PARALLELIZING NEURAL NETWORK TRAINING FOR CLUSTER SYSTEMS

George Dahl
Computer Science Department
Swarthmore College
Swarthmore, PA 19081
email: gdahl@cs.swarthmore.edu

Alan McAvinney
Computer Science Department
Swarthmore College
Swarthmore, PA 19081

Tia Newhall
Computer Science Department
Swarthmore College
Swarthmore, PA 19081
email: newhall@cs.swarthmore.edu

ABSTRACT
We present a technique for parallelizing the training of neural networks. Our technique is designed for parallelization on a cluster of workstations. To take advantage of parallelization on clusters, a solution must account for the higher network latencies and lower bandwidths of clusters as compared to custom parallel architectures. Parallelization approaches that may work well on special purpose parallel hardware, such as distributing the neurons of the neural network across processors, are not likely to work well on cluster systems because communication costs to process a single training pattern are too prohibitive. Our solution, Pattern Parallel Training, duplicates the full neural network at each cluster node. Each cooperating process in the cluster trains the neural network on a subset of the training set each epoch. We demonstrate the effectiveness of our approach by implementing and testing an MPI version of Pattern Parallel Training for the eight bit parity problem. Our results show a significant speed-up in training time as compared to sequential training. In addition, we analyze the communication costs of our technique and discuss which types of common neural network problems would benefit most from our approach.

KEY WORDS
Parallel Neural Network Training, Cluster

1 Introduction

Artificial neural networks (ANNs) are tools for non-linear statistical data modeling. They can be used to solve a wide variety of problems while being robust to error in training data. ANNs have been successfully applied to hosts of pattern recognition and classification tasks, time series prediction, data mining, function approximation, data clustering and filtering, and data compression.

ANNs are trained on a collection of {input, desired output} pairs called training patterns. The set of training patterns is typically quite large. Backpropagation [7] is one of the most widely used training algorithms for ANNs. It can take a very long time to train an ANN using backpropagation, even on a moderately sized training set. Our work addresses the long training times of sequential ANN training by parallelizing the training in a way that is optimized for cluster computing.

Parallelization of the training task is a natural response to the issue of long training times. One way to parallelize neural network training is to use a technique called Network Parallel Training (NPT). In this approach the neurons of the ANN are divided across machines in the cluster, so that each machine holds a portion of the neural network. Each training pattern is processed by the cluster machines in parallel. To process a single training pattern, communication is required between any cluster nodes containing neurons that are connected by an edge (see Figure 1).

Network Parallel Training attacks the training time problem by improving the time to process a single training pattern. It has the potential to work well when implemented on special-purpose parallel hardware, however, it is much less likely to work well on a cluster of workstations connected by a LAN. In order to benefit from being split up across multiple machines, the neural network must be large enough to prevent the cost of communication between neurons on different cluster nodes from overwhelming the potential speedup from parallelizing the computation. The network latency and bandwidth on most cluster systems will significantly limit the degree of parallelism possible for training even large sized ANNs. When the original ANN is of small or medium size, there is even less benefit from an NPT approach as there will be little local computation between communication points.

Our approach, called Pattern Parallel Training (PPT) is more appropriate for cluster systems. In PPT the full ANN is duplicated at every cluster node and each node locally processes a randomly selected subset of patterns from the full training set. After a round of local processing of a subset of training patterns, a node exchanges its weight updates with other nodes. Each node then applies the weight updates to its copy of the ANN and determines if training is complete or if another round of local training is necessary (see Figure 2). By having each node apply the weight updates from every other node, we ensure that all copies of the ANN are identical. PPT uses parallelism to speed-up the processing of the entire set of training data by having each cluster node locally process a fraction of the the full set of training patterns. By having each node deal only with a subset of the training set each round, Pattern Parallel Training results in a decrease in the time to process

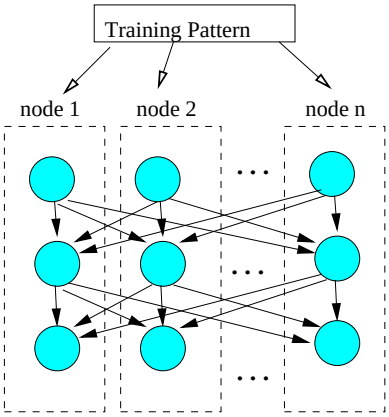

Figure 1. Network Parallel Training. *The neurons of the Artificial Neural Network are distributed across cluster nodes. The nodes work in parallel to process each pattern of the training set. Inter-node communication is required for all edges that span two cluster nodes.*

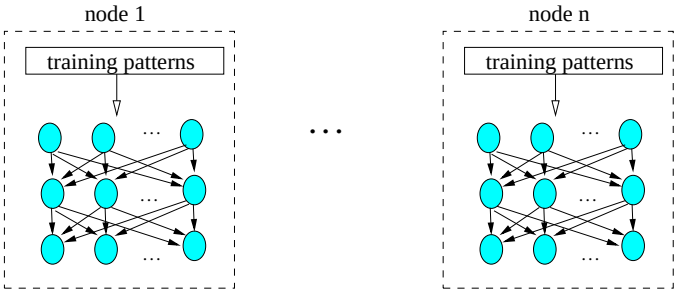

Figure 2. Pattern Parallel Training. *The ANN is fully duplicated at each node, and each node in parallel locally processes a subset of the patterns from the full training set. At the end of each local round of processing, each node broadcasts its weight updates to other nodes.*

the entire training set while incurring no communication costs within a round of training (unlike NPT that requires communication on each pattern). On cluster systems, PPT reduces the amount of communication necessary to process the set of training patterns, and as a result will allow for a higher degree of parallelism than NPT.

Pattern Parallel Training on a cluster of workstations can be an effective solution to all the causes of long training times in sequential ANN training; PPT should provide speedups when the neural network is very large, when the training set is very large, and when a large number of epochs (passes through the entire training set) are required to complete training. In addition, PPT will work well even when the number of neurons in the ANN is small (one case in which NPT does not work well).

The remainder of the paper is organized as follows. In Section 2 we present related work in parallelizing neural network training. In Section 3 we briefly present background information on sequential neural network training. In Section 4 we present the details of our solution and we present an implementation of our solution. In Section 5 we present results of applying our PPT approach to the eight bit parity problem. Finally, in Section 6 we present future directions for our work.

2 Related Work

Most of the previous work in parallelizing neural network training has focused on creating special purpose neural network hardware [2, 4] and using an approach similar to what we call Network Parallel Training where the neurons of the ANN are parallelized. The obvious problem with special-purpose hardware is that it is very expensive to acquire and time consuming to build.

There is some work on parallelizing ANN training on cluster systems. Omer et. al. [6] uses genetic algorithms to parallelize ANN training on a cluster of workstations. They use a hybrid approach that combines genetic algorithms and backpropagation. They create a diverse population of ANNs that are distributed across the nodes. Each node, in parallel, performs an independent full sequential training of its ANN. At the end of a parallel training round, a single master node collects results and chooses good candidates for generating a new population of ANNs to independently train in the next round. They use parallelism to sequentially train multiple, different ANNs in parallel and choose the best result. We use parallelization to speed up training of a single ANN.

The work most similar to ours, Suri et. al. [8], uses parallel ANN training to learn a pattern classification problem. They use a parallelization technique similar to our Pattern Parallel Training. However, our system differs in several ways. First, they use a central coordinator to which worker nodes send weight updates and receive new weight values. Our system is completely decentralized, thus reducing the potential bottleneck of a centralized approach. Second, they deterministically partition patterns from the training set across nodes. Our approach uses a random sampling on each node of the full training set each round of local training. By randomly selecting from the full training set, our solution will not suffer from anomalies that can occur when less than the full set of training data is processed each round. Finally, they parallelize the learning task by taking advantage of the parallel structure of the Levenberg-Marquardt training algorithm. In our work, we use the backpropagation training algorithm, which results in a different approach to parallelizing ANN training.

3 Sequential Training of Artificial Neural Networks

An Artificial Neural Network is a computer model for learning that is inspired by the brain. An ANN consists of a set of neurons, each neuron has a number of weighted input edges and a number of weighted output edges. A neu-

ron computes its "activation" based on an activation function (generally the logistic sigmoid) applied to the weighted sum of its inputs. Its activation value is then sent along the outgoing edges where it serves as input to other neurons.

Backpropagation is the most widely used model for ANNs. Generally, backpropagation is applied to simple feed-forward networks which are composed of layers of neurons. The neurons in one layer are fully connected to the neurons in the layers before and after it. Typically, there are three or more layers: an input layer of neurons that take a training pattern as its input; multiple hidden layers in the middle; and an output layer that computes the result of the ANN. The ANN learns by adjusting its weights after processing training patterns. An epoch is a single presentation of the entire set of training patterns ({input, output} pairs). Throughout an epoch, error values on each pattern are summed to obtain the error on the training set. At the end of each epoch, this error value is used to compute the changes to the weights and to determine if another round of training is necessary. Typically, the ANN takes many epochs before it has learned the problem to within an exceptable error. Sometimes learning is not successful due to a particular random initialization of the connection weights. In these cases learning is stopped after some max number of epochs, and new random weights are computed for another try.

There are a variety of factors that work together to cause very long training times for backpropagation. The primary cause is due to the large number of tunable parameters (the connection weights), which can be in the hundreds or thousands. Backpropagation performs a gradient descent on the error surface generated by evaluating the mean squared error of the neural network on the training set. Gradient descent can be fooled by local minima of the error surface and can take a very large number of epochs to converge to an acceptable minimum. It is not uncommon for more difficult problems to take hundreds of thousands or even millions of epochs. Additionally, training can take a long time due to a large training set size. Also, just processing a single training pattern can take a long time on large ANNs.

4 Pattern Parallel Training

Pattern Parallel Training is a technique for parallelizing the training of artificial neural networks that is designed to work well on cluster computers. In PPT the full ANN and the full set of training data is duplicated at each node. Each node then trains its local copy of the ANN on a randomly selected subset of the training data. When the local computation is complete, nodes broadcast their final weight updates to other nodes. In our system, an epoch consists of the local computation of the weight updates on a subset of the patterns and the broadcast of these weight updates. At the end of each epoch, every node applies the weight updates from the other nodes to its ANN and determines if the training is complete or if another training epoch is needed based on the error condition. The speed-up in training is achieved by shortening the time of performing a single epoch; in parallel each node evaluates just a subset of the full training data each epoch. By communicating only at the end of each epoch, the communication costs of Pattern Parallel Training are minimized.

4.1 Neural Network Issues

In designing our solution, we needed to address several issues related to parallelizing the ANN. These include calculating weight updates and error when the the training is distributed, determining the stopping condition, ensuring that the duplicated ANNs are identical, and determining how many training patterns should be presented each epoch.

In a training epoch in backpropagation, the gradient of the error on the training data with respect to the connection weights is used to compute the new weights. The weight updates are defined as the difference between the new weights and the old weights. For each pattern in a batch of data, the incremental weight updates for that pattern are added to a running total of weight updates. The total weight updates are applied only once at the end of the epoch. This means that when the batch is split across multiple processes, the weight updates from each process can be summed to produce a single final set of weight updates.

In our system, the neural network being trained is replicated across processes, so each process must communicate with every other process once every epoch to ensure that all processes get the same final set of weight updates.

The stopping condition for training is triggered by either the max number of epochs being reached (an unsuccessful training attempt), or by the error being below some threshold. When an ANN is trained serially, keeping track of error on the training set is trivial because the mean training set error must be computed to find the correct weight updates anyway.

Estimating error in Pattern Parallel Training is more complicated than in sequential training because each process is working with a randomly selected subset of the training set. All processes must agree to stop at the same time, and the union of the subsets that each process works with in one epoch does not necessarily equal the full training set.

Our solution is to have each process maintains an average mean squared error over the last kn/p training patterns that it has processed where n is the training set size, k is a small arbitrary constant (between 1 and 10, for example), and p is the number of processes. The purpose of k is to make sure the error history is large enough to have a high probability of including almost all of the training set. Each process piggybacks its mean squared error estimate onto the broadcast of its weight updates. The error estimates from all the processes are averaged to create an estimate of the error on the training set. When the global error estimate gets below the stopping threshold, each process tests the neural network on a test set of patterns to verify that the

stopping condition has been met.

Another issue that we need to address has to do with when weight updates should be performed. In serial neural network training, on some problems networks learn faster when weights are updated after each training pattern (incremental training), and others learn faster when weights are updated after the entire training set has been presented (batch training). It has been argued in [1] that suitably designed on-line (incremental) learning algorithms will asymptotically outperform batch learning as the amount of training data grows without bound. However, there exist many learning tasks for which batch learning is superior.

To take advantage of parallelism, our approach requires that the number of pattern each node processes per epoch be some fraction of the total training set size. If the number of patterns is close to 1, we approximate incremental learning, but at the expense of more communication overhead because the weight updates are communicated more frequently. If the number of patterns is close to the full size of the training set, then the time to process a single epoch approaches the per epoch processing time of serial batch training. By tuning the number of the training patterns to the particular problem, our approach has the potential to work well for problems that perform well with incremental training and problems that perform well with batch training.

Another issue is how a node chooses a subset of training patterns from the training set each epoch. In serial batch learning, sequentially sampling the data from the training set is slightly more effective than randomly sampling because it ensures that each training pattern will be seen the same number of times. However, in incremental serial training it is essential to sample the data randomly to avoid learning an artifact caused by the ordering of the data. Because our approach is somewhere between these two extremes, we choose random sampling of the full test set to avoid problems with learning artifacts that could be caused by either a static partitioning of the training set across nodes or by having each node process patterns in some pre-defined order.

Finally, we wanted to have support for adding a momentum term to the weight updates. Adding a momentum term to the weight update formula is a common way of improving the performance of backpropagation. In serial training with momentum, weights are changed in the normal way except that the weight changes from the previous time step multiplied by some decay factor (the momentum) are added to the weight changes computed for the current time step. In Pattern Parallel Training because all processes apply all the weight changes at once, each node simply stores the previous weight changes to apply the momentum factor (just as it would be done in serial training).

4.2 Parallelization Issues

In applying Pattern Parallel Training to an ANN, we must consider how to partition the problem in such a way that we balance the use of network and CPU resources to achieve maximum speedup. In our system the amount of data that each node needs to communicate at the end of each epoch is fixed for a given problem (it is its set of weight updates and error values for the ANN), but the length of each epoch and the total number of epochs necessary to learn can vary. Based on the particular problem and the on the particular cluster system, optimal epoch size and degree of parallelism will vary.

Generally, increasing the number of patterns trained each epoch decreases the time per pattern presentation because less time spent on communication between processes. However, increasing the the number of patterns trained each epoch can increase the number of epochs required for the network to learn the problem successfully and at the extreme approaches serial training. The exact relationship between the time per training pattern presentation and the number of patterns trained each epoch depends on cluster specific parameters, primarily the ratio of processor performance to network performance.

How fast the ANN training converges to an acceptable solution and how consistently it does so are important metrics for evaluating performance, and the number of training patterns per epoch will effect both of these. Unfortunately, the optimal number of training patterns per epoch is highly variable and impossible to absolutely determine at runtime since it requires one to have already solved the problem the neural network is trying to solve. Our current solution is to make it a user-adjustable parameter. However, for a given system and a given ANN, we expect that our PPT system can automatically generate good guesses for the degree of parallelism and the patterns per epoch. We plan to investigate this further as part of our future work.

Another issue related to parallelization of training has to do with maintaining consistent copies of the ANN across nodes. It is essential to keep the copies of the neural network consistent at each node, otherwise it is unlikely that learning will terminate. In order to maintain consistency, all nodes must communicate with all other processes at the end of each epoch, and no node can start its next epoch until it has applied the weight updates from all other nodes. This can be a costly synchronization point, however, we can take advantage of multicast mechanisms to improve the performance of communicating weight updates. In our implementation we use MPI routine Allgather that makes use of Ethernet multicasting and avoid using a more expensive point-to-point solution for communicating weight updates.

The Allgather routine is a synchronous broadcast, which allows us to keep the neural networks completely consistent. However, the speed of execution of the entire system is limited by the slowest node. On a dedicated homogeneous cluster, these limitations should not be much of an issue. However, on heterogeneous or non-dedicated clusters this may be a problem. In such systems a load balancing scheme can be used to determine the initial degree of parallelism and process placement in the system. Also, periodic profiling and re-balancing could be performed to

improve performance after the application begins.

4.3 Implementation

Currently, our system is implemented as an MPI [3] application that uses the FANN [5] open source neural network training library. We use the FANN library to perform the local training of the ANN on each node. Our system handles the initial replication of the full neural network and training data across all processes. It also determines the local set of training patterns for each local epoch by randomly selecting them from the full set. In addition, it handles broadcasting the weight updates at the end of each local epoch, applying the weight updates from other nodes to the local ANN, and computing the error to determine if another epoch of learning is necessary. We use MPI Allgather to synchronize epochs across cluster nodes. To implement our system, we needed to modify FANN to export some of its internal structures so that we could communicate weight values at the end of each epoch. Our plan is to eventually implement a Pattern Parallel Training Library as an extension to FANN. ANN programmers could then use our library to parallelize the training of their ANN on clusters.

5 Results

We present results using Pattern Parallel Training to train an ANN for the eight bit parity problem. Our experiments were run on an eight node cluster with 1 Gigabit Ethernet switch [1]. We compare total time to learn on different numbers of nodes, and we evaluate the effects of different numbers of patterns per epoch on training.

We chose the eight bit parity problem because while it is very simple to specify, it also takes many training epochs to learn. Thus, it is a good candidate for our parallelization technique. Additionally, eight bit parity is commonly used as a benchmarking problem for neural networks. We ran all of our final experiments with a three layer ANN containing one hundred nodes in the hidden layer. The hidden layer is probably larger than required for this problem and it may allow the neural network to memorize all the training examples it sees instead of generalizing. However, we feel that this is not a major issue for our purposes because we are attempting to test the advantages of parallelization, not the capabilities of neural network generalization. The learning rate was set to ten percent and the momentum factor was set to thirty percent.

Table 1 shows the total execution time to train the ANN for different numbers of nodes. The results are shown for the best number of patterns per epoch for each number of nodes (i.e. in the 8 node case the best average training times occurred with the local epoch size is 32, and in the 2 node case they occurred when the local epoch size is 128).

[1] Each node has a Pentium4 processor, 80GB Seagate Barracuda7200 IDE disk drive, and 512MB of RAM.

Number of Nodes	Total Time	Local Epoch Size
1	689	NA
2	190	128
4	177	64
8	65	32

Table 1. Total Execution time (in seconds) for Pattern Parallel Training to learn the 8 bit parity problem. *Each row shows the average execution time (column 2) when run on a different number of nodes (column 1). The 1 node case is the time to perform a sequential batch training of the ANN. Time values are the average time for 10 successful runs, for the best local epoch size (given in column 3) for each number of nodes.*

The one node version is the serial batch training version of the problem. The results show a steady improvement as the number of nodes increases. PPT results in significant speed-ups over sequential version. The best speed-up of 10.6 was achieved by the eight node version of PPT (65 seconds vs. 689). The speed-up of almost 11 on eight nodes was due in large part to the parallel training, but also due to our version requiring fewer total epochs than the sequential batch version. Unfortunately, we do not have access to more nodes, so we are unable to test 16 and 32 node versions. If we had done so, we would expect that the speed-up gains would level off at some number of nodes as the communication costs start to out way the benefits of higher degrees of parallelism.

Table 2 shows results evaluating the local epoch size for the eight node version of the 8-bit parity problem. We show the number of epochs, the total training time, and the percent of the total time due to communication for different local epoch sizes. The results show that for smaller local epoch sizes (number of patterns each node processes per round), the benefit of parallelism is greater. For example, for a local epic size of 32, we get the fastest training time of 64.7 seconds. However, as expected, as the local epoch size decreases the communication cost increases (80.6% of the total time for a local epoch size of 32 vs. 42% of the total time for a local epoch size of 256). For a given problem, there likely will be a point where decreasing the local epoch size will result in little benefit to further parallelization because communication costs will dominate. However, for the 8 bit parity problem, we are still seeing a good speed-up with a relatively small epoch size of 32 even though a significant amount of time is due to communication. These results support our approach to parallelizing training.

5.1 The Types of Problems for which PPT will do well

Our results show that Pattern Parallel Training is a promising approach to speeding up the training time of ANNs. In general, we expect PPT to work well for problems that require a fair amount of computation time between communications. Batch training problems with large training

Local Epoch Size	Num Epochs to Train	Total Training Time	Percent Time Communicating
32	19,891.2	64.7	80.6%
64	44,441.6	86.0	71.7%
128	104,448.0	133.1	57.5%
256	109,568.0	91.9	42.6%

Table 2. Training Results for different Local Epoch Sizes on 8 nodes. *Each row lists the total number of epochs to train (column 2), the total time to train in seconds (column 3) and the percent of the total training time spent communicating weight updates (column 4) for different local epoch sizes (column 1). Local epoch size is the number of training patterns each node processes per round. Results are the average of 10 runs.*

sets, or problems that require a large amount of computation to process each pattern from the training set are likely to perform well under PPT as each ANN no longer has to process the entire training set each epoch.

Even though we expect PPT to perform well for these types of applications, our results show that even when PPT resulted in over 80% of the total time due to weight update communication, we are still seeing a significant improvement in total training time. As a result, PPT may do well for problems with much less local computation time than we originally had thought.

6 Conclusions and Future Work

We have shown that our Pattern Parallel Training technique can be used to significantly speed-up training of Artificial Neural Networks. Our results for the 8 bit parity problem show up to a factor of 11 speed-up in training time. Because backpropagation is such a widely used training algorithm, we expect that Pattern Parallel Training can be used to improve training time of a large number of ANN learning problems.

Some areas of future work include trying PPT on a larger set of ANN problems. We would like to examine problems with much larger training set sizes than the eight bit parity problem. This will allow us to further analyze our approach at randomly selecting patterns from the training set at each epoch and it will allow us to perform better tests on choosing epoch size. We also want to examine how well PPT works on problems that perform well using incremental serial training. This is a set of problems for which we are less confident our approach will always work well. Our goal is to more completely characterize the types of ANN problems for which PPT works particularly well. We would also like to examine using PPT on training algorithms other than backpropagation. Quickprop, Rprop and its variants, and Cascade Correlation would be obvious choices.

In addition to examining the types of ANN problems for which PPT works well, we plan to further develop our software. This will involve fully integrating PPT into the FANN library to provide a full library interface to PPT that an application developer could easily use to parallelize the training of their ANN. We also want to examine having our system automatically generate some of the parameters that are currently supplied by the user. We expect that by using cluster system performance data and information about the particular ANN, our system could automatically generate good values for epoch size and degree of parallelism.

A final area of future work involves examining combining Network Parallel Training with Pattern Parallel Training to try to take advantage of the strengths of both approaches.

References

[1] Leon Bottou and Yann LeCun. Large-scale on-line learning. In *In Advances in Neural Information Processing Systems 15*. MIT Press, 2004.

[2] Philipp Faerber and Krste Asanovi. Parallel neural network training on multi-spert. In *IEEE 3rd International Conference on Algorithms and Architectures for Parallel Processing*, 1997.

[3] Ewing Lusk. Programming with MPI on clusters. In *3rd IEEE International Conference on Cluster Computing (CLUSTER'01)*, October 2001.

[4] Niels Mache and Paul Levi. Parallel neural network training and cross validation on a cray t3e system and application to splice site prediction in human dna, 1995.

[5] Steffen Nissen. Implementation of a fast artificial neural network library (FANN), 2003.

[6] Bernhard Omer. Genetic algorithms for neural network training on transputers, 1995.

[7] D. Rumelhart, G. Hinton, and R. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing, MIT Press*, volume 1, 1986.

[8] N. N. R. Ranga Suri, Dipti Deodhare, and P. Nagabhushan. Parallel levenberg-marquardt-based neural network training on linux clusters - a case study. The Third Indian Conference on Computer Vision, Graphics, and Image Processing, 2002.