SoutheastCon 2001
Student Software Competition

Parallelizing Neural Networks

Nathan A. DeBardeleben, William M. Jones

March 30, 2001

1 Introduction

Welcome to SouthEastCon 2001! This years project will be the implementation of a parallel Feed-Forward Artificial Neural Network (FFANN) using first order gradient descent for the purpose of image recognition. This document contains information regarding the itinerary, software requirements, the parallel programming environment, evaluation guidelines, and background information on neural networks and parallel computing as it pertains to your project.

2 Itinerary

2.1 Friday

From 7:00PM to 10:00PM you will be meeting in Riggs Hall (3rd floor) to go over the project, review FFANNs, and get you familiar with the computing resources. *You will not be allowed to actually code during these three hours.* You will be allowed to take this document with you to review as you see fit during the night.

2.2 Saturday

From 8:00AM to 6:00PM you will be allowed to use the computing facility to code and debug your program. You will need to have your code in a "final" state, ready to be evaluated, as you will not be allowed to use the computing resources further. That night, the judges will review your code.

2.3 Sunday

From 8:00AM to 10:00AM we will be meeting in Riggs Hall (3rd floor) to judge the programs. You are invited to attend the judging. The judges will be available afterwards to discuss the project and answer questions.

3 Software Requirements

For this contest, you will being using the Message Passing Interface (MPI), specifically the implementation MPICH, to leverage the collective power of your Beowulf parallel computer. Your parallel machine is a cluster of Commodity-Off-The-Shelf (COTS) workstations. Your programming language will primarily be ANSI C. You will be provided with documentation online to facilitate your MPI programming needs. Additionally, you will have access to "*man*" pages to help with C function definitions.

The compilable program that you have been provided contains code to:

- Acquire command-line and file parameters for running the FFANN,
- Load training data, H, from image files,
- Time the simulation for evaluation purposes,
- Spawn master and slave processes, and
- Create random Gaussian variables with mean 0 and variance 1.

We suggest that you take this code as a foundation for creating your FFANN simulation. In particular, based on the mathematical equations that define FFANN functionality given in Section 6, you will be required to develop code for the master and slave nodes that correctly implements the neural network. This will include code for:

- Communication between the master and slave nodes using **MPI_Send** and **MPI_Recv**, and their variants,
- Data structures for network parameters,
- Algorithms for performing forward and backward propagation, and
- Calculations for error measures.

4 Evaluation Guidelines

The most important goal is to correctly implement the FFANN in parallel using MPI. After this goal is met, we will evaluate the groups based on these additional goals:

- Visualization of output layer of the FFANN,
- Well documented code,
- Standard software engineering practices, and
- Highly optimized code, manual and compiler-based (i.e. **runs faster!**).

5 Neural Network Overview

Artificial Neural Networks (ANN) can provide decision support that is both adaptive and capable of acquiring knowledge and solving problems. These neural networks are often implemented as a software simulation. As with any simulation, a set of well defined parameters describes the structure and flow of the system, and ANNs are no exception. Arguably the most fundamental network structure used in many applications is the Multilayer Feed-Forward ANN (FFANN). This network infrastructure is characterized by several features including number of layers, number of nodes in a given layer, interconnection between adjacent layers, and individual node parameters.* In addition to the network architecture itself, there is the significant burden of training the ANN. Classical learning algorithms such as first order gradient descent often converge rather slowly but are less computationally expensive than more sophisticated techniques. To add to the complexity of the situation, there is the issue of scalability of the algorithm as a function of layers and nodes. To accommodate larger networks, there is a need to decrease the time required to train the ANN. In order to speed-up the back-propagation (BP) phase of the training algorithm, you will use MPI to parallelize the FFANN. The goal of this contest is to decompose the sequential problem into parallel parts and then develop C code to implement the FFANN.

> ∗ **Some of these parameters are provided in the input parameter file, while others you will need to implement.**

6 Technical Background

6.1 Parallel Computing Paradigm

Before you can effectively parallelize the FFANN, you will need to analyze the structure of computations involved in solving the problem. As with any parallel implementation, the inherent granularity, or time between communications, of the sequential algorithm dictates the way in which the problem can effectively be decomposed into disjoint parallel parts. Another consideration for algorithm decomposition is the target parallel processing platform. Although there are several classifications, the vast majority of parallel systems can be divided into two, possibly overlapping, categories [3]: shared memory and distributed memory systems, depicted in Figure 1.* The salient distinguishing feature of these systems is the physical location of the memory. As the name implies, a distributed memory system typically has physically separated memories, with no global name-space. This forces a type of communication between related parallel tasks commonly referred to as explicit message passing. This paradigm is often best suited for solving problems with a large grain size. On the other hand, shared memory systems have localized and tightly coupled main memories that allow related parallel tasks to share a common global name-space. These systems are well suited for solving problems with a variety of grain sizes. Although these systems provide more flexibility, they are also more expensive and less scalable than their distributed memory counter-parts.

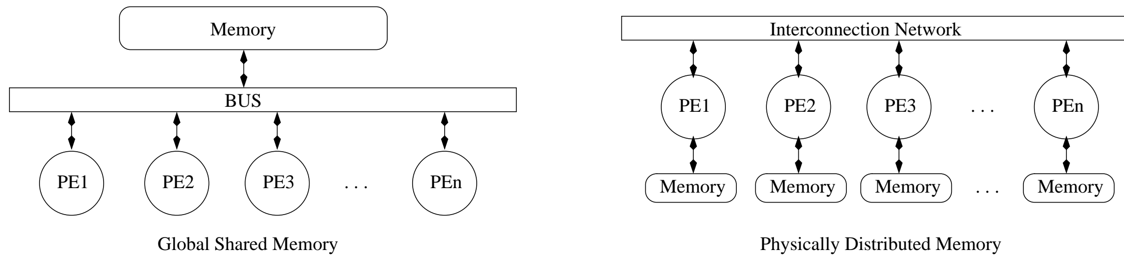

Figure 1: Two Parallel Processing System Architectures

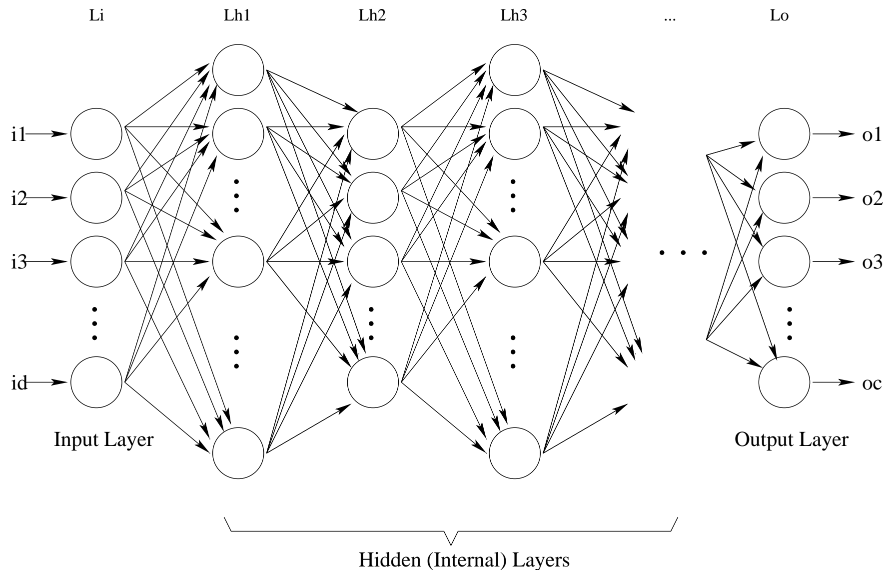

Figure 2: Generalized FFANN Architecture

$*$ **The Beowulf system you will be using is a distributed memory parallel system.**

6.2 FFANN Architecture

The basic building block of a neural network is the artificial neuron. From its inputs, these neurons form a weighted linear input combination, called the net activation net_i. This scalar value is then "squashed" by the activation function $f(net_i)$. The selection of this function depends on several factors including the training method. The choice of training algorithm constrains $f(net_i)$ to exhibit certain characteristics. For example, first-order gradient descent training requires $f(net_i)$ to be differentiable for all $(-\infty < net_i < \infty)$ as well as monotonically nondecreasing [4]. Collections of these neurons form a feed-forward network, i.e. a hierarchy of neurons organized in a series of two or more sets or *layers*. Figure 2 illustrates this layered concept.

Inside each neuron, there are weights w_{ij}, as well as other parameters in the activation function that can be adjusted to allow the neural network to "learn" the mapping from

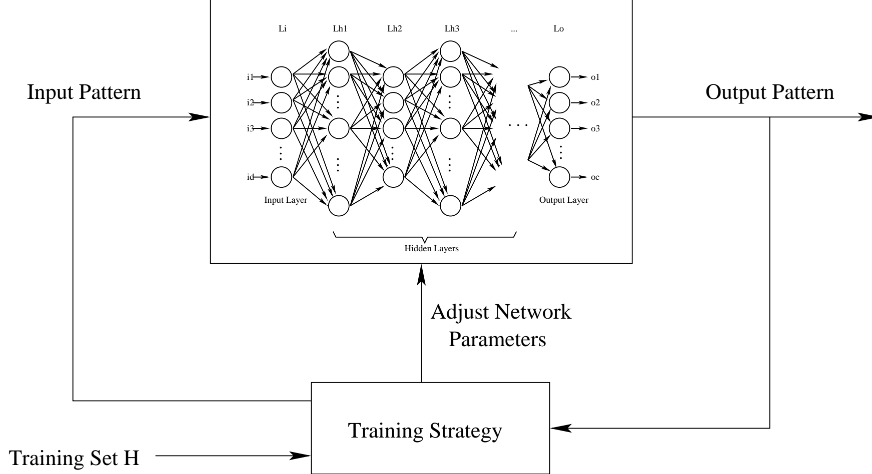

Figure 3: Generalized FF Training

the input space to the output space. The process of adjusting these parameters is formally known as training. The training set H normally includes a set of input/output pairs. The dimensionality of the input and output spaces are denoted here as d and c respectively. Using these I/O pairs, the training algorithm iterates over H and forms a, possibly synchronous, feed-back loop where network parameters are adjusted. Figure 3 illustrates this concept.

7 Methodology

7.1 Derivation of gradient descent

Before you begin the project, you may wish to review gradient descent training. The objective is to minimize the error obtained from *actual* outputs as compared to *desired* outputs. The objective function is therefore the total sum of squared error, formally expressed as

$$E = \sum_p E_p = \frac{1}{2} \sum_p \sum_j (o_{pj} - a_{pj})^2. \tag{1}$$

where, p corresponds of the training pattern, o_{pj} is the target output value of the jth unit in the output layer, and a_{pj} is the *actual* output value of the jth unit in the output layer. As mentioned earlier, the units form a weighted linear input combination (WLIC). This concept can be expressed formally by

$$net_j = \sum_i w_{ij} a_i. \tag{2}$$

Inside each unit, the net_j is "squashed" by some differentiable, increasing function $f(net_j)$. A function typically encountered in conjunction with FFANNs for this purpose is the sigmoid function

$$a_i = f(net_i) = \frac{1}{1 + e^{-net_i}}. \tag{3}$$

Since we wish to minimize error, the natural goal will be to move in the opposite direction of maximum increase on the error surface defined by equation 1. This means that the change in weight can be expressed using the gradient of the error surface. Applying the chain rule, we have

$$\Delta_p w_{ij} \propto -\frac{\partial E_p}{\partial w_{ij}} = -\frac{\partial E_p}{\partial net_{pj}} \frac{\partial net_{pj}}{\partial w_{ij}}. \tag{4}$$

From equation 2, we get

$$\frac{\partial net_{pj}}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \sum_k w_{kj} a_{pk} = a_{pi}. \tag{5}$$

Now define

$$\delta_{pj} = -\frac{\partial E_p}{\partial net_{pj}} = -\frac{\partial E_p}{\partial a_{pj}} \frac{\partial a_{pj}}{\partial net_{pj}}. \tag{6}$$

From equation 3, we get

$$\frac{\partial a_{pj}}{\partial net_{pj}} = f'(net_{pj}), \tag{7}$$

which is the first derivative of the squashing function f for the jth unit, evaluated at the point net_{pj}. This implies that if j is a unit in the output layer, using equation 1,

$$\frac{\partial E_p}{\partial a_{pj}} = -(o_{pj} - a_{pj}). \tag{8}$$

Substituting equations 7 and 8 into equation 6,

$$\delta_{pj} = (o_{pj} - a_{pj}) f'(net_{pj}). \tag{9}$$

> *Note:* The above derivation can be found in a number of resources including [2] as well as [4].

This derivation assumes of course that the error surface is quadratic, and that such a method will naturally lead to a global minimum. The error surface however, is not quadratic. This technique can, and often does, lead to sub-optimum local minima. One may initialize the weight parameters to random variables, so that each time the simulation is started, it will begin in a different location of weight space, hence a different location on the error surface.

This adjustment of weights during the BP phase follows layer by layer from the output layer to the input layer. A set of equations that follows from the basic derivation is referred to as the Generalized Delta Rules (GDR). They are summarized here [4]:

The objective function:

$$E^p = \frac{1}{2} \sum_p \sum_j (o_{pj} - a_{pj})^2. \tag{10}$$

The weight correction is

$$\Delta_p w_{ij} = \epsilon \delta_j^p a_i^p, \quad \text{where } \epsilon \text{ is the learning rate.} \tag{11}$$

For output units the $\delta' s$ are

$$\delta_j^p = (o_j^p - a_j^p) f_j'(net_j^p). \tag{12}$$

For internal units the $\delta's$ are

$$\delta_j^p = f_j'(net_j^p)\sum_n \delta_n^p w_{nj}, \quad \text{where } \delta_n^p\text{'s are from layer } (L_{k+1}). \tag{13}$$

Assuming a sigmoid activation (squashing) function,

$$f_j'(net_j^p) = a_j^p(1 - a_j^p). \tag{14}$$

8 Sequential Algorithm

Here we outline the basic sequential algorithm. It is slightly modified from the original in [2].

1. *Initialize.* Set number of units in input (N_0), output (M), and hidden ($N_k; k = 1..h$) layers, where h is the number of hidden layers. Set random values for all weights in network (except those in the input layer). A commonly used distribution is Gaussian with mean 0 and variance 1.

2. *Input.* Present input/output vectors $[i_0, ..., i_{N-1}]^T$, $[o_0, ...o_{M-1}]^T$.

3. *Calculate actual unit outputs.* For hidden layers ($x_i = i_i$ if unit i is an input unit):

$$a_i = f\left(\sum_{i=0}^{N_k-1} w_{ij}x_i\right) \qquad O \leq j \leq N_k \tag{15}$$

   For output layer:

$$a_i = f\left(\sum_{i=0}^{M_k-1} w_{ij}x_i\right) \qquad O \leq j \leq N_h \tag{16}$$

   These are the equations that formalize the forward propagation of the presented input vectors through the network.

   HINT: You can see here that the summation is forming the WLIC that is to be squashed by $f()$. In addition to the weights associated with the links from the previous layer, there should be an additional weight at each node (*with the exception of the input layer which only maps the input vector unmodified to the first hidden layer*). This additional weight is referred to as a *bias* weight. It is to be included in the WLIC summation. It is not scaled by any output from a prior layer, in other words, you can view it as scaled by 1 in the summation.

4. *Update δ-values.* If unit (j) is an output unit:

$$\delta_j^p = (o_j^p - a_j^p)f_j'(net_j^p). \tag{17}$$

If unit (j) is a hidden unit:

$$\delta_j^p = f_j'(net_j^p) \sum_n \delta_n^p w_{nj}, \quad \text{where } \delta_n^p\text{'s are from layer } (L_{k+1}). \tag{18}$$

5. *Update weights.* $w_{ij}(n+1) = w_{ij}(n) + \epsilon \delta_j a_i$, where $w_{ij}(n+1)$ is the weight in the link from unit i to unit j in the nth iteration and ϵ is the learning rate.

   HINT: The learning rate is a command-line parameter to the program.

6. *Repeat.* Goto Step 2.

9 Parallelization

With this foundation, you are ready to parallelize the overall procedure. During the training phase, the network parameters are adjusted according to the GDR equations and the procedure presented in 8. In your network architecture and BP kernel, the adjustable set of parameters includes the weight vector at each neuron. The weights can be updated as soon as one I/O pair is seen. This is referred to as training by pattern. By inspecting GDR equations, it becomes apparent that this updating can take place as late as after ALL patterns in H have been seen. This is accomplished by retaining the $\Delta_p w_{ij}'s$ for each pattern, then reducing these changes after all patterns are seen. This is referred to as training by epoch [4].

Given that the set of equations permit training by epoch, you can partition the training set H into disjoint subsets. Each of these subsets can be processed independently and therefore in parallel [1]. After each of these subsets is processed, the $\Delta w_{ij}'s$ associated with each subset can be combined. This would be considered one iteration of the training phase. The new weights can then be distributed to the processing elements for the next iteration. Figure 4 illustrates this concept.

HINT: The code template you have been provided includes the basic C code to load the image training data. The inputs to the FFANN will be vectors of row-concatenated images of people. The output of the FFANN will be a "selector" that indicates who the person is. The inputs and outputs form the traning set. Please refer to the *slave* code for more information.

10 Conclusion

We appreciate you attending the Student Software Competition portion of SoutheastCon 2001. **Good luck!**

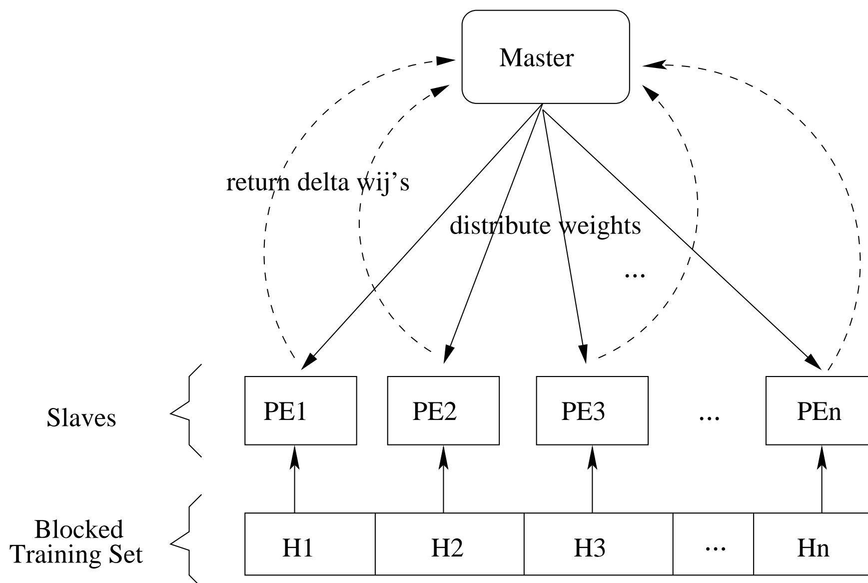

Figure 4: Problem Decomposition

References

[1] Douglas Aberdeen, Jonathan Baxter, and Robert Edwards. 0.92/MFlops/s, Ultra-Large-Scale Neural-Network Training on a PIII Cluster. In *Proceedings of the IEEE/ACM SC2000 Conference*. IEEE Computer Society, November 2000.

[2] Selwyn Piramuthu et. al. Learning Algorithms for Neural-Net Decision Support. In *ORSA Journal on Computing*, volume 5, 361-373, Fall 1993. Operations Research Society of America.

[3] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing*. Benjamin/Cummings, 1994.

[4] Robert J. Schalkoff. *Artificial Neural Networks*. McGraw-Hill, Inc., 1997.