To my parents and friends for support.
Acknowledgments

I would like to express my appreciation to Dr. Patricia A. Nava, who supervised this thesis project and provided guidance, for making the patience and efforts to become a success. Also I would like to thank Dr. David H. Williams and Dr. Helmut Knaust for being the members of the committee and being interested in this thesis.

Also, I thank the following people who provided the access to the system and support in the ECE UNIX Lab: Mr. Sergio Zapata, Mr. Leopoldo Hernandez, Mr. Ricardo Mesta, Mr. Nito Gumataotao, and Mr. Hugo Garcia. I would like to appreciate Mr. Manuel Silveyra for extended access and support during the testing phase. Additionally, I extend my appreciation to Mr. Shiu Hung Chan, who is known as George among us, for having insights of my project and providing various support including producing the graphics in this thesis writing. Not to forget, I would like to acknowledge Mr. and Mrs. Loya for giving me a lot of support both technically and personally.

Finally and most significantly, I would like to thank my parents for unconditional support and having me in higher education.
Abstract

The behaviors of a system based on Artificial Neural Network (ANN) are unlike conventional computing machines which require specific instructions for solving problems. Instead, the neural network system can solve a problem “by experience and learning” the input-output patterns provided by the user. This can have incredible advantages in solving highly complex nonlinear problems efficiently. However, the learning process of neural networks requires a massive number of repetitions in presenting the sample patterns until the neural network system can function correctly. In computing for multilayer backpropagation neural networks, a lengthy amount of time is required, especially during the learning process.

One approach to overcome this issue is to distribute and parallelize these computing tasks. Such concurrent processing gives the hope of shorter learning time of neural networks. In this thesis, we attempt to implement concurrent processing in a neural network application by designing software running on a computer host with multiple processors. In UNIX operating system platforms, Pthreads (POSIX threads) allow access of multiple processors by an application program, to achieve true parallel processing. Suggested future work involves the implementation of these functions using client-server processing, in order to provide opportunities for any networked workstation to access the parallel-execution server.

\[^{1}\text{UNIX is a trademark of AT & T Bell Laboratories.}\]
# Contents

Acknowledgments iv
Abstract v
List of Figures x

1 Introduction 1
  1.1 Background and Motivation 1
  1.2 Objective 2
  1.3 Road Map 3

2 Artificial Neural Networks 4
  2.1 Background 4
  2.2 Biological Neural Networks 5
  2.3 Artificial Neural Networks 7
    2.3.1 The McCulloch-Pitts Model of Neuron 8
    2.3.2 The Perceptron 9
    2.3.3 Artificial Neuron with Continuous Characteristics 10
    2.3.4 Single-Layer Network 12
    2.3.5 Multilayer Network 13
    2.3.6 Learning Processes 13
  2.4 Backpropagation Neural Networks 14
    2.4.1 Linear Separability and the XOR Problem 15
    2.4.2 Architecture of Backpropagation Networks 16
    2.4.3 Backpropagation Processing Unit 17
    2.4.4 Backpropagation Learning Algorithm 18
    2.4.5 Local Minimum Problem 23
    2.4.6 Generalization 23

3 Concurrent Processing 25
  3.1 Multiprogramming and Multitasking 25
    3.1.1 Processes 26
    3.1.2 Process Table 27
    3.1.3 Process State 28
    3.1.4 Running Processes 29
    3.1.5 Interprocess Communication 30
  3.2 Concurrency 31
    3.2.1 Threads 32
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.2</td>
<td>Processes vs. Threads</td>
<td>33</td>
</tr>
<tr>
<td>3.2.3</td>
<td>User-level and Kernel-level Threads</td>
<td>34</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Multi-threaded Applications</td>
<td>34</td>
</tr>
<tr>
<td>3.3</td>
<td>POSIX Threads</td>
<td>35</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Thread Creation</td>
<td>35</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Thread Termination</td>
<td>36</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Thread Identification</td>
<td>37</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Thread Attributes</td>
<td>37</td>
</tr>
<tr>
<td>3.4</td>
<td>Thread Synchronization</td>
<td>38</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Mutexes</td>
<td>38</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Condition Variables</td>
<td>40</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Other Synchronization Primitives</td>
<td>43</td>
</tr>
<tr>
<td>4</td>
<td>Parallel Neural Network Software Package</td>
<td>45</td>
</tr>
<tr>
<td>4.1</td>
<td>Parallel Neural Network Architecture</td>
<td>45</td>
</tr>
<tr>
<td>4.2</td>
<td>Parallel Neural Network Functions</td>
<td>47</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Configuration and Data Structure</td>
<td>47</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Weight Initialization Routine</td>
<td>48</td>
</tr>
<tr>
<td>4.2.3</td>
<td>run_pnnet() Routine</td>
<td>49</td>
</tr>
<tr>
<td>4.3</td>
<td>The Testing Module: Letter Recognition</td>
<td>52</td>
</tr>
<tr>
<td>4.3.1</td>
<td>The Motif Graphical User Interface</td>
<td>53</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Pre-Training Process</td>
<td>54</td>
</tr>
<tr>
<td>4.3.3</td>
<td>The Core Neural Network Processing</td>
<td>54</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Post-Training Process</td>
<td>55</td>
</tr>
<tr>
<td>4.4</td>
<td>Testing Results and Analysis</td>
<td>56</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Letter Recognition Performance</td>
<td>57</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Training Speed Performance</td>
<td>57</td>
</tr>
<tr>
<td>5</td>
<td>Results and Conclusions</td>
<td>65</td>
</tr>
<tr>
<td>5.1</td>
<td>Final Observations</td>
<td>65</td>
</tr>
<tr>
<td>5.2</td>
<td>Future Work</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td><em>References</em></td>
<td>67</td>
</tr>
<tr>
<td></td>
<td><em>Appendix A Source Code</em></td>
<td>70</td>
</tr>
<tr>
<td>A.1</td>
<td>Include File: pnnet.h</td>
<td>70</td>
</tr>
<tr>
<td>A.2</td>
<td>Initializing Weights: weight_init.c</td>
<td>72</td>
</tr>
<tr>
<td>A.3</td>
<td>Parallel Neural Network Function: pnnet.c</td>
<td>73</td>
</tr>
<tr>
<td>A.4</td>
<td>Other Low-level Functions</td>
<td>84</td>
</tr>
<tr>
<td>A.4.1</td>
<td>Forward Propagation: fp_layer.c</td>
<td>84</td>
</tr>
<tr>
<td>A.4.2</td>
<td>Neural Calculation: perceptron.c</td>
<td>86</td>
</tr>
<tr>
<td>A.4.3</td>
<td>Sum-of-Product Function: sop.c</td>
<td>87</td>
</tr>
<tr>
<td>A.4.4</td>
<td>Sigmoid Function: sigmoid.c</td>
<td>88</td>
</tr>
<tr>
<td>A.4.5</td>
<td>Backpropagation Routines: backprop.c</td>
<td>89</td>
</tr>
<tr>
<td></td>
<td><em>Curriculum Vitae</em></td>
<td>92</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Three Stages of Biological Neural System ........................................ 4
2.2 A Biological Neuron ........................................................................... 4
2.3 Linear Threshold Function .................................................................. 5
2.4 Symbolic Illustration of Linear Threshold Gate ................................... 6
2.5 More General Neural Model ............................................................... 7
2.6 Sigmoid Function .............................................................................. 7
2.7 Single Layer Neural Network ............................................................. 8
2.8 Multiple Layer Neural Network ........................................................ 8
2.9 Linearly Separable Pattern .................................................................. 10
2.10 Exclusive-OR Function ...................................................................... 10
2.11 Suggested Network for Solving XOR Problem .................................... 11
2.12 Backpropagation Neural Network ...................................................... 12

3.1 (a) Four Programs that are Loaded in Memory (b) Conceptual model of Four Independent Processes (c) Timing Model that Shows only One Program is Active at a Time .......................... 17
3.2 The State Diagram of a Process .......................................................... 18
3.3 System Calls to Control Processes ...................................................... 19
3.4 Interprocess Communication between Two Processes .......................... 20
3.5 Two Models of Concurrent Processing: (a) Three Processes, and (b) Three Threads ................................................................. 21
3.6 Basic POSIX Thread Management Functions ....................................... 22
3.7 Basic POSIX Mutex Functions ............................................................. 24
3.8 A Thread Accessing a Shared Variable with Mutex Lock ..................... 25
3.9 A Thread Using Mutex for Waiting .................................................... 26
3.10 Basic POSIX Condition Variable Functions ........................................ 26
3.11 A Thread Using Condition Variable for Waiting ................................. 27

4.1 A Neural Network Sliced by Neurons in Each Layer .............................. 30
4.2 Graphical Description of Data Structure for Implementation ................ 31
4.3 Weight Initialization Function Call ...................................................... 31
4.4 Parallel Neural Network Function Calls .............................................. 32
4.5 Flowchart of Thread Synchronization in run_pnet() ............................. 34
4.6 The Main Window of the User Interface .............................................. 34
4.7 The Dialog that Prompts for the Filename to Load ............................... 35
4.8 Neural Network Testing Output Display ............................................ 36
4.9 List of Measurements on kawaguc.mindspring.com (1 CPU, Linux) .... 39
4.10 List of Measurements with gettimeofday() on kawaguc.mindspring.com (1 CPU, Linux) ............................................................. 39
4.11 List of Measurements on eultra2.ece.utep.edu (1 CPU, Solaris) ............ 40
4.12 Plots of Measurements with gettimeofday() on eultra2.ece.utep.edu (1 CPU, Solaris) ................................................................. 40
4.13 List of Measurements on electra.ece.utep.edu (2 CPU’s, Solaris) ......... 41
Chapter 1

Introduction

1.1 Background and Motivation

Studies in Artificial Neural Networks (ANN), or commonly known as Neural Networks, have been attractive for applications in complex function modeling and classifications due to the fact that Neural Networks have very different computing approaches from traditional computing machines. The Neural Network System has an ability to construct the rules of input-output mapping by itself. Thus, the designer of the system does not need to know the internal structure and instructions of the system, or the functional rules like traditional systems. Instead, the Neural Network system requires the feed-in’s of input-output patterns to “learn” before the system can function correctly.

The main disadvantage of the Neural Network is that the time for the system to learn the input-output patterns correctly is very lengthy. This fact has been discouraging the implementation of Artificial Neural Networks to applications such as signal and image recognition, and other predicting systems.

One solution is to take the advantage of the architecture of Neural Networks. The Neural Networks are often known as massively parallel processors. For executing the simulation of the Neural Networks, not all the computing tasks are required to be performed in order. Several computing tasks can be done concurrently by partitioning parts of neural network into different tasks. Recent efforts have been accomplished by distributing computing tasks onto multiple computer hosts over the local area network (LAN) [Vel98].

In this thesis, the new form of parallel computing is introduced. Until recently, a standard form of developing multi-threaded application on UNIX platforms did not exist. POSIX.1c is such a standard developed and formed by Institute of Electrical and Electronic Engineers (IEEE) in 1996 [Ste98]. A multi-threaded application can access as many processors in the system as available, and achieves true parallelism. Our foremost interest is to observe the degree of improvements in neural network computation as we develop the neural network function with multiple threads.

1.2 Objective

Our primary objective is to develop a software package that simulates artificial neural networks by utilizing threads. First, the software should be portable across various UNIX platforms, including Solaris®, Linux, and others. Second, the multi-threaded application should be scalable. Third, the final form of the software should be somewhat reusable so that the developers in higher level of neural network application can link to this. Specifically, the software will be packaged into an Application Programming Interface.

* Solaris is a trademark of Sun Microsystems, Inc.
(API), and linkable to any higher-level applications. For this reason, the API should be very configurable in size and dimension of the neural networks, data, and other factors.

Once the software is developed, the testing is required for verifying the performance. We will introduce a testing application as a small digital image pattern recognition. First, we will verify the performance and behavior of neural network itself. Then, more importantly, we will measure the performance in execution time during the learning phase of the neural network to be simulated. In the verification of performance in time, we will vary some parameters under the certain fixed system environment, and will obtain the optimal set of parameters.

1.3 Road Map

This thesis is organized in five chapters. Chapter 2 and Chapter 3 will discuss the theoretical background. More specifically, Chapter 2 will deal with concepts of Artificial Neural Networks. We will begin with basic background concepts of Neural Networks, then approach more specific architecture called Backpropagation Neural Networks. In Chapter 3, the concepts of concurrent processing are discussed. We attempt to move from traditional multiprogramming / timesharing systems into more modern concurrent processing with threads. In Chapter 4, the actual software modeling, testing, and verification will be presented. Finally, Chapter 5 will summarize the outcome of testing results and suggested future works. Additionally, Appendix A will provide the source code of the core routines of the software, and the reader is strongly encouraged to refer to this as the discussion of software modeling progresses.
Chapter 2

Artificial Neural Networks

2.1 Background

The motivation of studies in neural networks lies in the flexibility and power of information processing that conventional computing machines do not have. The history of neural networks comes from attempts of modeling a system whose performance is analogous to the most basic functions of human brains. Although most computers can process faster and more precisely than human brains, people have ability to obtain experience then make more sensible decisions [Zur92]. Similar to the fact that the human brain generalizes the rules, the neural network system can “learn by examples and experience” and perform variety of nonlinear functions that are difficult to describe mathematically [Tay96].

Artificial neural networks are a narrow-sensed abstraction of the human brain, thus the organization of the artificial neural system is very similar to the one of biological neurons. The comprehensive understanding of biological neurons is not complete; however, the basic functionality that contributes to the learning ability of a system is implemented in artificial neural networks. The fundamental element, an artificial neuron, is a model based on known behavior of biological neurons that exhibit most of the characteristics of human brains that we are interested in [Vel98]. This is the most significant difference from conventional computers, which have internal fixed instructions to perform specific functions.

Artificial neural networks can be also described as highly parallel distributed computing models. The fundamental processing units, neurons, are highly connected with strengths, which are dynamically changed during the system’s learning process.

The discussion in following sections approach the engineer’s perspective of understanding the artificial neural networks. Though artificial neural networks are not an exact copy of biological human brain, it is important to begin with understanding fundamental concepts of biological neurons and the human brain.

2.2 Biological Neural Networks

The neural system of the human body consists of three stages: receptors, a neural network, and effectors. The receptors receive the stimuli either internally or from the external world, then pass the information into the neurons in a form of electrical impulses. The neural network then processes the inputs then makes proper decision of outputs. Finally, the effectors translate electrical impulses from the neural network into responses to the outside environment. Figure 2.1 shows the bidirectional communication between stages for feedback [Arb87].
The fundamental element of the neural network is called a neuron. As shown in figure 2.2, a neuron mainly consists of three parts: dendrites, soma, and axon. Dendrites are the tree-like structure that receives the signal from surrounding neurons, where each line is connected to one neuron. Axon is a thin cylinder that transmits the signal from one neuron to others. At the end of axon, the contact to the dendrites is made through a synapse. The inter-neuronal signal at the synapse is usually chemical diffusion but sometimes electrical impulses. A neuron fires an electrical impulse only if certain condition is met [Zur92].

The incoming impulse signal from each synapse to the neuron is either excitatory or inhibitory, which means helping or hindering firing. The condition of causing firing is that the excitatory signal should exceed the inhibitory signal by a certain amount in a short period of time, called the period of latent summation. As we assign a weight to each incoming impulse signal, the excitatory signal has positive weight and the inhibitory signal has negative weight. This way, we can say, “A neuron fires only if the total weight of the synapses that receive impulses in the period of latent summation exceeds the threshold.” [Arb87].

2.3 Artificial Neural Networks

The structure of artificial neural networks was based on the present understanding of biological neural systems. The computation is achieved by dense interconnection of simple processing units. To describe the attributes of computing, the artificial neural networks go by many names such as connectionist models, parallel distributed processors, or self-organizing system. With such features, an artificial neural system has great potential in performing applications such as speech and image recognition where intense computation can be done in parallel and the computational elements are connected by weighted links.

The artificial neuron, the most fundamental computational unit, is modeled based on the basic property of a biological neuron. This type of processing unit performs in two stages: weighted summation and some
type of nonlinear function. It accepts a set of inputs to generate the weighted sum, then passes the result to the nonlinear function to make an output.

Unlike conventional computing systems, which have fixed instructions to perform specific computations, the artificial neural network needs to be taught and trained to function correctly. The advantage is that the neural system can learn new input-output patterns and adjust the system parameters. Such learning can eliminate specifying instructions to be executed for computations. Instead, users simply supply appropriate sample input-output patterns to the network [Zur92].

The model of the entire artificial neural network is determined by the network topology, type of neural model, and learning rules. These are the main interests in designing artificial neural networks.

### 2.3.1 The McCulloch-Pitts Model of Neuron

The early model of an artificial neuron is introduced by Warren McCulloch and Walter Pitts in 1943. The McCulloch-Pitts neural model is also known as linear threshold gate. It is a neuron of a set of inputs and one output. The linear threshold gate simply classifies the set of inputs into two different classes. Thus the output is binary. Such a function can be described mathematically using these equations:

\[
\text{Sum} = \sum_{i=1}^{N} I_i W_i, \quad (2.1)
\]

\[
y = f(\text{Sum}). \quad (2.2)
\]

\(W_1, W_2, W_3, \ldots, W_m\) are weight values normalized in the range of either \((0,1)\) or \((-1,1)\) and associated with each input line, \(\text{Sum}\) is the weighted sum, and \(T\) is a threshold constant. The function \(f\) is a linear step function at threshold \(T\) as shown in figure 2.3. The symbolic representation of the linear threshold gate is shown in figure 2.4 [Has95].

The McCulloch-Pitts model of a neuron is simple yet has substantial computing potential. It also has a precise mathematical definition. However, this model is so simplistic that it only generates a binary output and also the weight and threshold values are fixed. The neural computing algorithm has diverse features for various applications [Zur92]. Thus, we need to obtain the neural model with more flexible computational features.
2.3.2 The Perceptron

In late 1950s, Frank Rosenblatt introduced a network composed of the units that were enhanced version of McCulloch-Pitts Threshold Logic Unit (TLU) model. Rosenblatt’s model of neuron, a perceptron, was the result of merger between two concepts from the 1940s, McCulloch-Pitts model of an artificial neuron and Hebbian learning rule of adjusting weights [BL96]. In addition to the variable weight values, the perceptron model added an extra input that represents bias. Thus, the modified equation from (2.1) is now as follows:

\[
\text{Sum} = \sum_{i=1}^{N} I_i W_i + b, \quad (2.3)
\]

where \(b\) represents the bias value.

2.3.3 Artificial Neuron with Continuous Characteristics

Based on the McCulloch-Pitts model described previously, the general form an artificial neuron can be described in two stages shown in figure 2.5. In the first stage, the linear combination of inputs is calculated. Each value of input array is associated with its weight value, which is normally between 0 and 1. Also, the summation function often takes an extra input value \(\theta\) with weight value of 1 to represent threshold or bias of a neuron. The summation function will be then performed as,

\[
x = \sum_{i=1}^{N} A_i W_i + \theta, \quad (2.4)
\]

The sum-of-product value is then passed into the second stage to perform the activation function which generates the output from the neuron. The activation function “squashes” the amplitude the output in the range of \([0, 1]\), or alternately \([-1, 1]\) [Hay99]. The behavior of the activation function will describe the characteristics of an artificial neuron model.

The signals generated by actual biological neurons are the action-potential spikes, and the biological neurons are sending the signal in patterns of spikes rather than simple absence or presence of single spike pulse. For example, the signal could be a continuous stream of pulses with various frequencies. With this kind of observation, we should consider a signal to be continuous with bounded range. The linear threshold function should be “softened” [BL96].
Figure 2.5: More General Neural Model

One convenient form of such “semi-linear” function is the logistic sigmoid function, or in short, sigmoid function as shown in figure 2.6. As the input $x$ tends to large positive value, the output value $y$ approaches to 1. Similarly, the output gets close to 0 as $x$ goes negative. However, the output value is neither close to 0 nor 1 near the threshold point. This function is expressed mathematically as follows:

$$y = \frac{1}{1 + e^{\text{exp}(-x)}}.$$  

(2.5)

Additionally, the sigmoid function describes the “closeness” to the threshold point by the slope. As $x$ approaches to $-\infty$ or $\infty$, the slope is zero; the slope increases as $x$ approaches to 0. This characteristic often plays an important role in learning of neural networks.

2.3.4 Single-Layer Network

By connecting multiple neurons, the true computing power of the neural networks comes, though even a single neuron can perform substantial level of computation [Ler91]. The most common structure of
connecting neurons into a network is by layers. The simplest form of layered network is shown in figure 2.7. The shaded nodes on the left are in the so-called *input layer*. The input layer neurons are to only pass and distribute the inputs and perform no computation. Thus, the only true layer of neurons is the one on the right. Each of the inputs $x_1, x_2, x_3, ..., x_N$ is connected to every artificial neuron in the output layer through the connection weight. Since every value of outputs $y_1, y_2, y_3, ..., y_N$ is calculated from the same set of input values, each output is varied based on the connection weights. Although the presented network is fully connected, the true biological neural network may not have all possible connections – the weight value of zero can be represented as “no connection”.

### 2.3.5 Multilayer Network

To achieve higher level of computational capabilities, a more complex structure of neural network is required. Figure 2.8 shows the *multilayer neural network* which distinguishes itself from the single-layer network by having one or more *hidden layers*. In this multilayer structure, the input nodes pass the information to the units in the first hidden layer, then the outputs from the first hidden layer are passed to the next layer, and so on.

Multilayer network can be also viewed as cascading of groups of single-layer networks. The level of complexity in computing can be seen by the fact that many single-layer networks are combined into this multilayer network. The designer of an artificial neural network should consider how many hidden layers...
are required, depending on complexity in desired computation.

2.3.6 Learning Processes

Perhaps, the most primary significance of a neural network is the ability to learn the incoming information and to improve the performance of processing information. The term learning refers to many concepts by various viewpoints, and it is difficult to agree on a precise definition of the term. In neural networks, we define learning as the following sequence of events: [Hay99]

1. Stimulation by an environment in which the network is embedded.
2. Changes in free parameters of the network as the result of stimulation.
3. Responses in a new way to the environment for improved performance.

A Learning algorithm is a prescribed set of well-defined rules for learning of a neural network. There are many types of learning algorithms; the common goal of learning is the adjustment of connection weights.

There are two classes of learning: supervised and unsupervised learning. Supervised learning requires an external source of information in order to adjust the network. On the other hand, in unsupervised learning, there is no external agent that oversees the process of learning. Instead, the network is adjusted through internal monitoring of performance. In this thesis, we mainly deal with supervised learning since understanding the backpropagation network, which focuses on supervised learning, is our goal.

2.4 Backpropagation Neural Networks

Backpropagation neural networks employ one of the most popular neural network learning algorithms, the Backpropagation (BP) algorithm. It has been used successfully for wide variety of applications, such as speech or voice recognition, image pattern recognition, medical diagnosis, and automatic controls. One of the most striking early applications was NETTalk by T. J. Sejnowski and C. R. Rosenberg in 1986. The NETTalk was able to learn the rules of phonetics, then the system produced a sound by reading from the sequence of given letters, with a behavior of a child learning to read aloud [Day90].

Backpropagation made a tremendous step forward from the single-layer perceptron network. With a more sophisticated learning rule, backpropagation networks overcome the limitations that single-layer networks have. Backpropagation is also the most suitable learning method for multilayer networks. Perhaps, the reason why the backpropagation made the major turning point is because the learning rule has a solid mathematical foundation and it is practical [Ler91].

2.4.1 Linear Separability and the XOR Problem

Consider two-input patterns \((X_1, X_2)\) being classified into two classes as shown in figure 2.9. Each point with either symbol of \(x\) or \(o\) represents a pattern with a set of values \((X_1, X_2)\). Each pattern is classified into one of two classes. Notice that these classes can be separated with a single line \(L\). They are known as linearly separable patterns. Linear separability refers to the fact that classes of patterns with \(n\)-dimensional vector \(x = (x_1, x_2, \ldots, x_n)\) can be separated with a single decision surface. In the case above, the line \(L\) represents the decision surface.

The processing unit of a single-layer perceptron network is able to categorize a set of patterns into two classes as the linear threshold function defines their linear separability. Conversely, the two classes must be linearly separable in order for the perceptron network to function correctly [Hay99]. Indeed, this is the main limitation of a single-layer perceptron network.
The most classic example of linearly inseparable pattern is a logical exclusive-OR (XOR) function. Shown in figure 2.10 is the illustration of XOR function that two classes, 0 for black dot and 1 for white dot, cannot be separated with a single line. The solution seems that patterns of \((X_1, X_2)\) can be logically classified with two lines \(L_1\) and \(L_2\) [BJ91].

2.4.2 Architecture of Backpropagation Networks

Our initial approach to solving linearly inseparable patterns of XOR function is to have multiple stages of perceptron networks. Each stage would set up one decision surface or a line that separate patterns. Based on the classification determined by the previous stage, the current stage can form sub-classifications. Figure 2.11 shows the network with two layers of perceptron units to solve the XOR problem [BJ91]. Node 1 detects the pattern for \((1,0)\), while node 2 detects the pattern for \((0,1)\). Combined, with these first-layer classifications, node 3 is allowed to classify XOR input patterns correctly [BJ91].

Generalizing the XOR case discussed above, the multilayer feedforward network seems to be the feasible network architecture for backpropagation. However, we still have to take into account how the learning is processed. Unfortunately, with multilayer perceptrons, the nodes in the output layer do not have access to input information in order to adjust connection weights. Because the actual input signals are masked off by the intermediate layers of threshold perceptrons, there is no indication of how close they are to the threshold point. For this reason, we need to modify a hard-limiting threshold function of the perceptron into a nonlinear function for backpropagation learning.
2.4.3 Backpropagation Processing Unit

The backpropagation processing unit should be in the form modified from a linear perceptron so that the activation function is nonlinear and smoothed out at the threshold point. The suggested form of the activation function is the sigmoid function as mentioned previously. With sigmoid function, we can obtain not only output from the neuron but also information about how close we are to the threshold point using the slope of the sigmoid function. Mathematically, we can derive the slope from the equation (2.5) as follows:

\[
\frac{d}{dx}f(x) = \frac{\exp(-x)}{(1 + \exp(-x))^2} \quad (2.6)
\]

\[
= \frac{1}{1 + \exp(-x)} \cdot \frac{\exp(-x)}{1 + \exp(-x)} \quad (2.7)
\]

\[
= \frac{1}{1 + \exp(-x)} \left[ 1 - \frac{1}{1 + \exp(-1)} \right] \quad (2.8)
\]

\[
= f(x) [1 - f(x)]. \quad (2.9)
\]

This will be the key information for the weight adjustments in the forthcoming discussions.

2.4.4 Backpropagation Learning Algorithm

The backpropagation algorithm trains a given feed-forward multilayer neural network for a given set of input patterns with known classifications. When each entry of the sample set is presented to the network, the network examines its output response to the sample input pattern. The output response is then compared to the known and desired output and the error value is calculated. Based on the error, the connection weights are adjusted. The backpropagation algorithm is based on Widrow-Hoff delta learning rule in which the weight adjustment is done through mean square error of the output response to the sample input [Vel98]. The set of these sample patterns are repeatedly presented to the network until the error value is minimized.

Refer to the figure 2.12 that illustrates the backpropagation multilayer network with \( M \) layers. \( N_j \) represents the number of neurons in \( j \)th layer. Here, the network is presented the \( p \)th pattern of training sample set with \( N_0 \)-dimensional input \( X_{p1}, X_{p2}, ..., X_{pN_0} \) and \( N_M \)-dimensional known output response \( T_{p1}, T_{p2}, ..., T_{pN_M} \). The actual response to the input pattern by the network is represented as \( O_{p1}, O_{p2}, ..., O_{pN_M} \). Let \( Y_{ji} \) be the output from the \( i \)th neuron in layer \( j \) for \( p \)th pattern; \( W_{jikh} \) be the connection weight from \( k \)th neuron in layer \((j-1)\) to \( i \)th neuron in layer \( j \); and \( \delta_{ji} \) be the error value associated with the \( i \)th neuron in layer \( j \).
The $i$th neuron in layer $j$

$Y_{(j-1)1}$ $W_{ji1}$ $\delta_{ji}$ $Y_{ji}$

$Y_{(j-1)2}$ $W_{ji2}$ $Y_{(j-1)N_{j-1}}$ $W_{jiN_{j-1}}$

$N_j = \#$ neurons

$X_{p1}$ $X_{p2}$ $X_{pN}$

$N_0$ $N_1$ $N_M$

$j = \text{layer } # \quad j = 0 \quad j = 1 \quad j = M$

$O_{p1}$ $O_{p2}$ $O_{pN}$

$T_{p1}$ $T_{p2}$ $T_{pN}$

Figure 2.12: Backpropagation Neural Network
The following is the outline of the backpropagation learning algorithm [BJ91]:

1. Initialize connection weights into small random values.

2. Present the \( p \)th sample input vector of pattern \( \mathbf{X}_p = (X_{p1}, X_{p2}, \ldots, X_{pN_0}) \) and the corresponding output target \( \mathbf{T}_p = (T_{p1}, T_{p2}, \ldots, T_{pN_M}) \) to the network.

3. Pass the input values to the first layer, layer 1. For every input node \( i \) in layer 0, perform:
   \[ Y_{0i} = X_{pi}. \]

4. For every neuron \( i \) in every layer \( j = 1, 2, \ldots, M \), from input to output layer, find the output from the neuron:
   \[ Y_{ji} = f \left( \sum_{k=1}^{N_{j-1}} Y_{(j-1)k} W_{jik} \right), \]
   where
   \[ f(x) = \frac{1}{1 + e^{xp(-x)}}. \]

5. Obtain output values. For every output node \( i \) in layer \( M \), perform:
   \[ O_{pi} = Y_{Mi}. \]

6. Calculate error value \( \delta_{ji} \) for every neuron \( i \) in every layer in backward order \( j = M, M - 1, \ldots, 2, 1 \), from output to input layer, followed by weight adjustments. For the output layer, the error value is:
   \[ \delta_{Mi} = Y_{Mi}(1 - Y_{Mi})(T_{pi} - Y_{Mi}), \] (2.10)
   and for hidden layers:
   \[ \delta_{ji} = Y_{ji}(1 - Y_{ji}) \sum_{k=1}^{N_{j+1}} \delta_{(j+1)k} W_{(j+1)ki}. \] (2.11)

The weight adjustment can be done for every connection from neuron \( k \) in layer \( (i - 1) \) to every neuron \( i \) in every layer \( i \):
   \[ W_{jik}^+ = W_{jik} + \beta \delta_{ji} Y_{ji}, \] (2.12)
   where \( \beta \) represents weight adjustment factor normalized between 0 and 1. The derivation of the equations above will be discussed soon.

The actions in steps 2 through 6 will be repeated for every training sample pattern \( p \), and repeated for these sets until the root mean square (RMS) of output errors is minimized.

We now attempt to derive the error and weight adjustment equations shown above. Let’s begin with the Root Mean Square (RMS) of the errors in the output layer defined as:
   \[ E_p = \frac{1}{2} \sum_{j=1}^{N_M} (T_{pj} - O_{pj})^2, \] (2.13)
for the \( p \)th sample pattern. In generalized delta rule [BJ91, Day90, Gur97], the error value \( \delta_{ji} \) associated with the \( i \)th neuron in layer \( j \) is the rate of change in the RMS error \( E_p \) respect to the sum-of-product of the neuron:

\[
\delta_{ji} = -\frac{\partial E_p}{\partial \text{net}_{ji}},
\]  

(2.14)

where \( \text{net}_{ji} \) represents the sum-of-product value. With the chain rule, we can obtain the rate of change in the RMS error \( \hat{E}_p \) in response to weight change:

\[
\frac{\partial E_p}{\partial W_{jik}} = \frac{\partial E_p}{\partial \text{net}_{ji}} \frac{\partial \text{net}_{ji}}{\partial W_{jik}}
\]  

(2.15)

\[
= -\delta_{ji} \frac{\partial}{\partial W_{jik}} [Y_{(j-1)0} W_{(j-1)i0} + \ldots + Y_{(j-1)k} W_{(j-1)ik} + \ldots]
\]  

(2.16)

\[
= -\delta_{ji} \frac{\partial}{\partial W_{jik}} Y_{(j-1)k} W_{(j-1)ik}
\]  

(2.17)

\[
= -\delta_{ji} Y_{(j-1)k}.
\]  

(2.18)

We can say that the weight change is proportional to this value above [BJ91].

\[
\Delta W_{jik} = \beta \delta_{ji} Y_{(j-1)k},
\]  

(2.19)

where \( \beta \) is a constant.

Thus, weight change can be performed as:

\[
W_{jik}^+ = W_{jik} + \Delta W_{jik},
\]  

(2.20)

which should match equation (2.12).

Now let’s get back to the equation (2.14) to find an error value associate with the neuron. Again, using the chain rule, we get:

\[
\delta_{ji} = -\frac{\partial E_p}{\partial Y_{ji}} \frac{\partial Y_{ji}}{\partial \text{net}_{ji}}.
\]  

(2.21)

For output layer, \( j = M \) and \( Y_{Mi} = O_{pi} \). Thus,

\[
\delta_{Mi} = -\frac{\partial E_p}{\partial O_{pi}} \frac{\partial Y_{Mi}}{\partial \text{net}_{Mi}}
\]  

(2.22)

\[
= -\frac{\partial}{\partial O_{pi}} \left[ \frac{1}{2} (T_{pi} - O_{pi})^2 + \ldots + (T_{pi} - O_{pi})^2 + \ldots \right] \frac{\partial}{\partial \text{net}_{Mi}} f(\text{net}_{Mi})
\]  

(2.23)

\[
= -\frac{\partial}{\partial O_{pi}} \left[ \frac{1}{2} (T_{pi} - O_{pi})^2 \right] f'(\text{net}_{Mi}).
\]  

(2.24)

Using equation (2.9),

\[
\delta_{Mi} = (T_{pi} - O_{pi}) f'(\text{net}_{Mi})[1 - f(\text{net}_{Mi})]
\]  

(2.25)

\[
= (T_{pi} - O_{pi})(O_{pi})(1 - O_{pi}).
\]  

(2.26)

This should correspond with equation (2.10). For error values associated with the hidden layer neurons, we cannot use target values. For this reason, the part \( \partial E_p / \partial Y_{ji} \) in equation (2.21) needs to be found using a
different approach. We use the chain rule applied to the sum-of-product values of neurons in the front layer (layer \((j+1)\)).

\[
\frac{\partial E_p}{\partial Y_{ji}} = \frac{\partial E_p}{\partial \text{net}_{(j+1)1}} \frac{\partial \text{net}_{(j+1)1}}{\partial Y_{ji}} + \frac{\partial E_p}{\partial \text{net}_{(j+1)2}} \frac{\partial \text{net}_{(j+1)2}}{\partial Y_{ji}} + ... \\
= \sum_{a=1}^{N_{j+1}} \left[ \frac{\partial E_p}{\partial \text{net}_{(j+1)a}} \frac{\partial \text{net}_{(j+1)a}}{\partial Y_{ji}} \right] \\
= \sum_{a=1}^{N_{j+1}} \left[ -\delta_{(j+1)a} \frac{\partial}{\partial Y_{ji}} (W_{(j+1)a} Y_{j0} + ... + W_{(j+1)a} Y_{ji} + ...) \right] \\
= \sum_{a=1}^{N_{j+1}} \left[ -\delta_{(j+1)a} W_{(j+1)a} Y_{ji} \right] \\
= \sum_{a=1}^{N_{j+1}} \left[ -\delta_{(j+1)a} W_{(j+1)a} \right]. \\
\text{(2.27)}
\]

Finally, combined with \(\frac{\partial Y_{ji}}{\partial \text{net}_{ji}}\) we get:

\[
\delta_{ji} = - \sum_{a=1}^{N_{j+1}} \left[ -\delta_{(j+1)a} W_{(j+1)a} \right] \frac{\partial Y_{ji}}{\partial \text{net}_{ji}} \\
= Y_{ji} (1 - Y_{ji}) \sum_{a=1}^{N_{j+1}} \left[ \delta_{(j+1)a} W_{(j+1)a} \right] \\
\text{(2.32)}
\]

This should concur with equation (2.11).

### 2.4.5 Local Minimum Problem

The backpropagation algorithm, as just described, employs gradient descent by following the slope of RMS error value \(E_p\) downward along with the change in all the weight values. The weight values are constantly adjusted until the value of \(E_p\) is no longer decreasing. Since the RMS error value is very complex function with many parameter values of weights, it is possible that the backpropagation network may converge into a local minima instead of the desired global minimum. This phenomenon of “learning paralysis” can be avoided with several solutions suggested [Gur97]. One is the matter of order in presenting training samples to the learning network. Adding noise to the weights while being updated could be also the solution. Another answer is to utilize momentum, which gradually increases the weight adjustment rate \(\beta\). All of these solutions are the way to escape from the trap of a local minimum.

### 2.4.6 Generalization

A trained backpropagation network is able to detect and classify an input pattern that has not been seen during learning. This feature is called generalization, borrowed from the psychology terms. Neural networks are known to be good at classifying noisy input patterns, but not at classifying a pattern that is intermediate between two solid patterns from the training samples. In other words, neural networks are good at interpolation but not extrapolation [BJ91]. Also, there may exist overfitted input data, the unseen input pattern such that it can be classified into one of the trained output response undesirably [Hay99]. Suggested solution includes modification of network architecture and more adequate training samples.
Chapter 3
Concurrent Processing

3.1 Multiprogramming and Multitasking

The most common modern computers consist of processors, memories, timers, disk or tape drives, terminals, and wide variety of other devices. Operating systems manage these hardware devices and resources, and provide controlled access to these devices for the user programs [Tan92]. The rapid growth in technology allows us to have faster hardware devices than past generations; however, not every type of device has increased its speed at the same rate. For example, disk drives have improved in speed of access, but due to the mechanical limit in rotation, the overall speed increase is not as significant as processors or other devices [RR96]. The fundamental difference between mechanical devices and semiconductor materials causes this disparity in performance.

Consider a user program waiting for the data to be fetched from a disk. Because of the time disparity between the disk drive and the processor, the program causes the CPU to be idle for a significantly long time. The concept of multiprogramming is inspired by the fact that while the CPU is idle for one program waiting for the same resource to be available, the CPU can execute instructions for other programs.

The modern operating systems like UNIX not only does multiprogramming, but also timesharing. Timesharing is a variant concept of multiprogramming, and means that the system appears to be executing multiple instances of programs simultaneously. Consider an analogy that a busy individual who needs to cook and do laundry. Two jobs, cooking and washing clothes, can be done one after another, but it is more efficient to be done at the same time, because some stages of both jobs may require waiting for something. While the individual is waiting for the clothes to be dried, it is possible to do something for cooking.

In a timesharing system like UNIX, a process becomes the basic active entity and the most central key object. The following will discuss the definition, modeling, and properties of processes.

3.1.1 Processes

A process is defined as an instance of execution of a program, which has started and not yet terminated [RR96]. When a program is loaded by the system, the system forms a process or processes by allocating the memory for the program image then keeps the entry of the process within the process table which is maintained by the operating system. In modern operating systems, it is often true that a program is formed with multiple processes. This is how a program becomes a process or a number of processes.

In multiprogramming environment, the CPU must switch back and forth between processes, and the system needs the maintain the table of process entries. Figure 3.1 illustrates four programs that become four
processes where each has its own flow of control [Tan92]. From the CPU’s point of view, the execution of each process is traced by a register that represents the program counter (PC), thus with four processes there are four program counters. Since only one process can be executed at a time by the CPU, the CPU saves and loads the PC (and other information about the process) in the system’s process table as context switch occurs.

### 3.1.2 Process Table

The operating system manages processes by the way of having the *process table*. For the system, each process is represented as an entry in the process table. Each entry of the given process is often known as *process control block (PCB)* or *process table field*. Each entry contains certain pieces of information about a process, and normally includes the following:

- **Process state**: How a process is being executed.
- **Registers**: Program counters (PC) and other sets of registers depending on the CPU architecture.
- **Scheduling information**: Priority, order of queue, and other scheduling parameters.
- **Memory Management**: Amount of allocated memory, etc.
- **Identification**: Process ID, user and group ID’s of who owns the process, and permissions.

![Figure 3.1: (a) Four Programs that are Loaded in Memory (b) Conceptual model of Four Independent Processes (c) Timing Model that Shows only One Program is Active at a Time](image-url)
3.1.3 Process State

The process state can indicate the status and condition of a process. Most operating systems form a set of process states, and figure 3.2 shows the graphical representation of the most typical process states.

When a program is loaded and gets the transformation to an active process, the process is in the new state. When such transformation is done, the process is ready to run. The process gets in the queue of ready processes. The process is running when it takes the turn for the CPU to be allocated by the scheduler.

At some point, the process must wait for some event to occur in order to proceed execution, such as when it makes an I/O request to the system. Until the event occurs or the I/O request returns, the process is blocked. Sometimes a process can voluntarily block by going to sleep. Once the process is out of blocked state, it is again ready to run.

Since the scheduler allocates the limited CPU usage to each process, the process must make a context switch when its CPU usage has run out and goes back to the ready state. When the process finishes execution, it will be terminated and transitions to the done state. This means all execution is done but the process entry is still in the system’s process table. This state is also known as the “zombie state” [RR96].

3.1.4 Running Processes

In UNIX platforms, the creation, control, and termination of processes can be accomplished by using the system calls shown in figure 3.3 [RR96]. The fork() function creates a new process by making a copy of the calling process, the parent process, in memory. The newly created process, the child process, inherits most of attributes and resources of the parent process, and has its own process table entry. Since the child process is a clone of the parent process and has the same instruction codes to execute, we need some way to distinguish between these processes. Indeed, upon successful creation of a child process, fork() returns the child process ID number with datatype of pid_t to the parent process, and 0 to the child process [Vah96].

The process can terminate itself through the exit() function. The parameter of exit() function indicates the status of process termination, where a status value of 0 often indicates normal exit. The wait() and waitpid() functions can be used to block the calling process until a child process terminates. While wait() waits for any child process that will exit first, waitpid() can specify child process to wait for.

Additionally, the exec() system call and its variants are often used by the child process created by fork(). The exec() function family starts a new program in place of the copy of the parent process.
```c
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

pid_t fork(void);
pid_t wait(int *stat_addr);
pid_t waitpid(pid_t pid, int *stat_addr, int options);
void exit(int status);
```

`fork()`, `wait()`, and `waitpid()` return the child PID or -1 on error.

**Figure 3.3: System Calls to Control Processes**

### 3.1.5 Interprocess Communication

One objective of designing a time-shared, multiprogramming operating system is to assure that processes that are allocated in separate address spaces should not interfere with each other. However, it is often true that we need to provide applications with a method for sharing resources between processes. For two processes to communicate each other, the operating system must provide some form of Interprocess Communication (IPC) where the system regulates the rules between processes in sharing resources [Ste90]. This means that an IPC object is created by and maintained within the operating system kernel. Figure 3.4 illustrates the communication between two processes through the system. Both the UNIX System V and POSIX standards provide the following methods of IPC:

- Named and unnamed pipes.
- Message queues.
- Named and unnamed semaphores.
- Shared memory space.

### 3.2 Concurrency

*Concurrency*, which refers to sharing resources in the same time frame, occurs at both hardware and application software levels. At the hardware level, concurrency is when multiple devices operate at the same time or when a processor has internal architecture that allows multiple instructions to be executed in parallel. At the software level, as described in previous sections, concurrency happens when sharing I/O communication channels (i.e. file descriptors) or passing information between processes [RR96].

A traditional approach of achieving concurrent processing in UNIX is to create multiple processes with `fork()` system call. The processes need to coordinate properly up to their termination. The coordination of processes is done by the Interprocess Communication (IPC) provided by the operating system.

The alternative to concurrent processing is to have multiple *threads* within a process. A thread refers to a flow of execution control, and sharing resources and communication between threads is much simpler than
with the multiple-process model. The shared resources between threads can be allocated within the same user address space. Unfortunately, there had not been a standard way of using threads until recently; however, more multiple-thread applications are being developed today because of the availability of standards and the advantages over the traditional multiple-process model [RR96].

3.2.1 Threads

Threads are often called lightweight processes. Each thread has its execution control – program counter, register contents, and stack. Each process has at least one thread. Multiple processes within a process share the address space, global variables, file descriptors, and so on. Figure 3.5 shows the two different models of concurrent processing where three jobs can be parallelized. In figure 3.5(a), the system can see three processes whereas there is only one process running in figure 3.5(b) [Tan92].

Since the number of instructions per thread is smaller than the number of instructions per process, operation of threads is relatively cheap in terms of CPU cost. However, care must be taken to coordinate threads because there is no protection between threads. This is due to the fact that (1) threads share the common user address space, and (2) it is impossible to have protection [Tan92].

Perhaps, one of the most significant advantages of multiple-thread application is the access to the multiple processors. By running each thread on a different processor, the application can achieve true parallelism [Vah96]. If the number of threads is greater than number of processors, the threads are multiplexed in CPU utilization.

3.2.2 Processes vs. Threads

The traditional method of concurrency is done by creating multiple processes with `fork()` system calls. With wider availability of thread usage today, it is important to realize the differences between multiple-process and multiple-thread applications:

- `fork()` is an expensive system call. Creating a new process requires more system memory space thus it causes more load on the operating system in keeping track of active processes [Ste98].

- In a multiple-process application, the only way for the processes to share resources is through an Interprocess Communication (IPC) object, which is maintained by and kept within the system. For
the user program, usage of these IPC objects is normally simple and abstract but causes heavy system overhead.

- On the other hand, in a multiple-thread application, all the threads share resources within user address space and load on the operating system is reduced. However, the synchronization between threads is much more complicated. Moreover, using system call functions in multiple-thread application may have some unusual implications [Vah96].

### 3.2.3 User-level and Kernel-level Threads

There are two distinct models of thread controls, and they are *user-level threads* and *kernel-level threads*. The thread function library to implement user-level threads usually runs on top of the system in user mode. Thus, these threads within a process are invisible to the operating system. User-level threads have extremely low overhead, and can achieve high performance in computation. However, using the blocking system calls like `read()` , the entire process would block. Also, the scheduling control by the thread runtime system may cause some threads to gain exclusive access to the CPU and prevent other threads from obtaining the CPU. Finally, access to multiple processors is not guaranteed since the operating system is not aware of existence of these types of threads.

On the other hand, kernel-level threads will guarantee multiple processor access but the computing performance is lower than user-level threads due to load on the system. The synchronization and sharing resources among threads are still less expensive than multiple-process model, but more expensive than user-level threads. The thread function library available today is often implemented as a *hybrid model,* as having advantages from both user-level and kernel-level threads. The design consideration of thread packages today consists of how to minimize the system overhead while providing access to the multiple processors [RR96].

### 3.2.4 Multi-threaded Applications

When developing an application, the programmer must consider how multiple threads can be utilized. One of the typical scenarios for using multiple threads is when there are multiple jobs that can be processed independently or asynchronously. Also, there is a situation where while the original thread is blocked, one thread can be executed for a different job.
3.3 POSIX Threads

POSIX is an acronym for “Portable Operating System Interface” and is a family of standards developed by the Institute of Electrical and Electronic Engineers (IEEE). POSIX began with specification of a system application program interface (API) in C in 1988, and added the second part in 1992. The third part is under development.

The first part of POSIX appended its extensions in 1990, 1993, and 1996. The 1996 edition covers the base API of 1990, real-time extensions of 1993, threads of 1995, and some technical corrections. The first part of current POSIX standard is known as POSIX.1 [Ste98]. Most UNIX systems today are somewhat compliant with POSIX and current versions of Linux and Solaris are no exception. It is always the best for programmers to use the POSIX functions whenever possible.

POSIX.1c, which covers threads, provides the standard library package that contains thread creation, destruction, and synchronization objects. Solaris 2 or later, and Linux kernel version 2.0 or later contain the POSIX thread library, Pthreads. All Pthread function calls and name of associated objects have the prefix pthread.

3.3.1 Thread Creation

When a program is loaded, there is only one thread. To create an additional thread, pthread_create() function is used. Each thread is identified by a thread identification number, with datatype of pthread_t. The function call returns the thread ID number by reference.

Each thread has its attributes that can specify priority, stack size, and others. The function call can also specify such attributes. In most cases, we can take the default thread attributes by passing NULL pointer value.

When a new thread is created, we also must specify what to execute. The thread creation function accepts the pointer to the routine to execute and its single parameter value. The types for func_routine and arg are shown in figure 3.6. The function routine func_routine accepts a single argument of type pointer to void, and also returns a pointer to void. In order to pass multiple data objects to this function, it is appropriate to pass an argument as pointer to some struct.

Note that the return value of POSIX thread functions are slightly different from typical system calls. These functions return 0 if OK. When error occurs, the return value is not -1 but some nonzero value that represents the error number. An Exxx value is returned instead of setting errno.
3.3.2 Thread Termination

pthread_join() waits for a specified thread to terminate. This is analogous to waitpid() function which waits for a specified child process to exit. pthread_join() must specify the thread ID to wait for, unlike waitpid() that has an option to leave the process ID unspecified with value of -1.

A thread can terminate itself by calling pthread_exit() function. The argument is the pointer to the status value, but the status value must not be stored locally to the function routine that started the thread. A thread can also terminate by either returning from the function routine, having the main() of the program return, or when one of the threads within a process calls exit() which terminates the entire process.

When non-null argument of status is involved in the pthread_join(), the joining function will return the pointer to the status value by reference.

3.3.3 Thread Identification

The thread that calls pthread_create() can obtain the ID of the thread created, but pthread_self() function is used to obtain the ID of the self. This is similar to getpid() function which obtains the process ID number of itself.

3.3.4 Thread Attributes

The POSIX thread has an associated attribute object that can represent properties. An attribute object can be associated with multiple threads, and there are functions to create, configure, and destroy the attributes. The following is the list of the types of configurable properties by the function calls [RR96]:

- Initialization,
- Stack size,
- Detach state,
- Scope,
- Inheritance,
- Schedule policy, and
- Schedule parameters.

3.4 Thread Synchronization

The threads that are created within a process can share data and resources such as global variables and file descriptors in the user address space of the process. However, we must be concerned about synchronizing the access to such resources and variables in order to avoid potential conflict and inconsistent result of data. There are two types of synchronization – locking and waiting. Locking is used to exclusively hold the resource for a short duration. On the other hand, waiting refers to blocking for unbounded duration of time until a certain event occurs [RR96].

The following sections discuss synchronization primitives provided by POSIX.1c that can be used for multi-threaded programs.
3.4.1 Mutexes

One of the goals in synchronization is to avoid race conditions. A race condition refers to the situation where two or more jobs are accessing a shared resource and the final result depends on the order of execution. In other words, the winner of the race holds the key of the final result. To avoid such situation, we need to guarantee that only one job can access the shared resource at a time, by providing certain rule of mutual exclusion.

Mutual exclusion can be accomplished by having a single, shared variable which represents a lock associated with each shared resource. The lock is enabled during the critical period where the shared resource is being accessed, and each job must check the lock before entering the access to the shared resource [Tan92].

POSIX.1c provides mutex, which represents mutual exclusion and protects a critical section of the program where the shared resource or variable is accessed by one thread at a time. POSIX mutex has a datatype of pthread_mutex_t, and can be statically allocated and initialized to the constant PTHREAD_MUTEX_INITIALIZER. We can control the mutex values using the four function calls as listed in figure 3.7.

A mutex must be initialized before using, and can be initialized using either pthread_mutex_init() function or static constant PTHREAD_MUTEX_INITIALIZER. Although the latter method is more efficient, the former function call method is used when a mutex is allocated dynamically in the situation like using malloc() or shared memory. pthread_mutex_init() function also allows initializing a mutex with the specified attributes.

pthread_mutex_lock() function locks the mutex. If the mutex is already locked by some other thread, pthread_mutex_lock() blocks until the mutex is available. pthread_mutex_trylock() function also locks the mutex; however, it is a nonblocking call and returns EBUSY if the mutex is locked already. pthread_mutex_unlock() unlocks the mutex [Ste99].

Each of the functions above are indeed executed as an atomic action, which is guaranteed to be completed without context switch. Also, pthread_mutex_lock() as a blocking function does not use busy waiting and does not consume CPU time while blocked. Instead, the calling thread is suspended. Finally, note that all four functions will return 0 upon success; otherwise, an error value is returned to indicate the cause of error.

The code segment in figure 3.8 illustrates the usage of a mutex to protect a critical section where a shared variable counter is incremented. Notice that the function thread_function() is executed by a thread that will increment the shared variable count using a mutex lock named count_lock.

Notice that the function thread_function() is executed by a thread that will increment the shared variable count using a mutex lock named count_lock.
```c
#include <pthread.h>

int count = 0;

pthread_mutex_t count_lock =
    PTHREAD_MUTEX_INITIALIZER;

thread_function(void)
{
    pthread_mutex_lock(&count_lock);
    count++;
    pthread_mutex_unlock(&count_lock);
}
```

Figure 3.8: A Thread Accessing a Shared Variable with Mutex Lock

### 3.4.2 Condition Variables

A mutex demonstrates an efficient method of controlling exclusive access to the shared resource, and may also be used for a thread to wait for a shared resource to assume a desired value. The sequence of the code in figure 3.9 allows the thread to wait until a shared counter becomes a desired value. The fundamental issue of this type of implementation for waiting is that we have no idea how many times the thread executes the sequence within the while loop. Nevertheless, this is busy waiting and waste of CPU time [Ste99].

We need a different type of synchronizing object for waiting. POSIX.1c provides a condition variable for this purpose which has a datatype of `pthread_cond_t`. Figure 3.10 lists the functions that can be used for basic condition variable operations. Similar to a mutex, a condition variable must be initialized using either `pthread_cond_init()` function or static constant `PTHREAD_COND_INITIALIZER`.

`pthread_cond_wait()` is used to wait for shared data to enter a desired state. Let’s define a Boolean predicate which indicates such a state. The predicate may be indirect – such as testing whether a counter has reached a certain value. The waiting thread first checks the predicate, and if the desired predicate is not true, the thread waits for the condition variable to be signaled. When checking the relevant predicate, a mutex must be associated with the condition variable. The segment of the program in figure 3.11 shows that the thread is waiting for a predicate that the counter has reached a desired value.

Note that the shared counter variable `count` is associated with a condition variable `count cond` and a mutex `count lock`. To check if `count` has reached the desired value `desired value`, the thread must first lock the mutex. The thread keeps waiting on the condition variable to be signaled, but also the mutex cannot be locked during the entire waiting period. Instead, `pthread_cond_wait()` function atomically unlocks the mutex then blocks. Similarly, as the function returns, the calling thread wakes up then locks the mutex [RR96]. This way, the other thread is allowed to access the counter to increment. Also, use of while loop is required to check the predicate once `pthread_cond_wait()` returns, because receipt
```c
while(1) {
    pthread_mutex_lock(&count_lock);
    if(count > desired_value) {
        pthread_mutex_unlock(&count_lock);
        break;
    }
    pthread_mutex_unlock(&count_lock);
}
```

Figure 3.9: A Thread Using Mutex for Waiting

---

```c
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *cond,
     const pthread_condattr_t *attr);
int pthread_cond_wait(pthread_cond_t *cond,
     pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

All returns 0 on success, otherwise error number.

Figure 3.10: Basic POSIX Condition Variable Functions
```c
int count = 0;
pthread_mutex_t count_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t count_cond = PTHREAD_COND_INITIALIZER;

void *waiting_thread_func(void)
{
    pthread_mutex_lock(&count_cond);
    while(count < desired_value)
    {
        pthread_cond_wait(&count_cond, &count_lock);
        pthread_mutex_unlock(&count_cond);
    }
}
```

Figure 3.11: A Thread Using Condition Variable for Waiting
of signal only means the change in condition variable and the counter count has not reached the desired value yet. This is to avoid rare occurring of such spurious wake-ups [Ste99]. Thus, we understand that in the code above, the thread executes only once in the while loop, if count has really reached the desired value when signaled.

Using pthread_cond_signal() function unblocks one of the threads that is waiting for the condition variable to be signaled. In situation where the multiple threads should be awakened, pthread_cond_broadcast() function should be used to awaken all the threads that are blocked. It is suggested that the mutex should be locked by the thread calling pthread_cond_signal() or pthread_cond_boradcast() function [Ste99].

3.4.3 Other Synchronization Primitives

Semaphores are the traditional method for synchronization between processes or between threads. A semaphore is normally and integer value and can be operated atomically. There are three types of semaphores: Unnamed semaphores, named semaphores, and System V semaphores. The first two types of semaphores are known as POSIX semaphores, provided by POSIX.1b, and the difference between them are whether the semaphore is stored in shared memory or identified by an IPC name. The third type of semaphore, a System V semaphore, is a more common form of semaphore and maintained within the operating system. System V semaphores can also be organized as a set of semaphores [Ste99]. The semaphores can be used for both locking and waiting and the operations are more abstract than mutexes or condition variables. By this fact, the semaphores are a more robust form of synchronization; however, semaphores are often allocated and maintained within the operating system and operations are often expensive. Indeed, semaphores are the form of Interprocess Communication (IPC) which is more likely to be used for multi-processed programs. Simpler usage of semaphores and expense of system resources are the trade-offs.

The coordinating threads can be synchronized with other forms of Interprocess Communications (IPC) and signals, but they also require more system-intensive operation than POSIX mutexes and condition variables. Finally, the pthread_join() function can be used for thread synchronization and it allows a thread to wait for a specific thread to complete its execution [Dig96].
4.1 Parallel Neural Network Architecture

Now with the information obtained from the previous two chapters, we are going to propose the software model of a backpropagation neural network that utilizes concurrent processing. The nature of computing architecture in neural networks allows distribution of tasks with small amount of effort. However, while distributing and parallelizing computing tasks can be accomplished very easily, the computing architecture of neural networks provides wide variety of choice in methods of task distribution. First, we examine several suggested methods of partitioning computation for simulating neural network functions.

The task distribution in backpropagation neural networks can be classified into two types: network partitioning and pattern partitioning [AS94]. In network partitioning, calculations of node activation, errors associated with nodes, and weight adjustments are parallelized. The coordination among individual tasks are intensive in order to maintain consistency in these values. On the other hand, in pattern partitioning, multiple sample patterns can be presented in parallel. Weight adjustments for different patterns can be done individually. At the end of being presented all sample patterns, the “comprehensive” weight adjustments are required. This method is highly suitable for applications like serial-input pattern recognition [Aus92].

Here, we focus on the methods in network partitioning, since there are more potential neural network applications on pattern recognition with parallel inputs (opposed to serial inputs) and wide availability of synchronization primitives in software modeling. Also, in network partitioning, scalability of parallel tasks is maintained regardless of the small number of sample patterns for training.

Now we can further categorize the network partitioning methods into two ways. The first approach is to place each layer into separate parallel computing tasks. The drawback of this is that there is often a load imbalance among tasks due to the different number of nodes in each layer. Also, the neural network requires a large number of layers to benefit the parallelizing tasks. We consider the second approach – slicing the network along the lines perpendicular to the layers as shown in figure 4.1.

For the software modeling which will be described in the next section, the computing tasks for neural network training will be done in network partitioning, by slicing the network against the direction of the layers. Furthermore, we assume that the distributed tasks should be balanced thus the number of neurons in each divided group in a layer is always equal.
4.2 Parallel Neural Network Functions

The Parallel Neural Network (pnn) software package will provide highly configurable, and scalable functionality of backpropagation neural networks utilizing POSIX threads. The package is expected to be used by developers of higher-level neural network applications who desire high-performance in neural network learning and function testing. The software is in the form of Application Programming Interface (API), meaning that users or developers can simply include the provided header into their source code prior to calling the parallel neural network function routine, then the code can be linked with the library included in the package, as if using a typical third-party development library. Finally and most importantly, the package is designed to be portable across various UNIX platforms. The software package should include a header (pnn.h) and two function calls (weight_init() and run_pnnnet()), which will be described in the following sections.

4.2.1 Configuration and Data Structure

Shown in figure 4.2 is the diagram that illustrates the data structure of the neural network. Figure 4.2 (a) is the structure of the entire network with $M$ layers, figure 4.2 (b) is the organization of neurons within a layer where $N_m$ indicates the number of neurons in layer $m$, and figure 4.2 (c) is a single neuron and its surrounding data objects such as weight, input, output, and the error values. In programming languages, the data array is often indexed starting with 0, instead of 1. Thus, this diagram is modified from the one in the previous chapter.

Let’s take a close look at the diagram of a single neuron. $OUT[m][n]$ is a two-dimensional array that represents the output value from neuron $n$ in layer $m$. The neuron $n$ in layer $m$ accepts the signal fired from the neurons in the previous layer. The weight value $w[m][n][l]$ is the connection weight from neuron $l$ in layer $m-1$ to neuron $n$ in layer $m$. Finally, the error value associated with the neuron is represented as $ER[m][n]$.

4.2.2 Weight Initialization Routine

Before executing the neural network function, the programmer must initialize the weight values that will be used for the neural networks. Once the weight value data structure is created and set, the function call weight_init() will initialize the three-dimensional array of weights into the random values normalized within the specified range. Shown in figure 4.3 is the proper syntax of calling weight_init(). The first argument, weights, is the pointer to the floating point type array of weights. The second argument size
Figure 4.2: Graphical Description of Data Structure for Implementation.

```c
#include <pnnet.h>
void weight_init(float *weights, int size, float min, float max);
```

Figure 4.3: Weight Initialization Function Call

is the size of the array of weights, not size in bytes. The last two arguments `min` and `max` are the maximum and minimum values for the random value to be normalized within. Note that the function does not need to know about the dimensional size of the weight values, but it only needs to know how many values are in the array.

### 4.2.3 run_pnnet() Routine

Once the weight values are initialized, the neural network function is ready to be executed. The execution of neural network function can be accomplished by simply calling the `run_pnnet()` function. The syntax is shown in figure 4.4. This function accepts one argument value which is pointer to the value with datatype of pnn_param. This datatype pnn_param is actually a struct and defined in pnnet.h, and the members of the struct are as follows:

```c
typedef struct {
    int nlayers; /* Number of layers, including input layer */
```
```c
#include <pnnet.h>
int run_pnnet(pnn_param *arg);
```

*run_pnnet() returns 0 on success, 1 when error occurs.*

**Figure 4.4: Parallel Neural Network Function Calls**

```c
int nneurons[MAXLS]; /* 1-D array of # of neurons */
float *wt; /* Pointer to the 3-D array of weight values */
    /* Dimension size by PNN_N_MAX_LAYERS *
    PNN_N_MAX_NEURONS * PNN_N_MAX_NEURONS */
int samples; /* Number of samples to teach the neural network */
float inputs[MAXSS][MAXNS]; /* 2-D array of sample inputs */
float targets[MAXSS][MAXNS]; /* 2-D array of sample targets */
float *outputs; /* Pointer to the array of outputs returned */
float bias; /* Bias value for this neural net */
float eratio; /* Error ratio */
float wratio; /* Weight adjustment ratio */
int iteration; /* # of iteration to loop in presenting samples */
int concurrency; /* Level of concurrency */
unsigned int debug_mode; /* Indicate debug mode during execution */
} pnn_param;
```

PNN_N_MAX_NEURONS, PNN_N_MAX_LAYERS, and PNN_N_MAX_SAMPLES are defined variables that represent maximum possible number of neurons, layers, and training samples respectively. MAXNS, MAXLS, and MAXSS are the aliases of those values. The details can be referred to Appendix A.

This data structure provides high level of configurability in the neural networks to be executed. The included information in this data structure can be categorized into the following:

- Neural network architecture – nlayers, nneurons[]
- Internal data for network training – *wt, bias, eratio, wratio, iteration
- Training sample patterns – samples, inputs[][], targets[][]
- Returned value by reference – *outputs
- Operation modes – concurrency, debug_mode

The neural network function is not just for training the network, but also can be used for simply testing the trained network. The function has the feature of “no training” mode by setting both iteration and samples into 1. In “no training” mode, the function will bypass the section of code where error calculation and weight value adjustments are executed.

The last item of this data structure, debug_mode, is for indicating the amount of information to be outputted to the console screen. The value can be set one or more bitwise OR combinations of any of the following values as defined in pnnet.h:
#define PNN_D_QUIET 0x0000 /* No debug mode */
#define PNN_D_INITWT 0x0001 /* Enable displaying initial weights */
#define PNN_D_ITER 0x0002 /* Enable displaying iteration # */
#define PNN_D_SAMP 0x0004 /* Enable displaying sample # */
#define PNN_D_FP 0x0008 /* Enable displaying status of FP */
#define PNN_D_OUTPUTS 0x0010 /* Enable displaying outputs */
#define PNN_D_MODE 0x0020 /* Enable displaying execution mode */
#define PNN_D_BP 0x0040 /* Enable displaying status of BP */
#define PNN_D_POSTWT 0x0080 /* Enable displaying final weights */
#define PNN_D_ALL 0x00FF /* All enabled */

The return value of the run_pnnet() function is either 0 or 1. Return value of 0 indicates the success of the operation, whereas 1 indicates some error occurred. The condition of errors will be displayed into standard output. The most typical causes of error include improper values of concurrency and number of neurons in each layer set.

Inside of the function routine, the specified number of threads are created at the beginning of the forward propagation, then all the threads are joined back at the end of the backward propagation. Each threads takes a partitioned section of the neural network as described previously. The coordination between threads is done utilizing POSIX mutex and condition variables [Ste99, RR96]. The synchronization is performed in such a way that all the threads must be performing neural calculations within the same layer and no thread is allowed to proceed to next layer until all threads are finished. More specifically, a global counter variable is set for each layer, for indicating how many threads are finished for the given layer. Each completed thread increments the counter with mutually exclusive access, and waits for the condition variable to be signaled by the thread that finishes last. The thread can find itself being the last thread by examining the value of the counter, and broadcasts the signal to other waiting threads. Figure 4.5 shows the flowchart representation of the synchronization algorithm just described above.

4.3 The Testing Module: Letter Recognition

This section will discuss about the actual testing of the neural network functions described above. A simple digital image recognition is chosen for the testing application. Specifically, the application is a 16 by 16 pixel letter image recognition. A user-friendly interface is provided using Motif Graphical User Interface (GUI), which allows the user to manipulate the image by “point and click”.

4.3.1 The Motif Graphical User Interface

Motif* is a product of Open Software Foundation (OSF) and is a set of graphical user interface built on X Window System [McM93]. Motif provides a set of components called widgets, which gives consistent look and feel of graphical user interface. For the software development, Motif allows flexibility in event-driven programming.

Shown in figure 4.6 is the main window of the user interface. This window consists of three areas: menu bar on the top, the command buttons on the right, and 256 toggle buttons in the main area for manipulating letter image. The menu bar contains “File” and “Help” pulldown submenus, where the “File” pulldown contains commands for data file manipulation and a command to exit application. The “Help” pulldown contains commands for displaying some information about the application program. Each toggle button in the main area is in one of two states, on or off. The button is toggled on or off every time it is clicked. We will discuss the command buttons on the right in the following sections.

* Motif,OSF/Motif, and X Window System are trademarks of the Open Group, the Open Software Foundation, Inc., and the X/Open Company Ltd.
enter the current layer

perform calculation for "my section" in this layer

lock mutex

FLG: shared variable representing # threads completed in this layer

FLG = # threads

No

unlock mutex

wait for condition of FLG = # threads

Wakeup

Yes

# threads = 1 ?

No

broadcast condition of FLG = # threads

unlock mutex

go to next layer

Figure 4.5: Flowchart of Thread Synchronization in run_pnnnet()
4.3.2 Pre-Training Process

In this letter recognition application, the user can construct the input pattern by clicking on the toggle buttons in the main area of the window. Once the desired input image is drawn, clicking the “Teach” command button allows the user to choose and specify the output pattern. The user can repeat this process until the set of sample patterns is completed for training the neural network.

Alternatively, the user can load the set of samples from a data file which contains pre-defined input-output patterns. The user can then add more pattern entries, and save the result in a file. Shown in figure 4.7 is the dialog window that prompts the user to select the file of the sample data to load. Finally, the “Review” command button will display the list of loaded patterns.

4.3.3 The Core Neural Network Processing

Once desired set of pattern samples are loaded, the user can begin training the neural network by pressing the “Train Net” command button (figure 4.6). Prior to actual neural network calculation, the program will prompt for the user’s confirmation and display the configuration of the neural network to train. The progress of network training will be displayed onto the standard output during the actual training. The neural network for this application is fixed to have three layers, including the input layer. Each input node of the network corresponds to each pixel in the image, and the number of output nodes is set to at least the maximum possible number of training samples that the program can accept. For example, the neural network should have at least 26 output nodes in order to be trained to classify 26 different alphabetical character images [Hay99]. The number of times presenting the samples can be configured either by command line argument or macro definition in the program header. The default is set to 1000 times. Similarly, the number of threads and the number of hidden layer nodes can be set through command line argument.
4.3.4 Post-Training Process

The neural network is properly trained and the weight values are in the correct state, the user can test the performance of the pattern recognition. Again, the user can draw the letter image with the toggle buttons, and the program will detect the given pattern by pressing the “Acquire” command button. The program displays the output dialog, shown in figure 4.8, that contains the detected pattern and values from the output neurons. Since each output neuron corresponds to each type of output response, the value from the output neuron indicates the possibility of “belonging” to the type of pattern.

4.4 Testing Results and Analysis

We are going to examine the result outcomes of the testing application with two distinct criteria. First is the actual neural network functionality. It is a must to obtain properly working neural networks regardless of the modeling. Then, more importantly, we will examine the performance in speed during the neural network training, which is the primary objective of this thesis.

4.4.1 Letter Recognition Performance

Generally speaking, the trained neural network is very insensitive to the additive noise or distortion in the input. With additive noise or distortion, the corresponding neural output value was slightly less than the output value without the noise or distortion. However, the neural network often gets “confused” when the testing image is a mixture of two distinct sample images. Such ambiguous decision could occur as a potential weakness of backpropagation neural networks [Hay99].

4.4.2 Training Speed Performance

Perhaps, it is our foremost interest to observe the performance in computation time of the neural network training, because the primary objective is to increase the performance in speed of the neural network function. The execution time is measured with both gettimeofday() and getrusage() functions [RR96]. The first timestamp is taken immediately before calling run_pnnet() function, and the second is taken immediately after the return of the function.
In our experiment, we have the following fixed parameters. First, the neural network is presented 26 distinct sample patterns of a 16 by 16 pixel letter image. For this reason, the backpropagation network is fixed to have 256 input nodes and 32 output nodes. Since we are going to experiment with 1, 2, 4, and 8 threads, the number of output nodes is chosen to be 32 to balance the load between threads. Also, the neural network is fixed to be presented the set of samples 1000 times.

We consider varying the following parameters. First, the experiment is done with 1, 2, 4, and 8 threads. We also vary the number of hidden layer neurons among 8, 16, 32, 64, 128, 256, and 512. The number of hidden layer neurons will indicate amount of computation. Finally, we execute the program on different hardware platforms, using different types and numbers of CPU’s.

Figure 4.9 is the list of execution time on kkawaguc.mindspring.com. The system has one AMD K6-2 CPU (300 MHz) with Linux 2.2 kernel. In each entry, three timings in seconds are shown, which are the execution time measured with gettimeofday(), user resource time measured with getrusage(), and system resource time with getrusage(), respectively. Also, figure 4.10 shows the corresponding measurement plots with gettimeofday(). Figure 4.10 shows that increasing the number of threads causes a slight increase in execution time on a uniprocessor machine. The increase in execution time is due to the amount of system calls required in creating, synchronizing, and joining the threads. With a smaller number of hidden layer neurons, the proportion of system overhead becomes more obvious. Additionally, the time required for the system calls is independent of the amount of computation. This can be verified by the fact that the system resource time with getrusage() is varied by only the number of threads. The same case is shown in figures 4.11 and 4.12, and shows the results when the problem is run on the system eeultra2 with a single UltraSPARC CPU (300 MHz) and Solaris 2.7 as the operating system.

Now let's take a look at the case running on systems with two processors. Figures 4.13 and 4.14 show the measurements taken on electra, which has two UltraSPARC CPU’s (360 MHz each) running Solaris 2.7. Take a close look at the plots. Considering the case of 1 thread as the baseline, we see that performance with 2, 4, or 8 threads are improved at large amount of computation. At smaller amount of computation, the multiple-thread cases are either making no difference or even worse due to overhead by system calls involved with thread manipulation. The similar observation can be done with the case running on vectra shown in figures 4.15 and 4.16. The machine has two Pentium Pro CPU’s (166 MHz) each and Linux 2.2 Kernel with two Symmetric Multiprocessor (SMP) support. For both cases on a bi-processor system, it seems that it is ideal to run with two threads. At this optimal condition and with a large amount of computation, the computation time is converging into near one-half of the one-thread case.

Lastly, figures 4.17 and 4.18 show the results of performance measurements taken on eesun, which has four SPARC CPU’s (Cypress CY605, 40 MHz each) running Solaris 2.5. Again, consider the case with one thread as the baseline again. With two threads, at larger amount of computation, it converges into one-half of computation time with the baseline. Additionally, with four and eight threads, we can see even more improvements from the case with two threads. With 512 hidden layer neurons, the computing time with four threads is only one-third of the baseline, as if it is converging into near one-fourth of the case with one thread eventually. The only difference between the cases with four and eight threads is due to extra system overhead for eight threads. With four processors, we can determine that the optimal amount of threads on this system is four.

Summed, we can conclude with the following observations. First, it seems that the ideal number of threads to execute the neural network function should be equal to the number of processor in the system. Excessive number of threads only causes extra system overhead for handling larger number of threads. Second, the required minimum amount of computation to gain benefits from the multi-threaded environment is relatively small. In other words, a wide range of applications are possible. And above all, it is important for the user to consider both amount of computation and number of threads along with the given system

---

* AMD and K6-2 are trademarks of Advanced Micro Devices, Inc.
* SPARC and UltraSPARC are trademarks of Sun Microsystems, Inc.
* Pentium, Pentium Pro, MMX, Celeron, and Pentium II are trademarks of Intel Corporation.
hardware configuration. Finally, the analysis of performance measurements with a multi-threaded application should be done carefully to consider background process running on the given system. For this reason, it is suggested to use both gettimeofday() and getrusage() to obtain information on background processes that may cause excessive overhead [Sun97].
<table>
<thead>
<tr>
<th>Number of Hidden Layer Neurons</th>
<th>1 Thread</th>
<th>2 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>11.79 / 0.18 / 0.87</td>
<td>17.87 / 0.54 / 1.21</td>
</tr>
<tr>
<td>16</td>
<td>18.60 / 0.34 / 0.46</td>
<td>24.49 / 0.54 / 1.07</td>
</tr>
<tr>
<td>32</td>
<td>31.97 / 0.42 / 0.56</td>
<td>38.00 / 0.58 / 1.04</td>
</tr>
<tr>
<td>64</td>
<td>58.58 / 0.43 / 0.55</td>
<td>63.00 / 0.49 / 1.03</td>
</tr>
<tr>
<td>128</td>
<td>113.28 / 0.43 / 0.78</td>
<td>120.25 / 0.83 / 1.54</td>
</tr>
<tr>
<td>256</td>
<td>235.28 / 0.59 / 0.83</td>
<td>235.37 / 0.88 / 1.35</td>
</tr>
<tr>
<td>512</td>
<td>471.73 / 0.55 / 0.91</td>
<td>481.02 / 0.70 / 1.19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Hidden Layer Neurons</th>
<th>4 Threads</th>
<th>8 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>32.29 / 1.01 / 2.10</td>
<td>55.83 / 1.67 / 4.30</td>
</tr>
<tr>
<td>16</td>
<td>38.00 / 0.75 / 2.07</td>
<td>67.96 / 1.86 / 4.76</td>
</tr>
<tr>
<td>32</td>
<td>50.61 / 0.86 / 2.27</td>
<td>78.43 / 1.93 / 4.81</td>
</tr>
<tr>
<td>64</td>
<td>77.95 / 0.81 / 2.08</td>
<td>103.21 / 1.71 / 4.73</td>
</tr>
<tr>
<td>128</td>
<td>133.10 / 0.90 / 1.96</td>
<td>155.13 / 1.89 / 4.13</td>
</tr>
<tr>
<td>256</td>
<td>252.48 / 1.13 / 2.44</td>
<td>265.74 / 2.00 / 5.83</td>
</tr>
<tr>
<td>512</td>
<td>495.34 / 1.10 / 2.12</td>
<td>508.02 / 2.09 / 5.22</td>
</tr>
</tbody>
</table>

Performance is measured in time (seconds): time with gettimeofday() / user time with getrusage() / system time with getrusage()

Figure 4.9: List of Measurements on kkawaguc.mindspring.com (1 CPU, Linux)

Figure 4.10: Plot of Measurements with gettimeofday() on kkawaguc.mindspring.com (1 CPU, Linux)
<table>
<thead>
<tr>
<th>Number of Hidden Layer Neurons</th>
<th>1 Thread</th>
<th>2 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>9.62 / 9.50 / 0.07</td>
<td>11.14 / 10.68 / 0.43</td>
</tr>
<tr>
<td>16</td>
<td>17.59 / 17.28 / 0.06</td>
<td>18.71 / 18.38 / 0.29</td>
</tr>
<tr>
<td>32</td>
<td>32.61 / 32.46 / 0.03</td>
<td>34.38 / 34.11 / 0.23</td>
</tr>
<tr>
<td>64</td>
<td>63.99 / 63.23 / 0.01</td>
<td>67.81 / 66.51 / 0.41</td>
</tr>
<tr>
<td>128</td>
<td>135.25 / 125.26 / 0.07</td>
<td>131.48 / 129.51 / 1.12</td>
</tr>
<tr>
<td>256</td>
<td>247.56 / 246.62 / 0.68</td>
<td>255.90 / 254.65 / 1.03</td>
</tr>
<tr>
<td>512</td>
<td>506.10 / 504.70 / 0.71</td>
<td>518.56 / 514.37 / 3.56</td>
</tr>
<tr>
<td>Number of Hidden Layer Neurons</td>
<td>4 Threads</td>
<td>8 Threads</td>
</tr>
<tr>
<td>8</td>
<td>14.77 / 13.43 / 1.32</td>
<td>23.50 / 19.38 / 4.07</td>
</tr>
<tr>
<td>16</td>
<td>22.23 / 21.33 / 0.86</td>
<td>28.47 / 26.14 / 2.31</td>
</tr>
<tr>
<td>32</td>
<td>39.24 / 37.28 / 0.88</td>
<td>48.32 / 43.57 / 3.77</td>
</tr>
<tr>
<td>64</td>
<td>75.73 / 69.99 / 0.85</td>
<td>82.77 / 75.77 / 3.56</td>
</tr>
<tr>
<td>128</td>
<td>137.82 / 134.77 / 1.78</td>
<td>149.71 / 144.79 / 4.63</td>
</tr>
<tr>
<td>256</td>
<td>265.67 / 260.95 / 4.40</td>
<td>284.65 / 271.56 / 12.80</td>
</tr>
<tr>
<td>512</td>
<td>530.42 / 521.26 / 8.52</td>
<td>552.27 / 533.22 / 18.27</td>
</tr>
</tbody>
</table>

Performance is measured in time (seconds): time with gettimeofday() / user time with getrusage() / system time with getrusage()

Figure 4.11: List of Measurements on eeultra2.ece.utep.edu (1 CPU, Solaris)

Figure 4.12: Plots of Measurements with gettimeofday() on eeultra2.ece.utep.edu (1 CPU, Solaris)
<table>
<thead>
<tr>
<th>Number of Hidden Layer Neurons</th>
<th>1 Thread</th>
<th>2 Threads</th>
<th>4 Threads</th>
<th>8 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8.37 / 7.85 / 0.83</td>
<td>8.73 / 10.33 / 4.28</td>
<td>68.60 / 108.12 / 3.49</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>15.45 / 14.83 / 0.81</td>
<td>12.21 / 16.89 / 3.89</td>
<td>68.60 / 108.12 / 3.49</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>27.82 / 27.00 / 0.94</td>
<td>20.37 / 30.38 / 3.98</td>
<td>12.21 / 16.89 / 3.89</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>53.86 / 52.42 / 0.97</td>
<td>35.45 / 56.15 / 3.57</td>
<td>68.60 / 108.12 / 3.49</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>104.96 / 102.94 / 0.73</td>
<td>68.60 / 108.12 / 3.49</td>
<td>68.60 / 108.12 / 3.49</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>208.41 / 204.94 / 0.86</td>
<td>132.77 / 217.31 / 3.49</td>
<td>132.77 / 217.31 / 3.49</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>413.83 / 407.19 / 0.84</td>
<td>249.98 / 424.70 / 3.59</td>
<td>249.98 / 424.70 / 3.59</td>
<td></td>
</tr>
</tbody>
</table>

Performance is measured in time (seconds): time with gettimeofday() / user time with getrusage() / system time with getrusage()

Figure 4.13: List of Measurements on electra.ece.utep.edu (2 CPU’s, Solaris)

Figure 4.14: Plots of Measurements with gettimeofday() on electra.ece.utep.edu (2 CPU’s, Solaris)
<table>
<thead>
<tr>
<th>Number of Hidden Layer Neurons</th>
<th>1 Thread</th>
<th>2 Threads</th>
<th>4 Threads</th>
<th>8 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>10.91 / 0.25 / 1.00</td>
<td>13.65 / 0.49 / 1.61</td>
<td>27.50 / 0.93 / 3.47</td>
<td>55.75 / 1.80 / 6.73</td>
</tr>
<tr>
<td>16</td>
<td>16.91 / 0.34 / 0.94</td>
<td>16.96 / 0.60 / 1.71</td>
<td>31.55 / 0.92 / 3.60</td>
<td>59.49 / 1.78 / 6.78</td>
</tr>
<tr>
<td>32</td>
<td>29.50 / 0.31 / 1.39</td>
<td>23.63 / 0.45 / 1.88</td>
<td>38.88 / 0.88 / 3.90</td>
<td>68.08 / 1.83 / 7.63</td>
</tr>
<tr>
<td>64</td>
<td>52.52 / 0.25 / 1.39</td>
<td>36.61 / 0.49 / 2.12</td>
<td>50.72 / 0.95 / 3.53</td>
<td>78.10 / 1.98 / 6.84</td>
</tr>
<tr>
<td>128</td>
<td>103.41 / 0.44 / 1.47</td>
<td>62.16 / 0.48 / 1.87</td>
<td>76.55 / 1.03 / 3.81</td>
<td>104.80 / 1.75 / 7.81</td>
</tr>
<tr>
<td>256</td>
<td>212.94 / 0.36 / 1.38</td>
<td>116.83 / 0.66 / 2.27</td>
<td>136.23 / 1.19 / 3.60</td>
<td>166.59 / 1.91 / 7.94</td>
</tr>
<tr>
<td>512</td>
<td>443.33 / 0.39 / 1.76</td>
<td>242.61 / 0.73 / 3.01</td>
<td>259.56 / 1.05 / 4.04</td>
<td>284.46 / 2.01 / 8.12</td>
</tr>
</tbody>
</table>

Performance is measured in time (seconds): `time with gettimeofday() / user time with getrusage() / system time with getrusage()`.

**Figure 4.15:** List of Measurements on vectra.ece.utep.edu (2 CPU’s, Linux)

**Figure 4.16:** Plots of Measurements with gettimeofday() on vectra.ece.utep.edu (2 CPU’s, Linux)
<table>
<thead>
<tr>
<th>Number of Hidden Layer Neurons</th>
<th>1 Thread</th>
<th>2 Threads</th>
<th>4 Threads</th>
<th>8 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>98.29 / 93.76 / 9.56</td>
<td>81.37 / 118.21 / 35.55</td>
<td>420.32 / 394.82 / 704.67</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>168.99 / 153.48 / 10.69</td>
<td>118.40 / 187.56 / 38.43</td>
<td>171.61 / 255.10 / 170.93</td>
<td>416.73 / 455.77 / 666.76</td>
</tr>
<tr>
<td>32</td>
<td>311.93 / 305.69 / 11.64</td>
<td>193.35 / 333.99 / 40.83</td>
<td>226.70 / 400.66 / 173.87</td>
<td>436.93 / 593.36 / 609.31</td>
</tr>
<tr>
<td>64</td>
<td>602.80 / 596.64 / 12.31</td>
<td>338.55 / 619.40 / 42.38</td>
<td>327.82 / 691.51 / 178.78</td>
<td>509.94 / 975.92 / 583.62</td>
</tr>
<tr>
<td>128</td>
<td>1174.60 / 1167.76 / 12.73</td>
<td>634.60 / 1201.76 / 43.60</td>
<td>517.68 / 1278.14 / 201.68</td>
<td>669.82 / 1451.92 / 555.65</td>
</tr>
<tr>
<td>256</td>
<td>2318.94 / 2311.60 / 13.01</td>
<td>1212.94 / 2355.97 / 43.40</td>
<td>882.48 / 2445.74 / 192.52</td>
<td>1008.96 / 2614.96 / 542.30</td>
</tr>
<tr>
<td>512</td>
<td>4609.30 / 4597.86 / 16.65</td>
<td>2373.32 / 4657.31 / 48.76</td>
<td>1506.65 / 4766.71 / 211.46</td>
<td>1680.89 / 4928.65 / 537.90</td>
</tr>
</tbody>
</table>

Performance is measured in time (seconds): time with gettimeofday() / user time with getrusage() / system time with getrusage()

Figure 4.17: List of Measurements on eesun.ece.utep.edu (4 CPU’s, Solaris)

![Figure 4.17: List of Measurements on eesun.ece.utep.edu (4 CPU’s, Solaris)](image)

Figure 4.18: Plots of Measurements with gettimeofday() on eesun.ece.utep.edu (4 CPU’s, Solaris)
Chapter 5

Results and Conclusions

5.1 Final Observations

At this point, we are going to summarize the results and the observations that we obtained from the last chapter. First, the results indicate that the optimal level of concurrency, or the number of threads in a program, is equal to the number of processors in the executing system. An excessive level of concurrency only causes unwanted extra system overhead. Second, the minimum amount of computation required for benefit from the multiple threads is very small. This is a significant difference from the previous work done by distributing tasks into computer hosts over the network [VeI98], where network communication causes large amount of system overhead.

Most importantly, with this software model, the improvement in performance was significant, although the computation time did not become exactly half or quarter of the baseline with two or four threads respectively. We understand that this was due to the system overhead in order to manipulate and synchronize the multiple threads.

Finally, the developed software package was functioning correctly as we expected. The backpropagation neural network was behaving as we expected in image pattern recognition, although some ambiguous decision making occurred. The function calls worked properly as an Application Programming Interface (API), as we proved with our own testing application. Also, the neural network functions were highly configurable and scalable as we desired.

5.2 Future Work

With this work as the baseline, we can suggest several future endeavors. Application-wise, our testing program of letter image recognition seems to motivate a more advanced image identification system. One example of such suggested application could be fingerprint identification. For software modeling of concurrent processing, it may desired to form this as client-server model. Instead of executing multi-threaded neural network function on the local computer host, it could be done on a remote host using Remote Procedure Calls (RPC) [Ste99]. This may benefit those who have a single-processor workstation who desire to access a dedicated execution server with multiple processors over a local area network. Finally, research on a more sophisticated algorithm in partitioning tasks in a neural network computing is highly motivated.
References


[WT95] Wagner, Tom and Towsley, Don. “Getting Started with POSIX Threads”. Department of Computer Science, University of Massachusetts at Amherst, July 1995.

Appendix A

Source Code

A.1 Include File: pnnet.h

/****************************************************************************
 *** pnnet.h: The header file for run_pnnet() function. This file ***
 *** contains necessary data structure in order to call ***
 *** run_pnnet(). ***
 ***
 *** Written by: Kiyoshi Kawaguchi ***
 *** Last update: 03/22/2000 ***
 *** 03/28/2000 ver. 3.6 ***
/****************************************************************************

/** Maximum number of neurons per layer **/
#define PNN_N_MAX_NEURONS 512
#define MAXNS PNN_N_MAX_NEURONS /* Don’t erase this, for internal use only */

/** Maximum number of layers **/
#define PNN_N_MAX_LAYERS 5
#define MAXLS PNN_N_MAX_LAYERS /* Don’t erase this, for internal use only */

/** Maximum number of training sample accepted **/
#define PNN_N_MAX_SAMPLES 50
#define MAXSS PNN_N_MAX_SAMPLES /* Don’t erase this, for internal use only */

/** Maximum number of threads (i.e., level of concurrency) **/
#define PNN_N_MAX_THREADS 8

/** These are the definitions of bit setting in execution modes **/
#define PNN_D_QUIET 0x0000 /* No debug mode */
#define PNN_D_INITWT 0x0001 /* Enable displaying initial weights */
#define PNN_D_ITER 0x0002 /* Enable displaying status (iteration #) */
#define PNN_D_SAMP 0x0004 /* Enable displaying status (sample #) */
#define PNN_D_FP 0x0008 /* Enable displaying status (FP layer #) */
#define PNN_D_OUTPUTS 0x0010 /* Enable displaying outputs */
#define PNN_D_MODE 0x0020 /* Enable displaying execution mode */
#define PNN_D_BP 0x0040 /* Enable displaying status (BP layer #) */
#define PNN_D_POSTWT 0x0080 /* Enable displaying final weights */
#define PNN_D_ALL 0x00FF /* All enabled */

/**< Struct of arguments to pass to run_pnnet() ***/
typedef struct {
    int nlayers;  /* Number of layers, including input layer */
    int nneurons[MAXLS];  /* 1-D array of # of neurons */
    float *wt;  /* Pointer to the 3-D array of weight values */
        /* Dimension size by PNN_N_MAX_LAYERS * PNN_N_MAX_NEURONS */
    int samples;  /* Number of samples to teach the neural network */
    float inputs[MAXSS][MAXNS];  /* 2-D array of sample inputs */
    float targets[MAXSS][MAXNS];  /* 2-D array of sample targets */
    float *outputs;  /* Pointer to the array of outputs returned */
    float bias;  /* Bias value for this neural net */
    float eratio;  /* Error ratio */
    float wratio;  /* Weight adjustment ratio */
    int iteration;  /* # of iteration to loop in presenting samples */
    int concurrency;  /* Level of concurrency */
    unsigned int debug_mode;  /* Indicate debug mode during execution */
} pnn_param;

/**< Function Prototype ***/
int run_pnnet(pnn_param *);
void weight_init(float *, int, float, float);

A.2 Initializing Weights: weight_init.c

/****************** weight_init.c: The function that will initialize the array ********************/
/**< of weight values in backpropagation neural networks. The parameters are: pointer to the array of weights, size of that array, and the maximum value of initialized random number. This function returns void. */
/**< Written by: Kiyoshi Kawaguchi  
/**< Electrical and Computer Engineering  
/**< University of Texas at El Paso  
/**< Last update: 09/28/99 for Version 2.0  
/******************

#include <stdlib.h>
#include <stdio.h>
# define RANDOM_MAXIMUM 2147483647

/***************************************************/
/** Function weight_init(): Returns void */
/** weights - pointer to the array of weight values in float */
/** to be returned by reference */
/** size - number of values in array 'weights' */
/** minval - the minimum value of the weight to set */
/** maxval - the maximum value of the weight to set */
/***************************************************/

void weight_init(float *weights, int size, float minval, float maxval)
{
    int i;
    int random_generated;

    for(i=0;i<size;i++) {
        random_generated = random();
        *(weights + i) = (maxval - minval) *
            (float)random_generated / RANDOM_MAXIMUM + minval;
    }

    return;
}

A.3 Parallel Neural Network Function: pnnet.c

/***************************************************/
/** pnnet.c: This file contains the routine for executing the ***/
/** neural network function in multiple concurrency ***/
/** manner. ***/
/** Written by: Kiyoshi Kawaguchi, BSEE ***/
/** Electrical and Computer Engineering ***/
/** University of Texas at El Paso ***/
/** Last update: 03/22/2000 */
/** 03/28/2000 for ver. 3.6 */
/** 04/04/2000 for ver. 3.71d */
/** 05/01/2000 for ver. 3.76 */
/***************************************************/

#include <stdio.h>
#include <math.h>
#include <pthread.h>
#ifdef SOLARIS_THREADS
#include <thread.h>
#endif
#include "pnnet.h"
typedef struct {
    int group;  /* Group # */
    int nlayers; /* # of layers */
    int *nneurons; /* Ptr. to the 1-D array of #'s of neurons */
    float *inputs; /* Ptr. to the 1-D array of sample inputs */
    float *targets; /* Ptr. to the 1-D array of sample targets */
    int sample_index; /* Indicates the index of sample sets */
    float *nvals; /* Ptr. to the 2-D array of neural values */
    float *weights; /* Ptr. to the 3-D array of weight values */
    float *errors; /* Ptr. to the 2-D array of error values */
    float bias; /* Bias value */
    float eratio; /* Error adjustment ratio */
    float wratio; /* Weight adjustment ratio */
    int level_conc; /* Level of concurrency */
    int oonly; /* Flag for output-only mode */
    int mode; /* Output mode */
} subnnet_args;

struct {
    pthread_mutex_t lock;
    pthread_cond_t cond;
    int flg;
} fd[MAXNS], bk[MAXNS];

int a, b;
int run_pnnet(pnn_param *params) {

    /************************************************************
     *** Variable / function definition and declaration
    ************************************************************/

    int i, j, k;
    float nval[PNN_N_MAX_LAYERS][PNN_N_MAX_NEURONS];
    float er[PNN_N_MAX_LAYERS][PNN_N_MAX_NEURONS];
    pthread_t tid[PNN_N_MAX_THREADS];
    subnnet_args th_args[PNN_N_MAX_THREADS];

    void subnnet(subnnet_args *);

    /************************************************************
     *** On Solaris system, set up the level of concurrency
    ************************************************************/

    #ifdef SOLARIS_THREADS
        i = thr_setconcurrency((*params).concurrency + 1);
        if(i) {
            fprintf(stderr,
                "ERROR: Couldn’t not set concurrency level due to error %d.\n",
                i);
            return 1;
        }
    #endif

    /************************************************************
     *** Verify the number of neurons in each layer
    ************************************************************/

    j = 0;
    for(i=0; i<(*params).nlayers ; i++) {
        if(((*params).nneurons[i] % (*params).concurrency)) {
            fprintf(stderr,
                "** Number of neurons in layer #%d is not divisible by 4.\n",
                i);
            j = 1;
        }
    }
    if(j) return 1;

    /************************************************************
     *** Display the information about the neural network to be
**executed. (# layers, # neurons / layer, & mode)**

```c
printf("There are %d layers in this network.\n", (*params).nlayers);
printf("There are ");
for(i=0; i<(*params).nlayers; i++)
    printf("%d ", (*params).nneurons[i]);
printf("neurons in each layer respectively.\n");

if((*params).debug_mode & PNN_D_MODE) {
    if((*params).iteration==1 && (*params).samples==1)
        printf("Output-only mode. Will not train.\n");
    else
        printf("Execute in training mode.\n");
}
```

**We assume that the user / programmer who calls this function**
**has already initialized the weight values, so we begin with**
**accepting the input values of the first sample entry.**

```c
if((*params).debug_mode & PNN_D_INITWT) {
    printf("### WEIGHT VERIFICATION ###\n");
    for(i=1; i<(*params).nlayers; i++) {
        printf("==> In layer #%d (%d neurons)\n",
                (*params).nlayers, (*params).nneurons[i]);
        for(j=0; j<(*params).nneurons[i]; j++) {
            printf("Into neuron #%d\n", j);
            for(k=0; k<(*params).nneurons[i-1]; k++) {
                printf("%3d%6.3f ", k,
                        (*params).wt + i*MAXNS*MAXNS + j*MAXNS + k);
                if(k%5 == 4) putchar('
');
            }
            putchar('
');
        }
    }
}
```

**Set up the loops in two dimensions.**

```c
for(a=0; a<(*params).iteration; a++) {
    if((*params).debug_mode & PNN_D_ITER) {
        if(!a%50) printf("\n===> Iteration #%d / %d: \n", a,
```
for(b=0; b<(*params).samples; b++) {
    if((*params).debug_mode & PNN_D_SAMP) {
        if(!(a%50)) printf("---> Sample #%d processing...
", b);
    }
}

/**************************** **********************************************************************
*** Now create threads, let each thread to perform partial *** forward and backward propagation.
**************************************************************************************************/
/* Initialize flags */
for(i=0; i<(*params).nlayers; i++) {
    fd[i].flg = 0; bk[i].flg = 0;
    pthread_mutex_init(&fd[i].lock, NULL);
    pthread_mutex_init(&bk[i].lock, NULL);
    pthread_cond_init(&fd[i].cond, NULL);
    pthread_cond_init(&bk[i].cond, NULL);
}
/* Create threads */
for(i=0; i<(*params).concurrency; i++) {
    th_args[i].group = i;
    th_args[i].nlayers = (*params).nlayers;
    th_args[i].inputs = &((*params).inputs[b][0]);
    th_args[i].targets = &((*params).targets[b][0]);
    th_args[i].sample_index = b;
    th_args[i].nvals = &nval[0][0];
    th_args[i].weights = (*params).wt;
    th_args[i].errors = &er[0][0];
    th_args[i].nneurons = (*params).nneurons;
    th_args[i].level_conc = (*params).concurrency;
    th_args[i].bias = (*params).bias;
    th_args[i].eratio = (*params).eratio;
    th_args[i].wratio = (*params).wratio;
    if(((*params).iteration==1 && (*params).samples==1)
        th_args[i].oonly = 1;
    else
    th_args[i].oonly = 0;
    th_args[i].mode = (*params).debug_mode;
    j = pthread_create(&tid[i], NULL, (void *(*)(void *))&subnnet,
                    &th_args[i]);
    if(j) {
        fprintf(stderr,
            "ERROR: Thread creation failed due to error %d.\n",
            i);
        return 1;
    }
}
/* Wait for every thread to finish its partial neural calculation */
for(i=0; i<(*params).concurrency; i++) pthread_join(tid[i], NULL);

/***************************************************************
*** If output-only mode (ie, both # samples and iteration are 1) is
*** enable, return the output values by reference.
***************************************************************/
if((*params).iteration==1 && (*params).samples==1) {
    for(i=0; i<(*params).nneurons[(*params).nlayers-1]; i++)
        (*params).outputs = &nval[(*params).nlayers-1][0];
    goto done;
}

/***************************************************************
*** @@@@ End of loops with indices a and b @@@@@
***************************************************************
} }
}
done:

/***************************************************************
*** Display post-training state of weight values
***************************************************************
if((*params).debug_mode & PNN_D_POSTWT) {
    printf("### These are weights after training ###\n");
    for(i=2; i<(*params).nlayers; i++) {
        printf("==> In layer #%d (%d neurons)\n", i,
               (*params).nneurons[i]);
        for(j=0; j<(*params).nneurons[i]; j++) {
            printf("Into neuron #d\n", j);
            for(k=0; k<(*params).nneurons[i-1]; k++) {
                printf("%3d%6.3f \n", k,
                       *((*params).wt + i*MAXNS*MAXNS + j*MAXNS + k));
                if(k%5 == 4) putchar(\'\n\');
            }
            putchar(\'\n\');
        }
    }
}

return 0;

/***************************************************************/
/************************************** ****** ****** ****** ****** ****** **
*** Function subnnet(): The section where a thread executes partial
*** forward propagation.
************************************** ****** ****** ****** ****** ****** **/

void subnnet(subnnet_args *params)
{
    int i, j, subnneurons;
    void fp_layer(float *, float *, float *, float, int, int, int);
    float errcalc_out(float, float, float);
    float errcalc_hidden(float, float *, float *, int, int, float);
    void weight_adj(float *, float *, int, float, float);

    /*============================*
    *=== Process the input layer
    *============================*/

    /*--- Find out how many neurons to take care of ---
    --- in this layer ---*/
    subnneurons = *((*params).nneurons) / (*params).level_conc;

    /*--- For those input neurons to be responsible for, ---
    --- pass input values ---*/
    for(i=0; i<subnneurons; i++)
        *((*params).nvals + (*params).group * subnneurons + i) =
            ((*params).inputs + (*params).group * subnneurons + i);

    /*--- For sync. with other threads ---*/
    pthread_mutex_lock(&fd[0].lock);
    fd[0].flg++;
    if(fd[0].flg != (*params).level_conc) {
        pthread_mutex_unlock(&fd[0].lock);
        pthread_mutex_lock(&fd[0].lock);
        while(fd[0].flg != (*params).level_conc)
            pthread_cond_wait(&fd[0].cond, &fd[0].lock);
        pthread_mutex_unlock(&fd[0].lock);
    } else {
        if((*params).level_conc != 1)
            pthread_cond_broadcast(&fd[0].cond);
        pthread_mutex_unlock(&fd[0].lock);
    }

    /*======================================= ==*
    *=== Process the hidden and output layers
    *======================================= ==*/

    for(i=1; i<(*params).nlayers; i++) {
        /*----------------------------------- ------ ------ --*/
        pthread_mutex_lock(&fd[0].lock);
        fd[0].flg++;
        if(fd[0].flg != (*params).level_conc)
            { /*----------------------------------- ------ ------ --*/
                pthread_mutex_unlock(&fd[0].lock);
                pthread_mutex_lock(&fd[0].lock);
                while(fd[0].flg != (*params).level_conc)
                    pthread_cond_wait(&fd[0].cond, &fd[0].lock);
                pthread_mutex_unlock(&fd[0].lock);
            } else {
                if((*params).level_conc != 1)
                    pthread_cond_broadcast(&fd[0].cond);
                pthread_mutex_unlock(&fd[0].lock);
            }

        /*======================================= ==*
        *=== Process the hidden and output layers
        *======================================= ==*/
    }
}
```c
*--- Figure out how many neurons to take care of
*--- in this layer
*-----------------------------------------------*
subnneurons = *((*params).nneurons + i) / (*params).level_conc;

/*-----------------------------------------------*
*--- Perform partial FP in this layer
*-----------------------------------------------*/
fp_layer(
(*params).nvals + (i-1)*MAXNS,
(*params).weights + i*MAXNS*MAXNS +
(*params).group * subnneurons * MAXNS,
(*params).nvals + i*MAXNS + (*params).group * subnneurons,
(*params).bias,
*((*params).nneurons + i - 1),
subnneurons, 0);

/*-----------------------------------------------*
*--- Synchronize with other threads once done
*-----------------------------------------------*/
pthread_mutex_lock(&fd[i].lock);
fd[i].flg++;
if(fd[i].flg != (*params).level_conc)
    pthread_mutex_lock(&fd[i].lock);
    while(fd[i].flg != (*params).level_conc)
        pthread_cond_wait(&fd[i].cond, &fd[i].lock);
    pthread_mutex_unlock(&fd[i].lock);
else {
    if( (*params).level_conc != 1)
        pthread_cond_broadcast(&fd[i].cond);
    pthread_mutex_unlock(&fd[i].lock);
}

} /*=== End of processing hidden and output layers ===*/

/*====================================== ==*
*=== Try to display outputs at this point
*====================================== ==*/
if(((a%50)) && (*params).mode & PNN_D_OUTPUTS)) {
    printf("Sample %d Partial outputs: ", (*params).sample_index);
    subnneurons =
*((*params).nneurons + (*params).nlayers - 1)
/ (*params).level_conc;
    for(i = (*params).group * subnneurons;
```
i < ((*params).group + 1) * subnneurons;
    i++)
    printf("[%d]%f ", i,
        *((*params).nvals + ((*params).nlayers - 1) * MAXNS + i));
    putchar(‘\n’);
}

/*======================================================================
*** Return if output-only mode is enabled
/*======================================================================
if((*params).oonly) {
    printf("Output-only mode. Thread ending....\n");
    return;
}

/*======================================================================
*** Begin Backpropagation
/*======================================================================
for(i=(*params).nlayers-1; i>=1; i--) {

/*======================================================================
*** Figure out how many neurons to be responsible for
/*======================================================================
subnneurons = *((*params).nneurons + i) / (*params).level_conc;

/*======================================================================
*** Perform error calc. and weight adj. for every neuron
*** in responsibility
/*======================================================================
for(j = (*params).group * subnneurons;
    j < ((*params).group + 1) * subnneurons;
    j++) {

/*======================================================================
*** Error calculation */
/*======================================================================
if(i == (*params).nlayers-1) {
    *((*params).errors + i * MAXNS + j) =
        errcalc_out(*{(*params).nvals + i * MAXNS + j},
            *((*params).targets + j),
            (*params).eratio);
} else {
    *((*params).errors + i * MAXNS + j) =
        errcalc_hidden(*{(*params).nvals + i * MAXNS + j},
            (*params).errors + (i+1) * MAXNS + j,
            (*params).weights + (i+1) * MAXNS + j,
            *((*params).nneurons + i),
            *((*params).nneurons + i + 1),
            (*params).eratio);
A.4 Other Low-level Functions

A.4.1 Forward Propagation: fp_layer.c

/************************************************************/  
/*** fp_layer.c: fp_layer() is a function that will perform the ***/  
/*** one-layer neural calculation. This function ***/  
/*** accepts the parameters of input array, weight ***/  
/*** array, # of inputs, # of outputs, and bias. ***/  
/*** Returns the output array by address. ***/  
/*** Usage: ***/  
/*** fp_layer(float *inputs, float *weights, ***/  
/*** float *outputs, float bias, ***/  
/*** int, num_inputs, int num_outputs); ***/  
/*** inputs: Pointer to float type array of ***/
```c
#include <stdio.h>
#include "pnnet.h"
#define N_MAX_NEURONS PNN_N_MAX_NEURONS

/***
*** Function fp_layer(): Returns void
*** inputs - pointer to the array of input values going to
*** this layer
*** weights - pointer to the 2-D array of weight values
*** outputs - pointer to the array of output values from
*** this layer
*** bias - bias value
*** num_inputs - # of inputs
*** num_outputs - # of outputs
***
*/

void fp_layer(float *inputs, float *weights, float *outputs, float bias,
              int num_inputs, int num_outputs, int concurrency)
{
    int i; /* Loop counter */
    float perceptron(float *, float *, int, float); /* Neuron function */

    /* Process for every neuron in this layer */
    for(i=0;i<num_outputs;i++) {
        *(outputs + i) = perceptron(inputs,
                                    weights + (i * N_MAX_NEURONS),
                                    num_inputs, bias);
    }

    return;
}
```
A.4.2 Neural Calculation: perceptron.c

#include <math.h>
#include <stdio.h>
#include <stdlib.h>

/**
 *** Function perceptron(): Returns neuron’s output value
 *** inputs - pointer to the array of input values going to
 *** the neuron
 *** weights - pointer to the array of weights associated with inputs
 *** size - number of input values to the neuron
 *** bias - bias for this neuron
 **/
float perceptron(float *inputs, float *weights, int size, float bias)
{
    float sum;
    float outval;
    float sop(float *, float *, int);
    float sigmoid(float);

    /* Perform the sum-of-product calculation of inputs and weights */
    sum = sop(inputs, weights, size);
    #ifdef DEBUG
    printf("Sum = %f ", sum);
    #endif
    /* Apply bias */
    sum = sum + bias;

    /* Then do the activation function, which is sigmoid */
    outval = sigmoid(sum);

    return outval;
}
A.4.3  Sum-of-Product Function: sop.c

/******************************************************************************
/** sop.c: The function that will perform sum-of-product calculation from the given two sets of arrays of floating point values. The function accepts three parameters, which are pointers to each array and number of pairs to multiply and accumulate. Returns floating point of result. */
/** function usage: */
/** float sop(float *first, float *second, int size); */
/** first: base address of the first array */
/** second: base address of the second array */
/** size: number of pairs to multiply then accumulate (must be <= 200) */
/** Written by: Kiyoshi Kawaguchi */
/** Electrical and Computer Engineering */
/** University of Texas at El Paso */
/** Last update: 09/27/99 for version 2.0 of BP-XOR program */
*******************************************************************************/

#include <stdlib.h>

float sop(float *first, float *second, int size)
{
    int i;
    float sum = 0.0;

    /* Perform the sum-of-product calculation here */
    sum = 0.0;
    for(i=0;i<size;i++)
        sum = (*(first + i)) * (*(second + i)) + sum;

    return sum;
}

A.4.4  Sigmoid Function: sigmoid.c

/******************************************************************************
/** sigmoid.c: This code contains the function routine sigmoid() which performs the unipolar sigmoid function for backpropagation neural computation. Accepts the input value x then returns it’s sigmoid value in float. */
/** function usage: */
/** float sigmoid(float x); */
/** x: Input value */
/** Written by: Kiyoshi Kawaguchi */
*******************************************************************************/
#include <math.h>

float sigmoid(float x)
{
    float exp_value;
    float return_value;

    /*** Exponential calculation /***
    exp_value = exp((double) -x);

    /*** Final sigmoid value /***
    return_value = 1 / (1 + exp_value);

    return return_value;
}

## A.4.5 Backpropagation Routines: backprop.c

### Including functions for error calculation and weight adjustments.

#### Usage: See below for the detail of each function routine.

#### Written by: Kiyoshi Kawaguchi

#### Electrical and Computer Engineering

#### University of Texas at El Paso

#### Last update: 01/07/2000

#### 03/12/2000 for minor changes (Ver. 3.03)

### Function errcalc_out(): Calculates err. of neuron in output layer

**Returns error value of a neuron**

```c
float errcalc_out(float output, float target, float ratio)
{
```
float error;

error = ratio * output * (1.0 - output) * (target - output);
#ifdef DEBUG
    printf("Error = %f. ", error);
#endif

return error;
}

/***
 *** Function errcalc_hidden(): Calculates error of a neuron in hidden layer
 *** output - output value of this neuron
 *** err_f - pointer to the error values of neurons in front layer
 *** wt_f - pointer to the weight values connected to front layer
 *** neurons - number of neurons in current layer
 *** neurons_f - number of neurons in front layer
 *** ratio - error ratio
***/

float errcalc_hidden(float output, float *err_f, float *wt_f, int neurons, int neurons_f, float ratio)
{
    int k;
    float sum = 0.0;
    float *error_front;
    float *weight_front;
    float error;

    for(k=0;k<neurons_f;k++) {
        error_front = (float *)(err_f + k);
        weight_front = (float *)(wt_f + k * N_MAX_NEURONS);
        sum = sum + *error_front * *weight_front;
    }

    error = ratio * output * (1.0 - output) * sum;
#ifdef DEBUG
    printf("Error = %f. ", error);
#endif

    return error;
}

/***
 *** Function weight_adj(): Adjusts the weights connected to this neuron
 ***/

void weight_adj(float *wt, float *in, int neurons_b, float err, float ratio)
{
    int k;
    float *ptr_weight, *ptr_input;
for(k=0;k<neurons_b;k++) {
    ptr_weight = (float *)(wt + k);
    ptr_input = (float *)(in + k);
    *ptr_weight = *ptr_weight + ratio * err * *ptr_input;
    #ifdef NNET_DEBUG_X
        if(k<6) printf("%6.3f ", *ptr_weight);
    #endif
}

return;
}
CURRICULUM VITAE

Kiyoshi Kawaguchi was born in Nishinomiya, Hyogo, Japan on August 19, 1973, as the oldest son of Mr. and Mrs. Toshihisa Kawaguchi. He entered Okayama High School, Okayama, Japan in April 1989. He participated in the American Institute for Foreign Study to become a foreign exchange student and to study abroad in the United States in 1991, and graduated from Pettus High School, Pettus, Texas. He entered the University of Texas at El Paso in August 1992, and he received the Bachelor’s Degree in Electrical Engineering in May 1997. He enrolled into the Graduate School of the University of Texas at El Paso to pursue Master’s Degree in Computer Engineering. During his graduate studies, he worked as a Teaching Assistant at the Department of Electrical and Computer Engineering for first two years. During the time between September 1999 and December 1999, he also worked as a part-time software engineer at Productive Data Solutions, LLC (El Paso, Texas), a division of Productive Data Systems, Inc., headquartered in Greenwood Village, Colorado.

Permanent address: 2-3-9 Satonaka-cho
Nishinomiya, Japan 663

This thesis was written and typed by Kiyoshi Kawaguchi using \LaTeX on Linux.