THE NEURAL SHELL: A NEURAL NETWORKS SIMULATOR

A Thesis
Presented in Partial Fulfillment of the Requirements for
the Degree Master of Science in the
Graduate School of the Ohio State University

by

Prakoon Chen, B.S.E.E.

* * * * *

The Ohio State University
1989

Master's Examination Committee:
Professor Stanley C. Ahalt
Professor Ashok Krishnamurthy
Professor Randolph L. Moses

Approved by:

Stanley C. Ahalt
Adviser
Department of Electrical Engineering
To my parents, Chen Yu Jar and Haung Sue Mae.
ACKNOWLEDGMENTS

I would like to express my sincere thanks to my advisor, Professor Stanley Ahalt, for his guidance and support throughout my graduate work. I would also like to thank Professor Ashok Krishnamurthy and Professor Randolph L. Moses for their many insights and for their careful reading of this thesis.

I wish to thanks my friends and fellow students, who have contributed to my thesis and made this project possible. I would also like to thank them for their moral support and encouragement. Special thanks go to Sarvajna Kazi, for his work on the Neural Shell V2.0, Doug Melton for his work on the FSCL net, and Tzyy-Ping Jung for his valuable assistance on image vector quantization research using the Neural Shell V2.1.
VITA

May 3, 1966 . . . . . . . . . . . . . . . . . Born — Bangkok, Thailand

August, 1987 . . . . . . . . . . . . . . . B.S.E.E.,
The Ohio State University,
Columbus, Ohio

September, 1987 – December, 1987 . . Graduate Teaching Assistant,
Department of Electrical Engineering,
The Ohio State University,
Columbus, Ohio

January, 1987 — Present . . . . . Graduate Research Associate,
Department of Electrical Engineering,
The Ohio State University,
Columbus, Ohio

FIELDS OF STUDY

Major Field: Electrical Engineering

Studies in Computer Engineering: Computer Architecture,
Digital Design,
Computer Software, and
VLSI Circuit Design.
# TABLE OF CONTENTS

**ACKNOWLEDGMENTS** iii  
**VITA** iv  
**LIST OF TABLES** viii  
**LIST OF FIGURES** ix

## I. INTRODUCTION 1

## II. OVERVIEW OF NEURAL SHELL 4  
2.1 Objectives of the Neural Shell 4  
2.2 Overview of the Neural Shell 5  
2.3 The Neural Shell Window 8  
2.3.1 The Selection Window 8  
2.3.2 The TTY Window 10  
2.3.3 The Message Window 11  
2.4 The Configuration Window 12  
2.5 The Graphics Display Window 17  
2.5.1 The Selection Window 21  
2.5.2 The Message Window 21  
2.5.3 The Display Window 23  
2.6 File Descriptions 24
III. IMPLEMENTATION OF NEURAL NET ALGORITHMS ON
THE CRAY SUPERCOMPUTER

3.1 The Hopfield Net ........................................ 29
  3.1.1 Learning Phase of the Hopfield Net ............... 30
  3.1.2 Performance Phase of the Hopfield Net .......... 31
  3.1.3 Implementation of the Learning Phase .......... 32
  3.1.4 Implementation of the Performance Phase ....... 33

3.2 The Back-Propagation Net ............................ 36
  3.2.1 The Feed-Forward Process ........................ 37
  3.2.2 The Back-Propagation Process .................... 38
  3.2.3 Implementation of the Feed-Forward Process ..... 41
  3.2.4 Implementation of the Back-Propagation Process 42

3.3 Frequency-Sensitive Competitive Learning Net ........ 44
  3.3.1 The Learning Phase of the FSCL Net ............. 45
  3.3.2 The Performance Phase of the FSCL Net .......... 46
  3.3.3 Implementation of the Learning Phase .......... 48
  3.3.4 Implementation of the Performance Phase ....... 50

3.4 Choice of Loop Control Variables .................. 51

3.5 Benchmarks of the Cray NNSP ........................ 52

IV. VECTOR QUANTIZATION OF IMAGES ................... 61

4.1 Introduction ........................................... 61

4.2 Vector Quantization ................................... 63

4.3 Measures of Fidelity and Data Compression .......... 64

4.4 The Linde-Buzo-Gray Vector Quantization Algorithm .. 65
<table>
<thead>
<tr>
<th></th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Summary of the file requirements of NNSP</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>Description of the Neural Shell Window's facilities</td>
<td>19</td>
</tr>
<tr>
<td>3</td>
<td>Description of the Graphic Display Window's facilities</td>
<td>23</td>
</tr>
<tr>
<td>4</td>
<td>Comparison of the NNSP on the Cray X-MP/28 and Sun 3/60</td>
<td>54</td>
</tr>
<tr>
<td>5</td>
<td>Comparison of performance between the LBG algorithm and FSCL net</td>
<td>69</td>
</tr>
<tr>
<td>6</td>
<td>Comparison of codewords generated by LBG algorithm and FSCL net</td>
<td>70</td>
</tr>
<tr>
<td>7</td>
<td>Comparison of performance between the LBG algorithm and FSCL net</td>
<td>71</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

1. Overview of the Neural Shell. ........................................... 6
2. The Neural Shell Window. ............................................... 9
3. The Configuration Window for the back-propagation net. .......... 13
4. Variation of the sigmoid function. .................................. 15
5. Using the save button. .................................................. 16
6. Sending commands to a Cray supercomputer. ...................... 18
7. The Graphics Display Window. ....................................... 20
8. The Configuration Window of the Graphics Display. ............. 22
9. The Neural Shell self documentation. ................................ 28
10. A typical node and the hard-limit threshold function of the Hopfield net. .................................................... 56
11. Execution of the vectorized code for learning phase of the Hopfield net. .................................................... 57
12. Execution of the nonvectorized Solution II. ....................... 58
13. Vertical summing applied to the computation of array $sum[j]$. 59
14. Execution of the vectorized Solution I. .......................... 60
15. Original image, 500 x 482 pixels, 8bits/pixel. .................. 72
16. Restored image using LBG algorithm (top) and FSCL net (bottom) at a bit rate of 0.5 bits/pixel. ......................... 73
17 Restored image using LBG algorithm (top) and FSCL net (bottom) at a bit rate of 1.0 bits/pixel. ............................................. 74
18 Restored image using LBG algorithm (top) and FSCL net (bottom) at a bit rate of 1.5 bits/pixel. ................................. 75
19 Original pixel values in the first scan line ............................. 76
20 Comparison of the pixel values in the first scan line. .................. 77
21 Original image, 500 x 482 pixels, 8 bits/pixel. .......................... 78
22 Restored image using LBG algorithm (top), and FSCL net with \( \mathcal{F}(u_i) = u_i \) (bottom). ............................................. 79
23 Restored image using FSCL net with \( \mathcal{F}(u_i) = u_i^{\frac{1}{2}} \) (top), and \( \mathcal{F}(u_i) = u_i^{\beta e^{-t/T}} \) (bottom). ......................... 80
CHAPTER I

INTRODUCTION

Neural networks are dense interconnections of simple processors, or nodes, which operate in parallel. They can be viewed as an effort to approximate human performance in complex tasks such as pattern recognition. In recent years, there has been an increase of interest in neural nets [1,2,3,4]. This is mainly due to the development of new neural network algorithms and the realization that neural nets have features which conventional von-Neumann computers lack.

In general, there are three major advantages that neural networks have over conventional computers. First, neural nets, massively interconnected and with numerous processors, are fault tolerant. Failures in the nodes or connections will not cause a catastrophic failure of the entire network. Second, neural nets do not require special programming for specific applications. Using general learning rules, neural nets can be trained by repetitively adapting to the given task. This adaptive ability is essential since in some applications such as pattern recognition, an algorithm to recognize arbitrary patterns can be quite complex. Third, neural nets operate in parallel. If neural networks are realized in hardware, many real-time applications are possible.

Because of the recent interest in neural networks and the need to understand these neural network algorithms, simulators for neural networks have been developed on diverse computing engines, ranging from special-purpose architectures
(neurocomputers) [5,6] to conventional computers. One such example of the simulator is the Rochester Connectionist Simulator [7,8]. The strength of the Rochester Connectionist Simulator lies in its flexibility, which allows arbitrary connections, activation functions, and data to be associated with each network node. The network is constructed by writing C code, which uses primitives, implemented as C library functions, to create nodes, name nodes, etc. Special functions, which are not provided in the library, can be written and incorporated by users. The C code is compiled and used for simulation. Nevertheless, this flexibility is achieved at the cost of time and memory. Each node in the network is simulated by a separate function call, and a generalized data structure must be provided so that arbitrary data structures can be represented.

Our simulator, the Neural Shell V2.1, takes a different approach. Rather than providing all of the flexibility required to construct an arbitrary network, the Neural Shell has a set of neural network simulation programs which implement algorithms such as the Hopfield net, the back-propagation net, and others. Although users cannot specify different computation rules for each node, they can specify the number of layers, number of nodes, learning rate, and other parameters. This reduced flexibility results in increased simulation speed.

Since simulating neural net algorithms in software on conventional computers is computationally intensive, we have investigated the use of the vector architecture of a Cray supercomputer. The simulation of neural networks on such an architecture is especially appropriate since many neural net algorithms can be viewed as consisting primarily of matrix multiplications.

This thesis describes our work in developing the neural network simulator, Neural Shell V2.1. In Chapter II, an overview of the Neural Shell is given. The different facilities and features of the simulator are discussed. Chapter III de-
scribes the implementation of the Neural Network Simulation Programs (NNSP) on the Cray supercomputer. Various issues relating to improving the simulation performance by code vectorization are discussed. Benchmark results of the NNSP obtained on the Sun 3/60 and the Cray XMP/28 are given. In Chapter IV, the Neural Shell is applied to the task of vector quantization. Vector quantization is introduced, and a study comparing the performance of the LBG algorithm and Frequency-Sensitive Competitive Learning (FSCL) net on image vector quantization is conducted. Chapter V concludes the thesis and suggests future research directions and enhancements for the Neural Shell.
CHAPTER II
OVERVIEW OF NEURAL SHELL

In this chapter, we give an overview of the Neural Shell. We begin by discussing the objectives behind the design and development of the Shell. Various features and facilities supported by the Neural Shell are discussed in detail. Examples are provided to illustrate the procedures concerned with the construction and simulation of neural networks using the Neural Shell. Finally, the file requirements for different neural network simulation programs are discussed.

2.1 Objectives of the Neural Shell

There were three major objectives in the design and development of the Neural Shell.

(1) The Neural Shell was designed to provide users with a rapid-prototyping environment for the simulation of neural net algorithms. It uses the mouse, simple keyboard sequences, and pop-up windows to simplify the construction of neural networks.

(2) Since many neural net algorithms are computationally intensive, the Neural Shell allows users to simulate large problems on the Cray supercomputer. With this facility, experimental neural networks can be prototyped on Sun workstations and then scaled to larger sizes or to more extensive training phases, and executed on the Cray supercomputer.
Finally, the Neural Shell was designed to encourage users to use it as a tool for experimentation and research into neural networks. Variable parameters such as the learning rate, threshold function, and the net configuration can be easily altered. This flexibility promotes increased use of neural networks, allows new applications to be found for neural networks, and promotes the development of better neural net algorithms.

2.2 Overview of the Neural Shell

The Neural Shell currently supports the following neural network algorithms: Hopfield, Hamming, back-propagation (BP), Kohonen Self-organizing Feature Maps (KSFM), and Frequency-Sensitive Competitive Learning (FSCL). The FSCL net was developed at The Ohio State University Department of Electrical Engineering [2].

The Neural Shell consists of a simulation window called the Neural Shell Window, a graphics-oriented window program called the Graphics Display Window, and nine independent, executable, text mode programs, which we collectively call the Neural Network Simulation Programs (NNSP). The Neural Shell Window and the Graphic Display Window can be viewed as an interface layer between users and the NNSP. Figure 1 illustrates an overview of the Neural Shell.

The Neural Shell Window is an interactive window which provides an interface for the construction and the simulation of neural network algorithms. During the construction of a neural net, users specify the neural network to be simulated, and select the appropriate simulation phase (learning phase or performance phase). A pop-up window then appears. This window allows users to modify and adjust the appropriate net parameters, which include the number of nodes, number of layers, learning rate, etc., and specify the NNSP's input and output files. When all of
Figure 1: Overview of the Neural Shell.
the parameters and file names for the simulation have been specified, the Neural Shell Window checks for inconsistencies in the inputs. If no errors are detected, the Neural Shell Window delegates the simulation task to the appropriate NNSP program, which performs the simulation. We discuss the Neural Shell Window in more detail in Section 2.3.

The Graphics Display Window is an interactive graphics display facility which can be called from the Neural Shell Window. It allows users to graphically view the interconnection weight matrices and results generated during neural net simulations. This facility helps users to intuitively understand the results obtained from running specific networks. Refer to Section 2.5 for more details.

The NNSPs are the core of the Neural Shell, since the collection of programs perform all of the computation tasks. The amount of computation associated with training a neural net can be quite extensive. This occurs for two primary reasons. First, neural networks are highly interconnected and, thus, the computation can grow exponentially with the size of the net. Second, some networks require extensive training in order to "learn" certain aspects of the input data. In order to provide fast turn-around when the user is training and prototyping new neural networks, there are two versions of the NNSP: a nonvectorized version which runs on the Sun workstations and a vectorized version for the Cray supercomputer. The latter allows large neural networks to be simulated with a substantial speedup. A vectorized version of the KSFM net is currently not included in the NNSP. In the future, it will be implemented and included in the Cray version of the NNSP.

The programs of the NNSP are divided into five groups, each of which simulates one of the five neural net algorithms currently supported by the Neural Shell. Furthermore, each neural net algorithm is separated into two programs, corresponding to the two phases of the algorithms, the learning and the perfor-
mance phases. For example, in a Hopfield net, the program which simulates the learning phase is called “ihop”, which stands for Learn HOPfield. Similarly, the performance phase is implemented in the program called “hopfield”. This convention holds for all of the other neural networks except the Hamming net which consists only of one program, “hamming”, due to the nature of the algorithm. Additional neural network algorithms can be added to the Neural Shell with few modifications.

2.3 The Neural Shell Window

The Neural Shell Window consists of three sub-windows. These are called the Selection Window, the TTY Window, and the Message Window. When the Neural Shell Window is first executed, a window similar to the one shown in Figure 2 appears in the SunTools environment. In the next three sub-sections, we describe each of the sub-windows of the Neural Shell Window.

2.3.1 The Selection Window

The Selection Window, located at the top of the Neural Shell Window, is the primary sub-window. In this window, users can select from the available neural net algorithms and can execute all of the simulation commands. As an illustration, in order to construct a back-propagation net, the cursor is used to select the back-propagation net from the menu located on the top right corner of the Selection Window. If the right mouse button is depressed, a list of all the neural networks algorithms appears, as shown in Figure 2. Dragging the mouse down the list until item Back-Propagation Net is highlighted, and releasing the right mouse button selects this net. This procedure should be familiar to those who work with SunTools,
Figure 2: The Neural Shell Window.
since it is the common method of invoking a menu and selecting an item from the menu.

In addition, associated with this sub-window are four other buttons. These are the Learn, Perform, Display, and Quit buttons. The Learn and the Perform buttons correspond to the two different phases of the neural net algorithms. These two buttons can only be selected after a particular neural net algorithm has been chosen, since they imply that the learning or performance phase of the selected net is to commence. As an example, to select the learning phase of the back-propagation net, we move the cursor over the Learn button, and depress the left mouse button. A pop-up Configuration Window, containing only the parameters which are appropriate for the learning phase appears. This window allows users to modify the net’s parameters, and is discussed in Section 2.4.

The Display button is used to call up the Graphics Display Window, and the Quit button terminates (exits) the Neural Shell.

Finally, the item Exec Dir:, appearing on the top-left corner of the Selection Window, is used to specify the directory where the NNSPs are located. This provides the flexibility of storing various versions of NNSP in arbitrary directories. The default value is set to the directory of the C-shell window in which the Neural Shell is called. In Section 2.4, we demonstrate the use of this item.

In the next section, we discuss the TTY Window of the Neural Shell Window.

2.3.2 The TTY Window

The second sub-window, the TTY Window, is located in the middle region of the Neural Shell Window. This is a normal shell window which can be used to execute any Unix command. To facilitate its use in a networked environment, a menu is provided to allow remote login, telnet, and ftp to the other machines.
connected to the network. For example, to telnet to the Ohio Supercomputer Center Cray, the right mouse button is first depressed in the TTY Window. A menu appears, listing all of the facilities for accessing the network. Dragging the mouse down until the item telnet is highlighted, and moving the cursor to the right causes a second menu to appear. This second menu, which can be customized by users, lists the names of frequently used remote machines. In this case, we select the item Cray X-MP/28 to access the Ohio Supercomputer Center Cray X-MP/28.

When the TTY Window is connected to a remote machine, all commands passed to this window are executed on the remote machine. Utilizing this technique, the Neural Shell Window can pass line commands to other remote machines. For example, if the user telnets to a Cray supercomputer through the TTY Window, subsequent neural network simulations take place on the Cray rather than on the local Sun (assuming that the appropriate versions of the NNSP reside on the Cray). This allows the Shell to use the graphical interface of the Sun, and at the same time allows use the computation power of other machines such as the Cray. Section 2.4 further clarifies these ideas.

2.3.3 The Message Window

The third sub-window, the Message Window, is located below the TTY Window and is used by the Neural Shell to display various messages. For example the Neural Shell generates an error message when it detects any inconsistencies in the parameters specified by the user. Other messages informs the user of the operations currently being executed by the Neural Shell. Because most messages are brief, the Message sub-window provides room for only four lines. A scroll bar is provided for reviewing previous messages.
2.4 The Configuration Window

Once the user selects a particular net and clicks on the Learn or the Perform button (refer to Section 2.3.1), a pop-up window similar to the one shown in Figure 3 appears. This pop-up Configuration Window allows the user to alter many of the basic parameters which affect the operation of the neural net. Since different nets require different set of parameters (configurations), the configuration window serves as a customized entry form for each net, showing the parameters which are applicable or appropriate for the type of network selected. For example, the Configuration window of the back-propagation net has a variable sigmoid threshold function, while the Hopfield net has a hard limit threshold function.

To illustrate this idea, in Figure 3 we show the Configuration window of the back-propagation net. The parameters which can be varied are the learning rate, the topology, the number of nodes in each layer, and the threshold function. The learning rate is the rate by which the interconnection weight matrix is changed (how fast the net learns or forgets).

The node topology determines how many nodes are to appear in different layers of the net, and how many layers the net is to consist of. In the example of Figure 3, a net with an input layer of 120 nodes, one hidden layer of 30 nodes, and 120 output nodes is being constructed. The user supplies integers which specify the number of nodes in each successive layer, starting with the input layer, then the hidden layers, and finally the output layer. Although in this example there are three layers of nodes, arbitrary numbers of layers of nodes can be simulated simply by adding additional numbers to the list. As an example, to simulate 4 layers of nodes with 120 inputs, 30 outputs, and two hidden layers of 20 and 10 nodes, the node configuration would be “120 x 20 x 10 x 30”.

12
Figure 3: The Configuration Window for the back-propagation net.
Table 1: Summary of the file requirements of NNSP

<table>
<thead>
<tr>
<th>Programs</th>
<th>.cfg</th>
<th>.xmp</th>
<th>.wgt</th>
<th>.dat</th>
<th>.rst</th>
<th>.mov</th>
<th>.rasters</th>
<th>.img</th>
<th>.nbr</th>
</tr>
</thead>
<tbody>
<tr>
<td>lhop</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hopfield</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hamming</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lbp</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bp</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lksfm</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>ksfm</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>lfscl</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>fscl</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

NOTE: *We have the option of choosing one of these files. For lfscl, we could choose the training file to be .xmp, .rasters, or .img. For fscl, we could choose the .dat, .rasters, or .img file as the training file.

For the threshold function (a sigmoid function), the user is allowed to vary the magnitude, the slope, and both the horizontal and vertical shift. Figure 4 demonstrates some of the effects of varying these four parameters.

The Configuration Window expects that the user supplies 1) the name of the directory where all of the input and output files appropriate to a particular net are located, and 2) the names of these files. This appears at the bottom portion of the Configuration Window. This facility serves as a reminder to the user, since during simulations, the learning and performance phases require slightly different sets of files. The list of files needed by each of the NNSP programs is shown in Table 1.

In addition, the Configuration Window has three buttons associated with it. They are labeled Save, Read, and Done. The Save button is used to save the configuration parameters (discussed above) into a file called the configuration file (a file with a .cfg extension). During a simulation, the NNSPs read this file to determine the configuration of the net. Since the Neural Shell does not currently
Figure 4: Variation of the sigmoid function.
auto-save changes, this must be done by the user after altering the parameters of a particular neural net. Failure to save these alterations results in a simulation which does not reflect the changes. In Figure 5, we illustrate the concept of saving the configuration to a .cfg file. Notice that all of the commands begin with a capital letter. The Read button (opposite of Save) is used to read the parameters from a configuration file and display them in the Configuration Window.

Finally, the Done button is used when the user is satisfied with the configuration of the net which is being designed and wishes to proceed to the learning or the performance phase. When the Done button is clicked, the Neural Shell Window checks for any inconsistencies in the parameters or the file names specified by the user. If no errors are detected, the appropriate NNSP program name and the file names are concatenated as a command line, which is passed to the TTY
Window. For the example shown in Figure 3, the Neural Shell Window concatenates "/u7/chen/neural_shellV2.1/sun_exec" (the directory where the NNSPs are located) specified by the item Exec Dir, "lbp" (the NNSP program for the learning phase of the back-propagation net), and the file names specified in the Configuration Window to form the line command, "/u7/chen/neural_shellV2.1/sun_exec/lbp t.bp.cfg t.bp.xmp t.bp.wgt 1000", which is output to the TTY Window. As discussed in Section 2.3.2, if the TTY Window is connected to a remote computer, such as a Cray, as discussed in Section 2.3.2, this command line is sent to the remote computer and executed. Figure 6 illustrates how the command is sent to the Cray Super Computer.

A complete listing of all the facilities provided by the Neural Shell is given in Table 2.

2.5 The Graphics Display Window

The Graphics Displays Window is a facility which allows users to view the interconnection weight matrices or the results generated by the Neural Shell in graphical forms. For instance, one useful function of the Graphics Display Window is to allow the user to validate or check results graphically, especially with image data. By displaying the results in a graphical form, the Graphics Display Window user can gain insight into how a particular net should be designed, or how it is performing. The Graphics Display Window, like the Neural Shell window, is divided into three sub-windows: the Selection Window, the Message Window, and the Display Window (see Figure 7). These sub-windows are discussed in the next three sub-sections.
1. Click the "Done" button.
2. Neural Shell checks for command errors before passing them to the TTY Window.
3. The command is sent to the Cray through the network and then executed.

Figure 6: Sending commands to a Cray supercomputer.
Table 2: Description of the Neural Shell Window’s facilities

<table>
<thead>
<tr>
<th>Feature</th>
<th>Hopfield</th>
<th>Hamming</th>
<th>Back-Prop.</th>
<th>FSCL</th>
<th>KSFN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution on Sun</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Execution on Cray</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>n/a</td>
</tr>
<tr>
<td>Arbitrary Nodes</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Arbitrary Layers</td>
<td>n/a</td>
<td>n/a</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Var. Learn. Rates</td>
<td>n/a</td>
<td>n/a</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Var. Thres. Func.</td>
<td>✓</td>
<td>n/a</td>
<td>✓</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Self Feedback</td>
<td>✓</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Dot Product</td>
<td>n/a</td>
<td>n/a</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Euclidean Dist.</td>
<td>n/a</td>
<td>n/a</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Self Documenting</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Global History</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Wgt. Vec. Saved</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Trace Mov. of Wgt.</td>
<td>n/a</td>
<td>n/a</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Process Image File</td>
<td>n/a</td>
<td>n/a</td>
<td>✓</td>
<td>✓</td>
<td>n/a</td>
</tr>
<tr>
<td>Pop Up Cfg. Win.</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Read/Save Cfg.</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Error Checking</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Error Reporting</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Menu - ftp, tel, rlogin, etc.</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

n/a – Not Applicable.

1. Hopfield and Hamming nets are single and double layers respectively. KSFN and FSCN net are two or three layer nets. The back-propagation net has an unlimited number of layers.

2. User can vary the Summation Threshold Function. For example:- back-propagation net, Adjustable sigmoid Function.

3. User can choose between Dot Product or Euclidean Distance for selecting the winner node.

4. When a file such as Weight, Result or Movement file is created by any neural net, the history of that file is automatically documented to the file. Similarly, a global file called “neural.trace” keeps track of all the executions done by any net.
Figure 7: The Graphics Display Window.
2.5.1 The Selection Window

At the top of the Graphics Display Window is the Selection Window. It is used for specifying the directory and the name of the file to be displayed. In addition, it contains various command buttons. Eight buttons: Display, Config, Forward, Rewind, Magnify, Shrink, Dump and Quit are used to manipulate the file being displayed. Most are self explanatory and refer to forwarding to the next frame, rewinding to the previous frame, magnifying a frame, shrinking a frame, dumping a frame to hardcopy, and quitting the Graphics Display. Here, a frame is defined as a rectilinear group of data. Only the Config button requires further explanation.

When the Config button is selected, a pop-up Configuration Window for the Graphics Display appears as shown in Figure 8. Much like the Configuration Window in the Neural Shell window, this pop-up window is used to specify how a frame should be displayed, specifically, the number of elements (pixels) per row or column, the size of the pixels, and the scale of magnification are chosen. For example, to display the pattern 0 as shown in Figure 7, it is necessary to specify that Num of Col be 10 and the Num of Row be 12. This is because the patterns being used in this problem were assumed to consist of frames of 12 rows, with 10 pixels per row (for a total of 120 elements). Obviously, attempting to display this pattern in another format causes the pattern to be distorted.

2.5.2 The Message Window

The Message Window, the middle sub-window, is used by the Graphics Display window to display various messages. For example, it is used to display error messages, the number of the frame being displayed, and the minimum and maximum values which occur in the file being displayed.
Figure 8: The Configuration Window of the Graphics Display.
Table 3: Description of the Graphic Display Window's facilities

<table>
<thead>
<tr>
<th>Features</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display Files</td>
<td>Display .wgt, .dat, .xmp, .rst files (numbers only)</td>
</tr>
<tr>
<td>Magnify Frame</td>
<td>Magnified a frame</td>
</tr>
<tr>
<td>Shrink Frame</td>
<td>Shrink a frame</td>
</tr>
<tr>
<td>Set Pixel Size</td>
<td>Use to set the size of a pixel</td>
</tr>
<tr>
<td>Forward Display</td>
<td>Display next frames</td>
</tr>
<tr>
<td>Backward Display</td>
<td>Display previous frames</td>
</tr>
<tr>
<td>Var. Shading</td>
<td>Vary shading with strength of the data point</td>
</tr>
<tr>
<td>Num. Pixels Horz.</td>
<td>Number of pixels in Horz. dimension of a frame</td>
</tr>
<tr>
<td>Num. Pixels Vert.</td>
<td>Number of Pixels in Vert dimension of a frame</td>
</tr>
<tr>
<td>Scroll Bars</td>
<td>Help view large picture (both Vert. and Horz.)</td>
</tr>
<tr>
<td>Dump Image</td>
<td>Save a frame to a .img file. (later process to .ps file)</td>
</tr>
</tbody>
</table>

NOTE: Multiple Graphic Display Windows can be called simultaneously.

2.5.3 The Display Window

The Display Window (bottom sub-window), is the area actually used for the display. This sub-window has both horizontal and vertical scrollbars so that a picture larger than the displaying window can be viewed. Associated with the Display sub-window is a menu which can be used to manipulate binary image files, such as standard raster files. This menu allows users to perform logical AND, OR, and EXOR of the image being displayed and other images stored in files. For example, in image processing applications, the EXOR operation can be used to generate error images by performing a bit-wise exclusive-or operation between the original image and the generated image.

Table 3 describes the facilities provided by the Graphics Display window. The input and outputs files requirements for all the NNSPs are discussed in the next section.
2.6 File Descriptions

There are currently 10 different types of files used by the NNSP. These are the configuration files, exemplar files, weight files, data files, result files, movement files, neighborhood files, connection files, raster files, and image files. These files conventionally have extensions which help to identify them. The file extensions are .cfg, .xmp, .wgt, .dat, .rst, .mov, .nbr, .con, .raster, and .img respectively. A brief description of their functions are given below. For complete details on these files, refer to the Neural Shell V2.1 User's Guide [9].

.cfg The configuration file contains the configuration parameters of a particular net. These include parameters discussed previously such as the learning rate, the topology and configuration of the net, and the parameters which control the shape of the threshold function. Different versions of the configuration file are used by different NNSPs. A configuration file is required for both the learning and the performance phases.

.xmp The exemplar file contains the set of exemplars, that is, the file containing the training data from which the user wishes the neural net to learn. The format of the file varies depends on the type of neural net selected by the user. For those neural networks which require supervised training, such as the back-propagation net, the .xmp file must contain the both the input data and, immediately following each input vector, the desired output vectors. This format results in the interleaving of the input and output vectors. For those nets which require no supervision during the learning phase, such as the Kohonen Self-organizing Feature Map or the Hopfield net, the .xmp file contains only the input vectors.
.wgt The weight file contains the interconnection weight matrix of a particular neural net. Originally, the weight file is either empty or pre-loaded with the initial values the net designer has selected. After the learning phase the weight matrix is saved into the weight file and can then be used in subsequent versions of the net. This means that a network can use previously adapted weights without having to be retrained. In addition, if further training is necessary, the net can use this weight file as a starting point, making it possible to progressively train the neural net.

.dat The data file contains the set of input vectors used during the performance phase. The neural net takes in one vector (the vector length equals to the number of nodes in the input layer of the net) and calculates the appropriate output vector corresponding to the input vector.

.rst The result file contains the output vectors generated by the neural net. Each vector has a corresponding output vector in the .rst file. Therefore, the .dat and .rst files contain an equal number of vectors.

.mov When a movement file is specified, the Neural Shell "traces" the movement of the interconnection weight matrix to this file. A snapshot of the weight matrix is taken after each iteration through the .dat file.

.con The connection file is used to describe the topological connection of neural units in the KSFM net. For example, the neural units might be connected as a square, a line, or another, more complex, topology. Refer to the Neural Shell V2.1 User's Guide for examples [9].

 nbr The neighborhood file describes the neighborhood of the neural units of the KSFM net. It specifies which neural units are at some distance from other
units. The .nbr file can be generated from the .con file using the program “con2nbr”. An example is given in the Neural Shell V2.1 User’s Guide [9].

.raster A .raster file is a standard bitmap file on the Sun. For example, it can be used with a FSCL or KSFM net as input data for the vector quantization of image data.

.img A .img file is an 8 bits/pixel gray scale image file. Like the .rst file, it might be used with the FSCL net for research into vector quantization of imaged data.

In addition to the facilities described in previous sections, the Neural Shell also provides a facility which documents the history of files generated by the Neural Shell. This self documentation facility is discussed in the next section.

2.7 The Self Documentation Facility

During the course of any project, it can be difficult to keep track of the data and files generated by various experiments. Consequently, the Neural Shell provides a self documenting (auto-trace) facility for all of the files which it generates. For example, when a file such as the weight file is generated by the Neural Shell, the information regarding the net which created this file is written into the .wgt file. At the same time, the same information is written to a global history file called the “neural.trace”. The “neural.trace” file thus serves as a backup copy and a log book of the history of all the neural networks executed by the Neural Shell. In Figure 9, we show two neural networks, lhop and lbp, which are undergoing training simultaneously. Assuming that lbp finishes first, it leaves a trace of its execution in the “neural.trace” file. At the same time, it stores the same information to the “t.bp.wgt” file. The “neural.trace” file is then closed. Later when lhop
finishes its execution, it reopens the "neural.trace" file and concatenates its own history to this file. Similarly, lhop writes a copy of the history to the "t.hop.wgt" file.

The information traced by the Neural Shell includes the date, the time, the user executing the net, the files, the configuration of the net, and other operational and performance statistics.
Figure 9: The Neural Shell self documentation.
CHAPTER III

IMPLEMENTATION OF NEURAL NET ALGORITHMS ON THE CRAY SUPERCOMPUTER

In this chapter, we describe the implementation of the Cray version of the Neural Network Simulation Programs (NNSP). We concentrate our discussion on three neural networks algorithms, the Hopfield net, the back-propagation net, and the Frequency-Sensitive Competitive Learning net. After each algorithm’s description, a simplified version of the code (written in C) is shown. The code is explained and various coding techniques which result in code vectorization are discussed.

3.1 The Hopfield Net

This implementation of the Hopfield net is a single layer binary input and binary output net. All nodes in the Hopfield Net are connected by weighted interconnections to other nodes. When an input vector is applied to the Hopfield net, each node transmits its value to all other nodes via these interconnections. The strength of the signal arriving at a destination node is the weight of that link multiplied by strength of the originating output signal. All incoming signals are summed, and the sum is passed to a hard limit threshold function. This algorithm is shown pictorially in Figure 10. The inputs and the outputs of this Hopfield net consist of only two values; +1, representing re-enforcement, and -1, representing inhibition.
The Hopfield net algorithm can be divided into two phases: the learning phase, and the performance phase. Extensive research on this net has been conducted [10,11,12]. A description of the algorithm is given in the next two sections.

3.1.1 Learning Phase of the Hopfield Net

The learning phase of the Hopfield Net begins by applying a training vector. The training vector’s dimension must equal the number of nodes in the Hopfield net. The strength of a particular weight in the Hopfield net is adjusted by taking the product of the outputs of the two nodes that the weighted link connects and adding this product to the previous value of the weight. This process is repeated for every node for each training vector. As an example, if node\(_i\) and node\(_j\) have the value +1 and -1 respectively and the weight \(w_{i,j}\) between them is 6, the strength of \(w_{i,j}\) is reassigned to \(6 + (+1 \times -1) = 5\). The learning phase of the Hopfield net is defined more formally below.

1. Given a training sequence of \(\{x(t); t = 0, ..., n - 1\}\) where \(n\) is the total number of training vectors. Each training vector \(x(t)\) is a row vector with dimension \(k\) and has the form

\[
x(t) = \{x_1, x_2, ..., x_k\}
\]  

(3.1)

2. Initialize all weights \(w_{i,j}\) between node\(_i\) and node\(_j\) to 0.

3. Present a training vector \(x(t)\) to the nodes of the Hopfield net.

4. The weights \(w_{i,j}\) are adjusted as:

\[
w_{i,j}(t) = w_{i,j}(t - 1) + \sum_{j=0}^{k-1} node_i \times node_j
\]  

(3.2)
(5) Repeat steps (3) through step (5) for each training vector. Once the learning phase is completed, the Hopfield net can be used in the performance phase, which is described in the next section.

3.1.2 Performance Phase of the Hopfield Net

Unlike the learning phase, where an input vector is presented only once and used to calculate the weight interconnection, the performance phase of the Hopfield net takes a “noisy” vector, a distorted or corrupted test vector, and iterates until all the outputs converge. The Hopfield net converges when all node values at time \( t - 1 \) equals those at time \( t \). In other words, further iterations no longer change the outputs. When the net converges, the node values of the Hopfield net may or may not represent one of the patterns it has learned. As noted by Hopfield, if many training patterns share common features, there is a tendency for the Hopfield net to converge to some untaught pattern [11]. The algorithm for the performance phase is given below.

(1) Present a noisy pattern \( y(t) \) of the form

\[
y(t) = \{y_1, y_2, \ldots, y_k\}
\]

(3.3)

to the nodes of the Hopfield net.

(2) Compute the new value of node \( i \) by:

\[
node_i(t + 1) = \mathcal{F} \left( \sum_{j=0}^{k-1} w_{i,j} \times node_j(t) \right)
\]

(3.4)

where \( \mathcal{F}(x) \) is some hard-limit threshold, commonly defined as

\[
\mathcal{F}(x) = \begin{cases} 
1 & \text{if } x > 0 \\
-1 & \text{otherwise}
\end{cases}
\]

(3.5)
(3) Repeat steps (2) until all the output values of the nodes converge. As noted, convergence occurs when \(\text{node}_i(t - 1) = \text{node}_i(t)\) for all \(i\).

3.1.3 Implementation of the Learning Phase

Implementation of the learning phase of Hopfield net is straight-forward. This is because each element \(w_{i,j}\) in the weight matrix of the Hopfield net is the summation of all multiplications of \(\text{node}_i\) with \(\text{node}_j\) for each training vectors. A simplified version of the code which implements the algorithm is shown below.

```c
int node[MAX_NUMBER_OF_NODES];
int total_number_of_nodes;
int weight[MAX_NUMBER_OF_NODES][MAX_NUMBER_OF_NODES];
int i,j;
{
   ... LOOP WHILE MORE TRAINING VECTORS */
   while (more_training_vectors)
   {
      READ_IN_ONE_TRAINING_VECTOR();
      /* COMPUTE THE WEIGHT MATRIX -- step (2) */
      for (i = 0; i < total_number_of_nodes; i++)
         for (j = 0; j < total_number_of_nodes; j++)
            weight[i][j] += node[i]*node[j];
   }
   ...
}
```

The above code consists of a `while` loop which iterates until no further training vectors are available. The nested `for` loop computes the weight matrix of the Hopfield net. The inner `for` loop, when compiled on a Cray supercomputer, will vectorize. In this example, switching the order of these two `for` loops will still result in vectorized code. However, in Section 3.1.4, we show another example in which the ordering of these loops is critical to code vectorization.

The execution of the above code could be described as a pipelined multiplication of a scalar, \(\text{node}_i\), which is held constant during the inner loop, and a
vector \textit{node}[j]. The results of the multiplication, also a vector, is then fed to the addition pipeline to be added with the vector \textit{weight}[i][j]. Due to the pipelined architecture of the Cray, which allows multiplication and addition to be performed concurrently through the chaining of different pipes, high computation speeds are obtained. Figure 11 demonstrates how the above code executes.

3.1.4 Implementation of the Performance Phase

The implementation of the performance phase of the Hopfield net is slightly more complex. In this section, we demonstrate how the ordering of the nested \textit{for} loops can determine whether the code can be vectorized. The listing of the code for the performance phase of the Hopfield net is shown below. We refer to this code segment as Solution I.

```c
int node[MAX_NUMBER_OF_NODES];
int total_number_of_nodes;
int weight[MAX_NUMBER_OF_NODES][MAX_NUMBER_OF_NODES];
int sum[MAX_NUMBER_OF_NODES];
int i,j;
    /* SOLUTION I */

READ_IN_ONE_NOISY_INPUT_VECTOR();
converge = FALSE;
while(! converge)
{

    /* SUM ALL MULTIPLICATIONS OF WEIGHTS & NODES */
    for (i = 0; i < total_number_of_nodes; i++)
        for (j = 0; j < total_number_of_nodes; j++)
            sum[j] += node[i] * weight[j][i];

    /* PASS THE SUMS THROUGH THE HARD-LIMIT THRESHOLD */
    for (i = 0; i < total_number_of_nodes; i++)
        if (sum[i] > 0)
            node[i] = 1;
        else
            node[i] = -1;

    DETERMINE_IF_THE_NET_HAS_CONVERGED();
} /* end while */
```

...
This code consists of a while loop which terminates when the Hopfield net converges. The main body of the while loop consists of two parts. The first portion is a nested for loop which performs the summation of the multiplications of the weight[j][i] and node[i]. The second portion performs the task of passing the summation sum[j] through the hard-limit threshold function \( F(x) \). There are two distinct characteristics of this code which warrant discussion. First, the array sum[j] is used to hold the temporary results of the summation for each node before passing them through the threshold function. Second, the summing and the thresholding processes are separated into two independent for loops, even though their loops vary with the same index \( j \). The efficiency of this coding technique might be questioned. But before this aspect of the algorithm is discussed, let us examine how this portion of the algorithm might otherwise be coded. The example below shows how we combine the two parts of the code into a single nested for loop, replacing the array sum[j] with a single variable sum. We call this sample code Solution II.

```c
...  
  /* SOLUTION II */
  /* SUM ALL MULTIPLICATIONS OF WEIGHTS & NODES AND
     PASS THE SUM THROUGH THE HARD-LIMIT THRESHOLD */

  for (i = 0; i < total_number_of_nodes; i++)
  {
    for (j = 0; j < total_number_of_nodes; j++)
      sum += node[j] * weight[i][j];

    if (sum > 0)
      node[i] = 1;
    else
      node[j] = -1;
  }
...  
```

Compared to Solution I, Solution II seems shorter, cleaner, and more efficient. There no longer exists the need to define sum as an array. Furthermore, a reduction
in the loop set-up time can be achieved, since there are only two for loops rather than three, as shown in Solution I. Nevertheless, Solution II, which is shorter and makes more efficient use of memory, does not vectorize. In Solution II, \( \text{sum} \) is computed iteratively. This means that the current iteration of \( \text{sum} \) depends on the previous \( \text{sum} \), and therefore there exists a data dependency between \( \text{sum}(t) \) and \( \text{sum}(t+1) \). On a vector architecture such as the Cray, this results in both the multiplication hardware completing the multiplication and the addition hardware finishing the addition for the current \( \text{sum} \) before the result can be used for the computation of the next \( \text{sum} \). When this occurs, special vector registers are not utilized, and pipelined multiplications and additions can not be performed. The result is substantial speed loss. Figure 12 demonstrates how the code for Solution II will be executed.

If the data dependencies of the summing process are removed, vectorized code results. Solution I demonstrates how this can be achieved at the expense of larger memory requirements and slightly more complicated code. In Solution I, we keep \( \text{sum} \) as an array, \( \text{sum}[j] \). Each value of the array is responsible for keeping the partial sum of each node. Thus \( \text{sum}[j] \) and \( \text{sum}[k], j \neq k \), are independent. That is, there are no data dependencies between elements of the array \( \text{sum} \), and the order in which \( \text{sum}[j] \) or \( \text{sum}[k] \) are calculated does not affect their results. This further implies that if \( \text{sum}[j] \) consists of a summation of \( N \) entries, we can arbitrarily halt the summation of \( \text{sum}[j] \) before entry \( n < N \), begin the computation for \( \text{sum}[k] \), and restart the summation of \( \text{sum}[j] \) without any side effects. A special case of this technique, where \( n = 1 \), is known as vertical summing, since if the summation is expanded, we will be computing the summation in a column-wise rather than row-wise fashion, as in Solution I. Figure 13 illustrates this technique, which can be compared to Figure 12.
In the first portion of the code, we compute the summation for each node while the inner most for loop varies with index \( j \). As discussed above, we sum the first entry of \( \text{sum}[j] \) then skip "down vertically" to sum the first entry of \( \text{sum}[j + 1] \). Since there is no data dependency between \( \text{sum}[j] \) and \( \text{sum}[j + 1] \), the multiplication \( \text{node}[i] \times \text{weight}[j][i] \), \( \text{node}[i] \times \text{weight}[j + 1][i] \), \ldots \) etc. can be done concurrently by chaining the pipelines. As a result, substantial speed-up can be obtained. Figure 14 demonstrates this concept.

The separation of the summing process and the thresholding process into two loops is necessary, since in a vectorizable for loop there cannot be any conditional statements. Thus, in Solution I, the summing process vectorizes, whereas the thresholding process does not. This demonstrates how the non-vectorizable portion of the algorithm can be isolated so that it will not effect the vectorizable portion.

Finally, it is important to note that the ordering of the nested loops in Solution I is crucial to the vectorization of the code. If we were to switch the two loops in the summing process, such that the inner loop is varying with \( i \) rather than \( j \), the code is not vectorizable since we end up with \( \text{sum}[i] = \text{sum}[i] + (\text{node}[i] \times \text{weight}[j][i]) \), a scenario similar to that described in the discussion of Solution II.

### 3.2 The Back-Propagation Net

The back-propagation net is a supervised, continuous valued input and output multi-layer net which utilizes the back-propagation algorithm during the learning phase [13,14]. Each layer of the net is fully connected to its subsequent higher layer. For example, the input layer (the first layer) is connected to the second layer. Similar to the Hopfield net, the back-propagation net can be viewed as consisting of two phases: the learning phase and the performance phase. The learning phase can further be subdivided into two processes: the feed-forward
process and the back-propagation process. During the learning phase, we present an input vector to the net and allows the vector to propagate to the output layer. This is known as the feed-forward process. The output vector generated is then compared with the desired output vector, which is provided by a “teacher”. The back-propagation process, the second process in the learning phase, propagates back the error so that the weight matrix can be adjusted.

The performance phase consists only of the feed-forward process. An input vector is presented and the corresponding output is generated. Both the feed-forward process and the back-propagation process are described below.

3.2.1 The Feed-Forward Process

In the feed-forward process, an input vector, whose dimension must equal the number of nodes in the input layer, is applied. The input vector is fed forward through the weights and is summed up as inputs to the second layer. The outputs of the second layer are determined by passing these sums through a non-linear function, such as the sigmoid function. The entire process is then repeated for subsequent higher layers until the last layer, which is the output layer. The feed-forward process is used for both the learning phase and the performance phase. The feed-forward process is described in more detail below.

1. Given a input sequence of \( \{x(t); t = 0, \ldots, n - 1\} \) where \( n \) is the total number of input vectors. Each input vector \( x(t) \) is a row vector with dimension \( k \) and has the form

   \[
   x(t) = \{x_1, x_2, \ldots, x_k\}
   \]  

   (3.6)

2. Initialize all weights \( w_{ji} \) between node \( j \) in a higher layer and node \( i \) in the lower layer, to some small random value.
(3) Present an input vector $x(t)$ to the input nodes of the back-propagation net.

(4) Perform the feed-forward process, from the input layer to the output layer.

The node values for all layers (except the input layer) are given by

$$node_j = \mathcal{F}(\sum_{i=0}^{N} w_{j,i} \ast node_i)$$  \hspace{1cm} (3.7)

where $N$ is the number of lower layer nodes, and $\mathcal{F}(x)$ is, in our implementation, a generalized sigmoid function given by

$$\mathcal{F}(x) = \frac{\text{Magnitude}}{1 + e^{-\text{Slope}(x-Hshift)}} + V_{\text{shift}}$$  \hspace{1cm} (3.8)

where Magnitude, Slope, Hshift, and Vshift are variables which the user specifies. Magnitude determines the envelope (limits) of the sigmoid function. Slope determines the rate of transition between the two asymptotes of the sigmoid function. Hshift allows the sigmoid function to be shifted horizontally along the $x$-axis. Finally, Vshift shifts the function vertically along $\mathcal{F}(x)$ axis.

During the learning phase of the back-propagation net, the error, generated at the outputs after the feed-forward process, must be propagated back to adjust the weight matrix. This is accomplished by the back-propagation process, which is described in the next section.

3.2.2 The Back-Propagation Process

The back-propagation process updates the weights by back-propagating the errors at each layer to subsequent lower layers. Before a back-propagation process can be performed, a feed-forward process is needed to generate the output vector, which is then compared with the desired output vector. The errors between
these two vectors are propagated back through the net, and used to adjust the weights. The back-propagation process is a relatively complex algorithm which uses derivatives to determine how the weights should be adjusted [13,14]. As a result, substantial learning time can be required. The back-propagation process is described below.

(1) Given an output, \( y(t) \), generated after the feed-forward process and the desired output \( \hat{y}(t) \), compute the change in error \( E \) due to any change in the output unit \( y_j \) for each output node \( j \), as:

\[
\frac{dE}{dy_j} = y_j - \hat{y}_j \tag{3.9}
\]

(2) Compute the change in error, \( E \), with respect to the change in the input, \( x_j \), of each output node.

\[
\frac{dE}{dx_j} = \frac{dE}{dy_j} \frac{dy_j}{dx_j} \tag{3.10}
\]

Note that \( \frac{dE}{dx_j} \) is computed using the chain rule. The value \( \frac{dE}{dy_j} \) is already computed in step (1), thus we need only to compute \( \frac{dy_j}{dx_j} \), where \( \frac{dy_j}{dx_j} = \frac{dF(x_j)}{dx_j} \), since the output \( y_j = F(x_j) \), from the feed-forward process step (4). Thus:

\[
\frac{dy_j}{dx_j} = \frac{dF(x_j)}{dx_j} = \frac{d}{dx_j} \left( \frac{\text{Magnitude}}{1 + e^{-(\text{Slope}(x_j - H_{\text{shift}})} \right) + V_{\text{shift}}} \\
= \frac{\text{Magnitude} \times \text{Slope} \times e^{-(\text{Slope}(x_j - H_{\text{shift}}))}}{(1 + e^{-(\text{Slope}(x_j - H_{\text{shift}}))})^2} \tag{3.11}
\]

The above equation for \( \frac{dy_j}{dx_j} \) is undesirable since it involves \( x_j \), which requires extra memory. In addition, it does not exploit \( y_j \), which is already known from the feed-forward process. Thus, a better way of expressing \( \frac{dy_j}{dx_j} \) is in terms of \( y_j \). Rewriting the above equation of \( \frac{dy_j}{dx_j} \) in terms of \( y_j \) yields:
\[ \frac{dy_j}{dx_j} = \frac{\text{Slope}(y_j - V_{\text{shift}})(\text{Magnitude} - (y_j - V_{\text{shift}}))}{\text{Magnitude}} \] (3.12)

This can be seen by substituting \( y_j \) with \( \frac{\text{Magnitude}}{1 + e^{-\text{Slope}(x_j - H_{\text{shift}})}} + V_{\text{shift}} \). Simplified the equation and we obtain Equation (3.11). It is clear that the second equation, Equation (3.12), for \( \frac{dy_j}{dx_j} \) is simpler and computationally less expensive.

(3) Compute the change in error, \( E \), with respect to change in the weight \( w_{j,i} \) between output node \( j \) and the input node \( i \) by chain rule. Thus:

\[ \frac{dE}{dw_{j,i}} = \frac{dE}{dx_j} \frac{dx_j}{dw_{j,i}} \] (3.13)

The term \( \frac{dx_j}{dw_{j,i}} \) is due to the output, \( y_i \), of node \( i \). Therefore, substituting \( \frac{dx_j}{dw_{j,i}} \) with \( y_i \) gives:

\[ \frac{dE}{dw_{j,i}} = \frac{dE}{dx_j} y_i \] (3.14)

(4) Adjust the weight \( w_{j,i} \) by:

\[ w_{j,i}(t+1) = w_{j,i}(t) - \varepsilon \frac{dE}{dw_{j,i}} \] (3.15)

where \( \varepsilon \) is the learning rate.

(5) Repeat the above process for subsequent lower layers. For lower layers, we no longer have the desired output \( \hat{y}(t) \). Thus, we propagate the error, backward:

\[ \frac{dE}{dy_i} = \sum_j \frac{dE}{dx_j} w_{j,i} \] (3.16)

In the following two sections, we discuss the implementations of the feedforward process and the back-propagation process.
3.2.3 Implementation of the Feed-Forward Process

The implementation of the feed-forward process is given below. The same technique for calculating the sum, as described for the Hopfield net, is utilized.

```c
int total_number_of_layers;
int num_of_nodes_in_layer[MAX_LAYERS];
int sum[MAX_NUMBER_OF_NODES];
int weight[MAX_LAYERS][MAX_NUMBER_OF_NODES][MAX_NUMBER_OF_NODES];
int node[MAX_LAYERS][MAX_NUMBER_OF_NODES];
int i, j, k;
{
  LOAD_IN_ONE_INPUT_VECTOR_INTO_THE_INPUT_LAYER();
  /* FEED FORWARD UNTIL THE OUTPUT LAYER */
  for (i = 0; i < total_number_of_layers; i++)
  {
    /* SUMMING ALL MULTIPLICATION OF WEIGHTS & NODES */
    for (j = 0; j < num_of_nodes_in_layer[i]; j++)
      for (k = 0; k < num_of_nodes_in_layer[i+1]; k++)
        sum[k] += node[i][j] * weight[i][k][j];
    /* PASSING THE SUM THROUGH THE SIGMOID FUNCTION */
    for (k = 0; k < num_of_nodes_in_layer[i+1]; k++)
      node[i+1][k] = SIGMOID_FUNCTION(sum[k]);
  }
  ...
}
```

The outer for loop terminates when all information has been fed forward through the intermediate net layers and propagated to the output layer. The main body of the code is divided into two portions, similar to the code for the Hopfield net. The first portion of the code performs the vertical summing of the multiplication of weight vectors and node vectors. This loop, as explained earlier, is vectorizable. The second portion of the code passes the summation results through the sigmoid function. Because the call to the SIGMOID_FUNCTION() prohibits vectorization, we isolate it as a separate loop. This is similar to the implementation of the thresholding process in the Hopfield net. Replacing the
sigmoid-function call with the parameterized sigmoid equation does not solve this problem since the function exp() is still required to compute $e^{-\text{slope}(x_j-H\text{shift})}$.

Note that the output value of each node $y_i$ is automatically saved in the two dimensional array node[i+1][k]. These values are needed in the back-propagation process for the calculation of $\frac{dE}{dy_j}$, $\frac{dE}{dx_j}$, and $\frac{dE}{dw_{j,i}}$. In the next section, we demonstrate how the back-propagation process is implemented.

3.2.4 Implementation of the Back-Propagation Process

The back-propagation process is used only during the learning phase. In the learning phase, the net first calls the feed-forward process before it could perform the back-propagation process to adjust its weights. A simplified version of the back-propagation process is given below.

```
int dEdy[MAX_NUMBERS_OF_NODES];
int dEdx[MAX_NUMBERS_OF_NODES];
{

  /* CALCULATING dE/dy FOR ALL j -- step (1) USE ONLY ONCE*/
  for (j = 0; j < num_of_nodes_in_layer[top_layer]; j++)
      dEdy[j] = node[top_layer][j] - desired_output[xmp_number][j];

  /* START WITH OUTPUT LAYER; LOOP UNTIL INPUT LAYER */
  for (curr_layer = top_layer; curr_layer > 0; curr_layer--)
  {
      lower_layer = curr_layer--;

      /* CALCULATE dE/dx FOR ALL j -- step (2) */
      for (j = 0; j < num_of_nodes_in_layer[curr_layer]; j++)
          dEdx[j] = dEdy[j] * Slope * (node[curr_layer][j] - Vshift) * 
                      (Magnitude - (node[curr_layer][j] - Vshift))/Magnitude;

      /* CALCULATE THE NEW WEIGHT -- step(3)(4) */
      for (i = 0; i < num_of_nodes_in_layer[lower_layer]; i++)
          weight[lower_layer][j][i] += - Epsilon * 
                      (dEdx[j] * node[lower_layer][i]);

      /* INITIALIZE dE/dy ARRAY TO 0 */
      for (i = 0; i < num_of_nodes_in_layer[lower_layer]; i++)
```

42
dEdy[i] = 0;

/* RECALCULATE dE/dy FOR LOWER LAYER -- step(5) */
for (j = 0; j < num_of_nodes_in_layer[curr_layer]; j++)
    for (i = 0; i < num_of_nodes_in_layer[lower_layer]; i++)
        dEdy[i] += dEdx[j] * weight[lower_layer][j][i];
}

For all nested for loops, only the inner-most loop can be vectorized using the Cray C compiler. In our implementation of the back-propagation process, we are able to vectorize all the inner loops. The first loop, which is only used at the beginning of the back-propagation process, initializes the array dEdy[j] to store the error terms $\frac{dE}{dy_j}$ between the desired output vector and the generated output vector. The computation of the array dEdy[j] is the vector subtraction of the vector node[top.layer][j] and the vector desired.output[xmp.number][j]. Since there are no data dependencies between dEdy[j] and the variables used to compute it, the loop is vectorizable.

The next for loop is a control loop, which allows the error to be back propagated through every layer in the back-propagation net. As discussed earlier, this outer loop is not vectorizable. Within this control loop, there are four more loops. We will discuss each of these loops, and explain why they all vectorize.

The first loop computes the array dEdx[j], and corresponds to step (2) in the algorithm. The equation for calculating the array dEdx[j] uses the variables from the two arrays on the right-hand side of the equation. They are dEdy[j] and node[curr_layer][j]. The remaining variables such as Magnitude, Slope, Hshift, etc, are held constant within this loop. These are known as invariants. As a result, the only factor which can prevent this loop from vectorization is the data dependency that might exist between dEdx[j] and dEdy[j] or node[curr_layer][j].
Since the arrays $dE_dx[j], dE_dy[j]$ and $node[curr\_layer][j]$ are distinct arrays, and the loop index varies with $j$, it is obvious that no data dependency exists between the arrays. Thus, this loop vectorizes.

The second loop adjusts the weight matrix of the back-propagation net, and corresponds to steps (3) and (4) of the algorithm. The weight matrix values are calculated using the vertical summing technique. During execution, the inner loop adjusts all the weighted interconnections between a node in the current layer and the nodes in the lower layer. The outer loop repeats this for all the nodes in the current layer. The fact that the weights are kept in a three-dimensional array rather than in one dimension, as in the case of the Hopfield net, does not prevent vectorization of this loop.

The third loop re-initializes the array $dE_dy[j]$ to the value 0. Since the assignment statement $dE_dy[j] = 0$ is a vector assignment, this loop is vectorizable. The final loop recalculates the error $\frac{dE}{dy_j}$ for the next lower layer. Once again, it employs vertical summing, which permits loop vectorization.

### 3.3 Frequency-Sensitive Competitive Learning Net

The Frequency-Sensitive Competitive Learning (FSCL) net is an unsupervised, single layer, continuous valued net. This layer is a winner-take-all layer, which receives its input from an input layer. The weighted interconnections between the input layer and the winner-take-all FSCL layer serve as reference vectors, which are used in selecting the winner node. In some applications, where output vectors are required, higher layers of a different type of neural net (e.g., a Grossberg layer [1]) can be cascaded to the FSCL net. The FSCL net consists of two phases, the learning phase and the performance phase. The algorithms for these two phases are discussed below.
3.3.1 The Learning Phase of the FSCL Net

In the learning phase, the input layer receives a training vector. The input vector is then compared to all of the weight vectors of the winner-take-all layer. When the comparisons are completed, one and only one node in the winner-take-all layer is chosen. The decision as to which node is selected depends on the fairness function $\mathcal{F}(u_i)$, and the “closeness” of the weight vector to the input vector. Two examples of measures which can be used to determine the “closeness” of the two vectors are the dot product and the mean square error (MSE). The fairness function, $\mathcal{F}(u_i)$, serves to ensure that all of the nodes in the winner-take-all layer are fully utilized [2].

When a node in the winner-take-all layer is chosen to be the winner, all other nodes are inhibited. The weight vector of the winner node is then adjusted to reflect the input vector by an amount proportional to the learning rate. The winner node then increments its update counter, $u_i$. Counter $u_i$ is subsequently used to influence the way the winner is chosen in the future. A summary of the FSCL learning phase algorithm is presented below.

1. Select the size of the FSCL neural network to be $(I, J)$ where $I$ is the number of neural units, and $J$ is the dimension of the input vectors. Initialize all the weights $w_{ij}$ between the input nodes $j$ and output nodes $i$ to some small random value. Then let $x(t)$, where $t = 0, ..., n - 1$, represent the training vector sequence.

2. Apply an input vector, $x(t)$.

3. Determine the distortion $d_i$ between the training input vector and the weight vector $w_i$: 45
\[ d_i = \mathcal{F}(u_i) \sum_{j=0}^{J-1} (x_i(t) - w_{i,j}(t))^2 \] (3.17)

where \( \mathcal{F}(u_i) \) is a function of \( u_i \) which determines the “fairness” of the weight vector use. An example of \( \mathcal{F}(u_i) \) is given below:

\[ \mathcal{F}(u_i) = u_i \] (3.18)

(4) Select the output unit \( i^* \), with the smallest distortion, and label it as the winner. Similarly, denote its weight vector as \( w_{i^*} \) and increment the frequency count, \( u_{i^*} \), by one.

(5) Adjust the selected \( w_{i^*} \):

\[ w_{i^*}(t + 1) = w_{i^*}(t) + \varepsilon(u_i)[x(t) - w_{i^*}(t)] \] (3.19)

where \( \varepsilon(u_i) \) is a decreasing learning rate, commonly defined as:

\[ \varepsilon(u_i) = Ae^{-(u_i/T)} \] (3.20)

where \( T \) is a fixed time constant.

(6) Repeat steps (2) through (5) for all training vectors.

Once the learning phase is completed, the FSCL net can be used in the performance phase. This phase is discussed in the next section.

3.3.2 The Performance Phase of the FSCL Net

The performance phase of the FSCL net is similar to the learning phase, except no adjustment of the weight vectors is required. A noisy input vector is presented to the input layer. This vector is then compared to all of the weight
vectors of the FSCL winner-take-all layer. The node which has the weight vector closest to the input vector is designated as the *winner*, and all other nodes are inhibited. When higher layers are present, the index of the *winner* node is passed to the higher layer and used to reproduce an output vector. A short description of this algorithm is given below.

1. Given a FSCL network of the size \((I, J)\) where \(I\) is the number of neural units, and \(J\) is the dimension of the input vectors. All weights, \(w_{ij}\), between input nodes \(j\) and output nodes \(i\) have been determined during the learning phase. Let \(x(t)\), where \(t = 0, ..., n - 1\), represent the noisy input vector sequence.

2. Apply an input vector, \(x(t)\).

3. Determine the distortion, \(d_i\), between the training input vector and the weight vector \(w_i\) as follows,

\[
d_i = \sum_{j=0}^{J-1} (x_i(t) - w_{i,j}(t))^2
\]  \hspace{1cm} (3.21)

4. Select the output unit, \(i^*\), with the smallest distortion and designate it as the *winner*. If higher layers are present, the index \(i^*\) of the *winner* node is used to reproduce the output vector.

5. Repeat steps (2) through (4) for all noisy input vectors.

The implementations of the learning phase and the performance phase are discussed in the next two sections.
3.3.3 Implementation of the Learning Phase

The implementation of the learning phase of the FSCL net, compared to the back-propagation net, is less complex. A technique similar to vertical summing is employed to vectorize of a major portion of the code.

```c
float weight[MAX_NUMBER_OF_NODES][MAX_NUMBER_OF_NODES];
float input_node[MAX_NUMBER_OF_NODES];
float mean_square_error[MAX_NUMBER_OF_NODES];
float fairness_measure[MAX_NUMBER_OF_NODES];
float mse_and_fairness[MAX_NUMBER_OF_NODES];
int u[MAX_NUMBER_OF_NODES];
int i, j, smallest, winner_index;
float Epsilon;
...
while (more_training_vectors)
{
    LOAD_IN_ONE_TRAINING_VECTOR();

    /* CALCULATE THE MEANS SQUARE ERROR -- step(3) */
    for (i = 0; i < nodes_in_layer[0]; i++)
        for (j = 0; j < nodes_in_layer[1]; j++)
            mean_square_error[j] += (input_node[i] - weight[j][i[i]]
                              (input_node[i] - weight[j][i]);

    /* CALCULATE THE FAIRNESS MEASURE -- step(3) */
    for (j = 0; j < nodes_in_layer[1]; j++)
        fairness_measure[j] = FAIRNESS_FUNCTION(u[j]);

    /* MULTIPLY FAIRNESS MEASURE WITH MEANS SQUARE ERROR (3)*/
    for (j = 0; j < nodes_in_layer[1]; j++)
        mse_and_fairness[j] = mean_square_error[j]*fairness_measure[j];

    /* FIND THE WINNER NODE -- ONE WITH SMALLEST PRODUCT OF
    MSE AND FAIRNESS MEASURE -- step (4) */
    for (j = 0; j < nodes_in_layer[1]; j++)
        if (mse_and_fairness[j] < smallest)
            { smallest = mse_and_fairness[j];
              winner_index = j;
            }

    /* PRECALCULATE THE LEARNING RATE FOR THE WINNER NODE */
    Epsilon = LEARNING_RATE_FUNCTION(winner_index);

    /* ADJUST THE WEIGHT VECTOR OF THE WINNER NODE -- step(4) */
    for (i = 0; i < nodes_in_layer[0]; i++)
        weight[winner_index][i] += Epsilon *
```
(input_node[i] - weight[winer_index][i]);

u[winer_index]++;
}

The while loop terminates when there are no additional training vectors. The body of the while loop consists of five for loops. Three of these loops vectorize. We discuss each of these below.

In the above sample of the code, the implementation of algorithm step (3) has been broken into three loops. This allows the vectorization of a large portion of the algorithm. In the first loop, we compute the square error for all the nodes in the winner-take-all FSCL layer, employing the vertical summing technique. The second loop serves two purposes. First, it computes the fairness measure for every node in the winner-take-all layer. Second, it isolates the procedure call to FAIRNESS_FUNCTION() within this loop. As discussed earlier, this second loop will not vectorize because of the procedure call. The third loop calculates the square error, computed in the first loop, with the fairness measure, computed in the second loop, and thus completes step (3) of the learning phase algorithm.

If we were to write step (3) as one single nested loop, vectorization is inhibited because of the call to FAIRNESS_FUNCTION(), which must be embedded within the loop. By breaking step (3) of the algorithm into three loops, it allows the first and third loops be vectorized.

The fourth loop, which determines the winner, does not vectorize because of the if – then statement. Finally, pre-calculating the learning rate, a function call, outside of the fifth loop allows the fifth, and last loop, to be vectorized. This loop corresponds to adjusting the weighted interconnections between the winner node and the nodes in the input layer.
This example shows how the non-vectorizable code can be isolated so that a major portion of the code is vectorizable. In the next section, we discuss the implementation of the performance phase.

3.3.4 Implementation of the Performance Phase

The code for the performance phase is similar to the code of the learning phase, except for a few for loop, which no longer is required. For example, the second and third loop which compute the fairness measure and the multiplication of this measure to the mean square error are eliminated. Similarly, the last loop, which adjusts the weight vector of the winner node, is replaced with a procedure call to PASS_WINNER_INDEX_TO_HIGHER_LAYER(), which passes the winner node’s index to higher layer such that output vector can be generated. The simplified code is given below.

```c
... while (more_noisy_input_vectors)
{
    LGAD_IN_ONE_NOISY_INPUT_VECTOR();

    /* CALCULATE THE MEAN SQUARE ERROR -- step (3) */
    for (i = 0; i < nodes_in_layer[0]; i++)
        for (j = 0; j < nodes_in_layer[1]; j++)
            mean_square_error[j] += (input_node[i] - weight[j][i]) * 
                                      (input_node[i] - weight[j][i]);

    /* FIND THE WINNER NODE -- ONE WITH SMALLEST MEAN SQUARE ERROR -- step (4) */
    for (j = 0; j < nodes_in_layer[1]; j++)
        if (mean_square_error[j] < smallest)
        {
            smallest = mse_and_fairness[j];
            winner_index = j;
        }
    PASS_WINNER_INDEX_TO_HIGHER_LAYER(winner_index);
} ...
```

Note that in all the implementations of the previous neural network algorithms, the loop control variable for all the vectorizable loops vary with the num-
ber of nodes in one of the layer of the net. In the next section, we discuss other variables which might be used as loop control variables.

3.4 Choice of Loop Control Variables

In general, it desirable to have long vectors for the pipelined operation on any vector computer architecture such as the Cray, as this reduces the set up cost of the pipeline and maximizes speed improvements. Nevertheless, this may not be possible due to constraints of the algorithm being implemented. For example, in the learning phase of many neural network algorithms, substantial speed-up can be achieved if the inner-most loop control variable can vary with the number of iterations or the number of training vectors. These two variables are excellent candidates for long vector pipelined operation, since most neural network algorithms have a large set of training vectors, or long training times. However, in the learning phase of most neural network algorithms, the weight matrix is adjusted iteratively with each training vector. This means that the new weight matrix can only be calculated after the computation of the previous weight matrix has been completed. In other words, there exists a data dependency between successive weight matrices and their predecessors. Thus, vectorization is not possible and the inner-most loop control variable cannot be made to vary with the number of iterations or the number of training vectors.

On the other hand, the performance phase is not constrained by the data dependencies found in adjusting the weight matrix, since the weights are not adjusted nor modified in this phase. Therefore, it is possible to vary the inner-most loop with the number of input vectors, at the expense of larger memory requirements. In some applications where a large number of input vectors are presented to the net, substantial speed-up can be achieved. As an alternative, the performance
phase of the FSCL net, given in Section 3.3.4, can be re-coded with the inner-most loop varying with the number of input vectors. This implementation technique has not been incorporated to the current version of NNSP. A possible coding of this technique is as follows.

```c
/* CALCULATE THE MEAN SQUARE ERROR FOR EVERY NODES IN 
WINNER-TAKE-ALL LAYER AND FOR EVERY INPUT VECTORS */
for (i = 0; i < nodes_in_layer[0]; i++)
  for (j = 0; j < nodes_in_layer[1]; j++)
    for (k = 0; k < number_of_input_vectors; k++)
      mean_square_error[j][k] += (input_node[i][k] - weight[j][i]) * 
                                (input_node[i][k] - weight[j][i]);

/* FIND THE WINNER NODE -- ONE WITH SMALLEST PRODUCT OF 
MSE AND FAIRNESS MEASURE -- step (4) */
for (k = 0; k < number_of_input_vectors; k++)
  for (j = 0; j < nodes_in_layer[1]; j++)
    if (mean_square_error[j][k] < smallest[k])
      { smallest[k] = mean_square_error[j][k];
        winner_index[k] = j;
      }
PASS_WINNER_INDEX_TO_HIGHER_LAYER(winner_index[k]);
```

Note that `mean_square_error` has been implemented as a two dimensional array. This is necessary to store all the mean square errors computed for all of the nodes of each input vector. Similarly, the `winner_index` is extended from a variable to a one dimension array, as needed to save all the indices of the winner nodes for all input vectors. Of course, the nested loop which determines the `winner` node does not vectorized because of the `if – then` statement embedded in this loop.

### 3.5 Benchmarks of the Cray NNSP

In all simulations, the performance of the implementations need to be justified. In the next section, we give the results of the benchmarks for the algorithms discussed in Section 3.1, Section 3.2, and Section 3.3.
In particular, we discuss the results obtained from simulations of neural networks using the NNSP on the Cray X-MP/28 and on the Sun 3/60. Comparisons are given in terms of Updates Per Second (UPS), measured during the learning phase and Interconnections Per Second (IPS), measured during the performance phase.

In a recent DARPA study [15], the term “IPS” is defined as the number of multiply-and-add operations that a neural network can perform per second. The study further states that a stated IPS rating may or may not include the time required for initializing a network, thresholding the sum, passing the sum through some non-linear function, determining the “winning” node, and other computational tasks that might be required by a neural network. Although ratings which exclude these tasks can be applied to any neural network topology, we believe that these measures do not reflect a measure of algorithm complexity. For example, if IPS or UPS is defined as the number of multiply-and-add operations per second, we would be simplistically benchmarking the speed of the hardware that a neural network is simulated on. As a result, the benchmarking of a neural network would yield only a benchmark of the speed of matrix multiplication.

Since we are interested in the complexity of the neural algorithm, the efficiency of the simulation code, and the suitability of a particular computer architecture for neural networks, we find more meaningful IPS and UPS ratings can be defined as follows. The IPS and UPS rating should include the time required for summation, calculation of any nodal non-linearity, selection of the winning node (where appropriate), updating all weights (for UPS), decreasing the learning rate, and any other function required by a particular neural algorithm. In other words, a benchmark should be performed on the minimum amount of code, the “core” of the algorithm, that the neural net requires for proper operation. Our measurements of IPS and
Table 4: Comparison of the NNSP on the Cray X-MP/28 and Sun 3/60

<table>
<thead>
<tr>
<th>Programs</th>
<th>Cray X-MP/28</th>
<th>Sun 3/60</th>
<th>Ratio Cray/Sun</th>
</tr>
</thead>
<tbody>
<tr>
<td>lhop*</td>
<td>16,981,132 UPS</td>
<td>86,403 UPS</td>
<td>196</td>
</tr>
<tr>
<td>hopfield*</td>
<td>11,650,485 IPS</td>
<td>78,549 IPS</td>
<td>148</td>
</tr>
<tr>
<td>lbp*</td>
<td>4,060,913 UPS</td>
<td>10,800 UPS</td>
<td>375</td>
</tr>
<tr>
<td>bp*</td>
<td>6,735,266 IPS</td>
<td>39,274 IPS</td>
<td>171</td>
</tr>
<tr>
<td>ifsc*</td>
<td>7,198,200 UPS</td>
<td>35,266 UPS</td>
<td>204</td>
</tr>
<tr>
<td>fscl*</td>
<td>7,317,073 IPS</td>
<td>36,001 IPS</td>
<td>203</td>
</tr>
</tbody>
</table>

NOTE: lhop* simulates the learning phase of the Hopfield net, while hopfield* performs the performance phase. Similarly, lbp* and bp* simulates the learning phase and the performance phase of the back-propagation net, respectively. In addition, ifsc* and fscl* perform the learning and performance phase of the FSCL net.

UPS exclude the time required for reading and writing files, initializations (e.g., random generation of weights), and other ancillary tasks.

Because there are no well known and accepted benchmarks for neural networks, we have benchmarked our neural network algorithms with the eight number patterns described by Lippman [1]. This problem requires a substantial number of nodes (120 input nodes) and, we feel, represents a reasonably realistic application. The result are shown in Table 1. The DARPA study [15] states that the maximum upper bound for Cray supercomputer doing neural network simulations is 50M IPS and for the Sun is 250K IPS. The DARPA results are comparable with ours since we have included the time required for all associated computation tasks, as discussed above.

We do not believe that our working definitions of IPS and UPS are perfect. Neither rating reflects the amount of training that a neural network requires for its learning phase in order to achieve “good” results. For instance, the IPS and UPS ratings for the back-propagation algorithm and the FSCL net are comparable. However, the back-propagation net requires a much longer learning phase than
does the FSCL net. Thus any measures of the IPS and UPS can be misleading.
Figure 10: A typical node and the hard-limit threshold function of the Hopfield net.
Figure 11: Execution of the vectorized code for learning phase of the Hopfield net.
\[ \text{sum} = \text{sum} + \text{node}[0] \times \text{weight}[0][0]; \]
\[ \text{sum} = \text{sum} + \text{node}[1] \times \text{weight}[0][1]; \]
\[ \text{sum} = \text{sum} + \text{node}[2] \times \text{weight}[0][2]; \]
\[ \vdots \]
\[ \text{sum} = \text{sum} + \text{node}[j] \times \text{weight}[i][j]; \]

Figure 12: Execution of the nonvectorized Solution II.
The order in which the array sum is computed

\[
\begin{align*}
\text{sum}[0] &= \text{node}[0] \times \text{weight}[0][0] + \text{node}[1] \times \text{weight}[0][1] + \ldots \\
\text{sum}[1] &= \text{node}[0] \times \text{weight}[1][0] + \text{node}[1] \times \text{weight}[1][1] + \ldots \\
\text{sum}[2] &= \text{node}[0] \times \text{weight}[2][0] + \text{node}[1] \times \text{weight}[2][1] + \ldots \\
\ldots &= \ldots \\
\text{sum}[j] &= \text{node}[0] \times \text{weight}[j][0] + \text{node}[1] \times \text{weight}[j][1] + \ldots 
\end{align*}
\]

Figure 13: Vertical summing applied to the computation of array sum[j].
Figure 14: Execution of the vectorized Solution I.
CHAPTER IV

VECTOR QUANTIZATION OF IMAGES

This chapter demonstrates an application of the Neural Shell to vector quantization of images. Vector quantization is introduced, and its use in image compression is described. Various advantages of neural network algorithms over conventional algorithms are discussed. A study, comparing the performance of the FSCL net and the Linde-Buzo-Gray algorithm, when applied to vector quantization of images, is conducted. The results of this study are discussed in detail.

4.1 Introduction

Increasingly large amounts of information are being transmitted and stored in a digital form. This data comes from a wide range of sources, including videoconferencing services, facsimile transmissions, ISDN services, satellite transmission of reconnaissance and weather photographs, and police records of finger-prints, to name a few. As a result, both transmission channel bandwidth and disk space are becoming limited. As these and other forms of communication, such as High Definition Television, gain in popularity, the problem of limited bandwidth and storage become even more apparent. It is therefore important that efficient compression algorithms be employed in order to reduce the transmission bandwidth and storage space that is needed.

In the past, a number of compression techniques such as Pulse Code Modu-
lation, Transform Coding, and Hybrid Coding have been applied to image data. Netravali and Limb [16] and Jain [17] have published extensive reviews of these techniques. Many of these techniques utilize scalar quantization. For example, Pulse Code Modulation technique samples the waveform and scalar quantize the sample amplitudes. However, according to Shannon's rate-distortion theory [18], better performance can be achieved by quantizing vector rather than scalar quantities.

An optimal vector quantization algorithm was proposed by Linde, Buzo, and Gray, and has come to be known as the LBG algorithm [19]. Extensive application of this algorithm to both speech and images has been studied [20,21], but in many cases, the computational complexity of this algorithm has restricted its use in real-time applications. Recently, there has been increasing interest in the use of neural networks to perform vector quantization in order to overcome these limitations [22,4,3].

The use of neural networks in vector quantization has many advantages. First, when the neural networks can be realized in hardware, vector quantization can be done in real time, since the networks are highly parallel. Second, due to the massive interconnections between processors in the network, neural nets are highly fault tolerant. Thus, a failure in one node or link will not significantly degrade the performance of the entire system. This is crucial in applications such as satellite systems and remote sensing devices. Finally, neural network vector quantizers can be used to realize adaptive quantizers.

These advantages prompted our interest into the research of vector quantization using neural network algorithms. In the course of our research, we have developed the Frequency-Sensitive Competitive Learning (FSCL) net [2] and Melton [22] has shown that the FSCL net can be applied to the vector quantization of
speech with results comparable to those obtained using the LBG algorithm.

4.2 Vector Quantization

The aim of any compression technique is to reduce the bandwidth needed for the transmission or storage of data. Vector quantization is one such technique. The goal of vector quantizer design is to find a set $Y$ of $N$ dimensional reference or codeword vectors such that the use of these codewords to represent the data will minimize the expected distortion. More formally, as described by Gersho [23], vector quantization is defined as a mapping of $k$-dimensional Euclidean space $\mathbb{R}^k$ into a finite subset $Y$ of $\mathbb{R}^k$. Thus,

$$Q : \mathbb{R}^k \rightarrow Y$$

where $Y = (\hat{x}_1, \hat{x}_2, ..., \hat{x}_N)$ and $N$ is the total number of codeword vectors in the codebook. Vector Quantization can be viewed as comprising of two parts: an encoder $E$ and a decoder $D$. Given an input vector $x$, the encoder maps the vector to an index $J$. The index is then transmitted and decoded back to $\hat{x}$.

$$E : \mathbb{R}^k \rightarrow J \quad \text{and} \quad D : J \rightarrow \mathbb{R}^k$$

The distortion which results from representing the original vector $x$ by the restored vector $\hat{x}$ can be measured by some distortion function, $d(x, \hat{x})$. An optimal quantizer, which may not be unique, is one which minimizes this distortion. One commonly used distortion measure is the square error, given by:

$$d(x, \hat{x}) = \sum_{i=0}^{k-1} (x_i - \hat{x}_i)^2$$

In many practical cases where $N$ is less than the number of possible input patterns, vector quantization represents a destructive compression. This means
there is some inherent distortion between the restored and the original vectors. In the next section, we discuss some measures which are useful for determining the effectiveness of various compression techniques.

4.3 Measures of Fidelity and Data Compression

Because vector quantization of images results in some degradation of the original image it is useful to employ some means of measuring the fidelity of the resulting image in order to determine the performance of the vector quantization algorithm. One common measure of distortion is the average mean square error (MSE). For an $N \times M$ size image the average mean square error is defined as:

$$MSE = \frac{1}{NM} \sum_{i=1}^{N} \sum_{j=1}^{M} (v_{i,j} - \hat{v}_{i,j})^2$$  \hspace{1cm} (4.4)$$

where $v_{i,j}$ and $\hat{v}_{i,j}$ represent the $N \times M$ original and restored images, respectively.

Another, perhaps more meaningful measure is related to the signal and noise power of an image. The signal-to-noise ratio (SNR) is defined as:

$$SNR = 10\log_{10} \frac{P^2}{MSE}$$  \hspace{1cm} (4.5)$$

where $P$ is the peak to peak pixel value of the original image. For gray-scale images, the peak-to-peak value is the difference between the maximum and the minimum value of the pixel intensity found in the images.

Although the above two performance measures provide some means for determining the distortion introduced by compression, they do not measure the effectiveness of the algorithm in reducing the information needed to represent the image. The amount that the data is compressed can be measured by the bit rate in bits/pixel, which is defined as:
\[ R = \frac{(\log_2 N)}{k} \]  

where \( N \) is the codebook size and \( k \) is the dimension of the reference vector or codeword.

The next section discusses a well established codebook design technique, and is followed by a description of a design technique based on neural network training.

4.4 The Linde-Buzo-Gray Vector Quantization Algorithm

Linde, Buzo, and Gray have proposed a vector quantization algorithm that has come to be known as the LBG algorithm [19]. This technique assumes an initial codebook of size \( N \) for an \( N \)-level vector quantizer. Linde et. al. [19] further proposed an initial guess method for the \( N \)-level vector quantizer using a "splitting" technique. The resulting codebook is optimal with respect to the initial centroid and the perturbation vector. The \( N \)-level vector quantizer as proposed by Linde, Buzo, and Gray is described in more detail below [19].

1. Let \( N \) be the number of codewords, and \( \epsilon \) be the convergence ratio. Given a training set, \( \{x_j; j = 0, ..., n - 1\} \), where \( n \) is the total number of training vectors. Assume an initial \( N \)-level codebook, \( \hat{A}_0 \).

2. Assume that the current iteration is \( m \), and the current codebook is given \( \hat{A}_m = \{y_j; i = 1, ..., N\} \), find the minimum distortion partition of the training set, \( x \). A vector, \( x_j \), in the training set is said to belong to a partition if the distortion between \( x_j \) and the codeword, \( y_j \), of that partition is minimum. The distortion measure is given in equation ((4.3)).

3. Compute the average distortion for all the partitions by:
\[ D_m = \frac{1}{n} \sum_{j=0}^{n-1} \min d(x_j, y) \text{ where } y \in \hat{A} \]  

(4.7)

(4) If \((D_{m-1} - D_m)/D_m \leq \epsilon\), halt with \(\hat{A}_m\) as the final codebook, else continue.

(5) Compute new codewords for the codebook, \(\hat{A}_{m+1}\), by assigning the codewords to the centroid of the partitions found in Step (2). Repeat Steps (2) - (5) until the average distortion is below the convergence ratio, \(\epsilon\).

Since the above algorithm requires an initial codebook of \(N\) codewords, Linde et. al. [19] proposed a “splitting” technique which can be used to obtain this initial codebook guess. The algorithm is described in more detail below.

(1) Start with the current codebook size \(M\) of 1 and let the codeword of this codebook be the centroid of the entire training set, \(x\).

(2) With the codebook \(\hat{A}(M)\) containing the codewords \(\{y_i; i = 1, ..., M\}\), “split” each codeword \(y_i\) into two close codewords of \(y_i + \epsilon\) and \(y_i - \epsilon\), where \(\epsilon\) is a fixed perturbation vector. Replace \(M\) by \(2M\).

(3) If the current codebook size, \(M\), equals the final codebook size, \(N\), halt the “splitting” algorithm and use the current codebook as the initial guess for the \(N\)-level quantizer, described previously. On the other hand, if \(M\) is less than \(N\), execute the \(M\)-level quantizer to reproduce a new codebook, and go back to step (2).

In the next section, we discuss the FSCL net and the mechanism used to solve a problem associated with neural network vector quantizers, the under-utilization of codewords.
4.5 The FSCL Neural Network

The Frequency-Sensitive Competitive Learning (FSCL) net was developed to solve some of the problems associated with other neural network vector quantizers. An example of these problems is the under-utilization of codewords. Underutilization of codewords occurs when a subset of the codeword vectors are consistently selected, leaving other codeword vectors unused. This phenomena can occur when there are bad initial guesses for the centroids of the data, a problem which is not uncommon since, in general, the distribution of the data is unknown. As a result, as training vectors are presented to the net, a portion of the codeword vectors start to adapt and represent the distribution of the incoming training data. For example, when the square error is used as the distortion measure, it is possible that codeword vectors which have previously been selected and modified will have a smaller distortion and will continue to be selected.

Most Competitive Learning neural networks algorithms have some mechanism which is used to solve this problem. For example, Kohonen [24] solves this in his Self-organizing Feature Maps by incorporating a decreasing neighborhood function, where the neighborhood of a codeword is defined by some topology. This technique forces reference vectors to be updated in a relatively uniform fashion. It also ensures that even with a bad initial guess for the data, all of the reference vectors will be used.

Frequency-Sensitive Competitive Learning solves the problem of the underutilization of codewords or reference vectors by using a frequency counter. Associated with each reference vector $W_i$ is an update counter $u_i$. When a reference vector $W_i$ is chosen during the learning phase, its update counter is also incremented. This counter is incorporated as one of the parameters in the selection of the closest
reference vector. As a result, occasionally, the least used reference vector will be chosen even though it might have a distortion measure larger than other reference vectors. Thus the FSCL net can be viewed as a compromise between distortion and entropy.

The FSCL net is an unsupervised learning net. The weights of the net are allowed to adaptively adjust to reflect the probability density function of the input vector patterns. The FSCL clustering algorithm has been previously discussed in Chapter III, Section 3.3.

In the next section we demonstrate the application of the FSCL net to the vector quantization of images. The results obtained from the FSCL net are compared with those generated by the LBG algorithm.

4.6 Results

The original images used in these experiments had a resolution of 500 x 482 pixels with 8 bits/pixels. The block size \( k \) (also the dimension of codeword) was set to 4 (a 2 pixel x 2 pixel pattern). Codebook sizes were varied to provide comparisons between the performance of the FSCL net and the LBG algorithm.

The convergence ratio \( \varepsilon \) of the LBG algorithm was set to 0.01. The parameters of the FSCL net were set to:

\[
\varepsilon(u_i) = 0.01e^{-(u_i/10,000)} \tag{4.8}
\]

\[
\mathcal{F}(u_i) = u_i. \tag{4.9}
\]

The "fairness" function \( \mathcal{F}(u_i) \) is nominally chosen to be \( u_i \) because of its simplicity and because reasonable results are experimentally obtained. When one weight vector is being updated frequently, its corresponding counter, \( u_i \), is increased. Since \( u_i \) is multiplied by the distortion, it serves as a penalty or cost
Table 5: Comparison of performance between the LBG algorithm and FSCL net

<table>
<thead>
<tr>
<th>Codebook Size</th>
<th>Bit Rate (bits/pix)</th>
<th>LBG Algorithm</th>
<th>FSCL Net</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No. Iter.</td>
<td>MSE (dB)</td>
<td>SNR (dB)</td>
</tr>
<tr>
<td>4</td>
<td>0.50</td>
<td>11</td>
<td>166.95</td>
</tr>
<tr>
<td>8</td>
<td>0.75</td>
<td>15</td>
<td>47.81</td>
</tr>
<tr>
<td>16</td>
<td>1.00</td>
<td>18</td>
<td>19.50</td>
</tr>
<tr>
<td>32</td>
<td>1.25</td>
<td>23</td>
<td>12.85</td>
</tr>
<tr>
<td>64</td>
<td>1.50</td>
<td>30</td>
<td>10.03</td>
</tr>
</tbody>
</table>

which deters frequently used weight vectors from being chosen, and thus allows other weight vectors to be chosen.

Table 5 lists the results we obtained using the FSCL and the LBG algorithms. The number of training iterations, mean square error, and Signal to Noise ratio are given for each of the algorithms. It can be seen that the results obtained from the FSCL net are comparable with the LBG algorithm.

In Figure 15, we show the original image (which has maximum and minimum pixel values of 183 and 0 respectively and 8 bits/pixel). Figure 16 shows the restored image at 0.5 bits/pixel for the LBG algorithm (top) and FSCL net (bottom) respectively. Figure 17 show the results obtained when the bit rate is 1.0 bits/pixel. Finally, Figure 18 show the results when the bit rate is 1.5 bits/pixel. Note that in this last example, the restored images are almost indistinguishable from the original image.

As a comparison between the codebooks generated by the LBG Algorithm and the FSCL net, in Table 6 we show the codewords in both codebooks when the dimension of the codebooks, $k$, is equal to 4. It is clear that very similar codewords are generated by both the FSCL net and the LBG algorithm.

In Figure 19, we show the original pixel values of the first scan line. Figure
Table 6: Comparison of codewords generated by LBG algorithm and FSCL net

<table>
<thead>
<tr>
<th>LBG Algorithm</th>
<th>FSCL Net</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_1$</td>
<td>$w_1$</td>
</tr>
<tr>
<td>$(151, 151, 151, 151)$</td>
<td>$(40, 40, 40, 40)$</td>
</tr>
<tr>
<td>$w_2$</td>
<td>$(46, 46, 45, 45)$</td>
</tr>
<tr>
<td>$w_3$</td>
<td>$(105, 105, 104, 104)$</td>
</tr>
<tr>
<td>$w_4$</td>
<td>$(7, 7, 7, 7)$</td>
</tr>
<tr>
<td>$(152, 152, 151, 151)$</td>
<td></td>
</tr>
<tr>
<td>$(101, 101, 100, 100)$</td>
<td></td>
</tr>
</tbody>
</table>

20 shows the pixel values of the reproduced image using the LBG algorithm and the FSCL net at 1.5 bits/pixel.

To further study the performance of the FSCL net, we conducted another experiment on a different image. In this experiment the image of a building with a resolution 500 x 482 pixels and 8 bits/pixel. Note that the first image contains few sharp edges, whereas the second has a large number of edges. The range of the pixel values for the second image is from 0 to 255.

We ran experiments for codebook sizes of 16, 32, and 64. The number of iterations were as described in the previous experiment. The “fairness” function was first set to be $F(u_i) = u_i$. The results obtained in the FSCL net are poorer than those obtained using the LBG algorithm (Table 7). We conjecture that the cause of is that this image consists of non-uniform cluster sizes and choosing a fairness function of $F(u_i) = u_i$ attempts to impose uniformity in the use of the weight vectors. To improve the performance of the FSCL net, we ran the experiment again with $F(u_i) = \sqrt{u_i}$, and the results in this case were comparable with the LBG algorithm. This demonstrates that the “fairness” function must represent the distribution of the data. Since the distribution of the data is unknown, it is undesirable to have a fixed “fairness” function. Therefore, a decreasing power function for $u_i$ of the form $F(u_i) = u_i^{\beta e^{-t/T}}$ can be used, where $t$ is the global count of training vectors presented to the net, and $T$, a fixed time constant. This ensures
Table 7: Comparison of performance between the LBG algorithm and FSCL net

<table>
<thead>
<tr>
<th>Codebook Size</th>
<th>Rate bits/pix</th>
<th>LBG MSE</th>
<th>$\epsilon = 0.01$ $u_i$ MSE</th>
<th>$\sqrt{u_i}$ MSE</th>
<th>$u_i^{\beta e^{-t/T}}$ MSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>1.00</td>
<td>174.63</td>
<td>197.52</td>
<td>176.01</td>
<td>180.54</td>
</tr>
<tr>
<td>32</td>
<td>1.25</td>
<td>93.28</td>
<td>116.41</td>
<td>89.48</td>
<td>93.67</td>
</tr>
<tr>
<td>64</td>
<td>1.50</td>
<td>46.15</td>
<td>71.39</td>
<td>56.57</td>
<td>49.19</td>
</tr>
</tbody>
</table>

mobility of the reference vectors at the beginning of the learning phase and at the same time, as training proceeds, the FSCL net gradually turns into competitive learning since $\lim_{t \to \infty} u_i^{\beta e^{-t/T}} = 1$. As a result, the net chooses the winner node as stated in step (3) of FSCL algorithm described in Chapter III, Section 4.5, with only the square error determining the error. For this experiment, $\beta$ has been set to 1 and the value $T$ has been set such that $F(u_i) = u_i^{\epsilon^1}$ when the FSCL net has gone through half of the learning process.

Figure 21 shows the original image. In Figure 22 we show the restored image using the LBG algorithm (top) and the FSCL net with “fairness” function $F(u_i) = u_i$ (bottom). Figure 23, we show the restored image using the FSCL net with “fairness” function $F(u_i) = \sqrt{u_i}$ and $F(u_i) = u_i^{\beta e^{-t/T}}$ respectively. It can be seen that the results are indistinguishable.

In this section, we have shown that the FSCL net can be applied to vector quantization of images with results comparable to those generated by the LBG algorithm. The FSCL net utilizes a fairness function $F(u_i)$ which ensures that all codewords are used. As one codeword or reference vector is being used, the update counter $u_i$ is incremented and $F(u_i)$ serves as a penalty or cost to hinder frequently chosen reference vectors from being chosen, and allows less frequently used vectors to be updated. In addition, we have shown that the fairness function
can have an a significant effect on the performance of the FSCL net. Rather than using a fixed power of $\beta$ for the fairness function, $u_i^\beta$, a decreasing function of $\beta$ seems to allow better adaptation to the underlying distribution. This allows for a high mobility of the reference vectors during the initial stages of learning and then allows the FSCL net to turn into a competitive learning net.

Figure 15: Original image, 500 x 482 pixels, 8bits/pixel.
Figure 16: Restored image using LBG algorithm (top) and FSCL net (bottom) at a bit rate of 0.5 bits/pixel.
Figure 17: Restored image using LBG algorithm (top) and FSCL net (bottom) at a bit rate of 1.0 bits/pixel.
Figure 18: Restored image using LBG algorithm (top) and FSCL net (bottom) at a bit rate of 1.5 bits/pixel.
Figure 19: Original pixel values in the first scan line
Figure 20: Comparison of the pixel values in the first scan line.
Figure 21: Original image, 500 x 482 pixels, 8 bits/pixel.
Figure 22: Restored image using LBG algorithm (top), and FSCL net with $\mathcal{F}(u_i) = u_i$ (bottom).
Figure 23: Restored image using FSCL net with $\mathcal{F}(u_i) = u_i^{\frac{1}{2}}$ (top), and $\mathcal{F}(u_i) = u_i^{\beta t/T}$ (bottom).
CHAPTER V

CONCLUSIONS

5.1 Summary

In this thesis, we described the Neural Shell, and its objectives. Various facilities and features of the Neural Shell were discussed in detailed. In addition, the implementations of the Cray versions of the Neural Network Simulation Programs were discussed, and techniques that result in code vectorization were explained. Sample codes were given to illustrate these techniques. Benchmarks of these implementations, measured in Interconnections Per Second and Updates Per Second, were presented. The results of the benchmarks shows that substantial improvement in performance can be achieved with a Cray supercomputer, compared to conventional computers. Furthermore, an application of the Neural Shell was described. Specifically, the performance of the FSCL net was compared to the LBG algorithm when both were applied to the vector quantization of images. The results generated by the FSCL net were shown to be comparable with the LBG algorithm. In the next two sections, we discuss future enhancements for the Shell, and future directions for our research.

5.2 Future Enhancements of the Neural Shell

Currently, all of the NNSP in the Neural Shell are implemented as independent programs. Common procedures or functions, which are used by all NNSP, are
not shared. This is because a library containing these procedures has not been implemented. As a result, copies of these procedures exist in different programs of the NNSP. During the initial development phase of the Shell, the lack of such a library did not pose a significant problem, since most of the programs are relatively small. Common procedures can be copied to different programs manually using a text editor. Nevertheless, as the Neural Shell has grown, it has become clear that a neural network library is needed.

There are many advantages in having such a common library. First, a copy of common procedures are not required for each of the NNSP. This decreases each program's size, and increases its readability. In addition, having the modules in a library will ease the task of developing new neural network algorithms by allowing existing modules, such as reading and writing of files, to be re-used. Only the "core" of the new neural network algorithms will need to be written. Second, the use of library's routines enhances modularity. This allows different programmers to work concurrently. New modules can be added to the library, and made available to all programmers. Testing and debugging programs, consisting of "well-tested" and "known-to-work" modules, are simpler since testing can be concentrated only on newly implemented modules. Furthermore, modularity also encourages layering and program abstraction. Implementation details of different modules can be hidden in the modules. Changes to an existing module need not effect other calling modules, if the input and output characteristics of the module remains unchanged. Therefore, it is possible to build a program composed of a hierarchy of different layers of these modules.

Another possible enhancement to the Neural Shell is the use of data structures. In the current version of the Shell, independent variables, such as array of floats or integers, are used to represents node values, weight matrices, etc. In most neural
network algorithms, these variables are related to one another. For example, in the FSCL net, the winner-take-all layer has an array of frequency counters \( u \), associated with each node. Using the structure facility provided in C, we can group these two related variables (the node value and the frequency counter) into a single structure as follows.

```c
struct fscl_node{
    float node_value;
    float u;
};
```

The winner-take-all layer could then be defined as an array of type “fscl_node”. The use of data structure improves readability and understandability by combining different variables into a single entity.

Furthermore, in Chapter III, Section 3.4, we discussed a new technique for vectorizing the performance phase of the neural network algorithms by having the inner most loop varying with the number of input vectors. This technique is not currently implemented in the current version of NNSP. In most neural network applications, where there are large number of input vectors, this implementation technique could result in substantial speedup, at the cost of large memory requirements. A study comparing the performance of this implementation with the current implementation should be conducted.

In the next section, we discuss other research directions that can further our understanding of the neural network algorithms.

5.3 Future Research Directions

As discussed earlier, the NNSP has been implemented on the Sun workstation and the Cray supercomputer. We have shown that substantial speedup can be achieved with a supercomputer. Nevertheless, a vector computer such as the
Cray supercomputer may not fully exploit the inherent parallelism in the neural network algorithms. Since neural networks operates in parallel, mapping of these algorithms to parallel machines could provide further insights to their communication requirements, and the speedup that can be achieved, when the nets are implemented in hardware. Two candidates for parallel computers are the Hypercube and the Transputer. Comparison of the performance achieved on the Cray supercomputer and these parallel computers can be investigated.

In addition, the role of the “fairness” function for the FSCL net needs further investigation. It has been shown that fairness function can significantly effect the performance of the FSCL net. Although, there is no current means of determining a specific choice for the fairness function, we find that a decreasing power function for $u_i$ seems to be appropriate. Further studies on the choice of the fairness function needs to be conducted.

Although neural network research is still in its infancy, we have illustrated in this thesis some of neural networks’ power and advantages over conventional computers. With further research, it might be possible that one day neural networks will perform some of the tasks, such as perception and recognition, that humans perform so well.
BIBLIOGRAPHY


86


