from neurons in the layer ahead of it in the MLP. These gradient signals are used to derive the local gradients of the error with respect to each parameter of the neuron and then another gradient signal is passed to neurons in the layer behind this neuron in the MLP. As this gradient signal propagates to the input layer of the MLP, each neuron is able to update its parameters after performing its local computations in order to attempt to decrease the error that was seen at the output layer of the MLP.

The computation for forward propagation and backward propagation performed locally on each neuron in the MLP is outlined in Algorithm 1 and Algorithm 2 respectively. Since the computation of a neuron depends on signals from all neurons in either the layer before or after it, computation must be synchronized at every layer before beginning any computation at the next layer of signal propagation.

Algorithm 1 Forward Propagation: Neuron in Layer ℓ

1: $activation := 0$;
2: **for** $i = 1$ to $N_{\ell-1}$ **do**
3: **if** ℓ is input layer **then**
4: $Z[i] \leftarrow i^{th}$ attribute of input file sample
5: **else**
6: $Z[i] \leftarrow$ recv output from i^{th} neuron in layer $\ell - 1$
7: $activation \mathrel{+}= W[i] * Z[i]$
8: $activation \mathrel{+}= bias$
9: $out \leftarrow 1/\left(1 + exp\left(-1 * activation\right)\right)$
10: $doutdB \leftarrow out * (1 - out)$
11: **if** ℓ is not output layer **then**
12: **for** $i = 1$ to $N_{\ell+1}$ **do**
13: send out to neuron i in layer $\ell + 1$

Algorithm 2 Backward Propagation: Neuron in Layer ℓ

$dEdout := 0$;
for $i = 1$ to $N_{\ell+1}$ **do**
  if ℓ is output layer **then**
    $d_i \leftarrow out_i - label_i$
  else
    $d_i \leftarrow$ recv $dEdz_i$ from neuron i in layer $\ell + 1$
  $dEdout \mathrel{+}= d_i$
$dEdb \leftarrow dEdout * doutdb$
$b \mathrel{-}= \eta * dEdb$
for $i = 1$ to $N_{\ell-1}$ **do**
  $dEdw_i \leftarrow dEdb * Z[i]$
  $dEdz_i \leftarrow dEdb * W[i]$
  $W[i] \mathrel{-}= \eta * dEdw_i$
  if ℓ is not input layer **then**
    send $dEdz_i$ to neuron i in layer $\ell - 1$

On initialization of forward propagation, the neurons at the input layer simply read their inputs from a file containing the input data. On initialization of backward propagation, the neurons at the output layer do an extra computation to calculate the error of their output with respect to a training label.

3.2 File I/O

In order to achieve optimal performance for reading the input data, MPI collective I/O operations are used, namely `MPI_File_read_at_all`. The benefit for this collective I/O is that file read requests can be pooled together into larger requests. These larger requests are more efficient as MPI can align request sizes to the block size of the file system, which has been shown to provide optimal performance [8].

We chose to have each rank read a contiguous portion of a given input file rather than the entire file because doing so requires much less memory. When it is time to process a sample from the file, the rank possessing it can broadcast it to the rest of the network. A single additional broadcast per sample is not a concern for scaling because of the large number of broadcast and reduce operations required to propagate the sample through the network.

We store training data for our MLP in a concatenated form, in which each training sample contains a feature vector concatenated with its one-hot encoded classification label. In order to read in sample data, both the feature vector and classification label must be consistent in data type. For the MNIST data set, we map the feature vectors to integers between 0 and 255 and therefore a single byte can be used to store each pixel. The MNIST data set contains images with dimensions of 28x28, which equates to a total of 784 pixels per image. Given this size, the feature vector will require 784 bytes per sample.

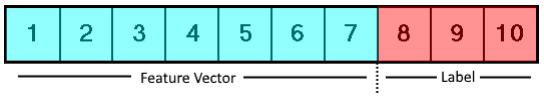

Figure 2: *Example of data layout for a 7 dimensional data set with 3 possible classifications. Total length of one sample is 10 bytes.*

As for the classification labels, this same byte stream method is used. Since each output neuron activation signals a classification, then the sample label can be expressed as the desired output layer state. See Fig. 2 as an example. For MNIST, there are a total of 10 classifications so the total size required for each sample label is 10 bytes. The classification label is appended to the end of the feature vector which gives a total of 794 bytes per sample.

We create several additional training data files of dummy data that are constructed the same way as our MNIST dataset. The dimensionality of a feature vector and a classification label is identical within each of these files which allows for simpler MLP architectures to be used when performing our scaling tests.

3.3 Parallel Implementation

The parallel implementation of the MLP is written in C and uses the MPI library for parallelization of the program into MPI ranks. Each MPI rank performs the computation outlined in Algorithm 1 and Algorithm 2 for one or more neurons in the MLP. Since neurons of any layer cannot process information until neurons in previous layers have processed their information, MPI broadcasting is used not only for rank communication, but to synchronize layer by layer processing.

For the forward propagation step, each rank is responsible for calculating the output of a subset of the neurons in each network layer using the forward propagation algorithm. Since the network is fully-connected, all active ranks for a layer transmit the output values of their neurons to all ranks using `MPI_Bcast` so the outputs will be available to the next layer.

For the backward propagation step, layers are processed in reverse order. For each layer, the MPI ranks calculate the results of the backwards propagation algorithm and send the appropriate error signals to the neurons in the preceding layer. Ranks only store the weights of the neurons for which they are responsible to avoid communication overhead when weights are updated. However, this means that the ranks must perform a series of `MPI_Reduce` operations to aggregate (sum) the error signals.

4 Experiments

4.1 Testing the MLP

We test our parallel MLP training program by comparing its results to TensorFlow. Fig. 3 shows the average error of the MLP with two hidden layers and 100 neurons in each hidden layer as it is trained on the MNIST dataset. The left plot in Fig. 3 is the result of training the MLP in TensorFlow and the right plot in Fig. 3 is the result of training the MLP with our parallel MPI program on the Kratos cluster at RPI with 10 MPI ranks. In each of the plots from Fig. 3, the MLP is trained on 300,000 iterations, equating to exactly 6 epochs through the dataset. We can see that the convergence plots of training the MLP in TensorFlow and training the MLP with our parallel program are very similar. We are able to achieve 92% accuracy on this MLP model with this training duration in TensorFlow and we are able to reproduce these results to get 92% accuracy after training the same MLP with our parallel program.

4.2 Scaling Experiments

In this section, we outline two scaling experiments that are performed using our parallel program to train several MLP models on the BG/Q. Since we vary the size of the MLP models in these experiments, we "train" the MLP using dummy data, and probe different parts of the program to observe its performance as it performs the same computations as if it were learning a real dataset.

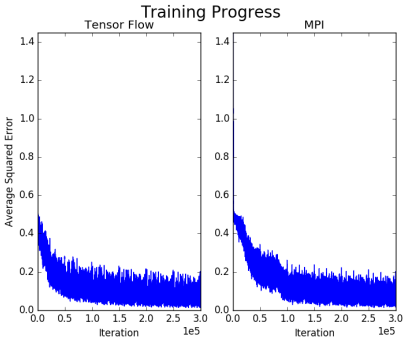

Figure 3: *Results of training the MLP on the MNIST dataset in TensorFlow and with our parallel MPI program.*

In each scaling experiment, we perform 2000 training iterations on five MLP's that have an equal number of neurons in each of their layers, hence the creation of dummy training data in which feature vectors and label vectors are the same length. The network widths of the five MLP's in each experiment varies from 32, 128, 512, 2048 to 8192 neurons and each MLP is trained multiple times using varying numbers of MPI ranks. For the first scaling experiment, we use up to 64 MPI ranks per BG/Q node and for the second scaling experiment we further distribute computation on the BG/Q and use up to 16 MPI ranks per BG/Q node.

During each training run within an experiment, we collect the time our parallel program spends doing forward propagation, backward propagation, MPI broadcasts and MPI reductions as well as the total time needed to perform the 2000 iterations of training. We calculate these times by using the cycle counter to time individual broadcasts etc. in each iteration. We take the the maximum time for each measurement across ranks using `MPI_Reduce` at the end of the program and then sum the times up over all iterations. Because the MLP must synchronize between processing each layer and the computations are nearly identical across ranks, the cycle counts vary relatively little. The total times calculated by taking the

minima instead of maxima across ranks differ by less than 2.5% for the forward and backward propagation for all experiment configurations. For the broadcast and reduce times, the differences are within 10% for 78% of the configurations. The greater differences generally correspond to the networks with (exponentially) larger widths and thus large times, so the main effects should not be perturbed.

4.3 Results and Analysis

Fig. 4 shows overall times to train the MLP on 2000 samples with different network widths (colors) and levels of parallelization (x axis). The training time increases with the network width, but the optimal level of parallelization increases as well. In fact, the MLP runs fastest with 8 neurons per MPI rank for all network widths except 512, for which the fastest run has 16 neurons per rank.

The training time reflects the combination of computation and communication times. For both forward and backward propagation, the computation per rank is $O(\frac{m^2}{n})$ where m is the network width and n the number of MPI ranks [13]. Depending on how the broadcast and reduce trees are constructed and the extent to which communications are limited by bandwidth or latency, the `MPI_Bcast` and `MPI_Reduce` operations may take $O(\log(n))$ or $O(\frac{m}{n}\log(n))$ time [6, 5]. Each rank performs one broadcast and reduce per iteration, so the communication time per iteration is $O(n\log(n))$ or $O(m\log(n))$. Setting the derivative of the combined computation and communication time with respect to n equal to zero yields $\frac{m}{n} = c\sqrt{1+\log(n)}$ or $\frac{m}{n} = c$ respectively where c is some constant. In either case the optimal ratio is a constant or nearly constant. Since the BG/Q has a 1.6 GHz clock rate and 2 GB/s torus network bandwidth it is reasonable that c, the ratio of the constant scaling factors for the communication and computation times, is within a couple of orders of magnitude from 1 (e.g. the observed ratio of 8).

Fig. 5 shows the speedup calculated as $\frac{T_{1,m}}{T_{n,m}}$ (dotted lines) and $\frac{T_{1,32}}{T_{1,m}}(\frac{m}{32})^2$ (solid lines) where $T_{n,m}$ is the overall execution time for the network with n ranks and width m . We include the second formula because the 1 rank experiment with a network width of 8192 did not complete in 50 min, so we are missing $T_{1,8192}$. To calculate the speedup relative to $T_{1,32}$ instead, we scale the speedup by the relative problem size $(\frac{m}{32})^2$ because we are interested in the network's strong scaling behavior. The

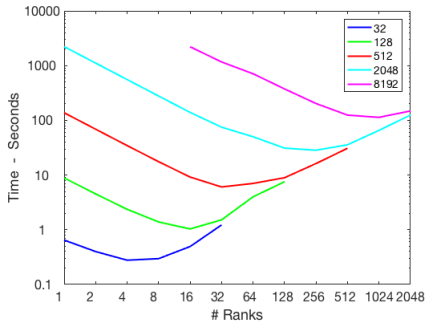

Figure 4: *Time to perform 2000 training iterations for various network widths and numbers of MPI ranks.*

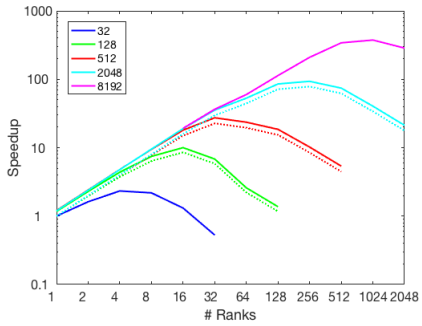

Figure 5: *Speedup achieved on 2000 training iterations for various network widths and numbers of MPI ranks.*

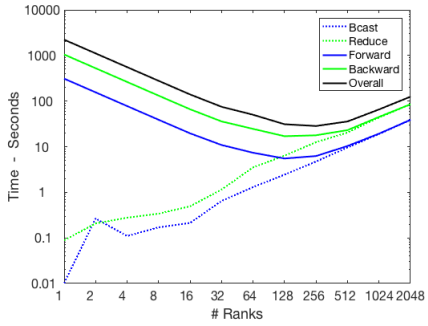

Figure 6: *Breakdown of the time spent in some important operations during 2000 iterations of training a network of width 2048*

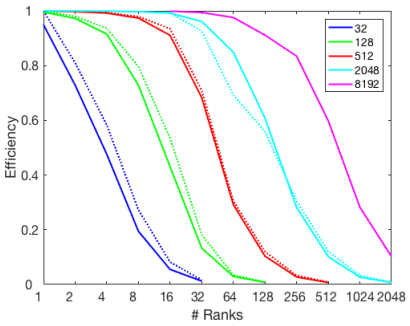

Figure 7: *Parallel efficiency on 2000 training iterations for various network widths and numbers of MPI ranks.*

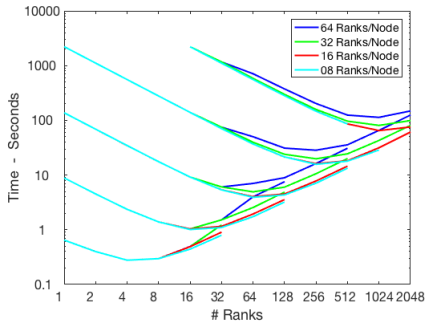

Figure 8: *Time to perform 2000 iterations of training for various network widths, numbers of MPI ranks, and distributions of ranks.*

maximum observed speedup, 375x, was for 1024 ranks and network width 8192.

Fig. 6 shows the total times spent on forward and backward propagation as well as the constituent broadcast and reduce times for a network of width 2048. Communication overhead takes over the bulk of the forward and backward propagation times for the experiments with more ranks. This pattern can be concisely captured with the parallel efficiency curves plotted in Fig. 7.

Fig. 7 shows the efficiency of the MLP calculated as $\frac{T_{1,m}}{nT_{n,m}}$ (dotted lines) and $\frac{T_{compute}}{T_{total}}$ (solid lines) where $T_{compute}$ is the overall time T_{total} minus the time spent on communications, i.e. `MPI_Bcast` and `MPI_Reduce`. The figure demonstrates that the first measure of efficiency reflects the fraction of execution time spent on "real computational work" as it should. The efficiency is greatest for 1 rank since there is no communication overhead. The greatest speedup is obtained at efficiencies between 0.29 and 0.68 depending on the network width, which makes sense since the minimum times are achieved at a compromise between computation and communication bottlenecks.

The scaling experiments described above use up to 64 ranks per BG/Q node. We repeat the experiments with 32, 16, and 8 ranks per node and plot the timing results in Fig. 8. Reducing the number of ranks per node improves running times of some configurations by reducing the communication overhead (verified with figures such as 6, not shown). Although decreasing the number of ranks per node may increase the physical spread of ranks across the BG/Q and thus communication latencies, it also means fewer ranks must share the network hardware of individual compute cards. Kumar et al. provide one explanation for the decrease in performance observed when over committing the BG/Q nodes with 64 ranks [7]. They note that since BG/Q nodes share a 32MB L2 cache used for intra-node communication, this memory can potentially become full when a large amount of compute-intensive ranks run on a BG/Q node. As a result, intra-node communication must utilize slower DDR memory. In data-dense applications such as training an MLP where there are many floating point parameters that must be stored in memory, it becomes much easier to fill the L2 cache on a BG/Q node.

Since communication overhead appears to be the limiting factor to scaling, we include one more set of experiments to investigate whether non-blocking broadcast and reduce operations provide better performance. The reasoning is that if all ranks post communication requests as soon as possible they may more fully utilize the network's bandwidth than if ranks sequentially perform broadcast or reduce operations. However, the BG/Q does not support asynchronous broadcast and reduce operations, so we perform these experiments on the RPI Kratos cluster with 1, 2, 4, and 8 ranks and networks widths of 32, 128, 512, and 2048. The asynchronous communications exhibit slightly faster run times (0.4-8.9%) in 12 of the 16 configurations but much slower run times (31-34%) in 3 configurations. The slower configurations do not have a clear patters since they include 3

different network sizes and 2 different numbers of ranks. The RPI Kratos cluster does not dedicate a subset of cores to a single application in contrast to the BG/Q, so interference from other users and processes may be responsible of the inconclusive results.

5 Future Work

Currently the tradeoff between computation parallelism and communication overhead limits the performance of the MLP. Multithreading is one avenue for improvement. With multiple threads it would be possible to parallelize neuron computations in the forward and backward propagation steps without introducing additional network communications because the memory is shared. Although multithreading does not remove the communication barriers to scaling, it could improve the performance and increase the optimal ratio of neurons per rank.

Earlier we mentioned that batch training is often used to train MLP's faster. By implementing the forward and backward propagation algorithms with batch updates, error gradients would be calculated from the propagation of multiple samples at a time and therefore we could drastically reduce the number of broadcast and reduce operations needed during training as well as improve the accuracy of the gradient computation used for each parameter update. As opposed to our current implementation where we propagate one training sample at a time, propagating batches of multiple samples at a time by sending larger messages during forward propagation could result in much faster learning. When performing back propagation, the gradient of the output error would be calculated from multiple samples and therefore the amount of backward propagation calculations would follow the reduction in forward propagation calculations, resulting in a decrease in reduce operations as well.

References

[1] Olivier Bousquet and Léon Bottou. The tradeoffs of large scale learning. In J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 161–168. Curran Associates, Inc., 2008.

[2] Gert Cauwenberghs. A fast stochastic error-descent algorithm for supervised learning and optimization. *Advances in neural information processing systems*, pages 244–244, 1993.

[3] I-Hsin Chung, Tara N Sainath, Bhuvana Ramabhadran, Michael Picheny, John Gunnels, Vernon Austel, Upendra Chauhari, and Brian Kingsbury. Parallel deep neural network training for big data on blue gene/q. *IEEE Transactions on Parallel and Distributed Systems*, 2016.

[4] George Dahl, Alan McAvinney, Tia Newhall, et al. Parallelizing neural network training for cluster systems. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks*, pages 220–225. ACTA Press, 2008.

[5] T. Hoefler, C. Siebert, and W. Rehm. A practically constant-time mpi broadcast algorithm for large-scale infiniband clusters with multicast. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, March 2007.

[6] Sameer Kumar, Amith Mamidala, Philip Heidelberger, Dong Chen, and Daniel Faraj. Optimization of mpi collective operations on the ibm blue gene/q supercomputer. *Int. J. High Perform. Comput. Appl.*, 28(4):450–464, November 2014.

[7] Sameer Kumar, Amith R Mamidala, Daniel A Faraj, Brian Smith, Michael Blocksome, Bob Cernohous, Douglas Miller, Jeff Parker, Joseph Ratterman, Philip Heidelberger, et al. Pami: A parallel active message interface for the blue gene/q supercomputer. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 763–773. IEEE, 2012.

[8] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock. I/o performance challenges at leadership scale. In

Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, pages 1–12, Nov 2009.

[9] Kyoung-Su Oh and Keechul Jung. Gpu implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314, 2004.

[10] Mark Pethick, Michael Liddle, Paul Werstein, and Zhiyi Huang. Parallelization of a backpropagation neural network on a cluster computer. In *International conference on parallel and distributed computing and systems (PDCS 2003)*, 2003.

[11] D.e. Rumelhart, G.e. Hinton, and R.j. Williams. Learning internal representations by error propagation. *Readings in Cognitive Science*, pages 399–421, 1988.

[12] Tara N Sainath, I-hsin Chung, Bhuvana Ramabhadran, Michael Picheny, John Gunnels, Brian Kingsbury, George Saon, Vernon Austel, and Upendra Chaudhari. Parallel deep neural network training for lvcsr tasks using blue gene/q. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[13] Abhinav Vishnu, Charles Siegel, and Jeffrey Daily. Distributed tensorflow with mpi. *arXiv preprint arXiv:1603.02339*, 2016.

[14] Feng Yan, Olatunji Ruwase, Yuxiong He, and Evgenia Smirni. Serf: efficient scheduling for fast deep neural network serving via judicious parallelism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 26. IEEE Press, 2016.