A SIMPLE ARTIFICIAL NEURAL NETWORK DEVELOPMENT SYSTEM FOR STUDY AND RESEARCH

by

David Southworth

Project submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science and Applications

Approved:

[illegible]

E. Haddad, Chairman

[illegible]

C. Egyhazy

[illegible]

R. Schneider

December 1991

Blacksburg, Virginia

c.2

LD
5655
V851
1991
S687
C.2

A SIMPLE ARTIFICIAL NEURAL NETWORK DEVELOPMENT SYSTEM FOR STUDY AND RESEARCH

by

David Southworth

Committee Chairman: Emile Haddad
Computer Science

(ABSTRACT)

This research paper proposes the design and implementation of an Artificial Neural Network (ANN) development system which will provide the foundation and a tool for study and research in ANN architectures and algorithms. A system was developed which allows for the implementation of networks and for the modification of common and interesting network parameters and algorithms. This establishes a versatile and effective model from which to proceed with ANN study.

The system design is an initial prototype providing a generic and dynamic interface which allows the versatility to implement simple networks on a Personal Computer and modify their parameters, architectures, and algorithms. It also allows the monitoring of the internal conditions of the network, providing a basis for detailed data collection and research.

Several different neural nets were implemented and trained on the system. Various feedforward networks using the

Delta Rule and the backpropagation learning algorithms were trained in the supervised mode to solve problems in pattern recognition. Modifications to these networks were then used to compare training and operational characteristics between the different architectures. A classic one layer Hopfield net using a recurrent feedback, associative memory architecture was also implemented and trained in the unsupervised mode. An unsupervised Hebbian learning algorithm was also implemented and tested on the system.

Several enhancements are proposed which will increase the versatility of the system and aid in the further study of Artificial Neural Network implementations and characteristics.

Acknowledgements

I would like to thank the members of my review committee for their support and many helpful suggestions, my father, John S. Southworth for his expert advice and, at times, well deserved criticism, and my wife Jeanie for her unswerving support throughout. I also especially thank Dr. David Andes of the Naval Weapons Center, China Lake, who's inspirational excitement about the possibilities of neural networks provided the spark which kindled my interest.

TABLE OF CONTENTS

LIST OF FIGURES

SECTION 1

INTRODUCTION

Artificial Neural Networks (ANNs) have been inspired by their biological counterparts, the real neural networks of the human brain. The mysteries of how the brain actually operates is little understood although its general operation, in a gross sense, has been studied thoroughly. Rough models of the brain and its functions form the pattern and the basis for Artificial Neural Networks.

In recent years, there has emerged a new interest in the use of Artificial Neural Networks as an innovative computational approach to problems that are considered very difficult, if not intractable, to solve on conventional von Neumann machines.
These applications include pattern matching and recognition, decision making in a fuzzy environment, and adaptive control.

The human brain solves these kind of problems in an elegant, although still unclear, fashion. Humans see an incomplete image, perhaps partially obscured by shadows or other objects, and can identify it almost immediately. No one of the billions of neurons in the brain is especially important, yet together they form a marvelous processor that outperforms machines in most recognition tasks.

One might argue that it is impossible to build an artificial neural network if real ones are not well understood. Even so, ANNs are being built and simulated on conventional digital computers. The first tentative steps for implementing them in hardware is also taking place. These are not exact models of real neural systems but they do generalize the concepts of actual "wetware". We do not fully understand how actual neural networks operate but still we can grasp many of the concepts well enough to simulate in software.

There is a great necessity for further research in artificial, as well as biological, neural networks. We must know much more about learning algorithms, interconnection philosophies, layering requirements, size requirements, and how best to implement all of those into hardware and/or software. All these issues need to be addressed along with the biological, chemical and behavioral study of actual networks if the artificial and real world are to ever approach convergence.

The purpose of this Artificial Neural Network Development System project is to construct a structural implementation which will provide a vehicle for the further study and research of Neural Network architectures. The development system provides a foundation for the student to create networks of different sizes and architectures, with variable parameters and computational algorithms, and see them in operation. It includes the capability to vary several of the parameters of ANN implementations which are of interest in current research. These include such things as the interconnection of the nodes within the network, the implementation of multi-layer networks, fully or partially connected networks, and recurrent (feedback) networks. Also included are neural body activation algorithms, learning algorithms of the network, and the number and size of layers within the network. A complete implementation, where these parameters are selectable and modifiable by the user, will provide a versatile and effective tool and a foundation for further study and research in ANN architectures and capabilities. This project is meant as a first step in the development of such a simulation system.

The intended users of this system are students studying the functions and use of neural networks. Such users may be studying, for example, the effects of various parameters on neural nets, exploring the use of different learning algorithms, different activation functions or connection architectures. The development system was created to support such capabilities.

The user interface for this system was partially completed in satisfaction of course requirements for Human Computer Interaction [CS 5714] finished in Fall 1989) [1]. Since that time, several enhancements, changes and additions have been made to the implemented system.

SECTION 2

ARTIFICIAL NEURAL NETWORK BACKGROUND

Figure 1 is a sketch of two simple neural cells with one interconnection (synapse). There could be many more such interconnections.

The neural body is the main computational element of both the biological neural network and its artificial counterpart. It receives electrical inputs from one, hundreds, or thousands of other neural bodies via its dendrites. The electrical pulses traveling along the dendrites may be excitatory or inhibitory. The neuron essentially averages its inputs over a short period of time. If the end result exceeds a certain threshold, the neuron "fires", sending an electrical signal out along its axons. Each axon is, in turn, connected through synapses to the dendrites of other neurons and the pattern continues. This simplistic description of actual neuron operations and interconnections is what most artificial networks are patterned after.

Memory is "distributed" throughout the neural network and is manifested in a multitude of neurons, their connections and weights, and their averaging functions which determine their time to fire. A single neuron, or even many neurons, with their multiple connections do not constitute the complete memory of a system. In fact, after an ANN has been trained,

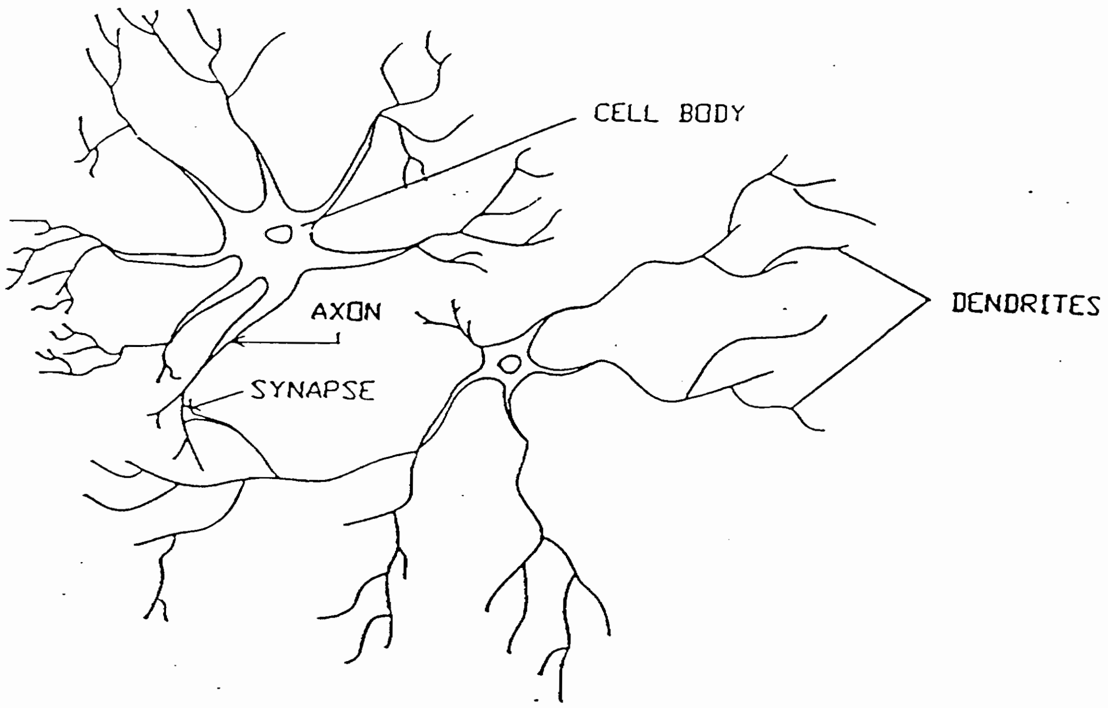


Figure 1.   Biological Neurons

many neurons and their connections may be removed with little or no effect on the outcome of the computations of the network. The memory of the system is truly distributed across, and throughout the network.

The artificial neuron, Figure 2, first envisioned and developed by Rosenblatt [2] was patterned after the functions of the biological neuron. Essentially, a set of inputs is applied with each input representing either the output of another neuron or a stimulus from the outside world. Each input is multiplied by a corresponding "weight". The weighted inputs are then summed, producing an output called SUM for purposes of this paper (the general term in the literature for SUM is NET). The set of weights associated with each neuron and corresponding to each input is a vector W. This weight vector is multiplied and summed with the vector of all the inputs, X, to the neuron. This forms the relationship:

$$[X]\ [W] = \mathrm{SUM}$$

The thresholding function (F) of the neuron, commonly called the neuron activation function, is then applied to SUM, which produces the output, or firing of the neuron, called OUT.

In the Artificial Neural Network, the neuron is represented by the summation process coupled with the activation function. The dendrites are simulated by the

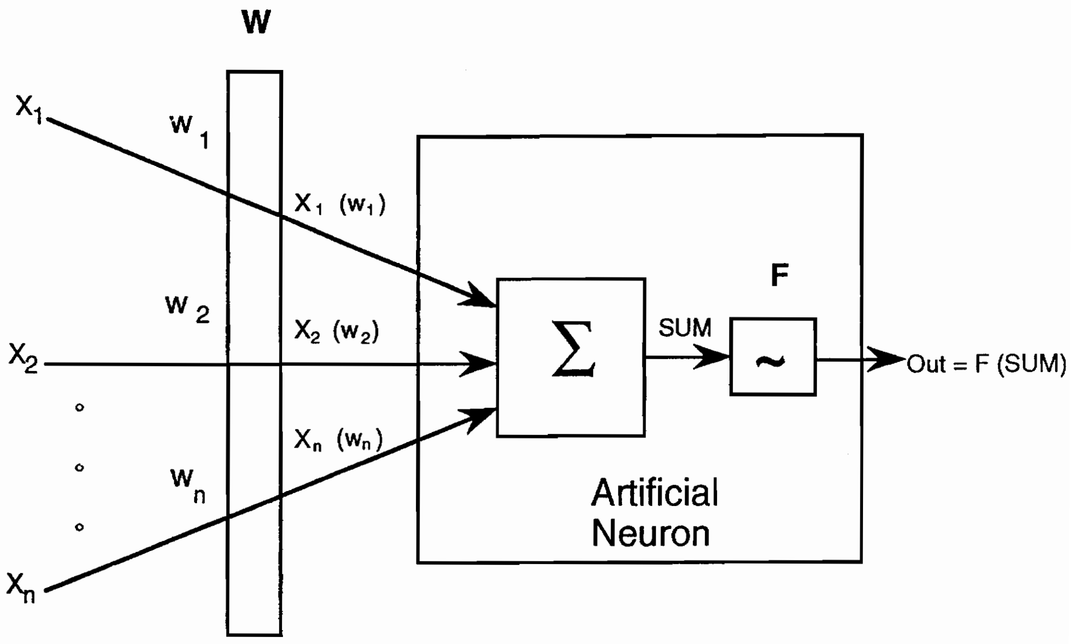


Figure 2. Artificial Neuron

inputs and weight vector to the neural body, while the axons are represented by the output lines of the neuron which become inputs to the weight vector of the next neural layer. The synapse connections are represented by the connectivity between the layers which may be fully or partially connected. In a fully connected network all neurons of one layer are connected to every neuron in the following layer. In a partially connected network, some of these connections will be missing. The connectivity and how it affects the computational power of certain networks is one of the objects of current research in the field.

The activation function (F) of the neuron may be a simple linear or non-linear function, or a more complex non-linear function. Multi-layer networks using a linear activation function, such as

$$OUT = C * SUM$$

where C is some constant, have been shown to have no more representational power than a single layer network with the same function [3]. Networks today universally use non-linear activation functions such as a simple threshold function (Figure 3a) where

$$\text{OUT} = 1 \text{ if SUM} >= \text{Threshold}$$

$$\text{OUT} = 0 \text{ if SUM} < \text{Threshold}$$

or other, more complex non-linear functions which permit more general and stable network activities.

Often, activation functions in current research follow the sigmoid model expressed as

$$\text{OUT} = 1/(1 + e^{-sum}) \qquad \text{(Equation 1)}$$

This form (Figure 3b) allows the network to handle a wide range of signals where the output is "squashed" to produce low gain for large inputs and a higher gain for smaller inputs. In Figure 3b, the neuron produces a continuous, compressed output (OUT) within the range of zero and one. Some algorithms requiring discreet, binary outputs may further apply a separate thresholding function to OUT. If OUT then exceeds this threshold level, the neuron fires and applies a signal to its axons or output lines. If below the threshold, the output is zero.

Figure 3c shows a modified Sigmoid function. This simple enhancement to the Sigmoid Function, as described by Stornetta and Huberman [4], allows for output values in the range of -.5 to +.5 which often reduces training time of the network since

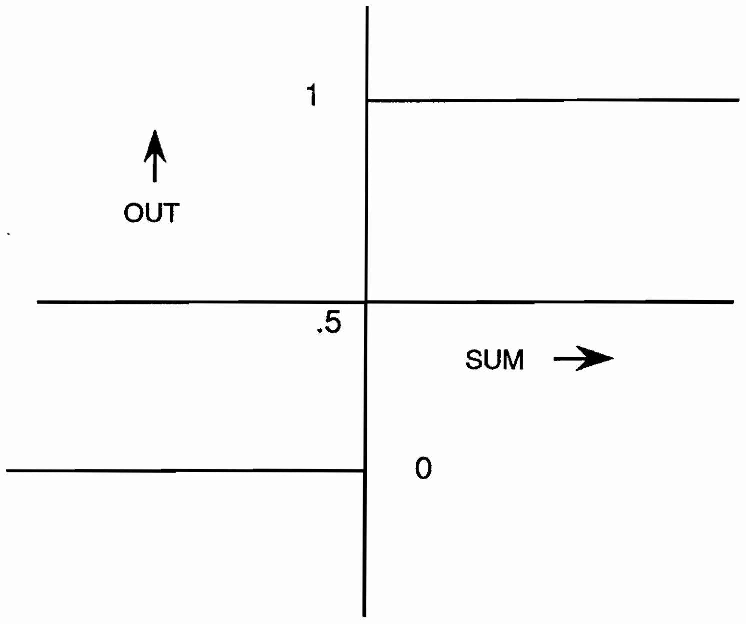


Figure 3a. Simple Threshold Activation Function

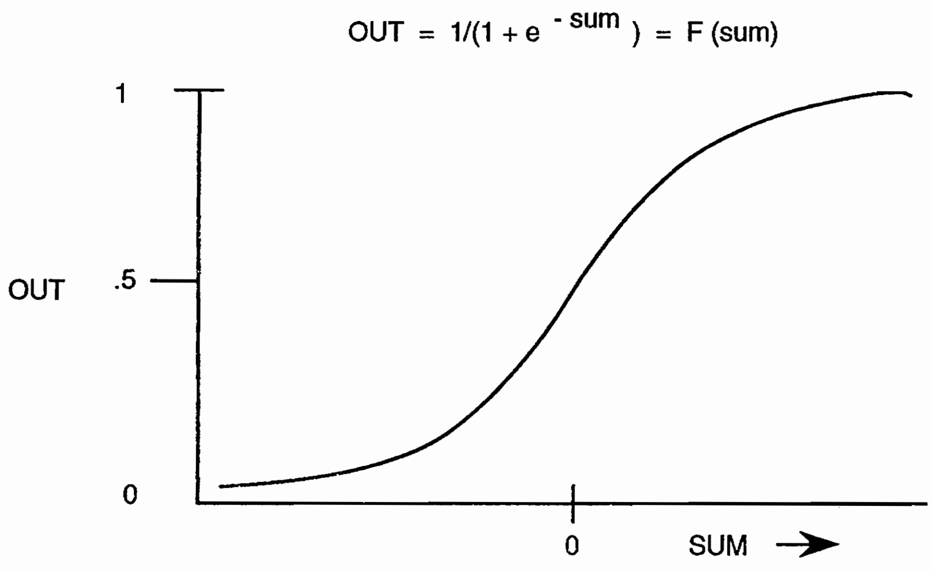


Figure 3b. Sigmoid Activation Function

$$OUT = -.5 + 1/(1 + e^{-sum}) = F(sum)$$

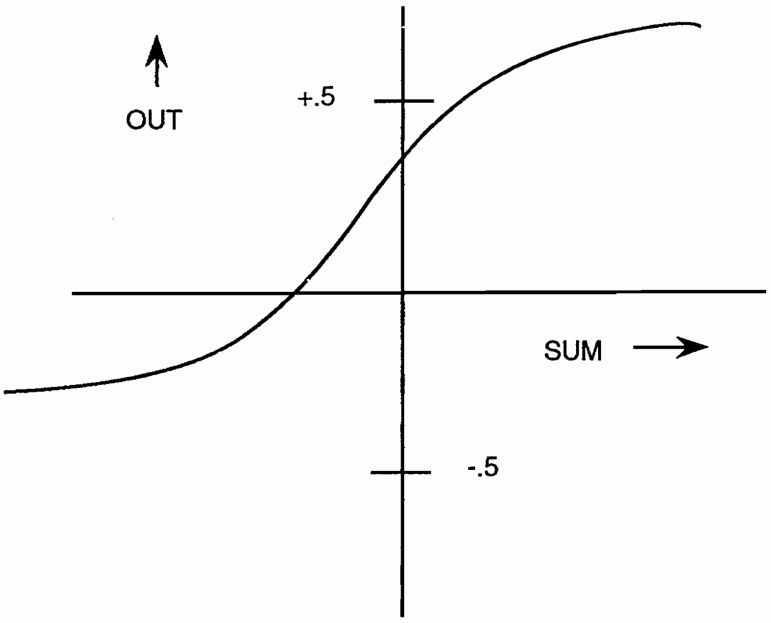


Figure 3c. Modified Sigmoid Activation Function

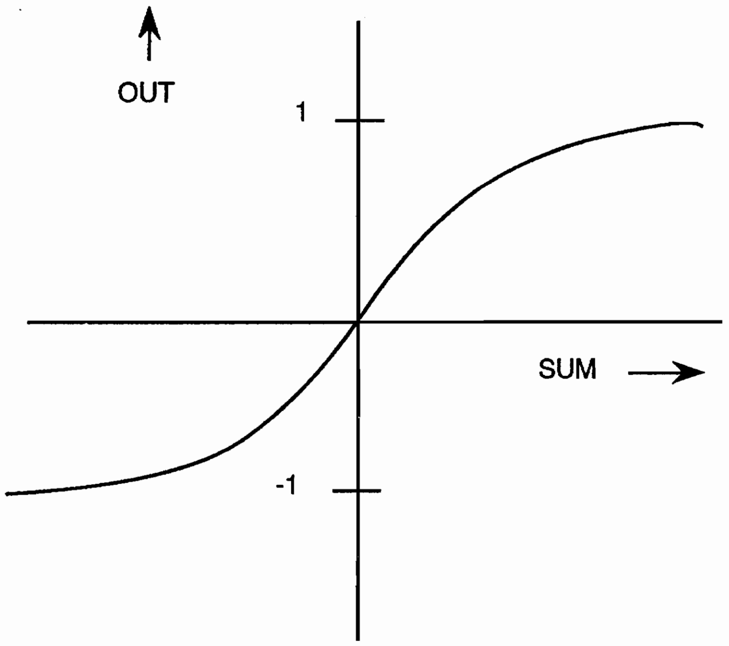


Figure 3d. Hyperbolic Tangent Activation Function

the majority of the values of OUT take on values other than values close to zero. Input range also is changed from 0/1 to -.5/+.5 which allows all weights in the first hidden layer to be trained. One common learning algorithm, backpropagation, produces training where weight changes are proportional to the output values of the input neurons. Thus, inputs of 0 allow no training of the weights through which those input values are connected. In several training algorithms, especially backpropagation, this modified activation function allows the weight to be multiplied by input values which are not close to zero. This tends to speed up training time.

Figure 3d shows another common activation function, the hyperbolic tangent function. It has the same general shape as the sigmoid. As with the Modified Sigmoid Function, this function also generates zero output when SUM = 0. However, in this case output values range from -1.0 to +1.0. OUT again generates either positive or negative numbers which allows for inhibitory signals, as well as excitatory ones.

A multi-layer, fully connected network is shown in Figure 4. A layer is considered to be a set of neural bodies and their associated input weight vectors (weight vectors are normally associated with the layer with which they affect the input). Thus Figure 4 possesses two layers. The input layer, commonly referred to in the literature as layer 0, in effect does no computation. It merely serves as a distribution

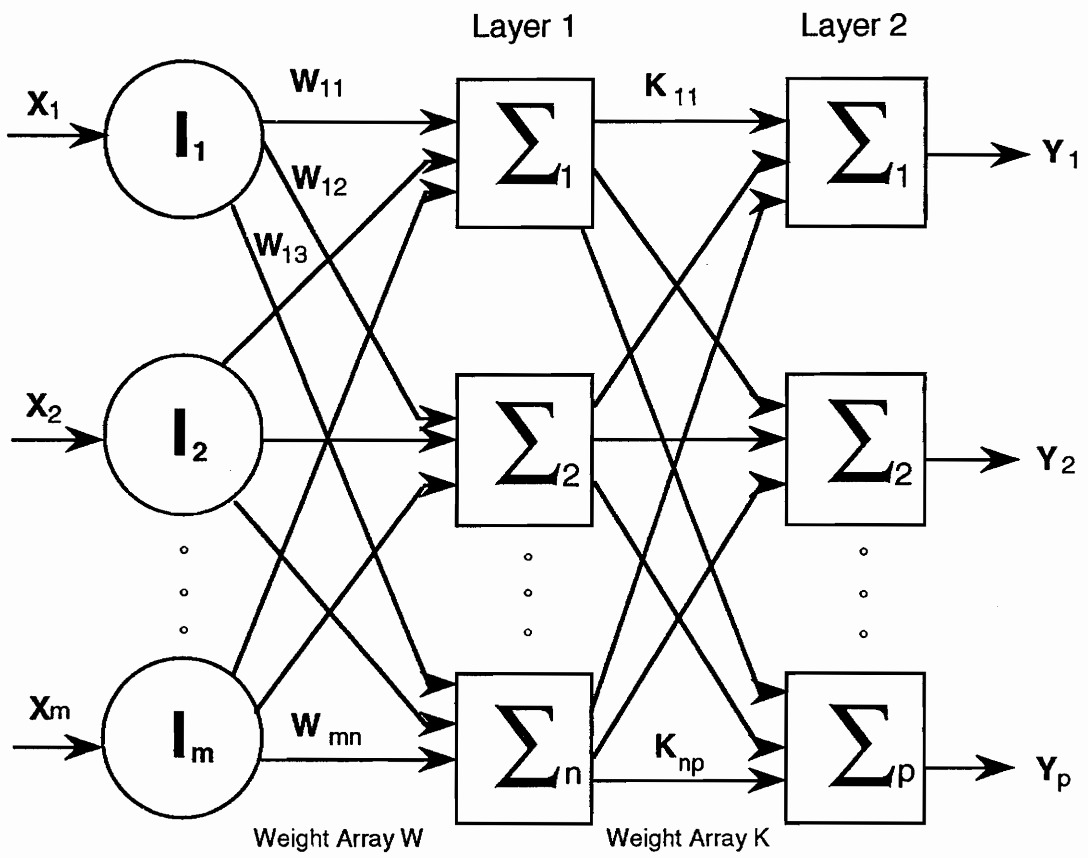


Figure 4. Two Layer Neural Network Model

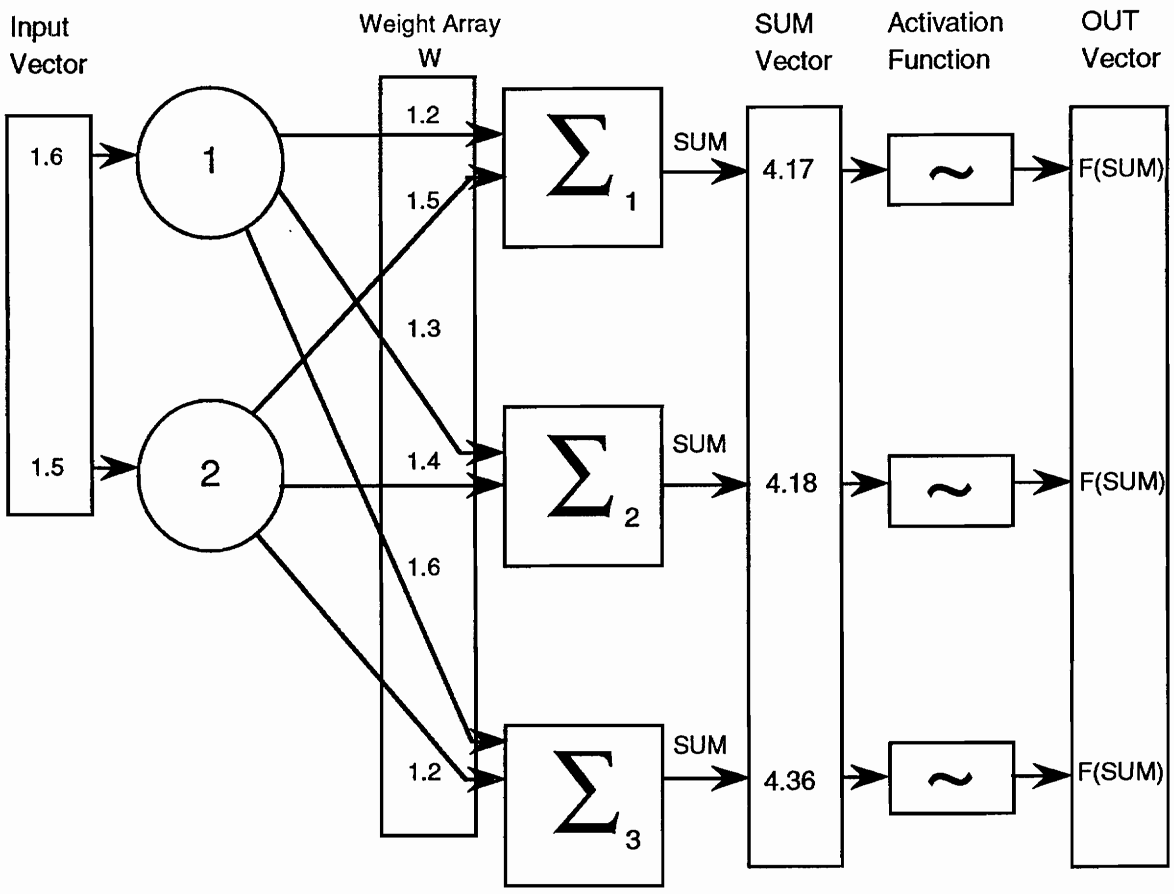
center for the first set of inputs and weights and is, therefore, not considered a true layer. Layer 0 does not posses an associated weight vector. The first set of weights is associated with layer 1. The output layer usually performs a computation through its associated weight vector and, therefore, functions as a layer. In Figure 4 the output layer is also layer 2. Layer 1 is often referred to as a "hidden" layer, since its input and output, and therefore its computational contribution to the network cannot be "seen" by the user.

The set of inputs to the network in Figure 4 has each of its elements connected to each neuron in the next layer through an individual weight, shown as W_{mn}. This fully connected network has a feedforward architecture since the output of one layer feeds forward only, as input to the next layer. Feedback, or recurrent, networks are also possible where the outputs of a layer may feed back, through an associated weight vector, to itself or to earlier layers. These feedback architectures may also be fully connected or connected to a lesser extent, as with feedforward connections.

Variations of these types of architectures are limited only by the researcher's imagination and the computational power of the machine on which the network is implemented. Often, in the machine representation, the weight vectors are considered to be elements of a matrix (W) with m rows and n columns where m is the number of inputs and n is the number of neurons. Thus $W_{2,1}$ would represent the weight connecting the second input to the first neuron in the layer. Lack of a connection between an input and a neuron would be signified by a 0 in the respective component of the matrix. Calculating the set of neuron outputs (SUM) for a layer is then a simple matrix multiplication with SUM = NW where N is a vector of inputs to the layer, W is the weight matrix, and SUM is the output vector. This calculation is illustrated in Figure 5. The activation function is then applied to SUM of each neuron resulting in OUT. This resulting output vector serves as the input vector to the next layer. In this manner, networks of any number of layers may be built.

Of all the capabilities of artificial neural networks, probably the most interesting is their apparent ability to learn. Many training algorithms have been developed but this remains an area of intense research.

Training algorithms are generally categorized as supervised training or unsupervised training. In supervised training, each set of inputs is paired with a set of correct outputs. A training set is presented, over and over, to the network until the network produces the correct outputs a satisfactory percentage of the time. The algorithm itself



$$\boxed{1.6 \quad 1.5} \times \boxed{\begin{matrix} 1.2 & 1.3 & 1.6 \\ 1.5 & 1.4 & 1.2 \end{matrix}} = \boxed{4.17 \quad 4.18 \quad 4.36}$$

Figure 5. Network Computation

feeds back the difference between the correct output and the actual output and modifies the weight values of each layer according to a predetermined procedure. Through this process, the outputs of the net gradually converge to an acceptable level. When the network produces the correct output for all inputs, it is said to have "converged" or to have reached convergence. Figure 6 shows the general, supervised learning procedure and how the results of the training are fed back through the network.

In unsupervised learning, the same general method is applied except there is no output for comparison. Again, the weights are changed according to a set algorithmic procedure until similar inputs produce outputs that are consistent. The training algorithm thus extracts statistical properties of the training set and groups the input vectors into classes. The final output patterns produced by the network must then be transformed into a comprehensible and usable form. This generally is not a difficult task but is a step not required in a supervised training scenario.

Feedforward, backpropagation networks are examples of supervised training while self-organizing networks are examples of unsupervised training. Hopfield nets (Figure 7) are single layer, recurrent networks that are often used in associative memory (content addressable) memory applications. The weights in a Hopfield net are first initialized via a very

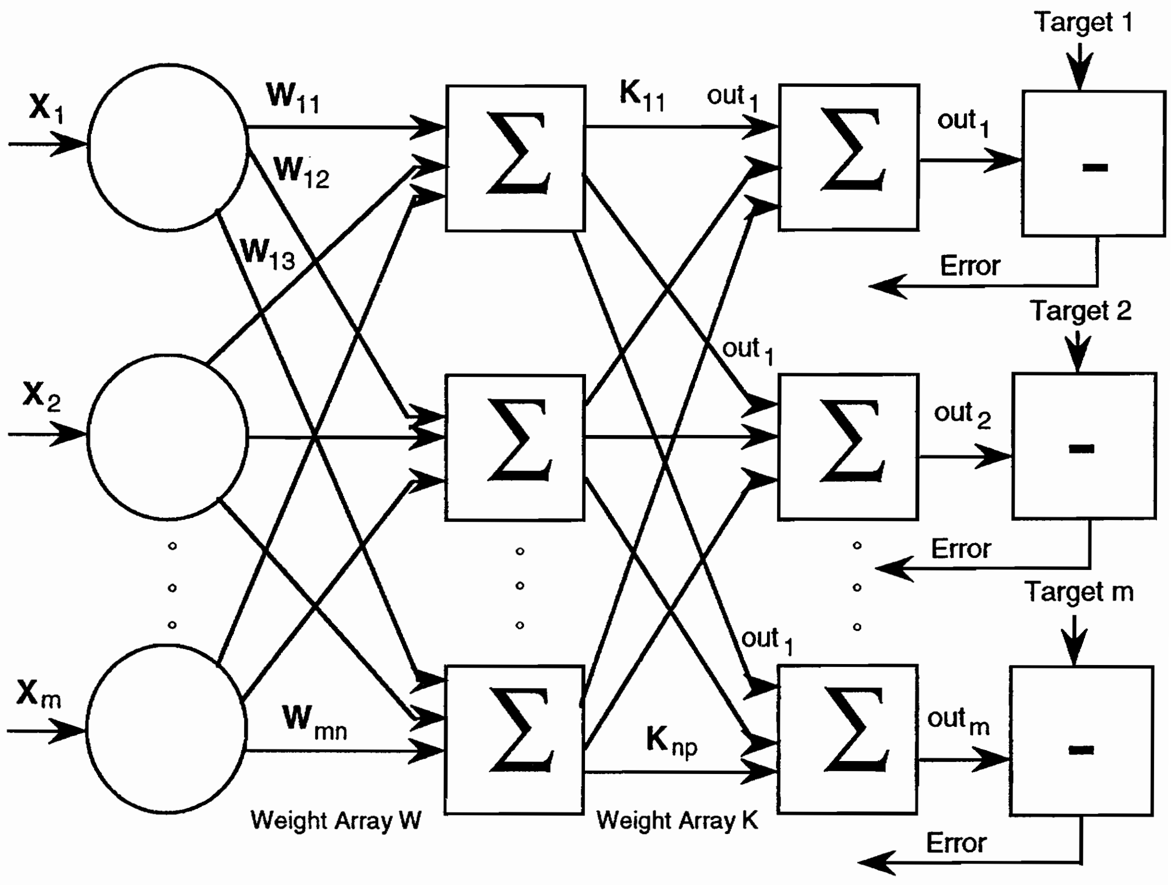


Figure 6. Network Learning

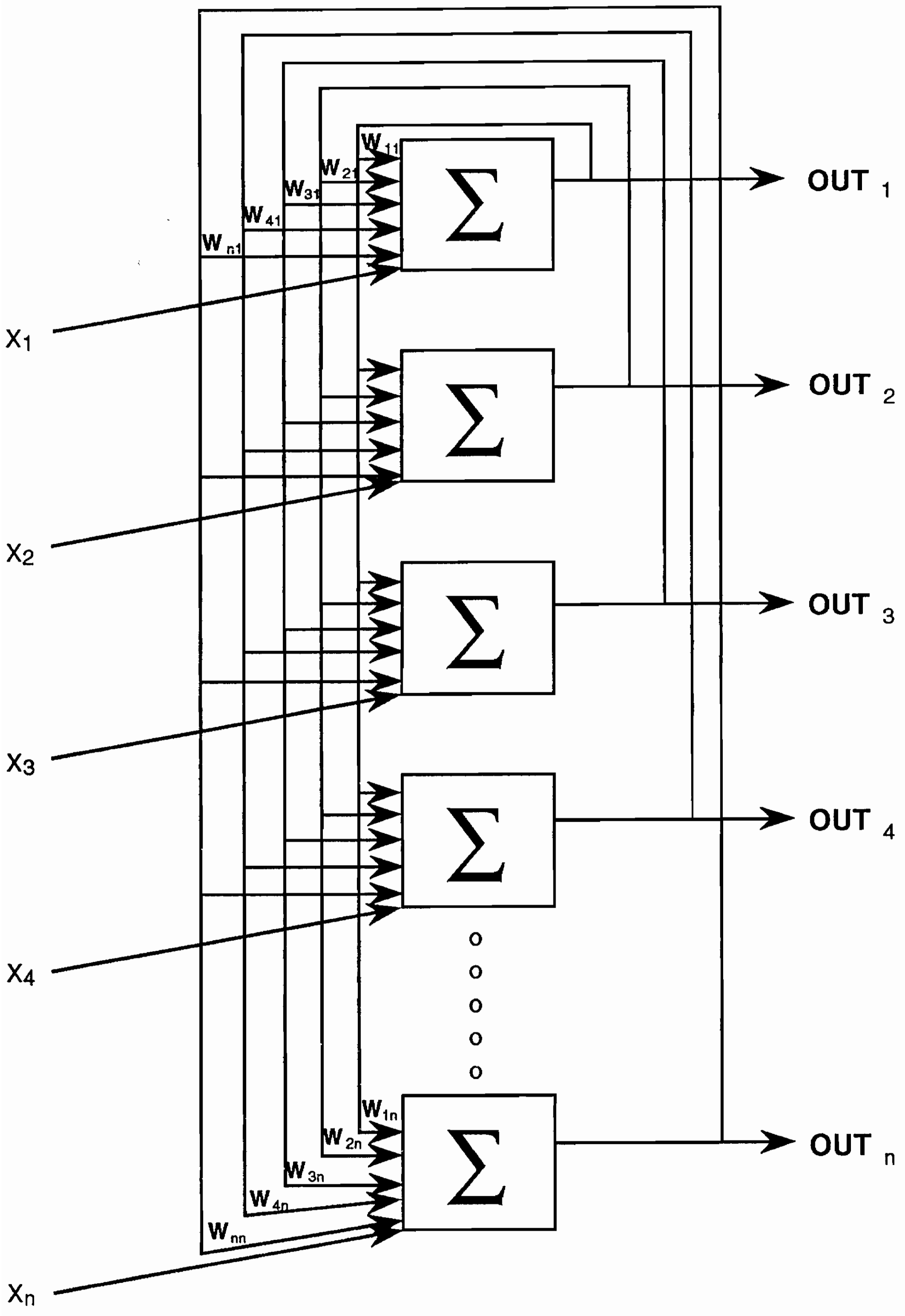


Figure 7. Hopfield Network

specific algorithm using exemplar, or example, patterns from all classes to be recognized by the network. The weights are not changed after this initialization. As such, the Hopfield net cannot be said to be trained in the normal sense. However, since this type of network is widespread in the study of ANNs it has been implemented on this development system.

The supervised training network in the form of a feedforward, backpropagation net and the recurrent, unsupervised Hopfield net will be discussed thoroughly throughout this report. Both have been implemented on the subject development system. Further implementations of self-organizing networks using the Hebbian learning algorithm will be briefly discussed. Other supervised and unsupervised nets using different training algorithms will also be discussed in the context of how they might be implemented on the prototype development system.

SECTION 3

DEVELOPMENT SYSTEM DESCRIPTION

This Artificial Neural Network Development System was designed and implemented on a personal computer. It is a system for use in research and investigation of the capabilities and functions of Artificial Neural Networks (ANN) and was designed as a tool to aid in that study. A modifiable ANN development system was evolved in which several of the common parameters in ANN implementations may be manipulated by the user without the necessity of a full software development. The ANN Development System consists of the network algorithms and computational functions, and the user interface. The interface to the system supports the selection of the variable parameters, provides a display of the parameters selected, and causes the selections to be invoked within the ANN computational software. The interface also provides for the normal input of data into the selected ANN implementation, displays that input, and displays the output of the final computation that the ANN performs on that input data. Additionally, it visually displays dynamic data as the Artificial Neural Network performs its computations.

Since the system implementation is meant to be a research and study tool, it not only supports the basic input and output common to any ANN, but it also implements the capability to vary the system's internal, normally hidden representations and keep the user informed of the choices he/she has made. These design criteria required an in-depth task analysis/use model of which ANN parameters should be variable, what the student needs to know about those variations, and how they should be displayed. Once the selections have been made by the user, the system allows data to be inputted to the ANN, computations to be made, and final net output to be displayed.

The interface for a conventional ANN--one which is meant to run a single application with a single architecture--must only support input and display the output in some format applicable to that specific application. It need only implement a single algorithm for training and output calculation. One such application was the Hopfield Net implemented by Springsteen in his Masters project [5]. The ANN development system for the present project requires a much more generic and dynamic interface which will allow the user not only to modify the parameters of the network, but also the algorithms themselves. This gives the user an adaptable, multi-purpose capability for the study of networks. The system developed for this project, therefore, has the potential for creating an unlimited number of vastly different networks, with different capacities and computational capabilities, running in different problem domains.

Since this system is intended for the study of multi-layered Artificial Neural Networks, the user will also be interested in the internal conditions of the network as it is performing its computations. He will want to be able to monitor the changing conditions of the "internals" of the network. The development system therefore also allows this monitoring of the hidden layers to take place.

Finally, the system also supports the normal input and output for the network. Its versatility will allow a wide range of input and output formats and the networks created can support a wide range of application domains.

The background section of this paper described some typical ANN configurations and how they operate. It presented the various functioning sections of an ANN and how they interrelate. These functional sections are what the user deals with when studying the effects of various factors on the networks. The user variables of an ANN include the following:

1. Number of Layers.
2. Size (number of neurons and associated weights) of each layer.
3. Connectivity (number of connections) of each layer to the next layer.
4. Direction of connectivity (forward, backward).
5. Neuron Activation Functions.
6. Learning Algorithms.

7. Various factors used to moderate, smooth, or vary the amount of training or speed of training. Momentum, training rate coefficient, saturation level, and tolerance will be discussed later in this report.

The user may have occasion to modify any one or all of the above variables in carrying out his studies of ANNs. The development system allows the user to vary these factors expeditiously in a logical fashion, as well as display the selections and parameters he has chosen.

Researchers are interested in how particular parameters vary and change during the learning cycle of an ANN. These parameters include the initial weight settings, the final weight settings, and the dynamics of how changes in weights for a particular layer occurred. They are also interested in firing rates of neurons within the net. The system allows this data to be collected and printed out at the end of a session. In addition, graphic display of data in real time during the training and operation cycle of the net has been implemented. This allows the user to visually see the net in action.

SYSTEM ARCHITECTURE

This ANN simulation system was developed to enable students to quickly implement artificial neural networks of interest, to vary network topologies and architectures, and to modify or develop new algorithms of interest in their study of the general area of Artificial Neural Networks. It has been implemented in Turbo Pascal 4.0, running on a 12 MHZ AT class machine with 1 Mb of main memory and VGA graphics capability. Transportability to other machines with CGA, EGA, or mono capability or with different system features was not a design goal. The system was designed with extensibility and ease of adding new network features in mind.

As system development progressed, it became evident that several broad categories of nets existed that were characteristically implemented in incompatible ways. This necessitated analogous categories of delineation within the system to allow for ease of implementation of these nets. Figure 8 shows the division of these categories.

Nets are first divided into the definition of either supervised or unsupervised training modes. Recurrent and feedforward nets are then specified within these categories. Within these areas, networks are usually similar in their implementation, only the specifics of basic algorithms being subject to change. The overall architecture and basic functional methodology remains similar, if not identical, across the ANNs within one of these specific categorizations. For example, a feedforward net using the backpropagation learning algorithm is basically the same net as a feedforward net using a Delta learning rule. Both use a supervised training sequence of application of the input, computation of the output, a comparing of the output to the required output, and an adjusting of the weights within the net in order to reduce the error. This sequence of events is iterated until the net output produces acceptable results. Within the two networks the specifics of the learning algorithm change, possibly even the activation function, the network topology, or the computation of the input to layers within the network, but the basic overall network paradigm remains consistent.

From the development system's standpoint, the biggest difference between supervised and unsupervised networks is the sequence of events and whether or not target vectors should be read from the input file. Specifying the networks produced on the development system in this manner enforces the sequence of events and permits a single format for the input file no matter what type of network is specified. This removes a major source for error in the development of a new network. Also, when viewed this way, networks are more easily modified and new algorithms are more easily implemented. The system

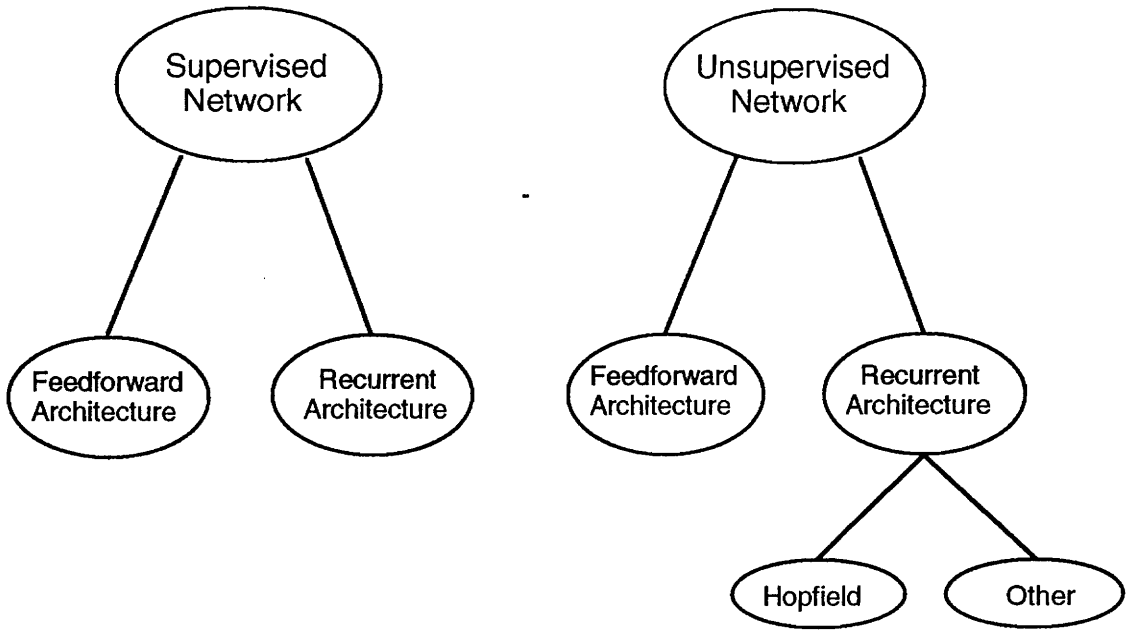


Figure 8. Network Implementation Categories

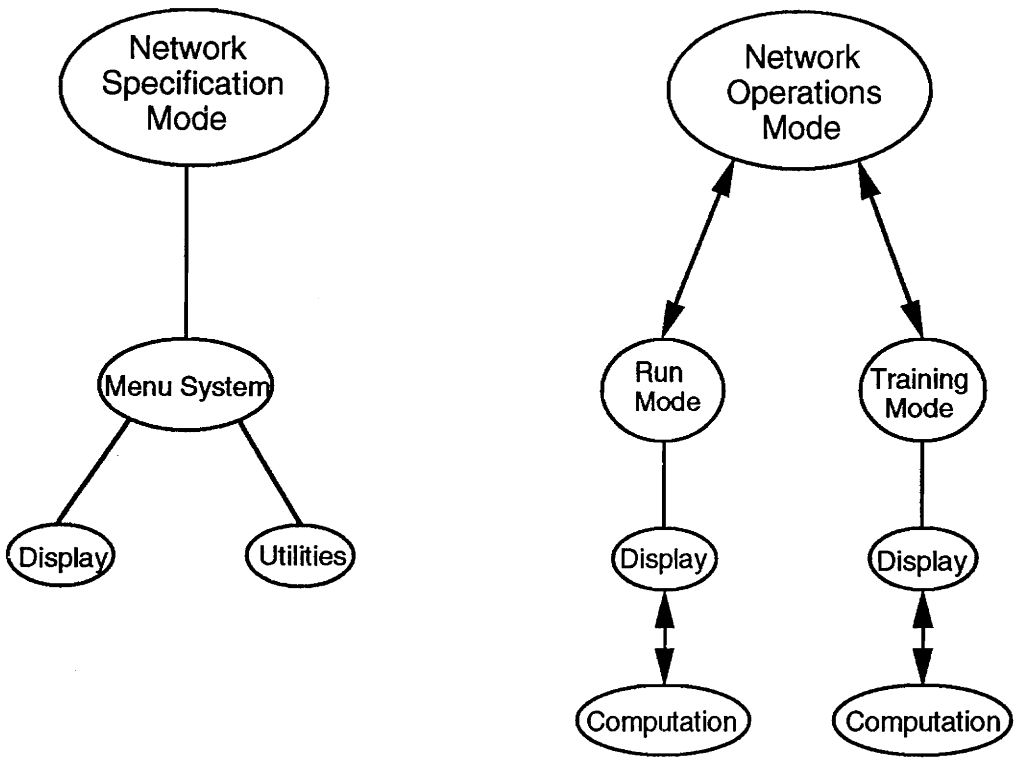


Figure 9. Development System Modes

becomes modifiable and extensible from the operator's point of view.

The system is comprised of four main functional sub-areas (Menu system, Display, Computation, and the I/O Utilities) specified in eight separate units. The source code for these units is fully reproduced in Appendices B through J. The units are:

1. unit nnglobal - global declarations
2. unit menus - menu system
3. unit weights - part of the menu system for weight setting and display
4. unit grafstuf - graphical display system
5. unit ioutil - input/output utilities
6. unit math - network computations and learning algorithms
7. unit ops_stuf - provides interface between the menu system (specification mode) and the display system (operations mode)
8. unit mathutil - initialization routines and other utilities

There are also two main modes of operations -- the Network Specification Mode and the Network Operations Mode (See Figure 9).

MENU SYSTEM

The Menu System is the primary user interface for creating a network and modifying its parameters. The Menu system is functionally rigid in the sequence displayed to the operator. This allows for a strict ordering of the sequence of events that take place and ensures that previous, required selections are made before dependent selections are made. The overall menu architecture and functionality is displayed in Figure 10.

The menu system itself is comprised of seven functional areas, and their associated menus. As shown in Figure 10, these are designated as the Main, Layer Data, Net Architecture, Connectivity, Weights, Activation Function, and Learning Algorithm menus. In many cases, sub-menus within these functional areas are available for individual parameter selection. Individual functional area menu and sub-menu displays are represented in Figures 12 through 24.

Figure 11 depicts the generic menu layout chosen for the development system. When the system is first turned on, the Main System Menu, Figure 12, is displayed and a default ANN is created. The operator is now in the Network Specification Mode of operation. A previously created network may now be loaded from disk or a network may be created manually through the use of the menu system. If a network is loaded from disk,

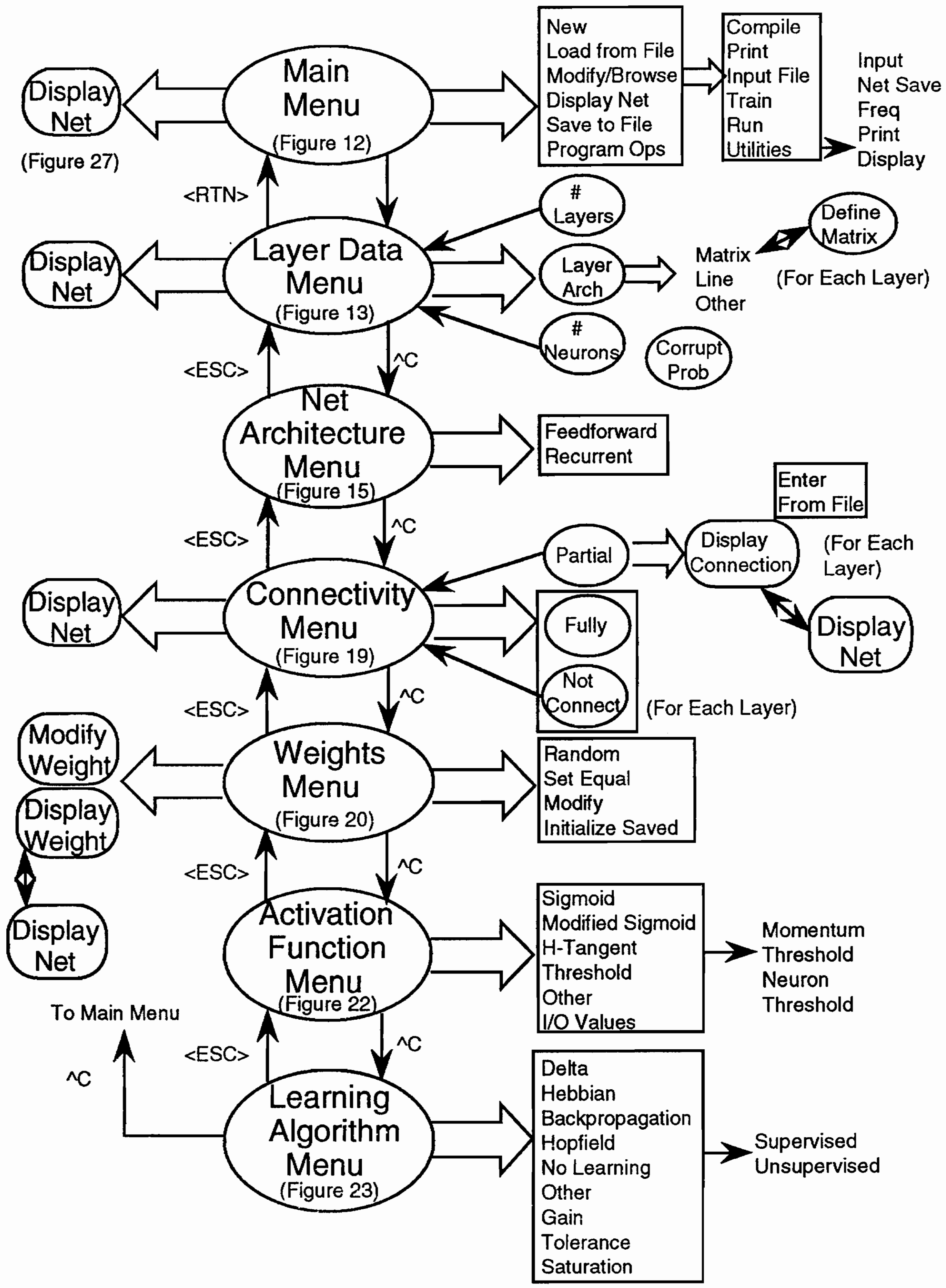
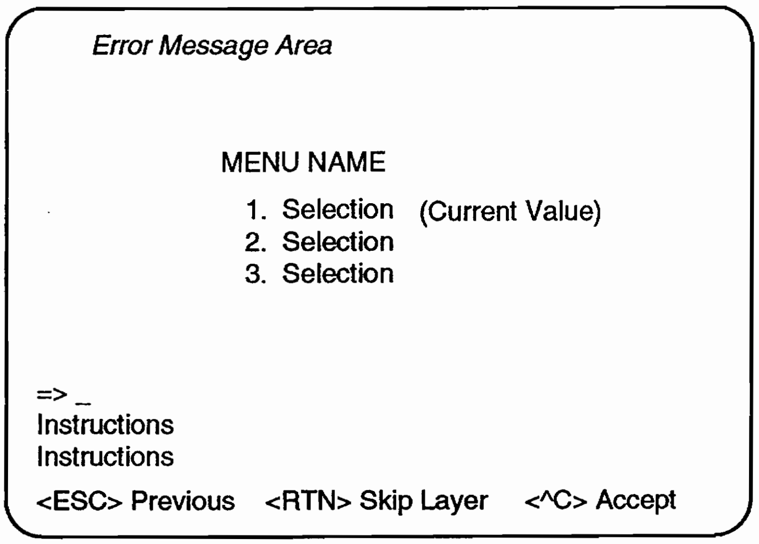


Figure 10. Menu System Interface Architecture

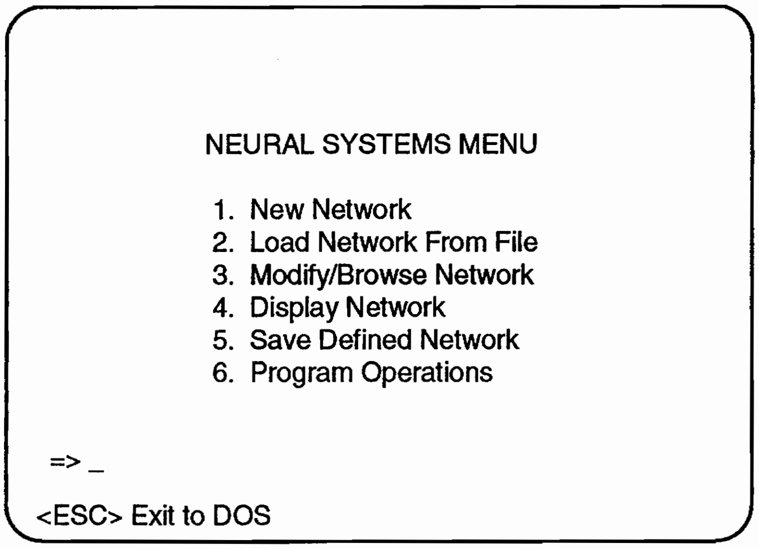

Error Message Area

MENU NAME

1. Selection (Current Value)
2. Selection
3. Selection

```
=> _
Instructions
Instructions
<ESC> Previous    <RTN> Skip Layer    <^C> Accept
```

Figure 11. Menu Template


NEURAL SYSTEMS MENU

1. New Network
2. Load Network From File
3. Modify/Browse Network
4. Display Network
5. Save Defined Network
6. Program Operations

```
=> _
<ESC> Exit to DOS
```

Figure 12. Main Menu

modifications may still be made via the sequence of menus. Networks may also be saved to disk from the Main System Menu (See Appendix A). The menu system allows the operator to create a network from established paradigms in a building block fashion and allows for the modification of specific parameters within the network, as discussed previously. Creation of new algorithms or modifications of existing ones for use within the net must be done via actual modification/addition to the Pascal code of the system. The code for these areas has intentionally been developed to be readily modifiable in order to accomplish this requirement. Only a certain specified number of easily identified and collocated procedures are required to be modified in order to add new algorithms. The system, as presented, does not include the capability to compile new code internally from within the system.

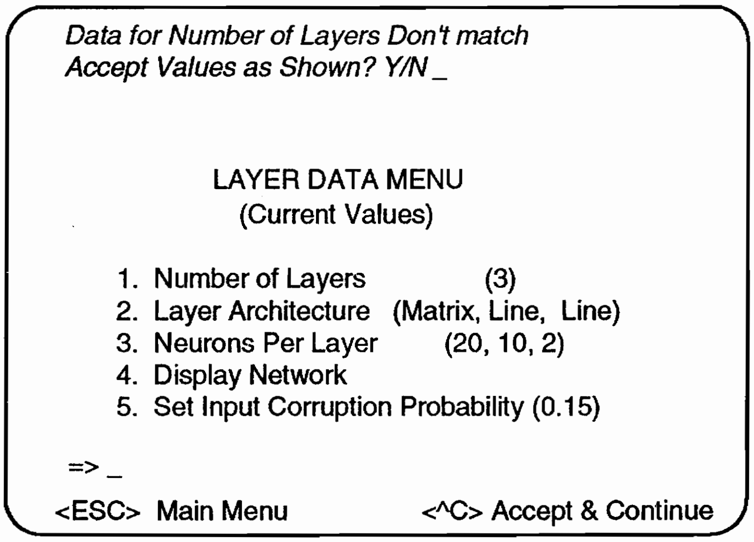
Once a major function (Layer Data, Net Architecture, Connectivity, Weights, Activation Function, Learning Algorithm) has been selected and completed, it is accepted via a ^C command. This leads to the next menu in the chain for further selections. An <ESC> from any of these menus returns to the previous menu although it maintains any changes made to the menu just vacated. A ^C command at the final menu (Learning Algorithm) unwinds the chain and returns the user to the Main menu. Sub-menus offer selections to be made within the main function but may be accepted by <ESC>, <RTN>, etc. depending on the individual menu. This selection is specified in all cases on the command line (line #24) of the individual menus.

As shown in Figure 10, at several steps along the way, a Display Net option is available to the operator to allow display of the network content. Figure 27, page 53, shows an example of the display that is available. This gives the operator a graphical representation of all entries. He can also browse through the menu system for a textual representation of the same data.

The overall layer specification is made from the Layer Data menu (See Figure 13). Here the number of layers is selected, the number of neurons per layer (currently a maximum of 80 due to computer memory limitations) and the layer architecture are specified. As shown in Figure 10, selection of a choice in the Layer Data menu leads to a sub-menu for that selection (See Figures 14, 15, and 17).

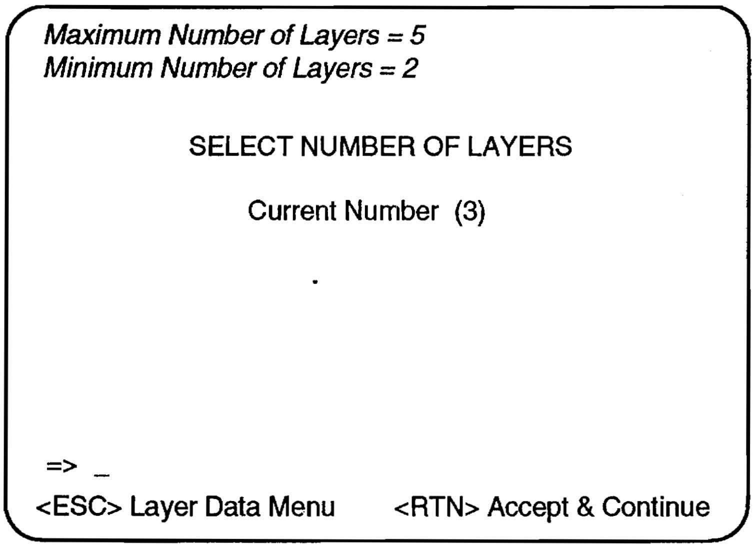
The layer architecture is specified as either a line or matrix. This is strictly for display purposes since the internal representation is simply a one dimensional array of neurons. If a matrix architecture is selected the Define Matrix menu, Figure 16, is displayed after acceptance of the layer data specification.

The convention for layer numbering in this development



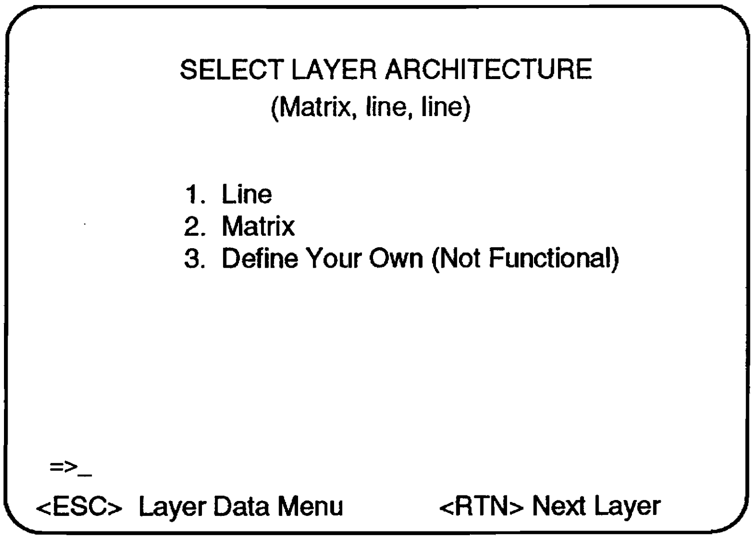
```
Data for Number of Layers Don't match
Accept Values as Shown? Y/N _


                    LAYER DATA MENU
                     (Current Values)

        1.  Number of Layers              (3)
        2.  Layer Architecture   (Matrix, Line,  Line)
        3.  Neurons Per Layer         (20, 10, 2)
        4.  Display Network
        5.  Set Input Corruption Probability (0.15)

 => _
<ESC>  Main Menu                <^C> Accept & Continue
```

Figure 13. Layer Data Menu



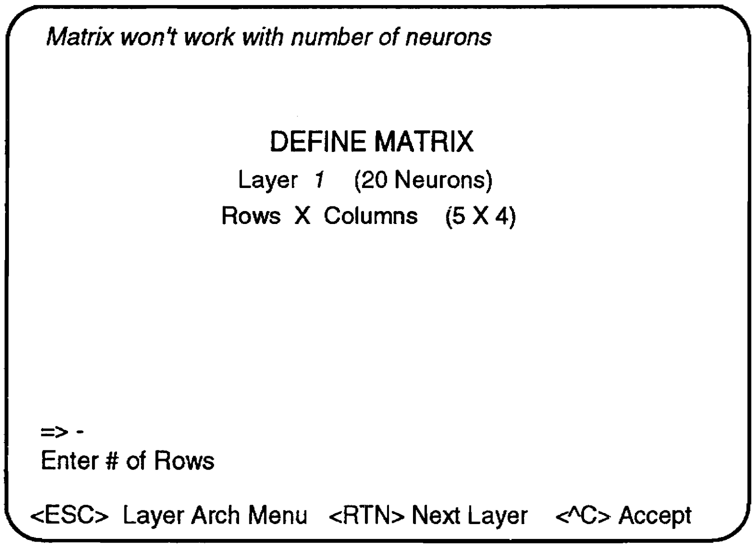
```
Maximum Number of Layers = 5
Minimum Number of Layers = 2

                SELECT NUMBER OF LAYERS

                    Current Number  (3)





=>  _
<ESC> Layer Data Menu        <RTN> Accept & Continue
```

Figure 14. Number of Layers Selection



```
                    SELECT LAYER ARCHITECTURE
                         (Matrix, line, line)

                    1.  Line
                    2.  Matrix
                    3.  Define Your Own (Not Functional)



 =>_
<ESC>  Layer Data Menu            <RTN> Next Layer
```

Figure 15. Layer Architecture Selection Menu



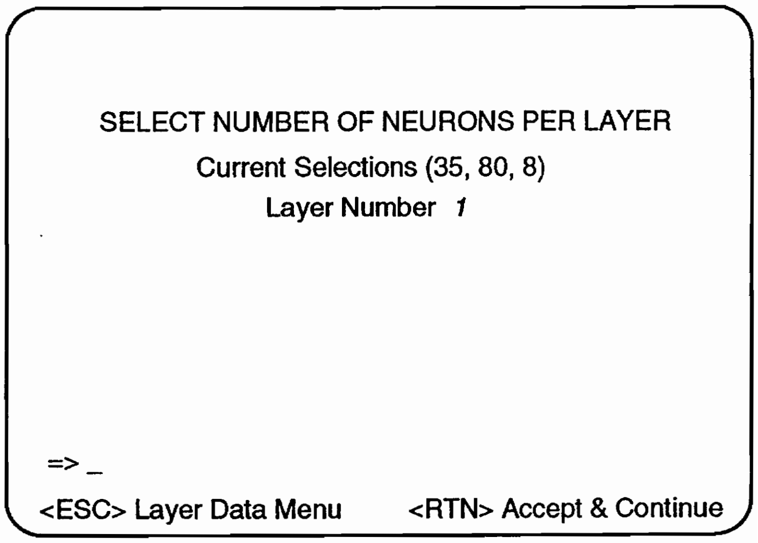
```
 *Matrix won't work with number of neurons*

                         DEFINE MATRIX
                      Layer  1    (20 Neurons)
                    Rows  X  Columns    (5 X 4)



=> -
Enter # of Rows
<ESC>  Layer Arch Menu   <RTN> Next Layer    <^C> Accept
```

Figure 16. Define Matrix Menu

system is for the input layer to be layer 1 (not as discussed in the background section, where it was noted that, in the literature, the input layer is normally layer 0). This convention was chosen to maintain consistency throughout the source code of the development system. All array and matrix indices specified throughout the source code also begin with 1 rather than 0. Thus all computations within a network specified on the development system will commence with layer 2. A single-layer network will actually be specified as having two layers. Layer 1 is always the input layer and provides no computational value, as was discussed previously in relation to layer zero. Layer one merely distributes the input to layer two, the first computational layer in the network.

The corruption probability selection from the Layer Data menu allows for the specification of a percentage of neurons in the input layer to be randomly selected for changing the binary value of their outputs. This simulates noise within the system and a less than perfect input in a random fashion and is often used in the study of how well certain networks generalize and produce correct results with partial input.

Following the Layer Data Selection Menu is the Network Architecture Menu (Figure 18). Feedforward and recurrent options are available for selection. Fully recurrent networks have not been implemented in this version of the system. If

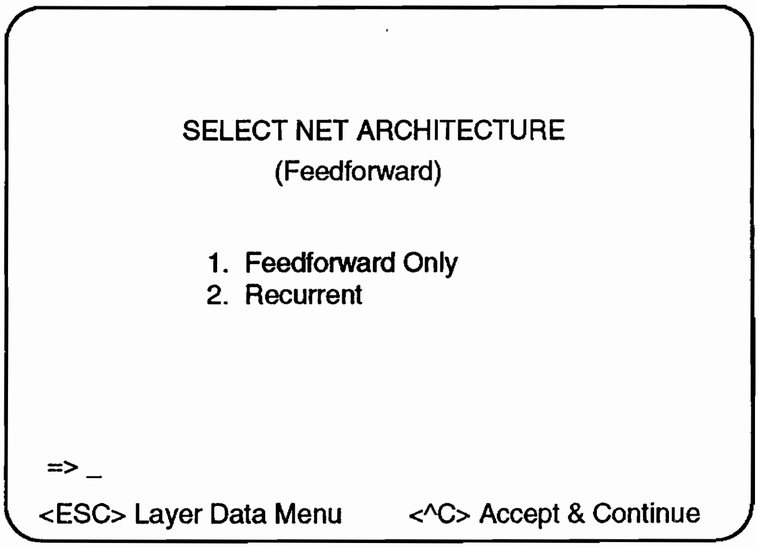


SELECT NUMBER OF NEURONS PER LAYER
Current Selections (35, 80, 8)
Layer Number *1*

=> _

<ESC> Layer Data Menu   <RTN> Accept & Continue

Figure 17. Neurons Per Layer Selection Menu



SELECT NET ARCHITECTURE
(Feedforward)

1. Feedforward Only
2. Recurrent

=> _

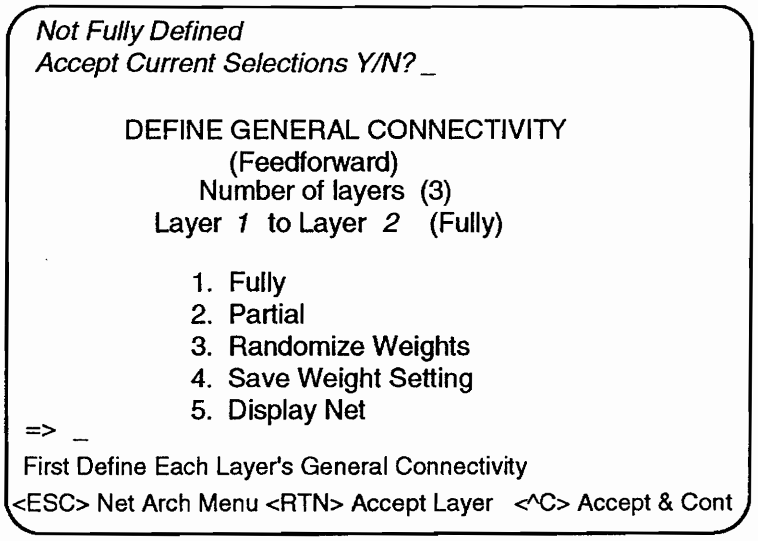
<ESC> Layer Data Menu   <^C> Accept & Continue

Figure 18. Network Architecture Selection Menu

a recurrent network is specified, the output of the output layer is fed back as input to the input layer. A fully recurrent network would allow the selection of various options to feed back output from any layer to any previous layer, as well as to itself. These various options were not implemented due to memory limitations and remain as future system enhancements to be developed for higher capacity machines. Each optional set of layer connections requires an 80 X 80 matrix of weight values, and, possibly, a separate 80 X 80 matrix for saved_weight values, depending on the training algorithm.

The Connectivity Menu (Figure 19) allows for the selection of fully connected or partially connected layers. Full connectivity is the default condition. Each layer may be sequenced through and specified as fully or partially connected. When a layer has been specified as partially connected, the operator may remove or replace individual connections as well as remove a random number of connections. He may also restore full connectivity. Figure 19a displays the menu for modifying the connections of a layer. This menu is invoked if the operator has defined one or more layers as partially connected. This same functionality is also available in the display mode while the network is in the run mode.

Each layer has a full matrix of weights available through



Not Fully Defined
Accept Current Selections Y/N? _

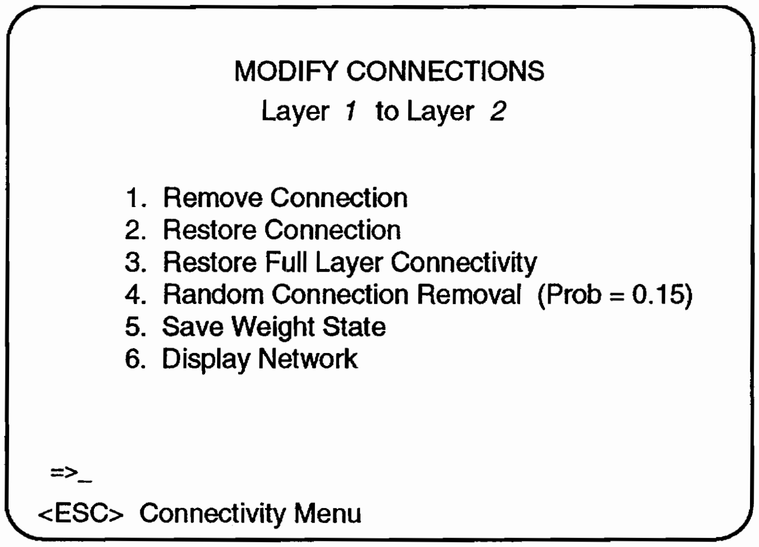
DEFINE GENERAL CONNECTIVITY
(Feedforward)
Number of layers (3)
Layer *1* to Layer *2* (Fully)

1. Fully
2. Partial
3. Randomize Weights
4. Save Weight Setting
5. Display Net

=> _

First Define Each Layer's General Connectivity
<ESC> Net Arch Menu <RTN> Accept Layer <^C> Accept & Cont

Figure 19. Define Connectivity Menu



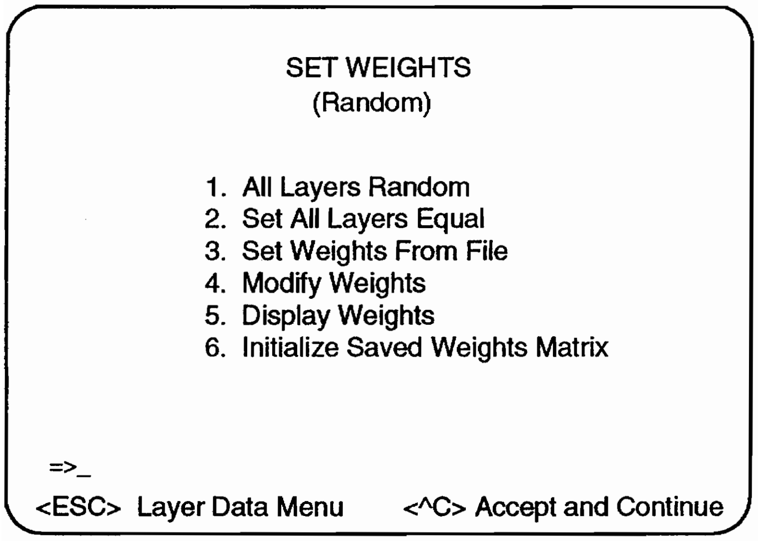
MODIFY CONNECTIONS
Layer *1* to Layer *2*

1. Remove Connection
2. Restore Connection
3. Restore Full Layer Connectivity
4. Random Connection Removal (Prob = 0.15)
5. Save Weight State
6. Display Network

=>_

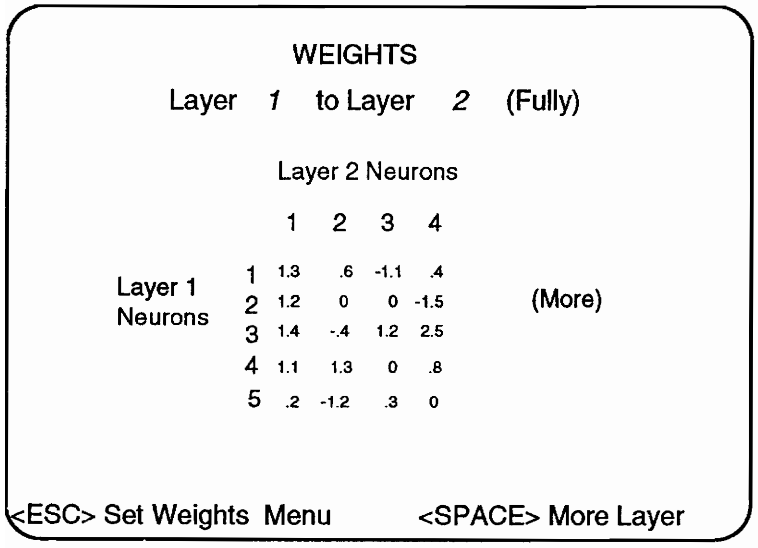
<ESC> Connectivity Menu

Figure 19a. Modify Connections Menu



```
                      SET WEIGHTS
                       (Random)

              1.  All Layers Random
              2.  Set All Layers Equal
              3.  Set Weights From File
              4.  Modify Weights
              5.  Display Weights
              6.  Initialize Saved Weights Matrix


 =>_
<ESC>  Layer Data Menu       <^C> Accept and Continue
```

Figure 20. Set Weights Menu



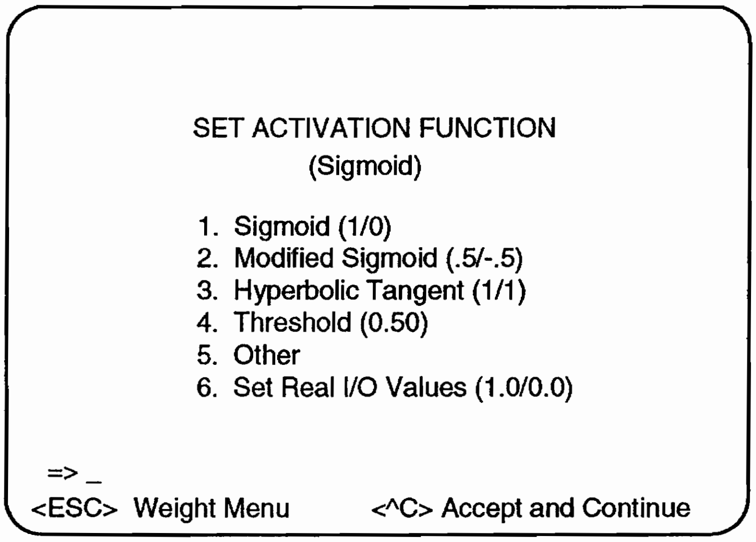
```
                       WEIGHTS
            Layer  1   to Layer  2   (Fully)

                   Layer 2 Neurons

                    1    2    3    4

 Layer 1       1   1.3   .6  -1.1   .4        (More)
 Neurons       2   1.2   0    0   -1.5
               3   1.4  -.4   1.2  2.5
               4   1.1   1.3  0    .8
               5   .2  -1.2   .3   0


<ESC> Set Weights  Menu         <SPACE> More Layer
```

Figure 21. Display Weights

which it connects to the previous layer. The weight matrices are implemented as dynamic variables, with pointers specified in the layer description (See Appendix B). Weight values may be displayed and/or modified via the menu selections (Figures 20 and 21). Separate dynamic weight matrices are implemented for saving weight values from previous iterations during training. These saved weight values are used in several training algorithms and are available for operator use if needed for his specific requirements.

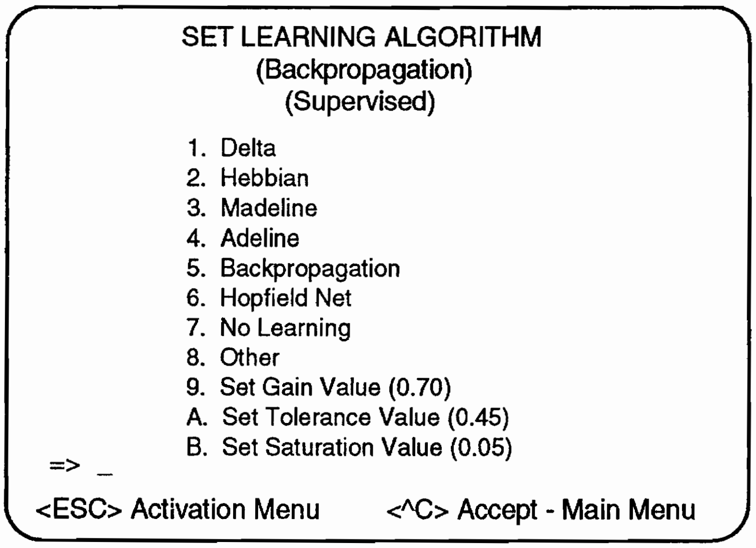
The current system uses the saved weight values in the backpropagation training algorithm. If they don't already exist, the operator may select an option to initialize a set for each network layer. This action creates the new dynamic variables and initializes their values to zero. This is often appropriate when training is restarted. Randomizing the weight matrices is also available. This is often done at the start of training to ensure that training will take place. It has been shown [3, 4] that training often cannot be accomplished if the initial set of weights are equal.

Various neuron activation functions are available for use, or the operator may specify and code his own. The interface specification must be adhered to for self coding. The activation functions implemented in the current system are depicted in Figure 3a, 3c, and 3d and are specified in Appendix H. Selections are made from the Activation Function



```
                SET ACTIVATION FUNCTION
                       (Sigmoid)

                1.  Sigmoid (1/0)
                2.  Modified Sigmoid (.5/-.5)
                3.  Hyperbolic Tangent (1/1)
                4.  Threshold (0.50)
                5.  Other
                6.  Set Real I/O Values (1.0/0.0)

  => _
<ESC>  Weight Menu          <^C> Accept and Continue
```

Figure 22. Activation Function Menu



```
               SET LEARNING ALGORITHM
                  (Backpropagation)
                    (Supervised)
               1.  Delta
               2.  Hebbian
               3.  Madeline
               4.  Adeline
               5.  Backpropagation
               6.  Hopfield Net
               7.  No Learning
               8.  Other
               9.  Set Gain Value (0.70)
               A.  Set Tolerance Value (0.45)
               B.  Set Saturation Value (0.05)
  =>  _

<ESC> Activation Menu        <^C> Accept - Main Menu
```

Figure 23. Learning Algorithm Selection Menu

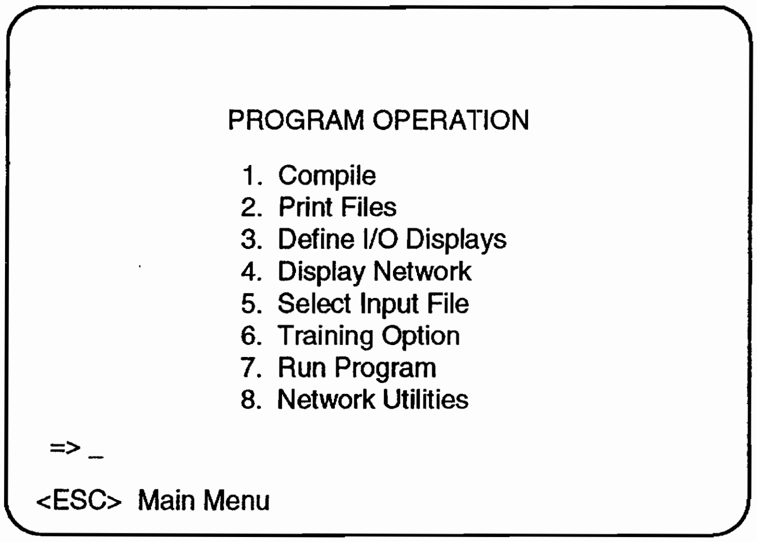
Menu, Figure 22. Default floating point input/output values to represent the binary value of the neuron are specified for each activation function (Sigmoid (1.0/0.0), Modified Sigmoid (0.5/-0.5), Threshold (1.0/0.0), Hyperbolic Tangent (1.0/-1.0)), but may be modified to fit the operator's desires and specific algorithm requirements in the Activation Function menu. When a particular activation function is selected, the operator is then cued to choose values for specific parameters required by the function (threshold values, momentum, etc.). Momentum is a parameter often used in the backpropagation algorithm for smoothing (See the discussion on backpropagation).

The final menu selection (Figure 23) is for the specific learning algorithm. Various implemented options are available and are specified in Appendix H. A no-learning option may be selected, but is not specifically required for simple operation of the net with no training. The operator may specify and code his own algorithm ensuring that he respects the interface specified in Appendix H. In this case the "other" selection would be appropriate for use and the new code for the learning algorithm would be placed in the "other" selector in the case statement for procedure Learning_Rule in the Math unit (See Appendix H). If the operator produces code for his own learning algorithm, he must specify whether it is a supervised or non-supervised algorithm as this will effect the sequence of events occurring within the network.

Hopfield nets are a special class of networks, as previously discussed, and have been coded into this system since they are often studied. Selection of Hopfield in this menu sets up a specific chain of events unique to the Hopfield network. This sequence of events is discussed under the Hopfield implementation section of this report.

With this same menu (Figure 23), the operator may also specify certain factors used in the training algorithms selected. Current choices available include gain, saturation, and tolerance. These equate to factor1, 2, and 3 respectively in the net_description data type (See Appendix B). These factors may be specified as the operator sees fit for user defined algorithms.

Acceptance of this final menu selection returns the operator to the Main Menu (Figures 10 and 12). The network is now fully specified. From the Main Menu the operator may return through any of the operations discussed previously, change any of the selections, or browse through the specification to ensure correct selections. Finally, the operator may select Network Operations which then puts him into the Network Operations Menu (Figure 24) and the Network Operations Mode (Figure 9). From this menu the operator may select the functions of the Operations Mode, including selection of an input file, printing of the network save_state



PROGRAM OPERATION

1. Compile
2. Print Files
3. Define I/O Displays
4. Display Network
5. Select Input File
6. Training Option
7. Run Program
8. Network Utilities

=> _

<ESC> Main Menu

Figure 24. Program Operation Menu

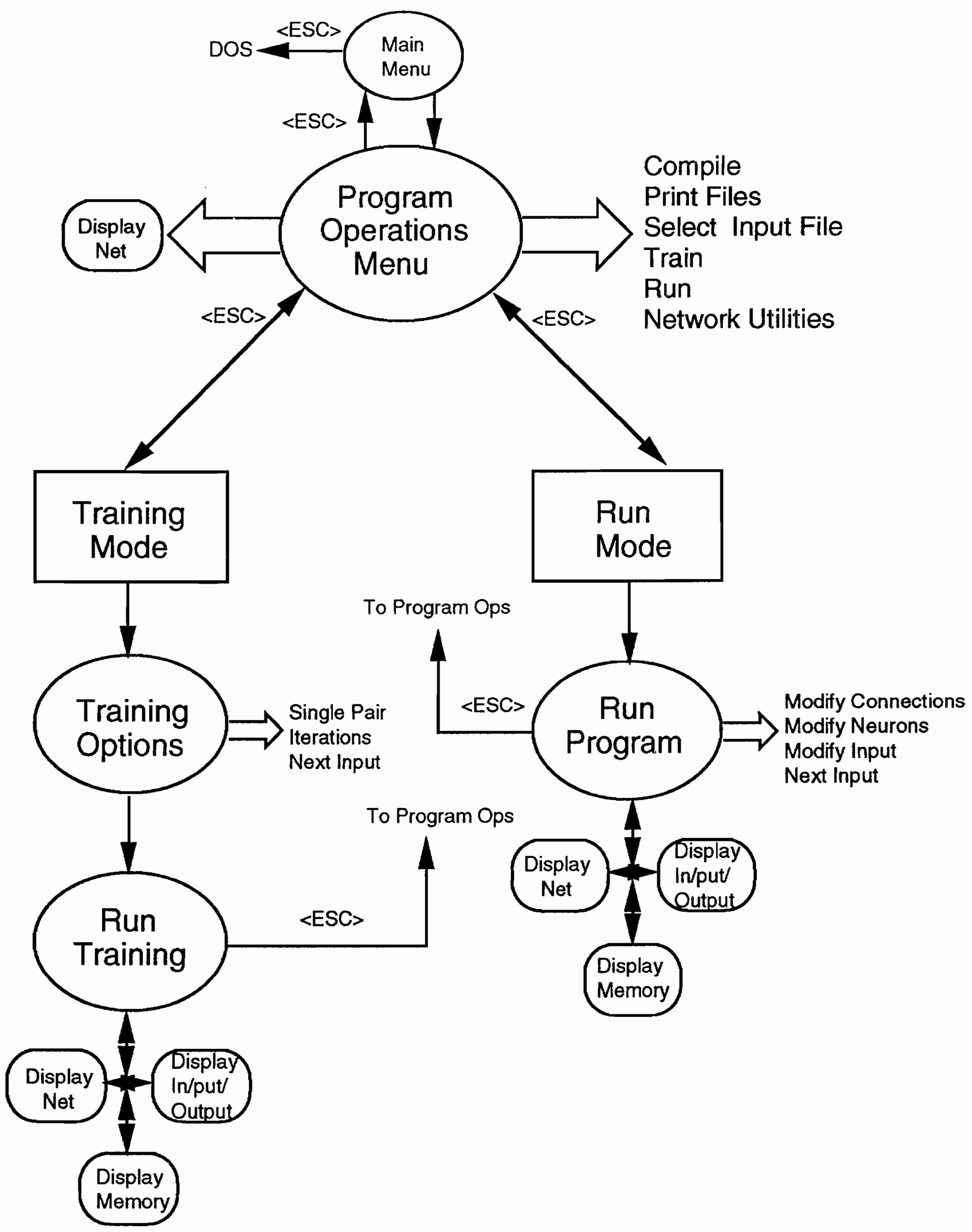


Figure 25. Operations Mode Control Flow

files, or Network Utilities. The compilation function is not currently available, as stated earlier.

Also available for selection are the Training mode and Run mode. Either one of these selections puts the operator into the display system and readies the system to run or train the specified network. If Run or Train Mode is selected before an input file is specified, an error message is generated and the system is returned to Operation Menu to select the file. Operational flow of control is depicted in Figure 25.

The network and neuron architectures of the Artificial Neural Network development system are embodied in two main data structures -- net_description_type and layer_type. The source code specification may be found in Appendix B. These two structures are records and have the following construction:

net_description_type (specifies network architecture)

```
net_name     : DOS file name
net_architecture : feedforward or recurrent
num_layers : number of layers in the network
activation : activation algorithm (threshold, sigmoid,
               mod_sigmoid, user defined, etc.)
threshold    : threshold value (used in most training
               algorithms)
learning     : type of learning algorithm to be used
               (Delta, Backpropagation, user-defined,
               etc.)
Factor      : Training rate coefficient (ie. n in Delta
               Rule - see below)
Factor1      : User_defined factor
```

```
Factor2       : User_defined factor
Factor3       : User_defined factor
momentum     : Factor used in Backpropagation Algorithm
               for smoothing
random_weights : boolean for initial weight settings
                 of random rather than hand set or
                 loaded from a file.
save_state : Boolean to save network state
             periodically
iterate       : How often to save the network state
one_value    : Real value used internally by the net to
               represent a binary 1 (usually 1.0 or
               0.5).  Default values depending on
               activation function or may be specified
               by the user.
zero_value : Real value used internally by the net to
             represent a binary 0 (0.0, -0.5, -1.0).
             See one_value.
```

layer_type (Specifies individual layer data - layers is an array variable [1..num_layers] of layer_type. Each layer - up to five - has a layer_type associated with it)

```
connection_kind : fully or partially connected to
                  next layer
neurons    : number of neurons in the layer
neural_set : array of record of values for each
             neuron in the layer.  The record
             contains arrays for the binary value
             (1/0), the real value used in net
             computations, and a threshold value
             for use in algorithms that make use
             of individual neuron thresholds.  Also
             included is a rate value which shows
             how often the neuron has fired.
architecture : line or matrix.  Used for display
               purposes.
arch_row_length : used for matrix architecture
arch_col_length : used for matrix architecture
weights : a pointer to the weights matrix
          representing each neuron's weight
          connection to each of the previous layer's
          neurons.
saved_weights : a pointer to a weights matrix
                similar to weights.  Used in some
                training algorithms to remember the
                previous set of weights before a
```

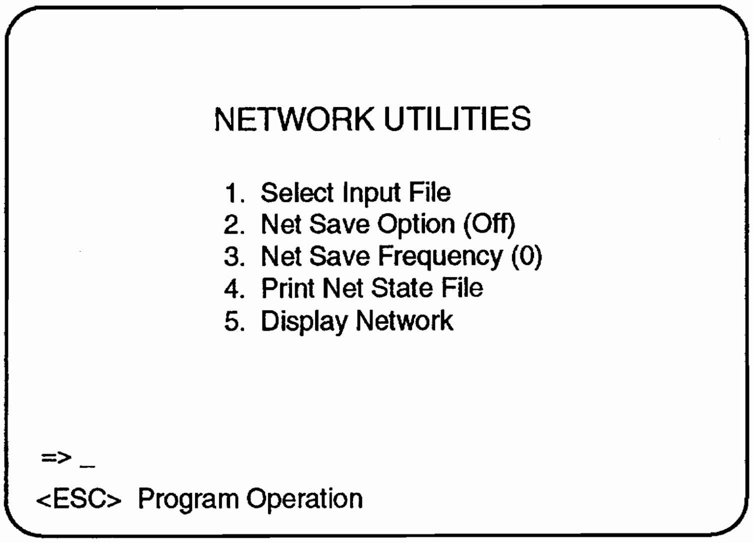
	training iteration. May be used for other user defined purposes.
delta_vector	: Array of real numbers to hold the Delta values. Used in several training algorithms.
connections	: a pointer to a connection matrix to represent if a connection is present between a neuron in layer n to a neuron in layer n + 1;
out_layer	: boolean value representing the last layer in the net - the output layer.

These two data structures, taken together, completely represent the state of the network. Specific data from these two structures are saved to disk files periodically during network operation and training if requested through the Network Utilities Menu, Figure 26. See Appendix A for a description of how the network is saved.

NETWORK UTILITIES

The Network Utilities menu, Figure 26, is selectable from the Program Operations menu and makes certain specific functions available to the operator. Currently implemented are selection of an input file (identical to the function available on the Program Operations Menu described earlier), Network Save, the frequency of state saves, a print option, and the Display Net option which is identical to that implemented throughout the system as described above.

The Net Save function is a toggle switch (boolean



```
NETWORK UTILITIES

1. Select Input File
2. Net Save Option (Off)
3. Net Save Frequency (0)
4. Print Net State File
5. Display Network

=> _
<ESC> Program Operation
```

Figure 26. Network Utilities Menu

variable in Net_Description) which tells the system if the operator wants to save the state of the network periodically to a file. This is most often used during training in order to have a record of how the network progresses toward convergence during the training cycle. It is most useful during continuous, automatic training. The frequency selection allows the operator to specify the frequency of the saves. This is a simple counter that saves the state of the network after the proper number of iterations have been made. This "snapshot" of the network is appended to the file. See Appendix A for a description of the network save state option. The data saved is basically the data in the net_description and layers data structures of the implemented network (see Appendix B). The data saved could then be used as the input for another program developed especially to analyze this data. The print function allows the operator to print out the history of net saves that he has maintained during network training. Appendix J provides an example of the printout of the saved network data.

DISPLAY

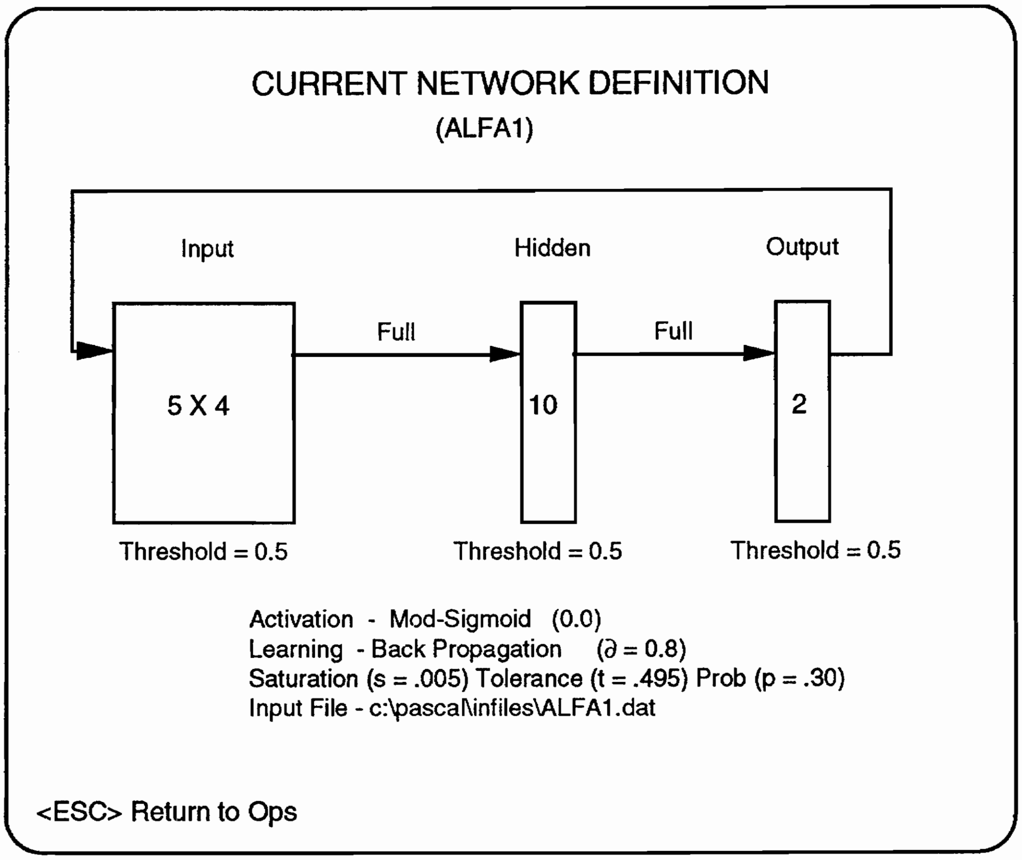
The Display system is the primary user interface during network operation, computation and training. As discussed before under the Menu System, included also in the display system is the graphical representation of the network architecture and parameters selected up to the point of display (Figure 27).

The display interface uses the Turbo Pascal graphics capability to display the network input and corresponding output. The display is divided into two sections and shows the input and output layers of the network, in either a line or matrix architecture as specified in the network parameters. An example display is shown in Figure 28.

Figure 28 depicts a network with a 6 X 4 neuron matrix architecture as the input layer and an 8 neuron line architecture as the output layer. A colored square represents a neuron with a binary output of 1 while a white or empty square represents a neuron with a binary output of 0.

When a network has been specified as requiring supervised learning and as having a line architecture as the output, two output displays are presented, as shown in Figure 28. The line on the left represents the actual output of the network while the one on the right is the target output as specified in the training set. This allows the operator to visually compare the two in order to see how the network is performing. This dual display of output is available in both the training and run mode of the network.

The mode line at the top of the display labels the input and the output and the mode of operation -- either the Run

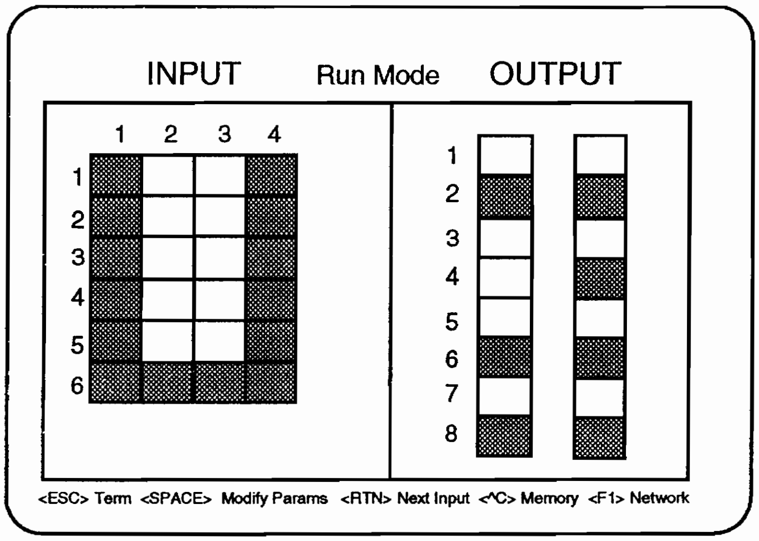


CURRENT NETWORK DEFINITION
(ALFA1)

Activation - Mod-Sigmoid (0.0)
Learning - Back Propagation (∂ = 0.8)
Saturation (s = .005) Tolerance (t = .495) Prob (p = .30)
Input File - c:\pascal\infiles\ALFA1.dat

<ESC> Return to Ops

Figure 27. Display Network Screen

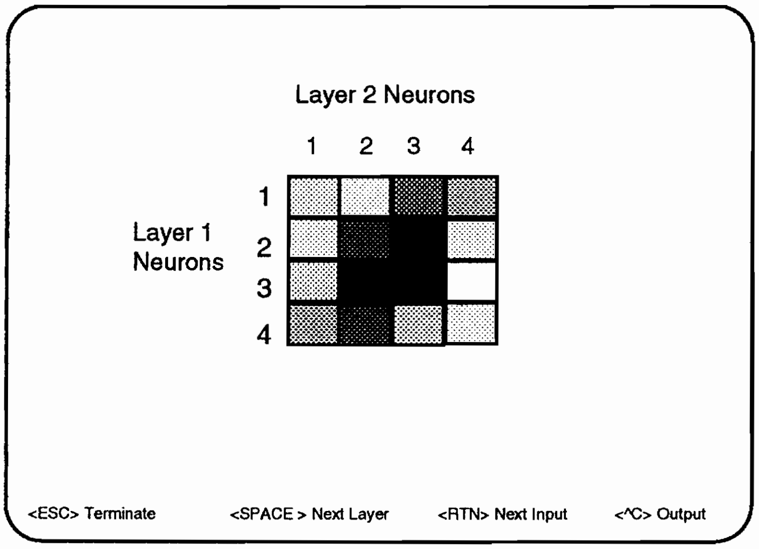


<ESC> Term <SPACE> Modify Params <RTN> Next Input <^C> Memory <F1> Network

Modify Params Command Line

<ESC> Display <F1> Mod Input <F2> Mod Neurons <F3> Mod Connections <F4> Corrupt

Figure 28. Network Operations Display



<ESC> Terminate <SPACE > Next Layer <RTN> Next Input <^C> Output

Figure 29. Memory Display

mode or the Training Mode. The Command Selection line at the bottom of the display allows for selection of the various functions available while in the Program Operations Mode. From here, if in the Run mode, selecting Next Input will cause the following events to occur -- the next input will be read, the system will calculate the output as specified by the network computational algorithm, and the screen will be updated to display the current input and associated, computed, output. The current computational algorithm is implemented as specified in Appendix H and displayed in Figure 5, since this is the algorithm universally used throughout the literature. Modifications to this could easily be incorporated into the system.

In the Training Mode, if in a supervised training system, the same sequence will occur except the appropriate training algorithm will be executed prior to the display update. The operator can thus see how the network output changes and converges toward a solution, during a training session. If using an unsupervised training network, the network will either operate as above if output comparisons are made or it may completely train the net for the specific set of inputs all at once, as is the case with the Hopfield net. It depends on the algorithm specified by the operator.

The Memory selection on the Command Line (Figure 28) changes the display from Input/Output to a representation of the network memory, a display of a layer's associated weight matrix. An example of this display is shown in Figure 29. The weights are shown as a matrix and are color coded according to their specific value. If space allows, the weight value itself is also displayed in the center of the square. Each layer is selectable for display. The operator may also run the next input from this display if so desired. If in the Training Mode he can then see how the weights are changed by the training algorithm for that specific input at that specific stage of the training cycle.

The Modify Parameters option on the command line (Figure 28) displays a new command line which will allow the operator to modify the input either through a manual means or through a corruption process which randomly corrupts the input as discussed earlier. The operator may also remove individual neurons from the net or disable specific connections between layers to see how a previously trained network is affected if these connections or neurons are eliminated. These disablings may also be used to monitor their affect while network training is taking place.

Also available, although not shown on the command line, is a multiple iteration capability which will operate the net for a specific, selectable number of iterations. This function is most useful during training where hundreds or even thousands of training iterations may be required to fully train the network. This function is selectable through the <SHIFT> T command. The operator is then cued to choose the number of iterations. Once initiated the network automatically runs through the number of cycles selected before the operator regains manual control.

Available also in the display mode is the capability to display the network architecture as described before and shown in Figure 27.

COMPUTATION

The network computation functions are implemented internally within the system and are not visible to the user except through their effects on the display system. All the network calculation and training algorithms are located within the Math Unit, as specified in Appendix H, and are thus localized for easy access and modification. To summarize the computational sequence, a set of inputs is applied to each neuron, each representing either the input from the outside world (the input layer) or the output from other neurons (a hidden layer). Each input is multiplied by a corresponding weight, all the weighted inputs to a single neuron are summed and then sent through an activation function to determine the output of the neuron. The output of each neuron (OUT), as well as the input from layer 0, is represented by a real value (layers[x].neural_set.real_value[i]) and a binary value (layers[x].neural_set.binary_value[i]). These data structures are specified in the Layer_type data structure as shown in Appendix B. The floating point values are normally used for network algorithmic computations while the binary values, representing an ON/OFF or FIRE/NOFIRE state, are used for display to the operator via the display mode. The output vector of a layer, which includes the outputs of each neuron in the layer, is then applied to form another weighted input to each neuron in the next layer (See Figures 2, 4, and 5).

Appendix H is the source listing for the algorithmic implementations (Math Unit). The above computation is implemented in the Calculate_Output procedure. This procedure calls Calculate_Activation which, in turn, applies the appropriate activation algorithm. Several popular, non-linear functions are implemented in the current version of the system, including the Threshold, Sigmoid, and Modified Sigmoid. Provision for the Hyperbolic Tangent function has been made but has not yet been implemented.

Provision for a user defined activation function has also been made and could be implemented in code for the user defined case in calculate_activation or through a called procedure defined by the user. The user must be careful to meticulously follow the interface definition of the calculate_activation procedure. The activation function is specified through the Activation Function Menu (Figure 22), as discussed previously, and is embodied in the activation field of the net_description record (Appendix B).

Most training algorithms are implemented through calls by the Learning_rule procedure (Appendix H). As with the activation function, the training algorithm to be applied to the network is specified through the Menu system and is embodied in the learning field of the net_description record (Appendix B). Three learning algorithms, Backpropagation, Delta, and Hebbian, have been implemented in the current version of the system. Additionally, a Hopfield net has been implemented although this may not be considered a true training algorithm since the weights are completely specified prior to network operation. Again, provision for a user defined algorithm has been made and is implementable through the "other" definition in the learning type.

In all cases, the call to the learning_rule procedure is controlled from within the Display Mode during network operations. If in the Training Mode, the learning algorithm will be called as required by the network specification (feedforward, recurrent, supervised, unsupervised). If in the Run Mode, the network simply computes its output from the applied input with no learning algorithm is exercised.

The Math Unit also contains all the matrix operations required to calculate appropriate network requirements. These include matrix multiplies, vector multiplication and addition, the inner product, matrix and vector transposition, and vector/matrix multiplication.

SECTION 4

IMPLEMENTATIONS ON THE DEVELOPMENT SYSTEM

Several different architectures and paradigms have been implemented on this prototype system in order to demonstrate its ability to support different neural net models. The networks implemented include the following:

1. A single layer, two input, single output network trained with the Delta rule learning algorithm to recognize binary functions (and, or, nand, nor, etc.) (Figure 30).
2. A two layer, two input, single output network, trained with the backpropagation learning algorithm to recognize the exclusive-or and exclusive-nor binary functions (Figure 34).
3. A multiple layer network consisting of 35 input nodes, 80 hidden layer nodes, and 8 output nodes, trained with the backpropagation learning paradigm to recognize the letters of the alphabet and the integers and produce the proper ASCII code for the character (Figure 35).
4. A four layer network consisting of 35 input nodes, 80 and 20 hidden layer nodes, and 8 output nodes to see if training would be different from that which took

place in number 3 above.

5. A single layer Hopfield net trained in the unsupervised mode, via the traditional Hopfield algorithm, to recognize several letters of the alphabet and recall them under various conditions of corrupted (noisy) input (Figure 38).

Although not specifically implemented, other network algorithms for both supervised and unsupervised modes are discussed as to their proposed implementation on the prototype system.

The following discussion elaborates on the various network implementations on the prototype system.

DELTA RULE

The single layer network was first described by McCulloch and Pitts in 1943 [6] and extensively studied by Rosenblatt in 1962 [2]. The basic model is as shown in Figures 2, 4 and 5 as described before. It uses a fully connected, feedforward architecture, with a hard limiting, non-linear threshold function as shown in Figure 3a. This model forms the basis for all neural system architectures today and was often called the perceptron model. The basic model multiplies the value of each input by an associated weight and sums to a total. If the resulting total is greater than a predetermined threshold, the model outputs a binary one, or if less, outputs a binary zero.

The simple perceptron learning procedure, first described by Rosenblatt and then generalized further to include continuous (vice binary) inputs and outputs, is called the Delta rule algorithm and is applied as follows:

1. Initialize the weights and threshold to some small, random value.
2. Present the input to the net. Each input vector to the input layer of the network has an associated target (correct) output vector for comparison. The two vectors, input and target output are collectively called the training pair.
3. Calculate the actual output of the net as depicted in Figure 5.
4. Compare the actual output vector to the target in the training pair and modify the weights according to the following algorithm:
   a. $\delta = T_j - A_j$
   b. $\blacktriangle_i = g \, \delta \, x_i$

c. $w_i = w_i + \blacktriangle_i$

where:

T_j = target output for jth neuron
A_j = actual output for jth neuron
g = gain factor, $0 <= g <= 1$ (Controls average size of weight change)
x_i = ith input value
$\blacktriangle_i$ = factor added to weight_i
w_i = weight associated with ith input

If the actual output matches the target output then $\blacktriangle_i = 0$ and no adjustment is made to the weight setting for that input.

5. Repeat the process until the network is trained satisfactorily.

This network and associated learning algorithm was the first to be made operational on the development system as a demonstration of the representational capability of the prototype. The initial network developed is as shown in Figure 30. Using the Delta rule as described and as specified in Appendix H, procedure delta_calculation, the network was trained to recognize 14 of the 16 binary functions for two inputs as follows:

	Input	Required Output	Function
	A B C D	A B C D	
1.	1 1/1 0/0 1/0 0	1/0/0/0	And
2.	"	1/1/1/0	Or
3.	"	0/0/0/1	Nor
4.	"	0/0/1/1	Not A
5.	"	0/1/0/1	Not B
6.	"	0/1/1/1	Nand
7.	"	1/0/1/1	B or not A
8.	"	1/1/0/1	A or not B
9.	"	0/0/1/0	B and not A
10.	"	0/1/0/0	A and not B
11.	"	1/1/0/0	A ignore B
12.	"	1/0/1/0	B ignore A
13.	"	0/0/0/0	Null
14.	"	1/1/1/1	All
15.	"	0/1/1/0	Exclusive-or
16.	"	1/0/0/1	Exclusive-nor

The final two functions, the exclusive-or and exclusive-nor, were shown by Minsky and Papert [7] to be linearly inseparable and therefore, not representable (or trainable) on a single layer network (See Figures 30 and 31). This limitation was quickly substantiated by the simple network under study.

The Exclusive-or function, as depicted in Figure 31, shows that all possible combinations of inputs and outputs are comprised of four points on a plane, A_1, B_1, A_0, and B_0, where the x coordinate value is one input and the y coordinate value is the other input. Using the single layer network in Figure

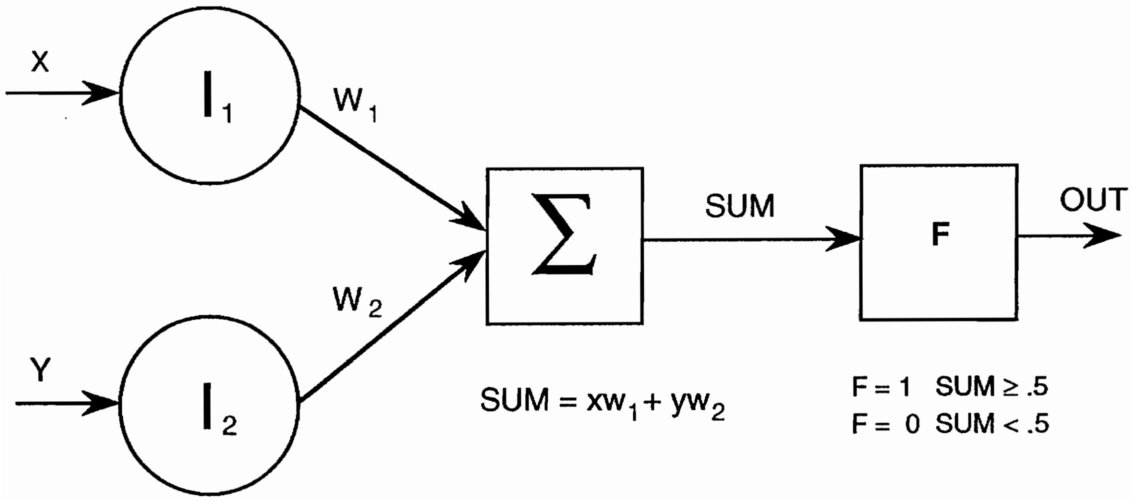
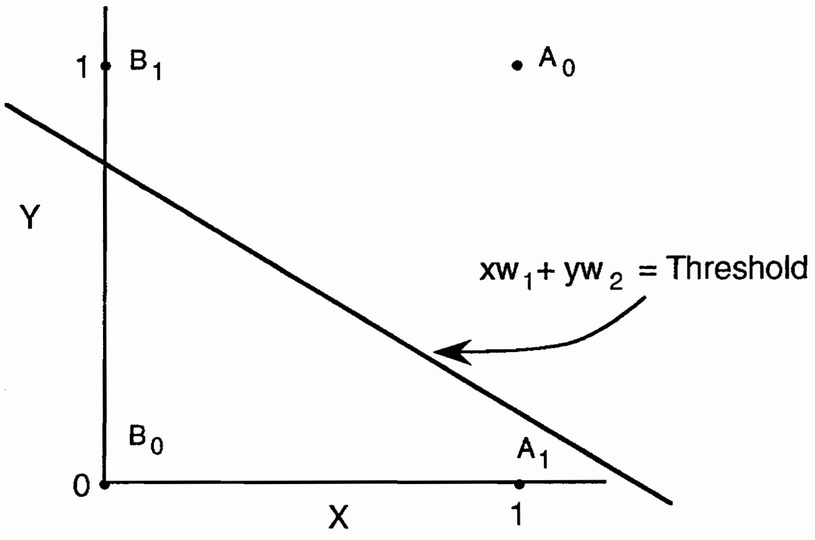


Figure 30. Single Layer Binary Function Network



	x Value	y Value	Required Output
A_1	1	0	1
B_1	0	1	1
A_0	1	1	0
B_0	0	0	0

Figure 31. Exclusive-or Function

30, where SUM = $xw_1 + yw_2$, it can be seen that no combination of values for the two weights, w_1 and w_2, will produce the relationship required by the table of Figure 31. The threshold line in the figure is where the equation F(SUM) must divide the plane. All values to the right of this line will yield a one whereas all value to the left will yield a zero. Changing the values of w_1 and w_2, as well as the threshold value, will change the slope and position of this line. However, no matter what values are used, the line cannot be positioned so that both A_1 and B_1 will fall to the right while A_0 and B_0 fall to the left, as required by the truth table, to satisfy the requirements of the exclusive-or (XOR) function. This is also the case with the exclusive-nor (XNOR) function. Conversely, a plot of the other 14 binary functions will reveal that they are linearly separable through manipulation of the weight and threshold values and are thus able to be represented, and trained, on a single layer network.

The XOR and XNOR class of functions is said to be linearly inseparable and is a major limitation of the single layer network. This limitation can be overcome through the use of multi-layer networks, and it was, in fact, known early on that multiple layer networks could learn any function that they could represent [2, 3], including the XOR function. However, until the backpropagation algorithm was developed, it remained a mystery how to systematically train these networks.

BACKPROPAGATION

The development of the backpropagation learning algorithm, with its ability to train multi-layer networks, brought about a resurgence of interest in Artificial Neural Networks. Rumelhart, Hinton, and Williams first presented the method in 1986 [8], although several others had discussed it earlier in 1974 and 1982 [3].

The basic architecture used in the backpropagation network implementation on the development system was described earlier and is as shown in Figures 2 , 3b, and 5. A fully connected, feedforward network was specified. The previously discussed method of neuron activation level calculations using SUM and a Sigmoid activation function for each neuron was developed and implemented.

During training, training pairs are presented to the network, one at a time. The input is applied to the first layer and neuron outputs are calculated, as in Figure 5, for each layer. The final calculation is for the output layer. The output of each neuron in the output layer is then compared to the target output. If there is a difference between the two, the network is trained and the weights are adjusted for each layer. The next input is then presented to the net, the output is calculated, the output again is compared to the new target vector, and the network is trained again. This continues until the net is satisfactorily trained to calculate the proper output for each input. Training may then be stopped and the network will continue to recognize the now familiar input patterns. No further training is necessary. This paradigm is illustrated in Figure 6.

The backpropagation algorithm, as coded in the development system, operates as the name implies (refer to Figures 32 and 33) [3, 9] moving backward through the network beginning with the computed output and the associated weights that produced that output. The weight values are first adjusted for the output layer. (Since the target value is known, this adjustment is made using a modification to the Delta rule already discussed). The output from a neuron in the output layer is first subtracted from its associated target value, producing an error signal. This error is multiplied by the derivative of the Sigmoid function for the output neuron (q) value, yielding a δ value (the derivative for the Sigmoid function is simply $(OUT_q)(1 - OUT_q)$. This simplicity, along with the compression attributes discussed before, is the reason the Sigmoid Function is often used as the network activation function of choice). The δ value, calculated as follows, is computed for each neuron in the

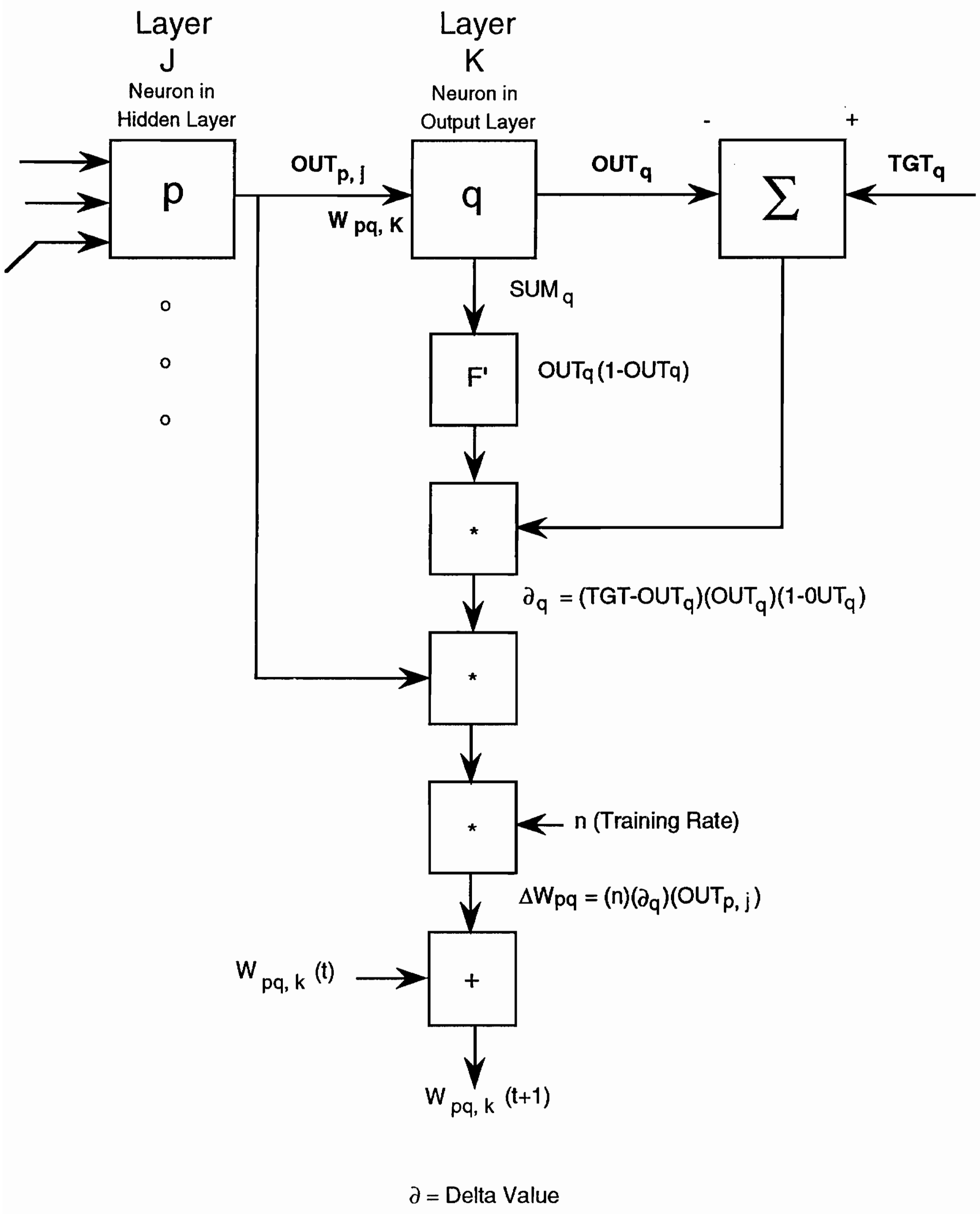


Figure 32. Backpropagation (Output Layer)

output layer:

$$\delta_q = OUT_q(1 - OUT_q)(Target - OUT_q) \qquad \text{(Equation 2)}$$

δ_q is then multiplied by OUT_p from the neuron in the previous layer for the weight to be modified. This product is multiplied by a training rate coefficient n (0.01 to 1.0) and the result added to the weight. The calculation, is

$$\blacktriangle w_{pq,k} = n * \delta_{q,k} * OUT_{p,j} \qquad \text{(Equation 3)}$$

$$w_{pq,k}(t+1) = w_{pq,k}(t) + \blacktriangle w_{pq,k} \qquad \text{(Equation 4)}$$

Where:

$w_{pq,k}(t)$	=	the value of the weight from neuron p in the previous hidden layer to neuron q in the output layer k at the time before adjustment.
$w_{pq,k}(t + 1)$	=	value of the weight at the time after adjustment.
$OUT_{p,j}$	=	the value of OUT for neuron p in the hidden layer j

(Note that p and q relate to specific neurons while j and k refer to a layer.)

After the weights have been adjusted for the output layer as described above, all weights must then be adjusted for each hidden layer. δs must be generated for each neuron in the hidden layers without the help of a target vector, since none exists for the hidden layers.

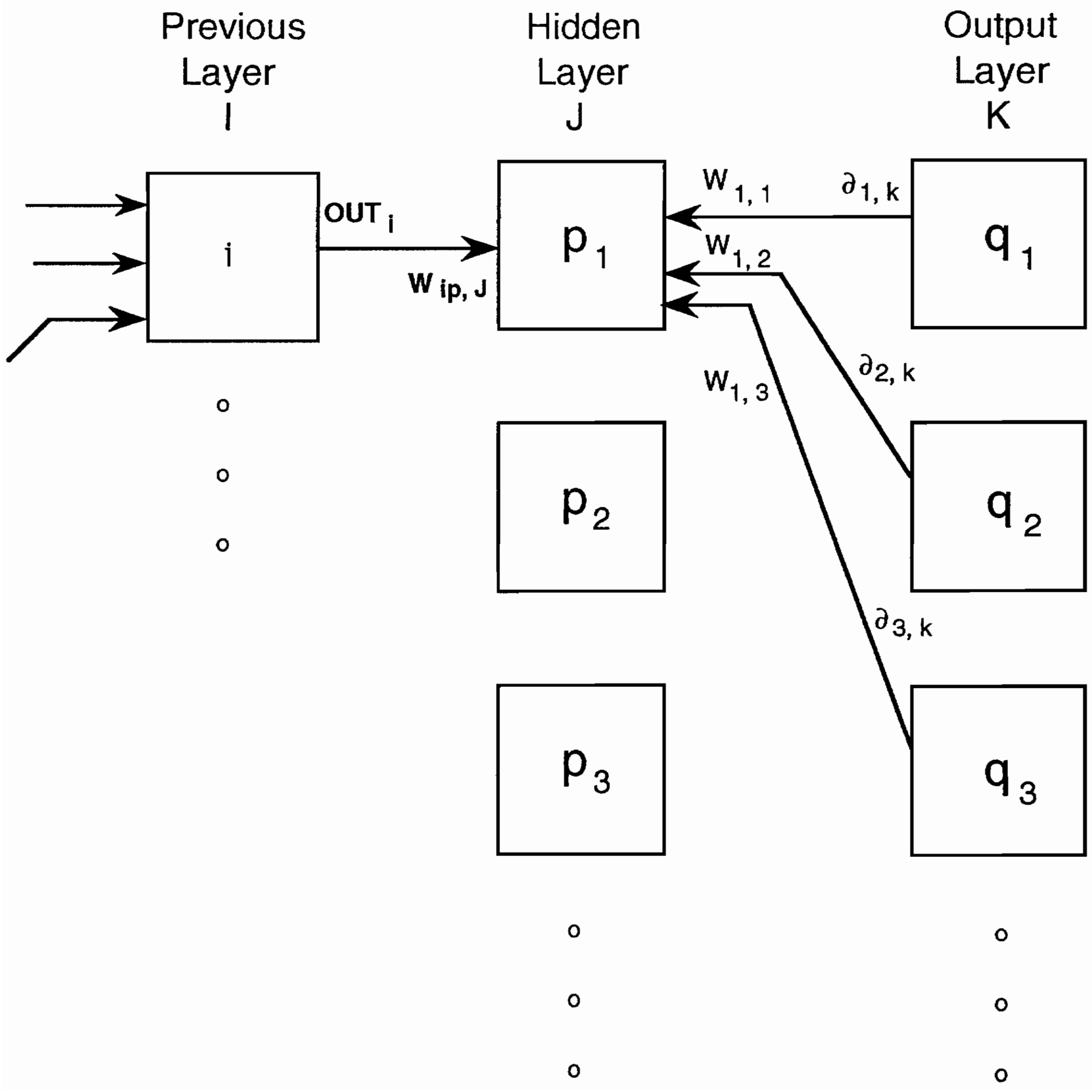
The Backpropagation learning algorithm trains each hidden layer by propagating the error determined by the target vector back through the network, determining a δ value for each neuron in each hidden layer and adjusting all weights involved. Equations 3 and 4 above are then used again to determine all new corresponding weight values.

Values for δ in the hidden layers are generated using Equation 5 as follows:

$$\delta_{pj} = OUT_{pj}(1 - OUT_{pj})(\Sigma\delta_{q,k}\ w_{pq,k}) \qquad \text{(Equation 5)}$$

This process is shown in Figure 33. Using this δ value calculated for each neuron in the hidden layer, Equations 3 and 4 can then be used to calculate the new weight values for that hidden layer. These new δ values are then used to propagate back through the weight values before adjustment in a similar manner to find new δ values for the next, previous hidden layer. This process is continued until the weights have been adjusted for all layers in the network.

There is no limitation to the number of layers that the backpropagation algorithm can effectively handle. Training time obviously increases with the number of layers as well as



$$\partial_{pj} = OUT_{pj}(1-OUT_{pj})(\sum^{q} \partial_{q,k} W_{p,q})$$

$$\Delta w_{ip} = (n)(\partial_{pj})(OUT_i)$$

∂ = Delta value

Figure 33. Backpropagation (Hidden Layer)

with the number of neurons in each layer.

The saved_weight data structure (See Appendix B - global declarations) was provided for the backpropagation algorithm as a means to have both the previous weights, W(t) (saved_weights) and the adjusted weights, W(t + 1) (weights data structure) available for the computation.

The saved_weights data structure also proved beneficial for the computation of a momentum term, which in some cases improves training time and enhances the stability of the process by smoothing out the calculations [5, 9]. The momentum process adds a term to the weight adjustment that is proportional to the amount of the previous weight change. Equation 3 is then modified to

$$\blacktriangle W_{pq,k}(t) = n * \delta_{q,k} * OUT_{p,j} + \alpha[W_{pq,k}(t-1)] \qquad \text{(Equation 6)}$$

where α is the momentum coefficient (0.0 - 1.0)

If a coefficient of 0 is used, then momentum is not a factor; if 1 is used, then the full weight change of the previous iteration is used in the calculation. Values from .7 to .9 are common when momentum is used.

The above process was used to train the net shown in Figure 34 to recognize the exclusive-or and exclusive-nor functions. Through several training sessions it became clear

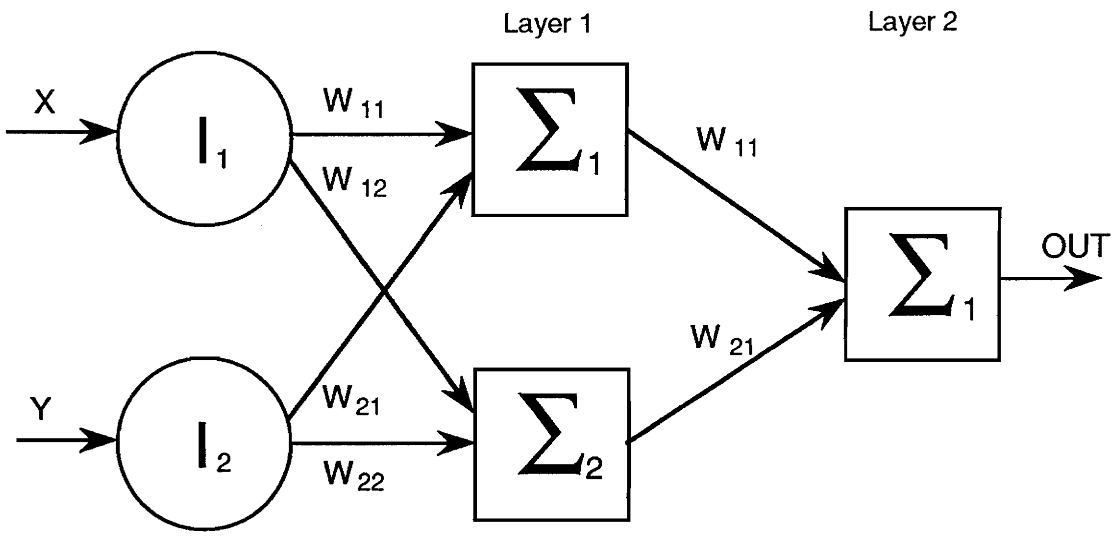


Figure 34. Two Layer Binary Function Network

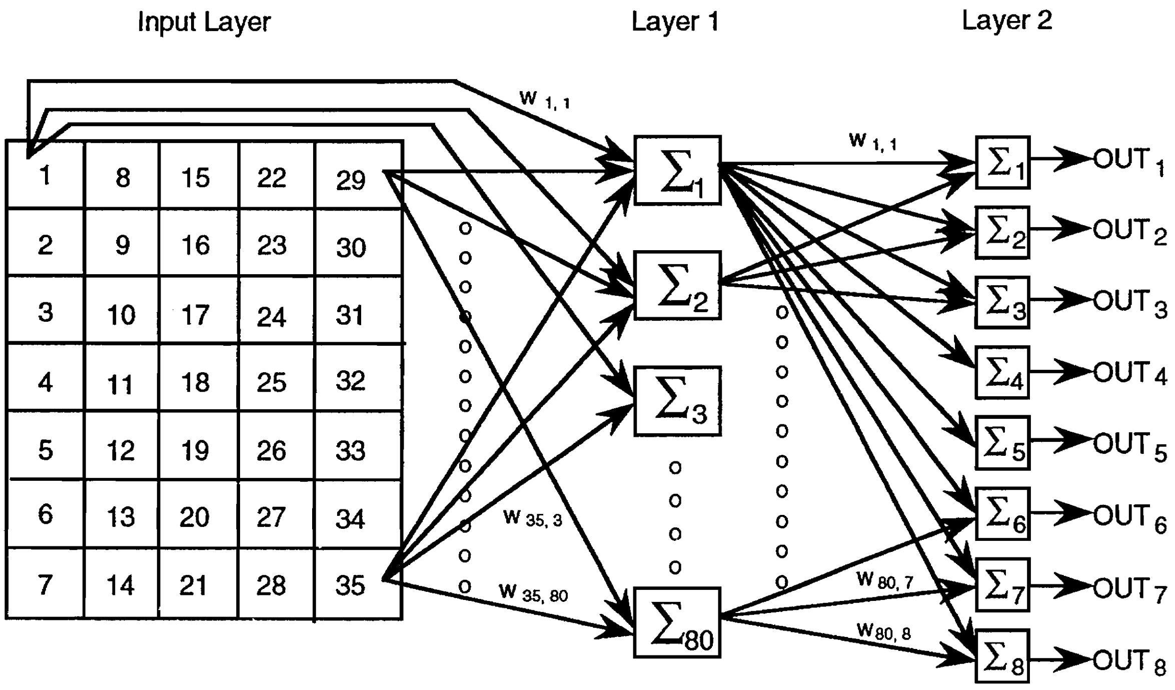


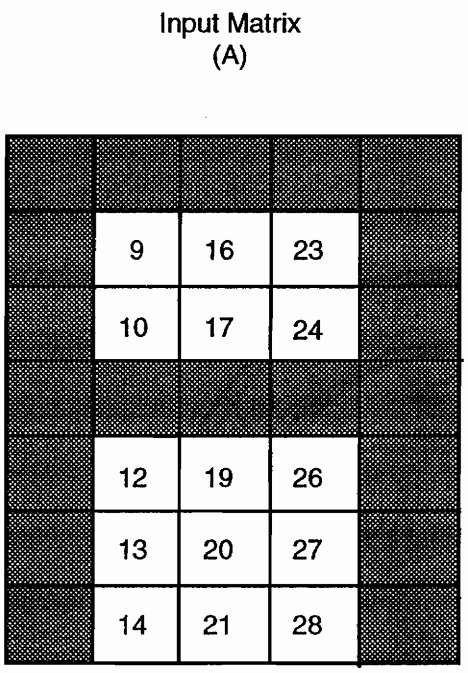
Figure 35. ASCII Code Two Layer Network

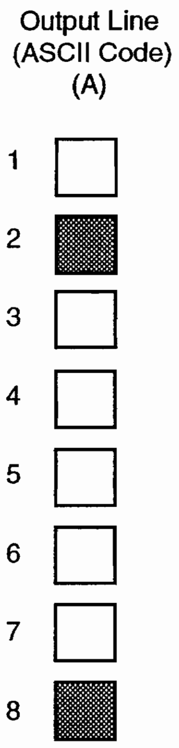
that the network would not always train as expected. The network would often oscillate back and forth and never settle on the correct weight values. In these cases, the only alternative was to reset the weights to low random values, initialize the saved weights to zero, and start the training sessions again.

This phenomenon of instability reflects some of the problems encountered with the use of the backpropagation algorithm. Rumelhart, Hinton, and Williams [8] have proven that if the solution to a problem can be represented by a network, then the backpropagation algorithm will converge to a solution. This proof however assumes infinitesimally small weight adjustment steps. This is impractical in real networks and often leads to problems of network paralysis and local minima. As the network trains, weights can be adjusted to very large values which tend to force neurons to operate at very large values of OUT. The derivative of the squashing function in this region is correspondingly extremely small (See Figures 3b and 3c) which tends to bring training to a standstill since training is proportional to this derivative. One method for overcoming this paralysis is discussed in the next section and is implemented through a saturation function which helps to bring these neurons out of saturation. Other methods for handling this paralysis include the use of smaller step sizes which is accomplished through a reduced gain factor (n). This however tends to increase training time. There is very little theory or practical guidance to show how to select step size for the training of networks. If too small, training time will be slow; if too large, training may reach paralysis or oscillate resulting in continuous temporal instability. This results in the network "unlearning" previously learned training sets.

There are several algorithms discussed in the literature which adaptively reduce step size automatically as training progresses. These algorithms could easily be integrated with those presented on the current prototype development system.

Next, a larger, more complex network was specified to recognize the alphabet characters and integers. This network (See Figure 35) consisted of an input matrix of 35 neurons arranged in a 5 X 7 grid , a hidden layer of 80 neurons, and an output layer of 8 neurons to represent the ASCII code of the characters presented to the net. Figure 36 displays a representative input and output for this network. A neural output of a binary one is represented, in Figure 36, by a colored square while an output of a binary zero is represented by a blank square. Thus, each character can be represented in the input matrix by a unique sequence of ones and zeros from the input file. Each training pair (character and ASCII representation) is specified by two lines in the input file, as shown in Figure 36.





<u>Input File</u>

1 1 1 1 1 1 1 1 0 0 1 0 0 0 1 0 0 1 0 0 0 1 0 0 1 0 0 0 1 1 1 1 1 1 1
0 1 0 0 0 0 0 1

Figure 36. ASCII Network Input/Output Representation

The network was fully connected, utilized a modified sigmoid activation function and the backpropagation learning algorithm. Utilization of the modified sigmoid activation function required the input layer of the network to be set to +.5 to represent a binary 1 (ON condition) and a -.5 to represent a binary 0 (OFF condition). This is accomplished in the software of the development system in order to maintain consistency of data input from the input file. Thus, the data input file uses the standard 1/0 for all network input no matter what algorithm is being used. The development system converts the input to the proper floating point value for use of the functions and algorithms specified through the menu system for that particular network. One_value and zero_value in net_description is set to these required values when the activation function is selected in the Menu system (See Appendix B). The neural floating point value is then set to these required values upon reading the input from the file. The neuron binary value remains as one or zero for display purposes. This is accomplished through the set_input_layer procedure in the ioutil unit, Appendix G.

The complete alphabet, in a form similar to that shown in Figure 36, plus all ten integers, was presented to the network. The network was satisfactorily trained to recognize all 36 characters. Some of the training data collected is presented in Figure 37.

During network training, large numbers of neuron outputs summed through the weight matrix for input to a single neuron in the next layer, tend to yield large values of SUM. This, when using a sigmoidal activation function, tends to drive the output of the next layer toward a saturated (maximum) value. Also, because the backpropagation training algorithm uses δ values which are based on the first derivative of the sigmoidal function, weight correction values tend toward zero at these saturated values. Network paralysis often results from these factors.

A method discussed by Wasserman [3] helps to solve paralysis problems without greatly affecting training already accomplished. Saturated neurons are identified through their OUT values. If the magnitude of a signal approaches the limiting value (called saturation in the development system), rather than training all weights in the normal fashion, weights feeding that neuron are operated on by a squashing function similar to the sigmoidal activation function. The saturation value is set in the Menu system (Figure 23) and is a value close to zero. This equates to a first derivative value of OUT close to the extremes of the function (see Figure 3c). The range of the function is set to +5/-5 and is implemented as follows:

$$w_{mn} = -5 + 10/[1 + \exp^{(-wmn/5)}] \qquad \text{(Equation 7)}$$

This function greatly reduces the magnitude of excessively large weights while not affecting small weights as much. It also preserves symmetry between the weights feeding the neuron, thereby maintaining relative differences between the weights. Weights associated with neurons that are not saturated are trained normally. The use of this procedure to draw neurons out of saturation greatly reduced training time in the backpropagation implementation of the alphabet recognition net.

While training the alphabet recognition network, it was noticed that output values would often oscillate back and forth between the correct level and an incorrect response. The results of such oscillations can be seen in some high numbers of training trials shown in Figure 37. As one letter became trained, another previously trained to a correct response would yield incorrect output. Due to the distributed nature of the memory of the network being resident within the weight matrix as a whole, as a letter would continue training, even though it had a correct response, the weight changes would tend to correct in the wrong direction for other letters, thus pulling them out of convergence (the correctly trained state).

Backpropagation Algorithm Results

26 Letter Alphabet
Gain Factor (n) = .7

Network Specification	Training Trials	Number of Letters Trained

Saturation = 0
Tolerance = 0
	500	6

	2800	17
	5000	12
	6000	15
	7000	17
	8000	18
	9000	17

Saturation = .001
Tolerance = 0
	500	20

	600	21
	700	24
	800	22
	900	22

Saturation = .001
Tolerance = .45
	500	25

	600	26
	700	26
	800	26

Figure 37. ASCII Code Network Results

With use of the Delta Rule in perceptron training, unlike with backpropagation, if the actual output matches the target output, then $\Delta_i = 0$ and no adjustment is made to the weight setting for that input. It was hypothesized that if the backpropagation algorithm could be made to act more like the Delta rule in this regard, where no training takes place once the response converges, then this correction in the wrong direction might be minimized, if not eliminated altogether.

An optional tolerance value was therefore created and integrated into the backpropagation algorithm. Thus, if the output of the net is within the specified tolerance for the difference between the target value and the output value (TGT - OUT), no training will take place for that particular neuron. If the difference is out of tolerance (there is a large variance between the output of a neuron and what it should be) then training will take place in the normal fashion. The algorithm was easily introduced as

```
If abs (tgt - out) >= tolerance factor
  then train
  else neuron $\delta$ value := 0
```

Setting the δ value to 0 for the output neuron in question ensured that the associated weights feeding that neuron would not be adjusted as the backpropagation algorithm progressed back through the layers.

Figure 37 shows some of the training data collected on network training with and without a saturation factor and a tolerance factor. The figure shows the number of letters trained (recognized correctly by the network) after specific numbers of trials had been completed. A trial consisted of one complete cycle of the presentation of a letter to the net, calculation of the network output, and training of the weights via the backpropagation algorithm. The alphabet was repeatedly sequenced through, one letter at a time. Thus, the network had seen each letter one time after 26 trials, twice after 52 trials, etc.

As shown in Figure 37, introducing the saturation and tolerance functions into the backpropagation algorithm not only greatly reduced training time but in fact was the only way the net could be fully trained without an inordinate number of training iterations. This was a good example of how the development system can be used to quickly implement enhancements that the student may wish to try in order to study or improve the capabilities of network functions.

One further network was implemented on the system to observe its effect on training time and network operation capabilities. This was a four layer network, with the same parameters as in the previous network, except that it had an additional hidden layer of 20 neurons between the 80 neuron layer and the output layer. Again, the network was trained on the ASCII character set described previously. The number of training cycles required to fully train the network did not change significantly (it took about 700) but the total training time did increase because of the additional layer involved in the training algorithm. The time taken to complete successive training cycles visibly decreased as training progressed due to the positive effect of the saturation and tolerance factors on the necessity for training on each cycle.

Although not fully investigated, it appears that this complex network is more robust than the three layer network (connections and neurons could be removed from this network without as deleterious an affect as such a removal had on the three layer network). This initial impression needs further study.

Use of the development system is an excellent and convenient vehicle for studying the effects of various parametric changes on the learning and performance of Neural Networks. Such use makes it an easy matter to modify and study, under controlled conditions, various factors such as gain (training rate), momentum, tolerance, saturation, removal of connections, activation algorithms, etc. Hidden layers can be added to the network, as was done above, or the number of neurons in the hidden layers can be varied to see how this would affect training time and the robustness of the network after training. Through the use of this system the student can now spend his time studying the effects of these parameters in a straight forward and simple fashion.

Another possible scenario for the backpropagation network would be to compare the use of the Sigmoid Function with that of the Modified Sigmoid Function to see if the premise for the development of the Modified Sigmoid, that of reduced training time, is true. This is easily accomplished on the development system by training a single basic network using the two different activation functions, all else remaining equal. Nothing more than a few keystrokes would be required of the system operator to make such a comparison.

HOPFIELD NET

A Hopfield net has also been implemented on the development system in order to demonstrate the system's versatility. Hopfield rekindled much of the interest shown in neural networks in the early 1980's through his extensive work with what is now known as the Hopfield Net [3, 10, 11, 12]. This network can be used as an associative memory or a content addressable memory. Its simplest implementation is a single layer, containing N nodes, using a simple threshold non-linear activation function (Figures 7, 3a), with binary inputs of 1 and -1. Items to be recognized, or recalled, are represented as N-bit binary numbers and are stored in the network in a distributed manner across the system of weight values.

The network developed on the prototype system is shown in Figure 38. It is obvious from the figure that the Hopfield net is radically different from the backpropagation network described earlier (See Figure 35). It consists of only a single layer and it is recurrent, that is, computed outputs are fed back as new inputs. The implementation of the Hopfield net on the development system actually calls for two layers, as is shown in Figure 38. Layer one, the input layer, is merely for input and distribution of those values to the next layer through the weight values. It performs no calculations. Layer two is the output layer. All outputs are routed back to the input layer as input for the next iteration. In the classic Hopfield Net, the output is fed back through weight values as shown in Figure 7. The architecture used for the development system implementation of the Hopfield Net (Figure 38) is, however, functionally equivalent to that depicted in Figure 7. Classic Hopfield network operation methodology is enforced in this prototype system.

The network is different from other common networks in that it uses neither a supervised training algorithm nor an unsupervised training algorithm. The weights are set prior to

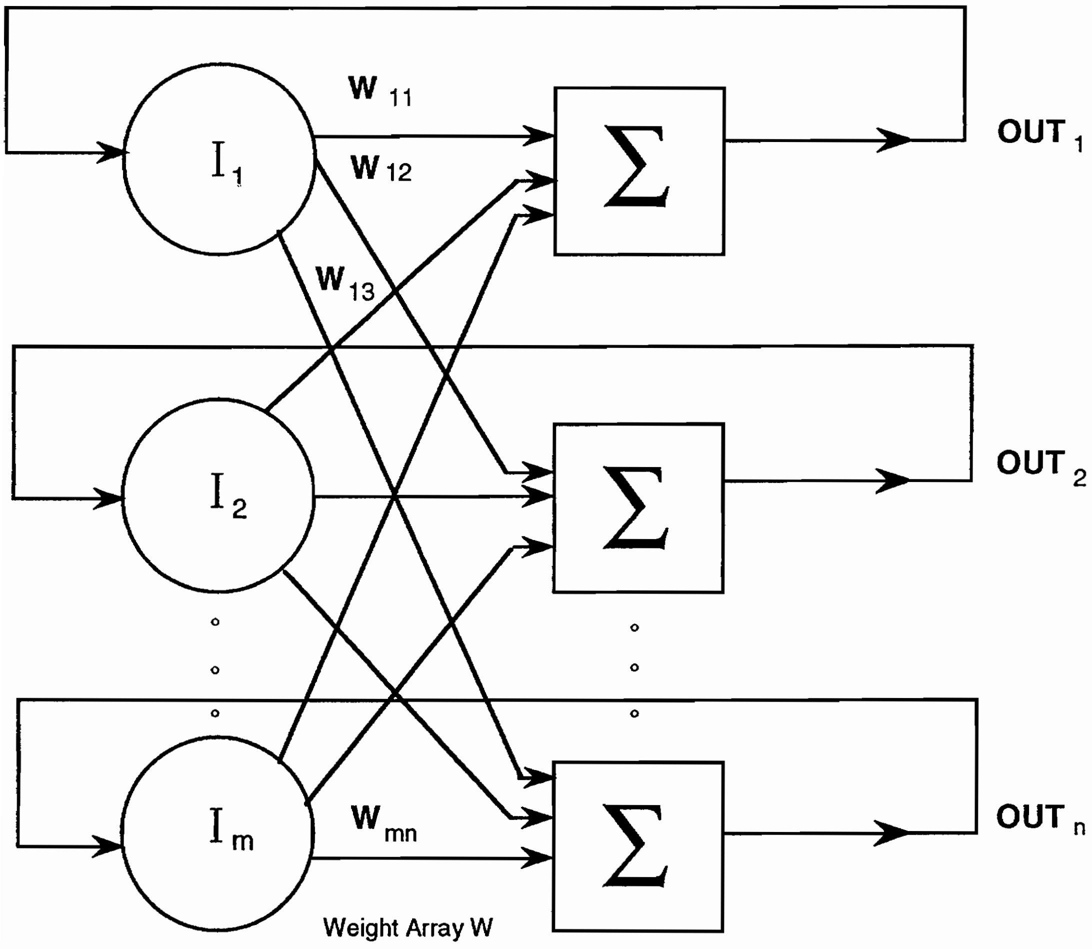


Figure 38. Hopfield Network Implementation

network operations using an algorithm that isolates unique characteristics of examples (called exemplars) of each class of input patterns to be recognized by the network. To set the weights, each exemplar is presented to the network, one after the other, and the following algorithm is applied:

$$W_{ij} = \sum_{s=1}^{m} x_is * x_js \quad \text{for } i <> j \qquad \text{(Equation 8)}$$

$$= 0 \quad \text{for } i = j$$

Where:

W_{ij}	is the weight connection of the ith neuron to the jth neuron
m	is the number of items to be stored in the network
s	is a particular item presented
x_is	is the ith binary digit of item s
x_js	is the jth binary digit of the same item s

When the weights are set in this fashion, they form a symmetric matrix of values with zeros on the main diagonal. In other words, $W_{ij} = W_{ji}$ for i <> j and $W_{ii} = 0$ for all i. Recurrent networks have been shown to be stable if they meet this criterion [3, 9], although it is not a necessary condition.

The output of a neuron is calculated as described earlier for the backpropagation network. A weighted sum of the inputs of the layer is taken, producing a SUM value which is then applied to an activation function -- in this case a simple threshold -- which produces the OUT value of the neuron. If the OUT value is greater then the threshold, output is 1, otherwise output is -1. The threshold value is normally set to 0 under these conditions.

The network computation equations are as follows and are similar to those described earlier for the feedforward networks:

$$SUM_j(t+1) = \Sigma w_{ij} * OUT_i(t) \qquad \text{(Equation 9)}$$

$$OUT_j(t+1) = 1 \text{ if } SUM_j(t+1) \geq T \qquad \text{(Equation 10)}$$

$$= -1 \text{ if } SUM_j(t+1) < T$$

Where:

T is the threshold value
t is time
t+1 is the next iteration after t

Since the network is recurrent, the output of each iteration becomes the input for the next iteration, which is the reason for the use of OUT at time t in Equation 9.

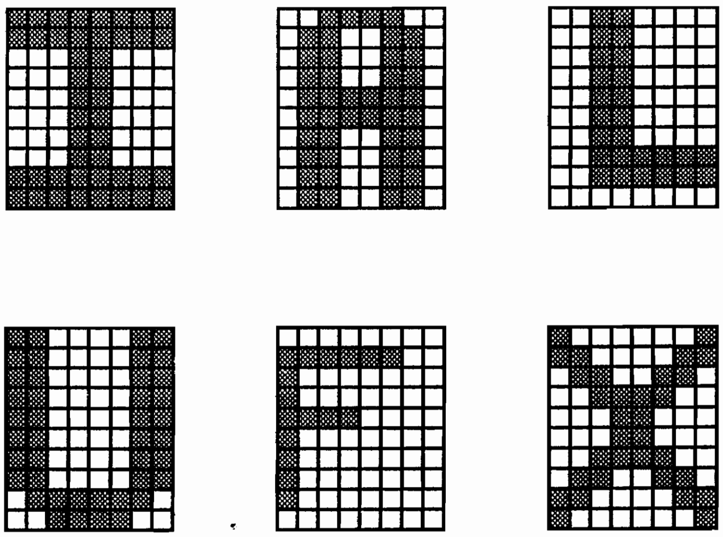
Once the network has been trained, a portion of an input data set can be withheld from the network and the full corresponding output data set will be returned. This is what is meant by associative memory. As with humans, where partial data such as a picture somewhat obscured by shadows produces a complete memory of that picture, so too the Hopfield net will produce a complete memory of the exemplar when a partially corrupted input is applied.

Hopfield network operations are classically performed in the following order. This methodology was enforced in the prototype system for the Hopfield net implementation. First the weights are initialized as specified in Equation 8 operating on the complete set of exemplars included in the input file. The network, by virtue of it being specified as a recurrent, unsupervised, Hopfield net is trained by iterating through each exemplar one time and setting the weights as per Equation 8. The weight matrix is said to be trained for the submitted set of exemplars and will not be altered during further work with that specific set. The Hopfield net is coded on the development system to train in this special fashion when the Training mode is entered. The flow of control within the system is described in the next section (See Figure 40).

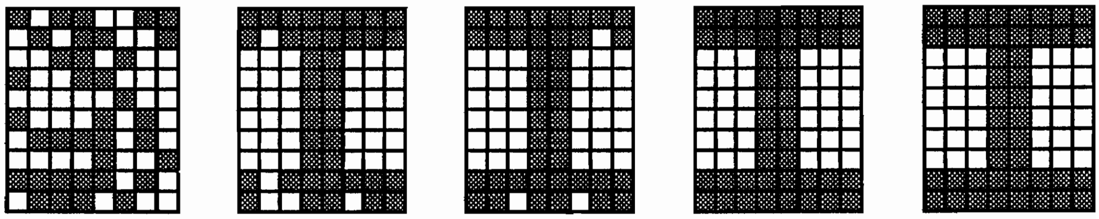
A trained Hopfield Net is then initialized by first specifying an input file (one exemplar per input file) and entering the Run Mode. This input pattern exemplar is applied as input to the net. This initialization may either be the complete exemplar or a corrupt version of that exemplar. If an exemplar is not corrupted, the output of the net will always reproduce the original exemplar pattern. An exemplar input may be corrupted by randomly reversing each bit (neuron output value) read from the input file with a specified probability. This corruption is applied through a selection option available in the Run Mode, Figures 28 and 42, and is accomplished in the corrupt_input procedure located in the math unit, Appendix H. Corrupt_input is called from the modify_parameters procedure in the grafstuf unit, Figure 42 and Appendix E. The corruption probability is set in the menu system as described earlier (Figure 13).

The net is then iterated (that is, output is computed, output is compared to the input, and then the input is set to the computed output) until the output shows no difference between itself and the input, at which time the net is said to have converged.

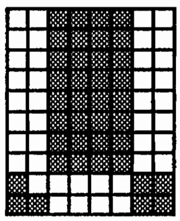
The network developed on the prototype system is shown in Figure 38 as described earlier. It included 80 neurons in its layer arranged in a matrix architecture of 10 rows by 8 columns (10 x 8). Thus, layer one and two are identical matrices, layer one being the input and distribution layer, layer two being the output and computational layer. Figure 39 depicts the 6 input patterns presented to the network to be trained. As described below, these patterns were hand crafted in order to produce good results on the network. Corrupted exemplars were then presented and were successfully recalled by the network when the corruption probability was at .20 or



Exemplars



Exemplar Iteration



One Common Spurious State

Figure 39. Hopfield Exemplars

below. As the probability increased, the successful recall deteriorated rapidly. A common spurious state for the letter I with corruption probabilities of greater then .25 is also depicted in Figure 39. This essentially reproduced and confirmed the results reported by Springsteen [5].

The Hopfield net has two major limitations when used in this fashion. First, the number of patterns that can be stored and recovered are limited. If too many patterns are stored, the net may converge to novel patterns not seen before. Such convergences are spurious states and will produce a "no match" output. Hopfield showed that this will occur rarely if the number of classes to be identified is kept below .15N, where N is the number of nodes in the net. For the prototype system with a maximum of 80 nodes in a layer, this equates to a limitation of 12 classes for identification. A second limitation is that the exemplar patterns will be unstable if they share many bits in common with other patterns. Springsteen [5] includes an excellent discussion of Hamming distance between exemplars to minimize the commonality between bits. The exemplars used in the example presented here were hand crafted to increase this Hamming distance. This greatly increased the effectiveness of the net. Data collected on exemplars with small Hamming distances confirmed that instability is indeed a problem with Hopfield nets.

HEBBIAN LEARNING

Hebbian learning is an unsupervised training method [3]. It involves local interaction between two neurons and requires no teacher and no global feedback system, as does backpropagation. In essence, the Hebbian Learning theory states that a connection between two neurons is strengthened (the weight factor w is increased) whenever those two neurons fire. This concept is expressed in the formula

$$w_{ij}(t + 1) = w_{ij}(t) + OUT_i OUT_j \qquad \text{(Equation 11)}$$

Where:

OUT_i	$= F(SUM_i)$	= output level of the source neuron
OUT_j	$= F(SUM_j)$	= output level of the destination neuron
t		= training iteration

With this algorithm, the synapse strength may be thought to increase according to the excitation levels of the two neurons that it connects. When the activation function F is of a sigmoid form, this learning algorithm is called Signal Hebbian Learning.

This learning algorithm was implemented on the development system using the modified sigmoid activation function. The Signal_Hebb procedure in the math unit, Appendix H, is called in the normal fashion through the Learning_Rule procedure, Figure 40. Since this is an unsupervised training algorithm however, when next_input is called, the setting of the output buffer for the target vector is skipped, as shown in Figure 41.

A network of 36 neurons in the input layer, arranged in a 6 x 6 matrix, 55 neurons in the hidden layer, and a line output layer of 5 neurons was established to demonstrate Hebbian learning on the development system. A series of 5 input sets, dividing the input matrix into five distinct patterns, was presented to the net. It was hoped that the training sessions would result in distinct outputs, thus essentially self-organizing the input into identifiable outputs. This result, however, was only partially achieved with two of the patterns yielding unique results, the other three resulting in no output at all.

The implementation of the Hebbian learning algorithm will require further analysis for its successful use, including the implemented algorithm itself as well as the network architecture used in the study. Once again, although unsuccessful, the development system provides an excellent vehicle for the further modification and analysis of this particular area of study. Further work in the area will provide insight into the Hebbian learning algorithm and the architecture used to implement it.

SECTION 5

DEVELOPMENT SYSTEM DATA FLOW

Figure 40 depicts the main data flow paths in the Training Mode and the Run Mode during network operations. As discussed in the previous sections, the system first determines if it is in the training mode or the run mode. This is accomplished in the Run_Network_Menu procedure in the Menu system which provides the Operations Menu, Figure 24. The global boolean variable "Train" is set to TRUE when the Training mode is selected and set to FALSE when the Run mode is selected.

As can be seen from Figure 40, the network functions in the operations mode are normally controlled from Display_Net_Output procedure specified in the grafstuf unit, Appendix E. The system calls appropriate procedures as determined by whether the network is characterized as a supervised or unsupervised net and then whether it is a recurrent or feedforward type architecture. These procedures for the operation of the network are all specified in the math unit, Appendix H. If it is an unsupervised, recurrent network, the further characterization of whether or not it is a Hopfield net is necessary in order to determine the sequence and procedures for reading input files and setting the input layer to the proper values. This proper sequencing of events

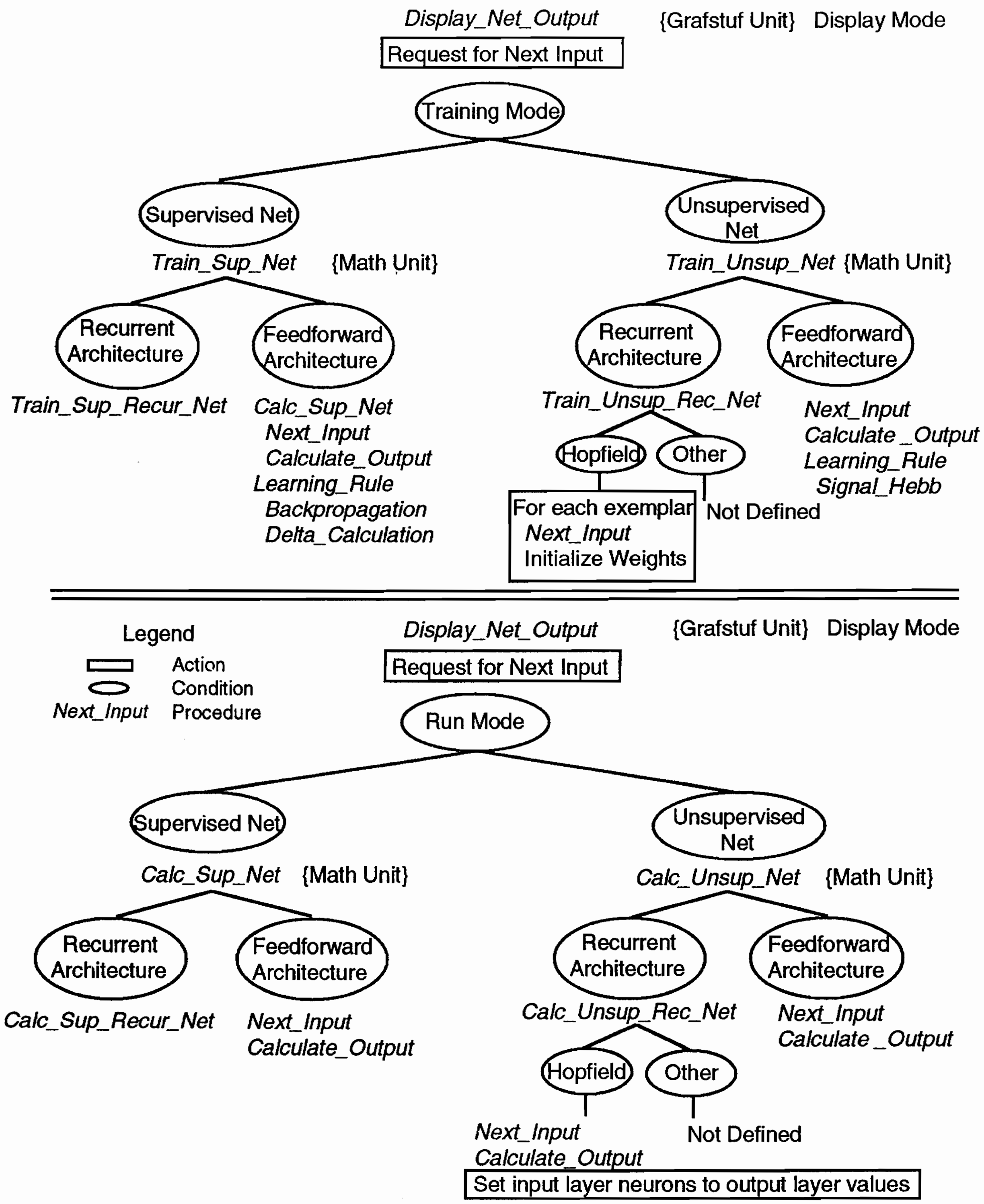


Figure 40. Display Mode Data Flow

is important primarily for the input file read operations, whether it is read at all, whether the target vector is read or skipped, etc. as explained previously and shown in the implementation categories depicted in Figure 8. This allows the system to require only one format for input files, thus enforcing consistency and easing extensibility requirements. If in the training mode, the final course of action is to call the Learning_Rule procedure, also located in the math unit, which in turn calls the proper learning algorithm as specified in the Net_Description variable (See Appendix B).

Figure 41 depicts the data flow through two important procedures for carrying out network operations. The first is calculate_output, specified in the math unit, Appendix H. This procedure establishes the basic algorithm for calculating the output of the network and is the same for all networks no matter what their characterization. The second is the next_input procedure, specified in the ioutil unit, Appendix G. This procedure also helps to maintain the proper sequencing of events by determining whether a target vector should be read from the input file or skipped. Aside from that determination, it also sets the input and output buffers to the values read from the input file, and then calls set_input_layer which sets the input layer neuron binary and real values to the proper equivalent numbers.

Figure 42 displays the procedure nesting and calling

Calculate_Output {Math Unit}

```
For each layer
  - weight_matrix_multiply
    Multiply input vector by
    weight matrix of layer

  - For each neuron in
    the layer
      Calculate_Activation

  - Set neuron output to
    the activation level to
    be input for next layer
```

Next_Input {ioutil unit}

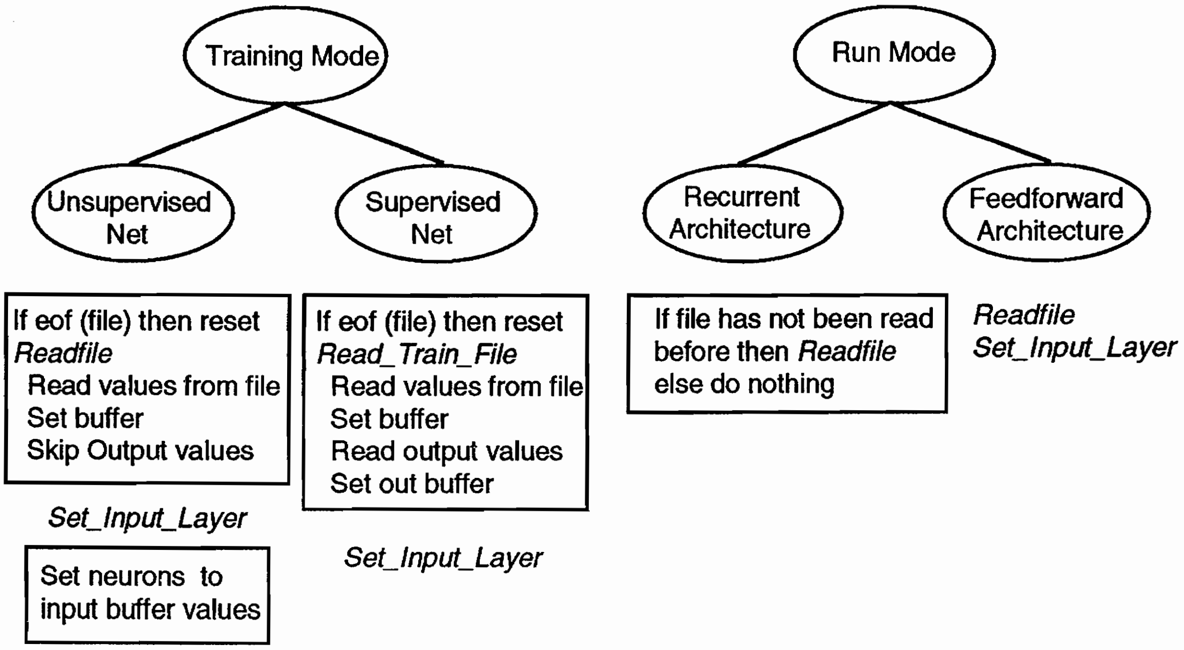


Figure 41. Calulate_Output and Next_Input Procedures

sequence schema specified in the display_net_output and the display_memory procedures (Display Mode). The Display Mode is entered from the run_network_menu (Program Operations Menu) through either the display_memory or display_net_output procedure when the Run Program or Training Option is selected (Figure 24). Similar capabilities exist whether the system is displaying a memory (weight) map or the network input and calculated output.

While in the display mode, the operator may toggle between the memory display and the input/output display. If toggled, the occurrences within the system are as follows: if in the display_net_output procedure, the procedure is exited, thus returning to the run_program procedure in the ops_stuf unit. A flag is set and the display_memory procedure is entered from the run_program procedure. The same sequence of events occurs while in the display_memory procedure (See Appendix E). The ops_stuf unit includes the procedures which supply the interface between the Menu system and the Display system.

When first entered, the options available to the operator in the display_net_output are Display Memory, Next Input, Save State, Modify Parameters, and Display Network (See Figure 28). As discussed previously, the Save State option is available through the <SHIFT> S command or automatically through the selection of periodic saves (Figure 26). The <SHIFT> S command is not displayed on the command line in the Display Mode. For various selections made, Figure 42 shows the sequence of procedure calls initiated by that selection for the Run Mode. A similar sequence also occurs in the Training Mode.

```
Run_Program      (Set Train Flag to False)                 {Ops_Stuf Unit}
   (Called from the Program Operations Menu)

  Display_Memory               {grafstuf unit}
     Train_Sup_Net             {math unit}  initiated by next input selection
     Calc_Sup_Net                   "            "
     Train_Unsup_Net                "            "
     Calc_Unsup_Net                 "            "
     Save_State                   {ioutil unit}
       (Save the state of the network to file)

  Display_Net_Output           {grafstuf unit}
     Train_Sup_Net             {math unit}  initiated by next input selection
     Calc_Sup_Net                   "            "
     Train_Unsup_Net                "            "
     Calc_Unsup_Net                 "            "
     Save_State                   {ioutil unit}
     Modify_Parameters         {grafstuf unit}
        Modify_Input
          (Manually set input of each neuron in input layer)
        Modify_Neurons         {grafstuf unit}
          (Delete/Restore Neurons in each layer)
           Get_Removal_Probability
           Remove_Random_Neurons
           Display_Network
              Net_Display
                 Draw_Layer_Box
                 Draw_FFArrows
        Modify_Connections     {grafstuf unit}
          (Delete/Restore individual connections)
           Get_Removal_Probability
           Remove_Random_Connections
           Display_Network
        Corrupt_Input          {math unit}
          (Randomly reverse input values)
           Calculate_Output    {math unit}

  Display_Network              {grafstuf unit}
     Net_Display
        Draw_Layer_Box
        Draw_FFArrows
```

Figure 42. Operations Run Mode Procedure Map

SECTION 6

PROPOSED IMPROVEMENTS

As stated in the introduction, the development system is a prototype. As such, it is meant to be an initial version that supports the concept of a general system that will allow the creation and study of diverse network architectures and paradigms. The described networks created on the system support this stipulation. The development system should be exercised additionally with several other network types and training algorithms such as Bidirectional Associative Memories (BAMs), Counterpropagation nets, Outstar and Instar, and Self-Organizing nets to get a better idea of the system's generality, extensibility, and I/O compatibility with different network paradigms.

As network implementation and study of different network operations progressed, several improvements to the current development system were envisioned to be helpful to future users.

First among these would be the capability to display the output of the hidden layers. This would greatly enhance the user's view and understanding of the network's operation. Current capabilities only allow for the display of the input layer and the output layer. It would be of benefit to be able to see the outcome of the network computations for each hidden layer in a similar manner to how weight values are displayable for all layers. This is especially true when developing new training algorithms -- to be able to see if the computation and therefore, the implementation is correct. With multi-layer networks it becomes especially tedious to determine correct performance without this capability. Suggested methodology for this enhancement would be similar to the implementation for the memory display, allowing the operator to cycle through the layers displaying layer n to the left in Figure 28 and layer n + 1 to the right. The methodology for displaying hidden layers on the screen will need to be thoroughly analyzed, as in the case of the 80 neuron layer line architecture that will not comfortably fit on the display screen. Perhaps splitting it into 4, 20 neuron lines for display would be the best approach. Whatever the method of display, it should be made as clear as possible without adding unnecessary confusion for the user. A display of the actual numerical output of these hidden layers would also be helpful.

Additionally, the current system procedure for removal of a connection between two neurons is to set the weight value for that connection to zero. This, in effect, disconnects that neuron from participation in the computation of the output for the neuron in the next layer. A more reasonable method for implementation is through the use of connection matrices as specified in the layer_type definition (Appendix B). The definition of connections, through a dynamic variable connection matrix associated with each layer (similar to the weight matrix associated with each layer), 1 specifying a connection, 0 specifying no connection, would be a cleaner, more extensible way of defining connectivity. For each computation of a neuron's output through the associated weight to a neuron in the next layer, a check would be made of the connection matrix. If connected (matrix value 1) the computation would be made; if not connected, that computation would be skipped. Such a positive approach would be especially useful when studying the effects of partial connectivity on training of the network. With the present system, the study of the effects of connectivity is a cumbersome task, not satisfactorily solved, since the system cannot distinguish between a weight value of zero meaning no connection and a transient zero value resulting from the state of training. A drawback with this proposed implementation on the current system, of course, is its affect on available memory, and therefore maximum network size.

A second enhancement would allow for the display of individual neuron firing rates in a fashion similar to the display of the weight (memory) matrices. This desirable capability would allow for visual presentation of active and less active neurons in the network. The display could be color-coded to visually depict activity rates. (See the discussion below concerning the memory display and the

Suggestions for Further Research for a discussion concerning neuron firing rate data).

A third proposed improvement to the system would be the inclusion of full recurrency in the modeling capabilities of the program. As discussed previously, the only recurrency allowed in the present system is from the output layer back to the input layer. There are many experimental networks under current study that make use of recurrency between the hidden layers as well as within a single layer back to itself. This would require a full design review to determine how best to implement such a capability. However, with the current system's isolation of network categories, as discussed in the system architecture section and as depicted in Figures 8 and 40, this should be a task of only modest difficulty.

A weakness in the current system's design and its effect on extensibility, concern the use of different activation functions and their impact on the implementation of learning algorithms and network computation. For instance, in the case of the backpropagation algorithm, as shown in Figure 32 and 33 and in Equations 2 and 5, the computations depend on the derivative of a Sigmoid function. One of the benefits of the Sigmoid function, as described earlier, was the simplicity of the derivative (Equation 2). This derivative value (the slope of the sigmoid curve at the computed OUT value) is dependent on the value of OUT which, in turn, varies depending on whether the activation function is a Sigmoid or a Modified Sigmoid. These computation problems are currently handled within the backpropagation algorithm (Appendix H). The addition of new activation functions will therefore also require a modification of the learning algorithms which they affect. In fact, the simple threshold activation function cannot be used with the current implementation of the backpropagation algorithm since the derivative formula does not hold for that function. This hampers the ease with which new functions and algorithms may be incorporated into the system. The system design should be reviewed to see if extensibility in this area can be improved.

The display of weight matrices (memory) associated with each network computation layer can also be improved. The original desired functionality was to color-code weight values to allow the user to observe patterns and concentrations as well as the relative strength of the weight values. While pretty, the display implemented (grafstuf unit [Appendix E]; procedure display_weights) does not always clearly convey to the user these relative weight values. The human interface aspects of these color displays should be reviewed as well as the range of weight value differentiation in order to provide the most meaningful information to the user. Methods for display of actual weight values should also be investigated since the current procedure (displaying them if there is room, not displaying them if the weight boundary squares are too small) is less then satisfactory.

A final suggested improvement to the system is to develop the capability to compile and link new code into the system through the menu system. This optional selection has been made available in the Program Operations Menu (Figure 24), although it is not currently functional. The desired functionality of this capability is to allow the user to design and write new source code from within the development system, save it to a pre-specified file of skeleton source code, and compile it online as a unit. The scope of this functionality would be limited to a few specific areas such as activation functions and learning algorithms. These would be compiled as separate units and then linked to the rest of the system. The basis for this has been established through the (generally) complete separation of the menu system from the operations mode of the development system. These two functional areas are interfaced only through the two procedures in the op_stuf unit (Appendix I). The separation of activation functions and learning algorithms through case statement calls (Appendix H) also supports this desired functionality. Developing this capability would require an extensive redesign of the system. Nevertheless, its implementation would give the user a simple and straight forward method to experiment with different algorithms.

SECTION 7

SUGGESTIONS FOR FURTHER RESEARCH

The use of this Development System, among other things, allows the student to easily see the affect of different conditions on various networks. Some additional areas of study were suggested in the discussion of the backpropagation algorithm and in the proposed improvements section. Additionally, some other uses that have not been exercised in the described system implementations are the following:

1. A modification of current algorithms to include the collection and display of neuron firing rate data. This variable is included in the individual neuron specification (Appendix B) but has not been included in network algorithm implementations.
2. Use of individual neuron thresholds in training algorithms and network computations. The current networks use a global threshold to determine the binary output of a neuron. The effect of individual thresholds is a topic of current research in the area. This variable is also included in the individual neuron specification and is available for implementation in network algorithms.

3. Modification of existing learning algorithms, or implementation of new learning algorithms, to include the training of the neuron threshold. Varying the threshold in some consistent manner could speed training time significantly. This could be done for the global threshold or individual neuron thresholds and is another topic of ongoing research in ANNs.
4. Investigation of the relative effects of other desaturation algorithms on network training. As discussed in Wasserman [3], there has been no effort to optimize the desaturation function used in the current system. Other functions may prove to be superior to the one implemented in the present Development System. In order to provide this capability to offer different desaturation functions, a further design review should be made to decide how best to incorporate it.

SECTION 8

SUMMARY

A prototype Artificial Neural Network Development System has been proposed, designed, and implemented on an AT class Personal Computer. This system is meant to be an aid to students studying the use and operations of various ANNs that are of interest. The proposed system permits the simple modification of common parameters of neural networks as well as the implementation of novel architectures and algorithms. The system design, operation, and data flow are all described and documented. Specific implementations of the Delta Rule and backpropagation learning algorithms, an unsupervised Hebbian learning algorithm, and Hopfield nets are also described. Improvements to the system are suggested as well as proposed research topics that may be realized using the system as presented.

The system has been shown to provide a viable approach to the study of neural networks without requiring a full system development for each network or variable of interest.

The complete source code for the system and algorithms is presented in Appendices B through J to this report.

LITERATURE CITED

1. Southworth, D. M. The Development of the Human/Computer Interface for a Neural Network Implementation on a Personal Computer, Course Project for Human Computer Interaction, VPI CS5714, December 1989.

2. Rosenblatt, F. Principles of Neurodynamics, Spartan Books, 1962.

3. Wasserman, P. D. Neural Computing Theory and Practice, Van Nostrand Reinhold, 1989.

4. Stornetta, W. S. and Huberman, B. A. "An improved three-layer backpropagation algorithm", IEEE First International Conference on Neural Networks, 1987.

5. Springsteen, S. Neural Network Simulation, VPI Masters Project for Computer Science, May 1989.

6. McCulloch, W. W. and Pitts, W. "A logical calculus of the ideas imminent in nervous activity", Bulletin of Mathematical Biophysics, Vol 5, 1943.

7. Minsky, M. L. and Papert, S. Perceptrons, MIT Press, 1969.

8. Rumelhart, D. E., Hinton, G. E. and Williams, R. J. Learning internal representations by error propagation, Parallel Distributed Processing, Vol 1, MIT Press, 1986.

9. Lippmann, R. P. "An Introduction to Computing with Neural Nets", IEEE ASSP Magazine, April 1987.

10. Hopfield, J. J. "Neurons with Graded Response have Collective Computational Properties like those of Two-State Neurons", Proceedings of the National Academy of Sciences USA 81, May 1984.

11. Hopfield, J. J. and Tank, D. W. "Computing with Neural Circuits: A Model", Science, Vol 2, August, 1986.

12. Abu-Mostafa, Y. S. and St. Jacques, J. "Information Capacity of the Hopfield Model", IEEE Transactions on Information Theory, July 1985.

OTHER REFERENCES

13. Waltz, D. and Feldman, J. A., eds. Connectionist Models and Their Implications, Ablex Publishing Co., 1988.

14. Grossberg, S., ed. Neural Networks and Natural Intelligence., The MIT Press, 1988.

15. Touretzky, D. S. Advances in Neural Information Processing Systems 1, Morgan Kaufmann Publishers, Inc., 1989.

16. Vemuri, V. "Artificial Neural Networks: An Introduction", IEEE Computer Society Press Technology Series, Artificial Neural Networks: Theoretical Concepts, 1988.

17. Anderson, J. A. "Cognitive and Psychological Computation with Neural Models", IEEE Transactions on Systems, Man, and Cybernetics, September/October, 1983.

18. Yoh-Han Pao, Adaptive Pattern Recognition and Neural Networks, Addison-Wesley Publishing Co., Inc., 1989.

19. Hecht-Nielsen, R. "Neurocomputing: Picking the Human Brain", IEEE Spectrum, March 1988.

20. Linsker, R. "Self-Organization in a Perceptual Network", Computer, IEEE Computer Society, March, 1988.

21. Johnson, R. C. Cognizers, Neural Networks and Machines That Think, John Wiley & Sons, Inc., 1988.

22. Rietman, E. Experiments in Artificial Neural Networks, Tab Books, Inc., 1988.

23. Nelson, M. M. and Illingworth, W. T. A Practical Guide to Neural Nets, Addison-Wesley Publishing Co., Inc., 1991.

HUMAN COMPUTER INTERFACE SOURCES

24. Rubenstein, R. and Hersh, H. The Human Factor, Digital Press, 1984.

25. Brown, J. R. and Cuningham, S. Programming the User Interface, John Wiley & Sons, Inc., 1989.

APPENDIX A

Development System Specifications

DEVELOPMENT SYSTEM SPECIFICATIONS and INFORMATION

.	Maximum Number of Layers (max_layers)	5
.	Maximum Number of Neurons per Layer (max_bodies)	80
.	Input limited to binary values	(1, 0)

. No continuous value inputs

. Output values for each neuron are available through the neuron's floating point value - layers[x].neural_set.real_value[y].

. Input values in the data file must equal 1 or 0. These values are then converted to a real value (1.0/-1.0, 1.0/0.0, .5/-.5 etc.). Default values for these real numbers are included and depend on the type of training and activation function. The conversion values are assigned to net_description.one_value and net_description.zero_value in the net_description data variable. They may also be specified by the operator through the menu system. All network computations use the floating point values. Integer (binary) values are used only for input and output displays.

. A matrix architecture for the input layer and the output layer still uses a line as internal representation and computations for the neurons of that layer. Thus a 5 X 5 matrix designated by the operator is represented internally by the system as a line of 25 neurons. The matrix architecture specified by the user is for display purposes only. Thus a number, letter, or other graphic may be displayed as the input or the output of the network. Input data is read directly from the input file and placed in the line representation of the layer. Order for display of the matrix is by column, thus column 1 is read first from the line, then column 2, etc. In the example above, input 1 through 5 would represent the first column in the matrix, 6 through 10 the second column, etc.

. The system, when reading the input file, expects every other line to be either a line of data for supervised training or an empty line. Thus, if the data has no input line for a supervised training pair (as would be the case in an unsupervised training mode), the data **must** still include a blank line (RTN) after each data line.

- The computation of output for all networks is the same. Input data is read and placed in the binary values (layers[x].neural_set.binary_value[y]) for each neuron in the layer. Real value equivalents, as specified by the operator or defaulted by the program are placed in layers[x].neural_set.real_value[y]. Computation for each layer in the network is accomplished by summing the layer weights multiplied by the output value of each neuron in the previous layer [x]. This value is then sent through the activation function which determines the value of layers[x+1].neural_set.real_value[j] for all j (neurons) in layer x+1. Thresholding functions for each activation function then determine the value of layers[x+1].neural _set.binary_value[j]. This process is shown in Figure 5. The process continues for each layer until the output layer is reached. Output is displayed in binary format (on, off) according to layers[out_layer].neural_set.binary_value[j] for all j (neurons) in the output layer. As with input data, output data is displayed column by column for matrix architectures.

- A specified network and all its data can be saved manually by the operator. When saved, it is automatically saved to four DOS files. The File Name is specified by the operator and file name extensions (.1, .2, .3, and .4) are automatically added to the filename. Filename.1 is of type net_description and holds all net description data. Filename.2 is of type layer_type and holds all the layer data. Filename.3 is of type connection_matrix and holds the neuron connection data for each layer. Currently, the connection matrix is not used in the development system. Therefore, Filename.3 is saved as an empty file. Connections that have been removed are specified as weights equal to zero in the weight matrix. Filename.4 is of type weight_matrix and holds all the weights for each layer in the net.

- When the operator specifies periodic saves of the network state, each save is appended to file netstate.3 and netstate.4. Netstate.1 and netstate.4 are overwritten since this data does not change as the network operates. Once netstate files are opened they are not closed until done so manually by the operator or the network is exited to DOS. The operator must therefore be careful to close the files if a new network is specified without exiting and states of the old network are desired to be saved.

- In order to print the netstate file the operator is required to manually select this option through the menu system. The system reads the values of each data file (netstate.1 through netstate.4) and creates a network conforming to those values. Thus, in order to print the previously saved network states, the current network in the system is destroyed to allow the creation of the new one for printing. The operator is alerted to this fact and can choose either to continue, or skip the printing until he saves the current network.

- Saving the network and saving the state of network, although similar in function, are two distinct operations. Saving the state periodically saves the values of the network to four distinct files called netstate.1 through netstate.4. It is used to later analyze the operation of the network, normally after the completion of a training session. Saving the network saves the same data but only once, and then to a DOS file specified by the operator (filename.1 through filename.4). This function is used to save a network in order to recall it later so the operator need not re-specify and train the network each time. Basically, the final state of the network is saved when the operator selects this option.

APPENDIX B

Development System Main Program
and
Global Declarations

Appendix B through Appendix J constitute the complete development system program listing

```
Program Neural_System;

Uses Crt, Graph, nnglobal, grafstuf, menus, mathutil;


begin    {Neural_System}
  origmode := lastmode;
  TextColor (yellow);
  Initialize_graphics;{set the graphics mode-grafstuf unit}
  Initialize_stuff;     {Mathutil unit}
  Initialize_layers;    {Mathutil unit}
  RestoreCrtMode;  {Reset to textmode-toggle back and forth}
  System_Menu;     {Menus unit}
  closegraph;
  textmode (origmode);
  textbackground (black);
  TextColor (yellow);
end.     {Neural_System}

{-----------------------------------------------------------}
```

```
unit nnglobal;

interface

Const

  learning_kind : packed array [0..7] of string[16] =
                  ('Delta', 'Hebbian', 'Madaline',
                   'Adaline', 'Back Propogation',
                   'Hopfield', 'No Learning', 'Other');
  training_kind : packed array[0..2] of string[12] =
                  ('Supervised', 'Unsupervised', 'None');
  activation_kind : packed array [0..4] of string[16] =
                    ('Sigmoid', 'Modified Sigmoid',
                     'HypTangent', 'Threshold', 'Other');
  rows        = 80;   {Max Rows in a matrix}
  columns     = 80;   {Max Columns in a matrix}
  linesize    = 80;   {Max neurons in a line}
  max_layers  =  5;   {Max number of layers in network}
  stringsize  = 20;
  max_bodies  = 80;   {Max neurons in a layer}

Type

  activation_type = (Sigmoid, Mod_Sigmoid, HypTangent,
                    Threshold, you_define);
  learning_type   = (delta, hebb, madaline, adaline, backp,
                    Hopfield, none, other);
  training_type   = (supervised, unsupervised, zip);
  layer_arch_type = (Lines, Matrices);
  connection_type = (Fully, Partial, Not_Connected);
  layerange       = 1..max_layers;
  linerange       = 0..linesize;
  linetype        = packed array [linerange] of char;
  BinaryType      = -1..1;
  DeltaVectorType = packed array[1..max_bodies] of real;
  BodyValueType   = record
                      Binary_value : packed array
                          [1..max_bodies] of BinaryType;
                      Real_value : packed array
                          [1..max_bodies] of real;
                      threshold  : packed array
                          [1..max_bodies] of real;
                      rate   : integer;
                    end;
  net_arch_type   = (feedforward, recurrent);
```

```
net_description_type = record
                         net_name   : string [80];{MS Dos}
                         net_arch   : net_arch_type;
                         num_layers : layerange;
                         activation : activation_type;
                         threshold  : real;   {Activation}
                                       {threshold value}
                         learning   : learning_type;
                         training   : training_type;
                         factor     : real;  {various}
                                       {factors for}
                                        {Learn Algorithms}
                         factor1    : real;
                         factor2    : real;
                         factor3    : real;
                         momentum   : real;   {Momentum}
                         random_weights : boolean;
                         set_weights    : boolean;
                         save_state     : boolean;{State}
                                     {saved periodically?}
                         iterate        : integer; {Save}
                                              {how often?}
                         one_value      : real;    {I/O}
                                    {value representation}
                         zero_value     : real;
                       end;
strng             = packed array [1..stringsize] of char;
row               = 1..rows;
column            = 0..columns;
connection_matrix = record
                      value : packed array [1..max_bodies,
                               1..max_bodies] of '0'..'1';
                                     {Connected or not}
                      col_length : integer;
                      row_length : integer;
                    end;
weight_matrix     = record
                    value : packed array [1..max_bodies,
                             1..max_bodies] of real;
                    col_length : integer;
                    row_length : integer;
                  end;
connection_ptr    = ^connection_matrix;
weight_ptr        = ^weight_matrix;
```

```
  layer_type       = record
                        FFConnection_kind : Connection_type;
                              {Full or Partial}
                        Neurons     : integer;  {Number per}
                                                  {layer}
                        Neural_Set : BodyValueType;
                        Arch        : layer_Arch_type; {Layer}
                                                  {Architecture}
                        Arch_row_length : integer;     {Matrix}
                                                    {Row length}
                        arch_col_length : integer;     {Matrix}
                                                    {Col length}
                        weights          : weight_ptr; {Points}
                                              {to weight vector}
                        saved_weights    : weight_ptr;{weights}
                                                 {from time t-1}
                        delta_vector     : DeltaVectorType;
                        connections      : connection_ptr;{Ptr}
                                         {to connection matrix}
                        out_layer        : boolean;{last layer}
                     end;
  IntegerBufferType = packed array [1..max_bodies] of
                          BinaryType;
  RealBufferType    = packed array [1..max_bodies] of real;

Var

  M             : weight_matrix;  {For general matrix calcs}
  arch_set      : layerange;
  neuron_set    : layerange;
  ColNo         : linerange;
  ColsOnLine    : linerange;
  EndOfLine     : boolean;
  layers        : packed array [layerange] of layer_type;
                                             {Layer Data}
  lin           : linetype;
  net_name      : string [12];{filename for saving to disk}
  net_description : net_description_type; {Network}
                                           {specification}
  NextChar      : char;
  num_layers    : layerange;            {Layers in the net}
  train         : boolean; {for reading files in training}
                                 {mode}
  s             : string [80];
  InFileName    : string [80];
  IOCode        : integer;
  graphdriver   : integer;  {Following for Graphics Use}
  graphmode     : integer;
  errorcode     : integer;
  MaxX, MaxY    : integer;
  origmode      : word;
  in_buffer     : IntegerBufferType;       {Buffer for input}
  out_buffer    : IntegerBufferType;      {Buffer for output}
  r_out_buffer  : RealBufferType;          {Buffer for reals}
  infile        : text;                         {Input file}
  savefile1     : file of net_description_type;{Save Network}
  savefile2     : file of layer_type;          {Save Network}
  savefile3     : file of connection_matrix;   {Save Network}
  savefile4     : file of weight_matrix;       {Save Network}

implementation

end.
```

APPENDIX C

Development System Menu Listing
(Unit Menus)

Appendix B through Appendix J constitute the complete development system program listing

```
{-----------------------------------------------------------}

{This Unit (menus) provides the dialogue menus for the}
{system}

{-----------------------------------------------------------}

Unit Menus;

interface

uses graph, crt, grafstuf, printer, ops_stuf, nnglobal,
                              mathutil, ioutil, weights;

Procedure System_Menu;

implementation

{-----------------------------------------------------------}

Procedure Number_of_Neurons;

var
  ch          : char;
  i, j, int   : integer;

begin    {Number_of_Neurons}
  repeat
    clrscr;
    gotoxy (20, 8);
    writeln ('SELECT NUMBER OF NEURONS FOR EACH LAYER');
    gotoxy (20, 10);
    write ('Current Selections (');
    write (Layers [1].neurons);
    for i := 2 to num_layers do
      write (', ', Layers [i].neurons); {write number for
                                          each layer}
    write (')');
    gotoxy (10, 24);
    writeln ('                      <RTN> Skip Layer');
    gotoxy (1, 21);
    write ('=> ');
    i := 1;
    while i <=  num_layers do {Number of layers selected or
                                         default value}
      begin
        gotoxy (20, 11);
        clreol;
        write   ('        ');
    reverse;
    writeln ('Layer Number ', i);   {reverse video}
    normvid;
    gotoxy (4, 21);   {place cursor after =>}
    clreol;
    getline;
    if (ColsOnLine > 0) and (Nextchar in ['0'..'9'])
      then
        begin
          process_number (int);
          if int > max_bodies
            then
              begin
                gotoxy (10, 2);
                reverse;
                write ('Exceeds Max Neuron Limit (',
                                          max_bodies, ')');
                normvid;
                gotoxy (1, 21);
                clreol;          {Erase =>}
                gotoxy (1, 23);
                write ('<RTN> to continue');
                readln;
                i := i - 1;   {Set to repeat layer}
                gotoxy (1, 23);
                clreol;         {Erase <RTN>..}
                gotoxy (10, 2);   {Clear error message}
                clreol;
                gotoxy (1, 21);
                write ('=> ');
              end
            else
              begin
                layers[i].neurons := int;
                gotoxy (20, 10);
                clreol;
                write ('Current Selections (');
                write (Layers [1].neurons);
                for j := 2 to num_layers do
                  write (', ', Layers [j].neurons);
                                          {write each layer}
                 write (')');
              end;
        end;
    i := i + 1;
  end;
 neuron_set := num_layers;    {Need to initialize}
 gotoxy (20, 11);
 clreol;         {clear layer number status}
```

```
      gotoxy (10, 24);
      writeln ('<ESC> Layer Data Menu          <RTN> Modify
                                                    Numbers');
      gotoxy (1, 21);
      clreol;
      write ('=> ');
      ch := readkey;
   until ch = chr (27)  {ESC to exit to system menu}
end;     {Number_of_Neurons}

{---------------------------------------------------------}


{Number_of_Layers establishes the menu for specifying the}
{Number of layers}

Procedure Number_of_Layers;

var
  ch : char;
  i  : integer;

begin    {Number_of_Layers}
  repeat
    clrscr;
    gotoxy (20, 8);
    writeln ('SELECT NUMBER OF LAYERS');
    gotoxy (20, 10);
    writeln ('   Current Number (',
                      net_description.num_layers, ')');
    gotoxy (10, 24);
    write ('                              <RTN> Accept and
                                                 Continue');
    gotoxy (1, 21);
    write ('=> ');
    ch := readkey;
    while not (ch in ['1'.. '9', chr (13)]) do
      repeat_input (ch);
    case ch of
       '1'     : begin
                   gotoxy (10, 2);
                   reverse;
                   write ('Minimum Number of Layers = 2');
                   normvid;
                   gotoxy (1, 23);
                   write ('<RTN> to Continue');
                   readln;
                   gotoxy (1, 23);
                   clreol;     {Clear <RTN>..}
```

```
                    end;
       '2'..'9' : begin
                    if ord (ch) - 48 > max_layers
                      then
                        begin
                          gotoxy (10, 2);
                          reverse;
                          writeln ('Layers Exceed Maximum (',
                                    max_layers, ')');
                          normvid;
                          gotoxy (1, 23);
                          write ('<RTN>  to Continue');
                          readln;
                          gotoxy (1, 23);
                          clreol;
                        end
                      else num_layers := ord (ch) - 48;
                                          {Convert to integer}
                    end;
    end;   {case}
    for i := 1 to (num_layers - 1) do
      layers[i].out_layer := false;  {Set layers to not the
                                               last one}
    layers[num_layers].out_layer := true;  {set last layer
                                                    to true}
    net_description.num_layers := num_layers;
  until ch = chr (13);  {RTN to exit to layer menu}
end;     {Number_of_Layers}

{--------------------------------------------------------}

Procedure Get_Training_Mode;

var
  i  : integer;
  ch : char;

begin    {Get_Training_Mode}
  repeat
    clrscr;
    gotoxy (20, 8);
    writeln ('SELECT TRAINING MODE');
    gotoxy (20, 9);
    for i := 1 to (10 - (length (training_kind
                         [ord(net_description.training)])
                                      Div 2)) do
      write (' ');   {center the name}
    write ('(');     {write the kind of training}
    write (training_kind [ord(net_description.training)],
                                                    ')');
    gotoxy (1, 21);
    write ('=> ');
    gotoxy (20, 11);
    write ('1.  Supervised');
    gotoxy (20, 12);
    write ('2.  Unsupervised');
    gotoxy (20, 13);
    write ('3.  None');
    gotoxy (10, 24);
    writeln ('<ESC> Learning                        <^C>
                                                    Accept');
    gotoxy (1, 21);
    clreol;
    write ('=> ');
    repeat
      ch := readkey;
    until ch in ['1'..'3', #27, #3];
    case ch of
      '1' : net_description.training := supervised;
      '2' : net_description.training := unsupervised;
      '3' : net_description.training := zip;
    end;
  until ch in [#27, #3]  {ESC or ^C to exit to leraning
menu}
end;     {Get_Training_Mode}

{--------------------------------------------------------}

Procedure Learning (var flag : char);

var
  ch     : char;
  i, y   : integer;
  gain   : real;
  result : integer;

begin   {Learning}
repeat
    clrscr;
    gotoxy (20, 6);
    write ('SET LEARNING ALGORITHM');
    gotoxy (20, 7);
    for i := 1 to (10 - (length (learning_kind
                           [ord(net_description.learning)])
                                          Div 2)) do
      write (' ');   {center the name}
    write ('(');     {write the kind of learning}
write (learning_kind [ord(net_description.learning)],
                                                   ')');
gotoxy (20, 8);
for i := 1 to (10 - (length (training_kind
                      [ord(net_description.training)])
                                    Div 2)) do
  write (' ');    {center the name}
write ('(');      {write the kind of training}
write (training_kind [ord(net_description.training)],
                                                   ')');
y := 10;
for i := 1 to 8 do
  begin
    gotoxy (20, y);
    write (i,'.  ', learning_kind [i-1]);
    y := y + 1;
  end;
i := i + 1;
gotoxy (20, y);
write (i, '.  Set Gain Value (',
                       net_description.factor:4:2, ')');
i := i + 1;
y := y + 1;
gotoxy (20, y);
write ('A.  Set Tolerance Value (',
                      net_description.factor1:4:3, ')');
i := i + 1;
y := y + 1;
gotoxy (20, y);
write ('B', '.  Set Saturation Value (',
                      net_description.factor2:4:4, ')');
gotoxy (10, 24);
writeln ('<ESC>  Activation Menu       <^C>  Accept - Go
                                          to Main Menu');
gotoxy (1, 22);
write ('=> ');
ch := readkey;
while not (ch in ['1'..'9', 'A', 'a', 'B', 'b', #27,
                                                #03]) do
  repeat_input (ch);
case ch of
  '1'     : begin
              Net_description.learning := delta;
              Net_description.training := supervised;
            end;
  '2'     : begin
              Net_description.learning := hebb;
              Net_description.training := unsupervised;
            end;
```

```
 '3'     : begin
             Net_description.learning := Madaline;
             Net_description.training := supervised;
           end;
 '4'     : begin
             Net_description.learning := adaline;
             Net_description.training := supervised;
           end;
 '5'     : begin
             Net_description.learning := backp;
             Net_description.training := supervised;
           end;
 '6'     : begin
             Net_description.learning := Hopfield;
             Net_description.training := unsupervised;
           end;
 '7'     : begin
             Net_description.learning := none;
             Net_description.training := zip;
           end;
 '8'     : begin
             net_description.learning := other;
             get_training_mode;
           end;
'9'..'b' : begin
              clrscr;
              gotoxy (20, 8);
              case ch of
                '9' : begin
                        write ('Enter Gain Value [n]
                                                   ');
                        write ('(',
                         net_description.
                                     factor:4:2, ')');
                      end;
           'A', 'a' : begin
                        write ('Enter Tolerance Value
                                               [t] ');
                        write ('(',
                          net_description.
                                    factor1:4:3, ')');
                      end;
           'B', 'b' : begin
                        write ('Enter Saturation Value
                                               [s] ');
                        write ('(',
                          net_description.
                                    factor2:4:4, ')');
                      end;
```

```
                    end;  {case}
                    repeat
                      gotoxy (1, 21);
                      write ('=> ');
                      {$I-};
                      readln (gain);{Fix for rtn or esc!!!!}
                      result := ioresult;
                      {$I+}
                      {Check for real number}
                      if result <> 0     {not A real number}
                        then
                          begin
                            gotoxy (1, 1);
                            reverse;
                            write ('Illegal real number');
                            normvid;
                            gotoxy (4, 23);
                            write ('<RTN> to Continue');
                            readln;
                            readln;
                            gotoxy (1, 23);
                            clreol;
                          end
                        else
                          case ch of
                          '9' : net_description.factor :=
                                     gain; {gain on delta}
                     'A', 'a' : net_description.factor1 :=
                                          gain; {Tolerance}
                     'B', 'b' : net_description.factor2 :=
                                          gain; {Saturation}
                          end;   {case}
                    until result = 0;
                 end;
      #3       : begin
                   flag := #6;{Set flag to jump to main menu}
                   exit;
                 end;
    end;   {case}
  until ch = chr (27);
end;    {Learning}

{-----------------------------------------------------------}
```

```
Procedure Activation_Function (var flag : char);

var
  ch          : char;
  activate    : real;
  momentum    : real;
  I, a, j     : integer;
  result      : integer;

begin {Activation_Function}
  activate := net_description.threshold;  {Default if
                                         threshold chosen}
  momentum := net_description.momentum;
  repeat
    clrscr;
    gotoxy (20, 8);
    write ('SET ACTIVATION FUNCTION');
    gotoxy (20, 9);
    case net_description.activation of
      Sigmoid     : write ('(Sigmoid [α = ',
                     net_description.momentum:4:2, '])');
      Mod_Sigmoid : write ('(Modified Sigmoid [α = ',
                     net_description.momentum
                                         :4:2, '])');
      HypTangent  : write (' (Hyperbolic Tangent)');
      Threshold   : write ('(Simple Threshold [',
                                  activate:4:2, '])');
    end;   {case}
    gotoxy (20, 11);
    write ('1.  Sigmoid  (1/0)');
    gotoxy (20, 12);
    write ('2.  Modified Sigmoid (.5/-.5)');
    gotoxy (20, 13);
    write ('3.  H-Tangent  (1/-1)');
    gotoxy (20, 14);
    write ('4.  Simple Threshold (',
                 net_description.threshold:5:4, ')');
    gotoxy (20, 15);
    write ('5.  Other');
    gotoxy (20, 16);
    write ('6.  Set Real I/O Values (',
                 net_description.one_value:2:1, '/',
                net_description.zero_value:2:1, ')');
    gotoxy (10, 24);
    writeln ('<ESC>  Weight Menu      <^C>  Accept and
                                           Continue');
    gotoxy (1, 21);
    write ('=> ');
    ch := readkey;
```

```
while not (ch in ['1'.. '6', #27, #3]) do
  repeat_input (ch);
case ch of
 '1', '2' : begin     {Sigmoid}
              if ch = '1'
                then net_description.activation :=
                                              sigmoid
                else net_description.activation :=
                                          mod_sigmoid;
              repeat
                clrscr;      {get momentum value}
                gotoxy (20, 8);
                write ('Enter Momentum Value  [α] (');
                write (net_description.momentum:4:2,
                                                  ')');
                gotoxy (1, 21);
                write ('=> ');
                {$I-};
                readln (momentum);{Fix for rtn or esc!!}
                result := ioresult;
                {$I+}
                {Check for real number}
                if result = 0     {A real number}
                  then net_description.momentum :=
                                             momentum
                  else
                    begin
                      gotoxy (1, 1);
                      reverse;
                      write ('Illegal real number');
                      normvid;
                      gotoxy (4, 23);
                      write ('<RTN> to Continue');
                      readln;
                      readln;
                      gotoxy (1, 23);
                      clreol;
                      momentum :=
                            net_description.momentum;
                    end;
              until result = 0;
              repeat
                clrscr;   {get threshold value}
                gotoxy (20, 8);
                write ('Enter Neuron Threshold Value
                                                   (');
                write (net_description.threshold:5:4,
                                                  ')');
                gotoxy (1, 21);
```

```
    write ('=> ');
    {$I-};
    readln (momentum);{Fix for rtn or esc!!}
    result := ioresult;
    {$I+}
    {Check for real number}
    if result = 0     {A real number}
      then net_description.threshold :=
                                   momentum
      else
        begin
          gotoxy (1, 1);
          reverse;
          write ('Illegal real number');
          normvid;
          gotoxy (4, 23);
          write ('<RTN> to Continue');
          readln;
          readln;
          gotoxy (1, 23);
          clreol;
          momentum :=
                net_description.threshold;
        end;
  until result = 0;
  repeat
    clrscr;   {get neuron threshold value}
    gotoxy (20, 8);
    write ('Enter Individual Neuron
                         Threshold Value (');
    write (layers[2].neural_set.
                      threshold[1]:5:4, ')');
    gotoxy (1, 21);
    write ('=> ');
    {$I-};
    readln (momentum);{Fix for rtn or esc!!}
    result := ioresult;
    {$I+}
    {Check for real number}
    if result <> 0     {A real number}
      then
        begin
          gotoxy (1, 1);
          reverse;
          write ('Illegal real number');
          normvid;
          gotoxy (4, 23);
          write ('<RTN> to Continue');
          readln;
```

```
                    readln;
                    gotoxy (1, 23);
                    clreol;
                    momentum :=
                    layers[2].neural_set.threshold[1];
                  end;
            until result = 0;
            for a := 2 to net_description.num_layers
                                                    do
              for j := 1 to layers[a].neurons do
                layers[a].neural_set.threshold[j] :=
                                              momentum;
            if ch = '1'  {Sigmoid activation}
              then
                begin
                 net_description.one_value := 1.0;
                 net_description.zero_value := 0.0;
                end
              else       {mod_sigmoid activation}
                begin
                  net_description.one_value := 0.5;
                  net_description.zero_value := -0.5;
                end;
          end;
'3'     : begin    {HypTangent}
            net_description.activation := HypTangent;
            net_description.one_value := 1.0;
            net_description.zero_value := -1.0;
          end;
'4'     : begin    {Simple Threshold}
            repeat
              clrscr;
              gotoxy (20, 8);
              write ('Enter Threshold Value   ', '(');
              write (net_description.Threshold:4:2,
                                                 ')');
              gotoxy (1, 21);
              write ('=> ');
              {$I-};
              readln  (Activate);{Fix for rtn or esc!}
              result := ioresult;
              {$I+}
              {Check for real number}
              if result = 0     {A real number}
                then
                  begin
                    net_description.threshold :=
                                             activate;
```

```
                          net_description.activation :=
                                                threshold;
                        end
                      else
                        begin
                          gotoxy (1, 1);
                          reverse;
                          write ('Illegal real number');
                          normvid;
                          gotoxy (4, 23);
                          write ('<RTN> to Continue');
                          write (#7);
                          readln;
                          readln;
                          gotoxy (1, 23);
                          clreol;
                          net_description.activation :=
                                               threshold;
                          activate :=
                                net_description.threshold;
                        end;
                  until result = 0;
                  net_description.one_value := 1.0;
                  net_description.zero_value := 0.0;
                end;
      '5'     : begin        {You_Define}
                  net_description.activation := you_define;
                  net_description.one_value := 1.0;
                                          {Modify for setting}
                  net_description.zero_value := 0.0;
                end;
      '6'     : begin
                  repeat
                    clrscr;        {get one_value}
                    gotoxy (20, 8);
                    with net_description do
                      begin
                        write ('Enter Real Value for 1 (');
                        write (One_value:2:1, ')');
                      end;
                    gotoxy (1, 21);
                    write ('=> ');
                    {$I-};
                    read (momentum);{Fix for rtn or esc!!!!}
                    result := ioresult;
                    {$I+}
                    {Check for real number}
```

```
    if result = 0     {A real number}
      then net_description.one_value :=
                                     momentum
      else
        begin
          gotoxy (1, 1);
          reverse;
          write ('Illegal real number');
          normvid;
          gotoxy (4, 23);
          write ('<RTN> to Continue');
          readln;
          readln;
          gotoxy (1, 23);
          clreol;
          momentum :=
                  net_description.momentum;
        end;
  until result = 0;
  repeat
    clrscr;   {get zero_value}
    gotoxy (20, 8);
    with net_description do
      begin
        write ('Enter Real Value for 0 (');
        write (zero_value:2:1, ')');
      end;
    gotoxy (1, 21);
    write ('=> ');
    {$I-};
    read (momentum);{Fix for rtn or esc!!!!}
    result := ioresult;
    {$I+}
    {Check for real number}
    if result = 0     {A real number}
      then net_description.zero_value :=
                                     momentum
      else
        begin
          gotoxy (1, 1);
          reverse;
          write ('Illegal real number');
          normvid;
          gotoxy (4, 23);
          write ('<RTN> to Continue');
          readln;
          readln;
          gotoxy (1, 23);
          clreol;
```

```
                                 momentum :=
                                         net_description.threshold;
                               end;
                         until result = 0;
                       end;
      #3       : Learning (flag);    {<^C>}
    end;   {case}
    if flag = #6       {Unwind out of menus to Main}
      then exit;
  until ch = chr (27);    {<ESC>}
end;  {Activation_Function}

{------------------------------------------------------------}

Procedure Set_Weights (var flag : char);

var
  ch   : char;
  i    : integer;
  update : boolean;  {flag for modifying weights}

begin    {Set_Weights}
  repeat
    update := false;
    clrscr;
    gotoxy (20, 8);
    write ('SET WEIGHTS');
    gotoxy (20, 9);
    if net_description.random_weights
      then write ('(Random)')
      else write ('(Set Weights)');   {File or console????}
    gotoxy (20, 11);
    write ('1.  All Layers Random');
    gotoxy (20, 12);
    write ('2.  Set All Weights Equal');
    gotoxy (20, 13);
    write ('3.  Set Weights from File');
    gotoxy (20, 14);
    write ('4.  Modify Weights');
    gotoxy (20, 15);
    write ('5.  Display Weights');
    gotoxy (20, 16);
    write ('6.  Display Saved_Weights');
    gotoxy (20, 17);
    write ('7.  Initialize Saved Weights Matrix');
    gotoxy (10, 24);
    writeln ('<ESC>  Connectivity Menu       <^C>  Accept
                                             and Continue');
    gotoxy (1, 21);
```

```
write ('=> ');
ch := readkey;
while not (ch in ['1', '2', '3', '4', '5', '6', '7',
                                          #27, #3]) do
  repeat_input (ch);
case ch of
  '1'     : begin
              if net_description.random_weights
                then
                  begin
                    gotoxy (10, 1);
                    reverse;
                    write ('Random Weights Already
                                          Set!');
                    gotoxy (10, 2);
                    write ('Set Again?  Y/N');
                    normvid;
                    ch := readkey;
                    if ch in ['N', 'n']
                      then
                        begin   {Don't do anything}
                        end
                      else
                        begin
                          net_description.random_weights
                                              := true;
                          net_description.set_weights :=
                                                false;
                          Set_Random_Weights;
                        end;
                  end
                else
                  begin
                   net_description.random_weights :=
                                                true;
                   net_description.set_weights := false;
                   Set_Random_Weights;
                  end;
            end;
  '2'     : begin
              net_description.random_weights := false;
              net_description.set_weights := true;
              Set_Equal_Weights;
            end;
  '3'     : begin
              net_description.random_weights := false;
              net_description.set_weights := true;
            end;
```

```
'4'      : begin
              ch := 'y';
              if layers[2].saved_weights <> nil
              then
                begin
                  clrscr;
                  gotoxy (1, 1);
                  reverse;
                  write ('WARNING - Saved_Weights should
                                           be deleted');
                  write (' - Y/N?');
                  normvid;
                  repeat
                    ch := readkey;
                  until ch in ['Y', 'y', 'N', 'n'];
                  if ch in ['y', 'Y']
                    then for i := 2 to
                           net_description.num_layers do
                      begin
                        dispose
                              (layers[i].saved_weights);
                        layers[i].saved_weights := nil;
                      end;
                end;
                if ch in ['Y', 'y']
                  then
                    begin
                      net_description.set_weights :=
                                                true;
                      update := true;
                      Display_weights (1, update, true);
                                        {weights unit}
                      gotoxy (1, 22);
                      clreol;
                      write ('Do You wish to initialize
                                 saved_weights??');
                      write (' - Y/N');
                      ch := readkey;
                      if ch in ['Y', 'y']
                        then init_saved_weights;
                                      {Mathutil unit}
                      update := false;
                    end;
            end;
'5'      : Display_Weights (1, update, true);
'6'      : Display_Weights (1, false, false);   {No
                          update/pass saved_weights}
'7'      : init_saved_weights;        {Mathutil unit}
#3       : Activation_Function (flag);   {Math unit}
```

```
    end;   {case}
    if flag = #6    {Unwind out of menus to Main}
      then exit;
  until ch = chr (27);
end;     {Set_Weights}

{---------------------------------------------------------}

Procedure Display_and_Enter_Partials;

begin {Display_and_Enter_Partials}
  clrscr;
  gotoxy (20, 3);
  write ('Connectivity For Layers Shown');
  gotoxy (20, 10);
  writeln ('This is a stubb for Recurrent Network
                              Connectivity Definition');
  write ('Press any key to continue');
  repeat
  until keypressed;
end;  {Display_and_Enter_Partials}

{-----------------------------------------------------------}

Procedure Define_Partial_Connectivity_Menu (var ch : char);

var
  i  : integer;

begin    {Define_Partial_Connectivity_Menu}
  repeat
    i := 1;
    repeat            {Feedforward direction}
      clrscr;
      gotoxy (15, 8);
      writeln ('DEFINE CONNECTIVITY FOR EACH PARTIALLY
                                  CONNECTED LAYER');
      gotoxy (15, 14);
      write ('1.  Modify Connections');
      gotoxy (15, 15);
      write ('2.  Display Network');
      gotoxy (1, 24);
      write ('<ESC> General Connectivity Menu       <RTN>
                                        Skip Layer');
      write ('      <^C> Accept');
      gotoxy (1, 21);
      write ('=> ');
     while (layers[i].FFconnection_kind <> partial) and (i <
                                        num_layers) do
      i := i + 1;  {Find next partially connected layer}
    if layers[i].FFconnection_kind = partial
      then
        begin
          gotoxy (20, 9);
          write ('Layer ');
          reverse;
          write (i:2);
          normvid;
          write (' to Layer ');
          reverse;
          write (i + 1:2);
          normvid;
          clreol;
          write ('  (Partial)');
          gotoxy (4, 21);
          ch := readkey;
          while not (ch in ['1', '2', #3, #13, #27]) do
            repeat_input (ch);
          case ch of
            '1' : Modify_Connections (ch, i, 'Connectivity
                                                    Menu');
            '2' : Display_Network (5);
             #3 : begin      {Provisions only}
                    exit;
                  end;
            #13 : begin     {<RTN> Skip this layer}
                    i := i + 1;
                  end;
            #27 : exit; {<ESC> exit to connectivity menu}
          end;  {case}
        end;
  until i = num_layers;
  if Net_Description.net_arch = recurrent
    then
      repeat
        i := 1;  {Stubb!!!!!!!!!}
        clrscr;
        gotoxy (20, 8);
        writeln ('DEFINE CONNECTIVITY FOR EACH PARTIAL
                                              CONNECTION');
        i := num_layers;
        gotoxy (20,10);
        write ('1.  Display and Enter Connections for the
                                                   Layer');
        gotoxy (8, 24);
        writeln ('<ESC> General Connectivity Menu  <RTN>
                              Skip Layer   <^C> Accept');
        gotoxy (1, 21);
```

```
          write ('=> ');
          while (layers[i].FFconnection_kind <> partial)
                            and (i <     num_layers - 1) do
          i := i - 1;
          if layers[i].FFconnection_kind = partial
          then
            begin
              gotoxy (20, 9);
              write ('Layer ');
              reverse;
              write (i:2);
              normvid;
              write (' to Layer ');
              reverse;
              write (i + 1:2);
              normvid;
              clreol;
              write ('  (Partial)');
              gotoxy (20, 11);
              {Display  the connectivity Matrix}
              gotoxy (4, 21);
              {Go through the recurrent layers}
              gotoxy (4, 21);
              ch := readkey;
              while not (ch in ['1', chr (03), chr (13), chr
                                                    (27)]) do
                repeat_input (ch);
              case ch of
                '1' : Display_and_Enter_Partials;
                 #3 : begin      {Provisions only}
                      end;
                #13 : begin     {Skip this layer}
                      end;
                #27 : exit;    {exit to connectivity menu}
              end;  {case}
            end;
        until i = 1;
  until ch = chr (27)  {ESC to exit to connectivity menu}
end;     {Define_Partial_Connectivity_Menu}

{-------------------------------------------------------------}
```

```
Procedure Net_Connectivity_Menu (var flag : char);

var
  ch : char;

{---------------------------------------}

Procedure Set_R_Arch (var ch : char);

var
  i, result : integer;

begin    {Set_R_Arch}
  gotoxy (1, 21);
  clreol;
  write ('=> ');
  ch := readkey;
  while not (ch in ['1'.. '4', chr (27), chr (03), chr
                                                    (13)]) do
    repeat_input (ch);
  case ch of
   '1' : begin
           Layers[i].FFconnection_kind := fully;
         end;  {Define structures}
   '2' : Layers[i].FFconnection_kind := Partial;
   '3' : Layers[i].FFconnection_kind := Not_Connected;
   '4' : Display_Network (4);
   #13 : begin     {<RTN> Skip to next layer connection}
         end;
   #27 : begin
         end;
  end;   {case}
  if (ch in [#27, #3, '3'])
    then exit;  {ESC Net Connectivity => Set_Weights}
end;     {Set_R_Arch}

{--------------------------------------------}

Procedure Set_FF_Arch (var ch : char);

var
  i,j  : integer;
  Part : boolean;  {Is this a partial connection?}

begin     {Set_FF_Arch}
  Part := false;
  for i := 1 to (num_layers - 1) do
    begin
      gotoxy (20, 12);
```

```
write ('Layer ');
reverse;
write (i:2);
normvid;
write (' to Layer ');
reverse;
write (i + 1:2);
normvid;
write ('    ');
clreol;
if layers[i].FFconnection_kind = fully
  then write ('(Fully)')
  else write ('(Partial)');
gotoxy (1, 21);
clreol;
write ('=> ');
ch := readkey;
while not (ch in ['1', '2', '3', '4', #27, #3, #13])
                                                    do
  repeat_input (ch);
case ch of
  '1' : if (layers[i].FFconnection_kind = partial) and
          (layers[i + 1].saved_weights = nil)
            then
              begin
                gotoxy (1, 1);
                reverse;
                write ('No saved_weights.. Cannot
                                     restore! <RTN>');
                repeat
                until keypressed;
                normvid;
                gotoxy (1, 1);
                clreol;
              end
            else if (layers[i].FFconnection_kind =
                                            partial)
              then
                begin
                  gotoxy (1, 1);
                  reverse;
                  write ('WARNING.. All weights in
                               this layer will be');
                  writeln (' restored!!');
                  write (' <RTN> to Continue   <ESC>
                                           to Quit');
                  repeat
                    ch := readkey;
                  until ch in [#13, #27];
                normvid;
                if ch = #13
                  then
                    begin
                      Layers[i].FFconnection_kind :=
                                                Fully;
                      layers[i + 1].weights^ :=
                       layers[i + 1].saved_weights^;
                    end;
                gotoxy (1, 1);
                clreol;
                gotoxy (1, 2);
                clreol;
                ch := '1';
              end;
'2' : Layers[i].FFconnection_kind := Partial;
'3' : begin       {Randomize Weight Settings}
        gotoxy (1, 1);
        reverse;
        writeln ('WARNING!  This will destroy all
                                weight settings');
        write ('<RTN> to Continue    <ESC> to
                                          Quit');
        repeat
          ch := readkey
        until ch in [#13, #27];
        normvid;
        if ch = #13   {RTN}
          then
            begin
              gotoxy (1, 5);
              write ('Working...');
              set_random_weights; {unit weights}
              net_description.random_weights :=
                                             true;
              net_description.set_weights := false;
            end;
        gotoxy (1, 1);
        clreol;
        gotoxy (1, 2);
        clreol;
        gotoxy (1, 5);
        clreol;
        ch := '3';
      end;
'4' : begin       {Save weight settings}
        for j := 2 to num_layers do
          begin
            if layers[j].saved_weights = nil
```

```
                      then new (layers[j].saved_weights);
                    layers[j].saved_weights^ :=
                                     layers[j].weights^;
                  end;
                gotoxy (1, 1);
                reverse;
                write ('Saved_Weights set to Weight values.
                                               <RTN>');
                normvid;
                repeat
                until keypressed;
                gotoxy (1, 1);
                clreol;
              end;
        '5' : Display_Network (4);
        #3  : begin
                For j := 1 to num_layers do
                  if layers[j].FFConnection_kind = partial
                    then Part := true;
                  if Part
                    then
                      begin
                        Define_Partial_connectivity_menu
                                                   (ch);
                        if ch = #27
                          then
                            begin
                              ch := #126;
                              exit;
                            end;
                      end
                    else exit;
              end;   {Else return to Connection menu passing
                                                      ^C}
        #13 : begin     {<RTN> Skip to next layer
                                              connection}
              end;
      end;   {case}
      if (ch in [#27, #3, '4'])
        then exit;  {ESC Net Connectivity => Set_Weights}
    end;
end;      {Set_FF_Arch}

{-------------------------------------}

begin   {Net_Connectivity_Menu}
  repeat
    clrscr;
    gotoxy (20, 8);
    writeln ('DEFINE GENERAL CONNECTIVITY');
    gotoxy (20, 9);
    case Net_Description.net_arch of
      feedforward : writeln ('     (Feedforward)');
      recurrent   : writeln ('      (Recurrent)');
    end;   {case}
    gotoxy (20, 10);
    write ('Number of Layers          (', num_layers, ')');
    gotoxy (20, 14);
    writeln ('1.  Fully');
    gotoxy (20, 15);
    write   ('2.  Partial');
    gotoxy (20, 16);
    if net_description.net_arch = recurrent
      then
        begin
          write ('3.  Not Connected');
          gotoxy (20, 17);
          write ('4.  Randomize Weights ');
          gotoxy (20, 18);
          write ('5.  Save Weight Settings');
          gotoxy (20, 19);
          write ('6.  Display Network');
          gotoxy (20, 19);
        end
      else
        begin
          write ('3.  Randomize Weights');
          gotoxy (20, 17);
          write ('4.  Save Weight Settings');
          gotoxy (20, 18);
          write ('5.  Display Network'); {All layers
                                 connected in feedforward}
        end;
    gotoxy (8, 24);
    write ('<ESC> Net Arch Menu   <RTN> Accept Layer   <^C>
                                    Accept and Continue');
    if net_description.net_arch = feedforward
     then Set_FF_Arch (ch)  {Pass ch to escape to net
                                  architecture from set..}
     else Set_R_Arch (ch);
   if ch = chr (03)
     then Set_Weights (flag);
   if flag = #6      {Unwind out of menus to Main}
     then exit;
until ch = chr (27);   {ESC Exit to Net_Architecture}
end;  {Net_Connectivity_Menu}

{------------------------------------------------------------}
```

```
Procedure Net_Architecture (var flag : char);

var
  ch : char;

begin   {Net_Architecture}
  repeat
    clrscr;
    gotoxy (20, 8);
    writeln ('SELECT NET ARCHITECTURE');
    gotoxy (20, 9);
    case Net_Description.net_arch of
      feedforward : writeln ('    (Feedforward)');
      recurrent   : writeln ('     (Recurrent)');
    end;   {case}
    gotoxy (20, 12);
    writeln ('1.  Feedforward Only');
    gotoxy (20, 13);
    write   ('2.  Recurrent');
    gotoxy (10, 24);
    writeln ('<ESC> Input Layer Data      <^C> Accept and
                                               Continue');
    gotoxy (1, 21);
    write ('=> ');
    ch := readkey;
    while not (ch in ['1', '2', chr (27), chr (03)]) do
      repeat_input (ch);
    case ch of
      '1' : Net_Description.net_arch := feedforward;
      '2' : Net_Description.net_arch := recurrent;
      #3  : net_connectivity_menu (flag);
    end;   {case}
    if flag = #6    {Unwind out of menus to Main}
      then exit;
  until ch = chr (27);   {ESC Exit to System_Menu}
end;      {Net_Architecture}

{-----------------------------------------------------------}

Procedure Define_Matrix (var ch : char);

var
  i, j, k, int, a  : integer;
  matrix_ok        : boolean;

begin   {Define_Matrix}
  repeat
    for i := 1 to num_layers do
      if layers[i].arch = matrices  {do only if matrix}
```

```
then
  repeat
    clrscr;
    gotoxy (24, 8);
    writeln ('DEFINE MATRIX');
    gotoxy (5, 24);
    writeln ('<ESC> Layer Data    <RTN> Next Layer
                      <^C> Accept & Continue');
    j := layers[i].arch_row_length;
    k := layers[i].arch_col_length;
    if j * k = layers[i].neurons
      then matrix_ok := true
      else matrix_ok := false;
    gotoxy (20, 10);
    write ('Layer ');
    reverse;
    write (' ',i, ' ');
    normvid;
    write ('   (', layers[i].neurons, ' Neurons)');
    gotoxy (20, 11);
    clreol;
    write ('Rows X Cols    (', j, ' X ', k, ')');
    gotoxy (20, 15);
    write ('1.  Number of Rows');
    gotoxy (20, 16);
    write ('2.  Display Net');
    gotoxy (1, 21);
    clreol;
    write ('=> ');
    gotoxy (4, 21);
    ch := readkey;
    while not (ch in ['1', '2', #3, #27, #13]) do
      repeat_input (ch);
    case ch of
      '1' : begin
              matrix_ok := false;
              gotoxy (1, 22);
              write ('Enter Number of Rows in the
                                Layer Matrix');
              gotoxy (4, 21);
              getline;
              if (ColsOnLine > 0) and (Nextchar in
                                        ['0'..'9'])
                then
                  begin
                    process_number (int);
                    if (int > max_bodies) or (int =
                                                  0)
                      then
    begin
      gotoxy (10, 2);
      reverse;
      if int > max_bodies
        then
          write ('Exceeds Max
             Neuron Limit (',
              max_bodies, ')')
        else write ('0 Rows
                   Illegal!');
      normvid;
      gotoxy (1, 21);
      clreol;         {Erase =>}
      gotoxy (1, 22);
      clreol;{Erase cue message}
      write ('<RTN> to
                     continue');
      readln;
      gotoxy (1, 22);
      clreol;   {Erase <RTN>..}
      gotoxy (10, 2);   {Clear
                  error message}
      clreol;
      gotoxy (1, 21);
      write ('=> ');
    end
  else
    begin
      j := int;
      gotoxy (1, 22);
      clreol;  {Remove cue}
      if (layers[i].neurons mod
                         j) <> 0
        then
          begin
            gotoxy (10, 1);
            reverse;
            write ('Matrix Not
                  Symetrical!');
            normvid;
            gotoxy (20, 11);
            write ('Rows X Cols
                (', j, ' X  ?');
            gotoxy (1, 21);
            write ('<RTN> to
                    Continue');
            readln;
            gotoxy (10, 1);
            clreol;
```

```
                                                  end
                                                else
                                                  begin
                                                    matrix_ok := true;
                                                    k :=
                                                      layers[i].
                                                         neurons div j;
                                                    gotoxy (20, 11);
                                                    write ('Rows X Cols
                                                      (', j, ' X ', k);
                                                    layers[i].
                                                   arch_row_length := j;
                                                    layers[i].
                                                   arch_col_length := k;
                                                  end;
                                            end;
                                      end
                                    else
                                      begin
                                        gotoxy (10, 2);
                                        reverse;
                                        write ('Illegal Entry');
                                        normvid;
                                        gotoxy (1, 21);
                                        clreol;          {Erase =>}
                                        gotoxy (1, 22);
                                        clreol;     {Erase cue message}
                                        write ('<RTN> to continue');
                                        readln;
                                        gotoxy (1, 22);
                                        clreol;         {Erase <RTN>..}
                                        gotoxy (10, 2); {Clear error
                                                              message}
                                        clreol;
                                        gotoxy (1, 21);
                                        write ('=> ');
                                      end;
                                end;
                          '2'  : Display_Network (3);
                         #27   : Exit;
                          #3   : begin
                                   matrix_OK := true;
                                   for a := 1 to
                                          net_description.num_layers do
                                     if layers[a].arch_row_length
                                         *layers[a].arch_col_length
                                         <> layers[a].neurons  {Not
                                                           symetrical}
                                       then matrix_OK := false;
```

```
                            if not matrix_OK
                              then
                                begin
                                  gotoxy (10, 2);
                                  reverse;
                                  write ('At least one matrix
                                             not symetrical!');
                                  normvid;
                                  gotoxy (1, 21);
                                  clreol;         {Erase =>}
                                  gotoxy (1, 22);
                                  clreol;     {Erase cue
                                                     message}
                                  write ('<RTN> to continue');
                                  readln;
                                  gotoxy (1, 22);
                                  clreol;        {Erase <RTN>..}
                                  gotoxy (10, 2);   {Clear error
                                                      message}
                                  clreol;
                                  gotoxy (1, 21);
                                  write ('=> ');
                                end
                              else exit;
                        end;
                end;   {case}
              until matrix_ok;
  until ch = chr (27);   {ESC Exit to System_Menu}
end;    {Define_Matrix}

{-----------------------------------------------------------}

Procedure Layer_Architecture;

var
  ch   : char;
  i, j : integer;

begin    {Layer_Architecture}
  repeat
    clrscr;
    gotoxy (20, 8);
    writeln ('SELECT LAYER ARCHITECTURE FOR EACH LAYER');
    gotoxy (20, 9);
    clreol;
    case Layers [1].Arch of      {write architecture code
                                         for each layer}
      lines    : write ('        (Line');
      matrices : write ('        (Matrix');
end;   {case}
for j := 2 to num_layers do
  case Layers [j].Arch of
    lines    : write (', Line');
    matrices : write (', Matrix');
  end;   {case}
write (')');
gotoxy (20, 13);
writeln ('1.  Line');
gotoxy (20, 14);
writeln ('2.  Matrix');
gotoxy (20, 15);
writeln ('3.  Define Your Own  (Not Functional)');
gotoxy (10, 24);
writeln ('<ESC> Layer Data Menu        <RTN> Next
                                          Layer');
gotoxy (1, 21);
write ('=> ');
for i := 1 to num_layers do   {Number of layers selected
                                     or default value}
  begin
    gotoxy (20, 11);
    clreol;
    write ('          ');
    reverse;
    writeln ('Layer Number ', i);   {reverse video}
    normvid;
    gotoxy (4, 21);
    ch := readkey;
    while not (ch in ['1', '2', '3', chr (13), chr
                                           (27)]) do
      repeat_input (ch);
    case ch of
      '1' : layers[i].arch := lines;
      '2' : layers[i].arch := matrices;
      '3' : begin      {Provisions only}
            end;
      #13 : begin     {Skip this layer}
            end;
      #27 : exit;    {exit to layer_menu}
    end;  {case}
    gotoxy (20, 9);    {Clear default values to reprint}
    clreol;
    case Layers [1].Arch of   {write architecture code
                                        for each layer}
      lines    : write ('        (Line');
      matrices : write ('        (Matrix');
    end;   {case}
    for j := 2 to num_layers do
```

```
          case Layers [j].Arch of
            lines     : write (', Line');
            matrices : write (', Matrix');
          end;   {case}
        write (')');
      end;
    arch_set := num_layers;
  until ch = chr (27);  {ESC to exit to system menu}
  end;     {Layer_Architecture}

{-------------------------------------------------------}

Procedure Get_Corruption_Prob;

  var
    prob   : real;
    result : integer;

begin    {Get_Corruption_Prob}
  repeat
    clrscr;      {get momentum value}
    gotoxy (20, 8);
    write ('Enter Corruption Probability  [p] (');
    write (net_description.factor3:4:2, ')');
    gotoxy (1, 21);
    write ('=> ');
    {$I-};
    readln (prob);   {Fix for rtn or esc!!!!}
    result := ioresult;
    {$I+}
    {Check for real number and correct probability}
    if (result = 0) and ((Prob <= 1) and (Prob >= 0)) {A
                                                  real number}
      then net_description.factor3 := prob
      else
        begin
          gotoxy (1, 1);
          reverse;
          write ('Illegal real number or probability');
          normvid;
          gotoxy (4, 23);
          write ('<RTN> to Continue');
          repeat
          until keypressed;
          gotoxy (1, 23);
          clreol;
          prob := net_description.factor3;
        end;
  until result = 0;
```

```
end;     {Get_Corruption_Prob}

{---------------------------------------------------------}

{Layer_Menu establishes the Neural Network Menu on the}
      {screen}

Procedure Layer_Menu (var flag : char);

var
  ch : char;
  i  : integer;
  mat : boolean;   {Matrix architecture for this layer?}

begin     {Layer_Menu}
  repeat
    clrscr;
    gotoxy (20, 8);
    writeln ('   LAYER DATA MENU');
    gotoxy (20, 9);
    writeln ('   (Current Values)');
    gotoxy (10, 12);
    writeln ('1.  Number of Layers                      (',
                                          num_layers, ')');
    gotoxy (10, 13);
    write   ('2.  Layer Architecture          ');
    case Layers [1].Arch of      {write architecture code
                                          for each layer}
      lines     : write ('(Line');
      matrices  : write ('(Matrix');
    end;   {case}
    for i := 2 to num_layers do
      case Layers [i].Arch of
        lines     : write (', Line');
        matrices  : write (', Matrix');
      end;   {case}
    write (')');
    gotoxy (10, 14);
    write   ('3.  Number of Neurons Per Layer       (',
                                       Layers[1].Neurons);
    for i := 2 to Num_Layers do
      write (', ', Layers [i].Neurons);
    write (')');
    gotoxy (10, 15);
    write ('4.  Display Network');
    gotoxy (10, 16);
    write ('5.  Set Input Corruption Probability   (',
                        net_description.factor3:4:2, ')');
```

```
gotoxy (10, 24);
writeln ('<ESC>  System Menu       <^C>  Accept and
                                          Continue');
gotoxy (1, 21);
write ('=> ');
ch := readkey;
while not (ch in ['1'..'5', #27, #03]) do
  repeat_input (ch);
case ch of
  '1' : Number_of_Layers;
  '2' : Layer_Architecture;
  '3' : Number_of_Neurons;
  '4' : Display_Network (2);
  '5' : Get_Corruption_Prob;
  #3  : begin
          if layers[2].weights = nil
            then Initialize_Weights
            else
              begin
                gotoxy (1, 21);
                clreol;
                write ('Initialize Weights?? - Y/N');
                ch := readkey;
                while not (ch in ['Y', 'y', 'N', 'n'])
                                                     do
                  repeat_input (ch);
                if ch in ['Y', 'y']
                  then Initialize_Weights;
              end;
          mat := false;
          if (arch_set <> num_layers) or (neuron_set <>
                                              num_layers)
            then
              begin
                gotoxy (10, 1);
                reverse;
                write ('Data Entered for Number of
                                Layers don''t Match!');
                gotoxy (10, 2);
                write ('Accept Values as Shown? Y/N');
                normvid;
                ch := readkey;
                if ch in ['N', 'n']
                  then
                    begin  {Start procedure over}
                    end
                  else
                    begin
                      arch_set := num_layers;{set flags}
```

```
                         neuron_set := num_layers;
                         for i := 1 to num_layers do
                           with layers[i] do
                             if arch = matrices
                               then mat := true
                             else
                               begin
                                 arch_row_length :=
                                                      neurons;
                                 arch_col_length := 1;
                               end;
                         if mat
                           then Define_Matrix (ch);
                         if ch = #27
                           then ch := #6   {Don't exit}
                           else Net_Architecture (flag) {^C
                                                      accepts}
                       end
                 end
               else
                 begin
                   arch_set := num_layers;  {set flags}
                   neuron_set := num_layers;
                   for i := 1 to num_layers do
                     with layers[i] do
                       if arch = matrices
                         then mat := true
                         else
                           begin
                             arch_row_length := neurons;
                             arch_col_length := 1;
                           end;
                   if mat
                     then Define_Matrix (ch);
                   if ch = #27
                     then ch := #6  {Don't exit}
                     else Net_Architecture (flag)   {^C
                                                  continues}
                 end;
           end;
    end;   {case}
    if Flag = #6    {Unwind out of menus to Main}
      then exit;
  until ch = chr (27);   {ESC Exit to System_Menu}
end;      {Layer_Menu}

{-----------------------------------------------------------}
```

```
Procedure Define_Displays;

begin    {Define_Displays}
end;     {Define_Displays}

{-----------------------------------------------------------}

Procedure Network_Utilities;

var
  ch : char;
  i  : integer;
  mat : boolean;   {Matrix architecture for this layer?}

begin     {Layer_Menu}
  repeat
    clrscr;
    gotoxy (20, 8);
    writeln ('   NETWORK UTILITIES MENU');
    gotoxy (20, 12);
    write ('1.  Select Input File');
    gotoxy (20, 13);
    write   ('2.  Net Save Option (');
    if net_description.save_state
      then write ('ON)')   {Save option on or off}
      else write ('OFF)');
    gotoxy (20, 14);
    write ('3.  Net Save Frequency (',
                     net_description.iterate, ')');
    gotoxy (20, 15);
    write ('4.  Print Net State Files');
    gotoxy (20, 16);
    write ('5.  Display Network');
    gotoxy (10, 24);
    writeln ('<ESC>  Program Operations');
    gotoxy (1, 21);
    write ('=> ');
    ch := readkey;
    while not (ch in ['1'..'5', #27]) do
      repeat_input (ch);
    case ch of
      '1' : Get_Input_File;          {Ioutil unit}
      '2' : Save_Net_Option;         {Ioutil Unit}
      '3' : Save_Freq_Option;        {Ioutil unit}
      '4' : Print_Files;             {Ioutil Unit}
      '5' : Display_Network (10);    {Grafstuf unit}
    end;  {case}
  until ch = #27;  {ESC exit to Program Operations}
end;  {Network_Utilities}
```

```
{----------------------------------------------------------}

{Run_Network establishes the high level Program Operations}
     {Menu}

Procedure Run_Network_Menu;

var
  ch   : char;
  flag : char;   {Set to return to main from final menu}

begin   {Run_Network}
  repeat
    flag := #0;
    clrscr;
    gotoxy (20, 8);
    writeln (' PROGRAM OPERATIONS');
    gotoxy (20, 10);
    writeln ('1.  Compile');
    gotoxy (20, 11);
    writeln ('2.  Print Files');
    gotoxy (20, 12);
    writeln ('3.  Define I/O Displays');
    gotoxy (20, 13);
    writeln ('4.  Display Network');
    gotoxy (20, 14);
    writeln ('5.  Select Input File');
    gotoxy (20, 15);
    writeln ('6.  Training Option');
    gotoxy (20, 16);
    writeln ('7.  Run Program');
    gotoxy (20, 17);
    write ('8.  Network Utilities');
    gotoxy (10, 24);
    writeln ('<ESC>  Exit to Main Menu');
    gotoxy (1, 21);
    write ('=> ');
    ch := readkey;
    while not (ch in ['1'..'8', #27]) do
      repeat_input (ch);
    case ch of
      '1' : begin
              Compile;
            end;
      '2' : Print_Files;
      '3' : Define_Displays;
      '4' : Display_Network (7); {grafstuf unit}
      '5' : Get_Input_File;      {ioutil unit}
      '6' : begin
              train := true;
              Training (Flag);    {Ops_Stuf Unit}
            end;
      '7' : begin
              train := false;
              Run_Program (Flag); {Ops_Suf unit}
            end;
      '8' : Network_Utilities;
    end;  {case}
  until ch = chr (27);    {ESC Exit to Main Menu}
end;    {Run_Network}

{----------------------------------------------------------}

{System_Menu establishes the top level System Menu }

Procedure System_Menu;

var
  ch   : char;
  flag : char;    {Set to return to main from final menu}
  S    : string;

begin     {System_Menu}
  repeat
    flag := #0;
    clrscr;
    gotoxy (20, 8);
    writeln (' NEURAL SYSTEM MENU');
    gotoxy (20, 10);
    writeln ('1.  New Network');
    gotoxy (20, 11);
    writeln ('2.  Load Network File From Disk');
    gotoxy (20, 12);
    writeln ('3.  Modify/Browse Network');
    gotoxy (20, 13);
    writeln ('4.  Display Network');
    gotoxy (20, 14);
    writeln ('5.  Save Defined Network');
    gotoxy (20, 15);
    writeln ('6.  Program Operations');
    gotoxy (10, 24);
    writeln ('<ESC>  Exit to DOS');
    gotoxy (1, 21);
    write ('=> ');
    ch := readkey;
    while not (ch in ['1', '2', '3', '4', '5', '6', chr
                                                 (27)]) do
      repeat_input (ch);
    case ch of
      '1' : begin
              Get_Network_Name (S);  {Mathutil unit}
              Net_description.net_name :=  S;
              Layer_Menu (flag);
            end;
      '2' : Load_File;              {ioutil unit}
      '3' : Layer_Menu (flag);
      '4' : Display_Network (1);
      '5' : Save_Network;           {ioutil unit}
      '6' : Run_Network_Menu;
      #27 : begin
              gotoxy (1, 21);
              reverse;
              write ('Do you really want to exit?');
              normvid;
              write (' Y/N ');
              ch := readkey;
              if ch in ['Y', 'y']
                then
                  begin
                    ch := #27;   {Exit to Dos}
                    clrscr;
                  end;
            end;
    end;  {case}
  until ch = chr (27);   {ESC Exit to Operating System}
  normvid;
  textcolor (yellow);
end;      {Neural_Menu}

{------------------------------------------------------------}

end.
```

APPENDIX D

Development System Weight Menu Unit Listing
(Unit Weights)

Appendix B through Appendix J constitute the complete development system program listing

```
{-----------------------------------------------------------}

{Unit weights manipulates the weight data and structures}
{via the menu system}

{------------------------------------------------------------}

unit weights;

interface

uses graph, crt, grafstuf, mathutil, nnglobal, ioutil;

Procedure Display_Weights (menu : integer; update, normal :
boolean);
Procedure Set_Random_Weights;
Procedure Set_Equal_Weights;

implementation   {weights}

{----------------------------------------------------------}

Procedure Set_Weight_Display (i, menu : integer; normal :
                               boolean);

const
  prev : packed array [1..2] of string[13] =
                       ('Set Weights', 'Equal Weights');

begin   {Set_Weight_Display}
  clrscr;
  gotoxy (30, 1);
  if normal
    then write ('WEIGHTS')
    else write ('SAVED_WEIGHTS');
  gotoxy (1, 50);
  write ('<ESC> ', Prev[menu], '  <SPACE> Rest of Layer
                                      <RTN> Next Layer');
  write ('  <F1> Display Net');
  gotoxy (15, 3);
  write ('Layer ');
  reverse;
  write (i - 1:2);
  normvid;
  write (' to Layer ');
  reverse;
  write (i:2);
  normvid;
  clreol;
  if Layers[i - 1].FFconnection_kind = partial
    then write ('  (Partially Connected)')
    else write ('  (Fully Connected)');
                                        {Center the Display!!!!}
  gotoxy (25, 6);
  write ('Layer ', i, ' Neurons  (', layers[i].neurons,
                                                            ')');
  gotoxy (1, 12);       {Fix for number of neurons}
  write ('Layer ', i - 1);
  gotoxy (1, 13);
  write ('Neurons');
  gotoxy (1, 15);
  write ('(', layers[i - 1].neurons, ')');
  gotoxy (16, 8);
 end;    {Set_Weight_Display}

----------------------------------------------------------}

Procedure Show_Weight_Sets (var ch : char; i, menu :
           integer; update, normal : boolean; w_ptr :
           weight_ptr);

var
  j, k, x, y        : integer;
  count1, count2    : integer;  {Tracks row/col blocks}
  count_r, count_c  : integer;  {Keeps track of rows and
                                 columns}

{-----------------------------------}

Function Get_Real_Number (lin : linetype; y : integer):
real;

var
  num, num1, j : real;
  int          : integer;
  x            : integer;
  neg          : boolean;

begin   {Get_real_number}
  neg := false;  {check for negative number}
  x := 0;
  int := 0;
  while x < y do  {y = length of string read in}
    begin
      x := x + 1;
      if lin[x] = '-'
        then
          begin
```

```
            neg := true;
            x := x + 1;
          end;
      while lin[x] in ['0'..'9'] do
        begin
          int := int * 10 + (ord (lin[x]) - ord ('0'));
          x := x + 1;
        end;
      if lin[x] = '.'
        then
          begin
            x := x + 1;
            num := int * 1.0;   {convert to real}
            if lin[x] in ['0'..'9']
              then
                begin
                  j := 1.0;
                  num1 := 0.0;
                  while (lin[x] in ['0'..'9']) and (x <= y)
                                                        do
                    begin
                      j := j * 0.1;  {set next decimal
                                                     place}
                      int := ord (lin[x]) - ord ('0');
                      num1 := num1 + (int * j);
                      x := x + 1;
                    end;
                  num := num + num1;
                end;
          end
        else
          num := int * 1.0;  {convert to real if no decimal}
    end;
  if neg
    then num := 0.0 - num;
  Get_Real_Number := num;   {return real number}
end;    {Get_real_number}

{----------------------------------------------}

Procedure Modify_Weights (c, r : integer);

var
  s                   : string [12];
  x, y, lgth, j       : integer;
  count_c, count_r    : integer;
  lgth1               : integer;   {length of previous entry}
  W                   : weight_ptr;
```

```
begin {Modify_Weights}
  count_c := c;
  count_r := r;
  gotoxy (1, 48);
  new (w);   {get new weight ptr}
  w^:= layers[i].weights^;
  reverse;
  write ('ENTER NEW WEIGHTS - <ESC> to cancel  <^C> to
                                                  Update');
  x := 14;
  y := 10;
  lgth1 := 0;
  repeat
    s := '';   {empty string}
    reverse;
    gotoxy (x, y);
    write (w^.value[c, r]:4:1);
    repeat
      ch := readkey;
    until ch in [#3, #13, #27, #0, '-', '.', '0'..'9']; {^C,
                                  RTN, ESC, Arrow, ., int}
    case ch of
      #0 : begin
             ch := readkey;
             case ch of
               #80 : begin  {down arrow}
                       normvid;
                       gotoxy (x, y);
                       write (w^.value[c, r]:4:1);
                       if (r = count_r + 34) or (r =
                                           w^.row_length)
                         then
                           begin
                             y := 10;
                             r := count_r;
                           end
                         else
                           begin
                             y := y + 1;
                             r := r + 1;
                           end;
                       gotoxy (x, y);
                       reverse;
                       write (w^.value[c, r]:4:1);
                       lgth1 := 0;
                     end;
               #77 : begin {right arrow}
                       normvid;
                       gotoxy (x, y);
```

```
              write (w^.value[c, r]:4:1);
              if (c = count_c + 9) or (c =
                                      w^.col_length)
                then
                  begin
                    x := 14;
                    c := count_c;
                  end
                else
                  begin
                    x := x + 5;
                    c := c + 1;
                  end;
              gotoxy (x, y);
              reverse;
              write (w^.value[c, r]:4:1);
              lgth1 := 0;
            end;
      #72 : begin  {up arrow}
              normvid;
              gotoxy (x, y);
              write (w^.value[c, r]:4:1);
              if r > count_r
                then
                  begin
                    y := y - 1;
                    r := r - 1;
                  end;
              gotoxy (x, y);
              reverse;
              write (w^.value[c, r]:4:1);
              lgth1 := 0;
            end;
      #75 : begin   {left arrow}
              normvid;
              gotoxy (x, y);
              write (w^.value[c, r]:4:1);
              if c > count_c
                then
                  begin
                    x := x - 5;
                    c := c - 1;
                  end;
              gotoxy (x, y);
              reverse;
              write (w^.value[c, r]:4:1);
              lgth1 := 0;
            end;
    end;  {case}
```

```
     end;
#13 : begin    {RTN}
        normvid;
        gotoxy (x, y);
        write (w^.value[c, r]:4:1);
        x := 14;
        c := count_c;
        if (r < (count_r + 34)) and (r <
                                  w^.row_length)
          then
            begin
              r := r + 1;
              y := y + 1;
            end;
        gotoxy (x, y);
        reverse;
        write (w^.value[c, r]:4:1);
        lgth1 := 0;
      end;
'.', '-', '0'..'9'  :  begin
                        s := '';
                        if ch = '.'
                          then
                            begin
                              s := '0.'; {Add leading 0
                                           for real number}
                              lgth := 2;
                            end
                          else
                            begin
                              s := ch;
                              lgth := 1;
                            end;
                        gotoxy (x, y);
                        write (ch);
                        x := x + 1;
                        repeat
                          repeat
                            ch := readkey;
                          until ch in [#13, #27, '.',
                                  '0'..'9']; {RTN, ESC..}
                          if not (ch in [#13, #27])
                                           {<RTN>, <ESC>}
                            then
                              begin
                                s := s+ch;
                                lgth := lgth + 1;
                                gotoxy (x, y);
                                write (ch);
```

```
                                  x := x + 1;
                                end;
                          until ch in [#13, #27];
                          if ch = #13  {RTN}
                            then
                              begin
                                x := x - length (s);
                                gotoxy (x, y);
                                for j := x to (x + lgth + 1)
                                                           do
                                  begin
                                    gotoxy (j, y);
                                    write (' ');
                                  end;
                                gotoxy (x, y);
                                write (s);
                                lgth1 := lgth;
                                for j := 1 to length (s) do
                                  lin[j] := s[j];  {transfer
                                                   to buffer}
                                w^.value [c, r] :=
                                    get_real_number (lin,
                                                length (s));
                              end;
                        end;
      #3 : begin                      {^C accepts new weights}
             layers[i].weights^ := w^;    {set to updated
                                                  weights}
             net_description.random_weights := false;
               {Search for weights set to 0.  If found set
                                               to partially}
               {connected.  To be Implemented}
           end;
    end;   {case}
  until ch in [#27, #3];    {ESC, ^C}
  dispose (w);
  normvid;
  gotoxy (1, 48);
  clreol;
end;  {Modify_Weights}

{------------------------------------------------}

begin   {Show_Weight_Sets}
  count1 := 0;
  count2 := 0;
  count_r := 0;
  count_c := 0;
  while count1 < layers[i].weights^.col_length do
```

```
begin
  while count2 < layers[i].weights^.row_length do
    begin
      j := count_c;
      Set_Weight_Display (i, menu, normal);
      while (j < count_c + 10) and (j <
                          w_ptr^.col_length) do
        begin
          j := j + 1;
          if j > 9
            then write (j, '    ')
            else write (j, '     ');
        end;
      y := 9;   {Seed for next loop}
      k := count_r;
      while (k < count_r + 35) and (k <
                          w_ptr^.row_length) do
        begin
          k := k + 1;
          x := 11;
          y := y + 1;
          gotoxy (x, y);
          write (k);
          x := 14;
          gotoxy (x, y);
          j := count_c;
        while (j < count_c + 10) and (j <
                          w_ptr^.col_length) do
            begin
              j := j + 1;
              write (w_ptr^.value[j, k]:4:1, ' ');
            end;
        end;   {while k...}
      count2 := count2 + 35;
      if (count2 < w_ptr^.row_length) or
         (count1 + 10 < w_ptr^.col_length)
        then
          begin
            gotoxy (72, 25);
            write ('(More)');
          end;
       if update
         then modify_weights (count_c + 1, count_r + 1);
       count_r := count_r + 35;
       ch := readkey;
       while not (ch in [#32, #13, #27, #0]) do  {SPACE,
                                             RTN, ESC, F1}
         repeat_input (ch);
```

```
          case ch of
            #0  : begin
                    ch := readkey;
                    if ch = #59   {F1}
                      then
                        begin
                           Display_Network (6);
                           textmode (lo(lastmode) +
                                     font8x8);  {Restore}
                                    {reduced size display}
                        end
                      else write (#7);
                  end;
            #13 : exit; {<RTN> Go to next layer display}
            #32 : begin {<SPACE> Finish the weight display}
                  end;
            #27 : exit;     {<ESC> to previous menu}
          end;   {case}
       end;        {while count2}
     count2 := 0;
     count1 := count1 + 10;
     count_c := count_c + 10;
     count_r := 0;
   end;   {while count1}
end;    {Show_Weight_Sets}

{---------------------------------------------------------}

Procedure Display_Weights (menu : integer; update, normal :
                                                    boolean);

var
  i         : integer;
  ch        : char;
  origmode  : word;
  w_ptr     : weight_ptr;

begin {Display_Weights}
  origmode := lastmode;
  textmode (lo(lastmode) + font8x8); {Reduce size for
                                     display of more data}
  normvid;
  new (w_ptr);
  repeat
    for i := 2 to Num_layers do
      begin
        if normal    {Display layer weights}
          then w_ptr^ := layers[i].weights^
          else w_ptr^ := layers[i].saved_weights^;
        Show_Weight_Sets (ch, i, menu, update, normal,
                          w_ptr);
        if ch = #27      {<ESC>}
          then
            begin
              textmode (origmode);
              directvideo := false;
              normvid;
              dispose (w_ptr);   {Destroy temporary pointer}
              exit;
            end;
      end;
  until ch = #27;          {<ESC>}
  dispose (w_ptr);         {Destroy temporary pointer}
  textmode (origmode);
  directvideo := false;
  normvid;
end;  {Display_Weights}

{--------------------------------------------------------}

Procedure Set_Random_Weights;

var
  i, j, k : integer;
  r, r1, c : real;
  ch : char;

begin   {Set_Random_Weights}
  randomize;  {Initialize random number generator to reals}
  for i := 2 to net_description.num_layers do
    begin
      for j := 1 to max_bodies do
        for k := 1 to max_bodies do
          begin
            r := (random (5) + random);
            c := random;
            if c < 0.20
              then r := 0.0 - r;  {negate}
            layers[i].weights^.value[k, j] := r;
          end;
      layers[i-1].FFConnection_kind := fully;
    end;
end;    {Set_Random_Weights}

{----------------------------------------------------------}
```

```
Procedure Set_Equal_Weights;

var
  i, j, k       : integer;
  weight        : real;
  ch            : char;
  result        : integer;

begin   {Set_Equal_Weights}
  repeat
    for i := 2 to num_layers do   {Number of layers selected
                                    or default value}
      begin
        clrscr;
        gotoxy (20, 8);
        writeln ('SET EQUAL WEIGHT FOR INDIVIDUAL LAYERS');
        gotoxy (10, 24);
        write ('<ESC> Set Weights Menu
                                        <RTN> Next Layer');
        gotoxy (30, 10);
        clreol;
        reverse;
        write ('Layer Number ', i);   {reverse video}
        normvid;
        gotoxy (20, 14);
        write ('1.  Set Weights');
        gotoxy (20, 15);
        write ('2.  Display Weights');
        gotoxy (1, 21);
        write ('=> ');
        ch := readkey;
        while not (ch in ['1', '2', #13, #27]) do
          repeat_input (ch);
        case ch of
          '1' : repeat
                  gotoxy (1, 22);
                  write ('Enter Weight Value');
                  gotoxy (4, 21);
                  clreol;
                  {$I-};
                  readln (weight);  {Fix for rtn or esc!!!!}
                  result := ioresult;
                  {$I+}
                  {Check for real number}
                  if result = 0     {A real number}
                    then
                      begin
                        for j := 1 to max_bodies do
                          for k := 1 to max_bodies do
```

```
                              layers[i].weights^.value[k, j]
                                                      := weight;
                          if weight = 0
                            then layers[i].FFConnection_kind
                                                := Not_Connected
                            else layers[i].FFConnection_kind
                                                       := fully;
                        end
                      else
                        begin
                          gotoxy (1, 1);
                          reverse;
                          write ('Illegal real number');
                          normvid;
                          gotoxy (1, 22);
                          clreol;
                          write ('<RTN> to Continue');
                          write (#7);
                          readln;
                          gotoxy (1, 1);
                          clreol;
                        end;
                  until (result = 0);
            '2' : Display_Weights (2, false, true);
            #13 : begin     {Skip this layer}
                  end;
            #27 : exit;    {exit to layer_menu}
          end;  {case}
        end;
    until ch = chr (27);  {ESC to exit to system menu}
end;    {Set_Equal_Weights}

{-------------------------------------------------------}
end.   {weights unit}
```

APPENDIX E

Development System Display Unit Listing (Unit Grafstuf)

Appendix B through Appendix J constitute the complete development system program listing

```
unit grafstuf;

interface

uses graph, crt, nnglobal, math, ioutil, mathutil;

Procedure Display_Network (i : integer);
Procedure Net_Display (i : integer);
Procedure Display_Net_Output (var ch : char; var semaphore :
integer);
Procedure Display_Memory (var ch : char; var semaphore :
integer);
Procedure Initialize_Graphics;
Procedure Modify_Connections (var ch : char; layer :
integer; str : string);

implementation

{----------------------------------------------------------}

Function Minimum (x, y : integer): integer;

begin  {Minimum}
  if x <= y
    then Minimum := x
    else Minimum := y;
end;   {Minimum}

{------------------------------------------------------}

Procedure LITTfont;   external;  {$L Litt.obj}
Procedure VGAdriver;  external;  {$L egavga.obj}

{-------------------------------------------------------}

Procedure Initialize_Graphics;   {Also in Unit Mathutil}

begin  {Initialize_Graphics}
  origmode := lastmode;
  clrscr;
  directvideo := false;
  if RegisterBGIfont (@LITTfont) < 0 then
    begin
      reverse;
      write ('Error In Registering font: ');
      normvid;
      write (graphErrorMsg (GraphResult));
      gotoxy (1, 21);
      write ('<RTN> to Continue');
      readln;
      clrscr;
      exit;  {Make flag to leave graphics also}
    end;
  if RegisterBGIdriver (@VGAdriver) < 0 then
    begin
      reverse;
      write ('Error In Registering driver: ');
      normvid;
      write (graphErrorMsg (GraphResult));
      gotoxy (1, 21);
      write ('<RTN> to Continue');
      readln;
      clrscr;
      exit;  {Make flag to leave graphics also}
    end;
  graphdriver := detect;
  initgraph (graphdriver, graphmode, '');
  errorcode := graphresult;
  if errorcode <> grok then
    begin
      Reverse;
      writeln ('Error In Initializing graphics');
      normvid;
      write ('Errorcode = ', errorcode);
      gotoxy (1, 21);
      write ('<RTN> to Continue');
      readln;
      clrscr;
      exit;  {Make flag to leave graphics also}
    end;
  TextColor (Yellow);
  SetColor (Yellow);
  MaxX := GetMaxX;    {Get Screen Resolution Values}
  MaxY := GetMaxY;
end;   {Initialize_Graphics}

{-----------------------------------------------------------}

Procedure FullPort;

begin  {FullPort}
  SetViewPort (0, 0, MaxX, MaxY, ClipOn);
end;   {FullPort}

{-----------------------------------------------------------}
```

```
{Writes title of the display sceen at the top of the
display}

Procedure Write_Header (Header : string; i : integer);

begin   {Write_Header}
  FullPort;
  SetTextStyle (SmallFont, HorizDir, 6);
  SetTextJustify (CenterText, TopText);
  OutTextXY (MaxX div 2, i, Header);
end;    {Write_Header}

{----------------------------------------------------------}

{Writes the command line at the bottom of the display}

Procedure Write_Command_Line (S : string; i : integer);

const
  Previous_Menu : packed array [0..12] of string [21] = ('',
                 'Main Menu', 'Layer Data', 'Define Matrix',
                 'Connectivity Matrix','Partial
                  Connectivity', 'Display Weights',
                 'Operations', 'Memory', 'Output', 'Network
                  Utilities', 'Modify Neurons', 'Modify
                  Connections');

var
  msg : string;

begin   {Write_Command_Line}
  SetTextStyle (SmallFont, HorizDir, 5);
  SetTextStyle (DefaultFont, HorizDir, 1);
  SetTextJustify (LeftText, BottomText);
  OutTextXY (0, MaxY - 10, S+Previous_Menu[i]);
end;    {Write_Command_Line}

{------------------------------------------------------------}

{Displays the graphic representation of the network
specified by the user}

Procedure Net_Display (i : integer);

type
  {'w' & 'x' start coords, 'y' & 'z' finish coords for
          arrows}
  coord_type = packed array [1..max_layers, 'w'..'z'] of
                                                   integer;
  coord_ptr   = ^coord_type;

var
  center_width  : integer;
  center_height : integer;
  width         : integer;
  height        : integer;
  x, y          : integer;
  j             : integer;
  Border        : integer;
  ViewInfo      : ViewPortType;
  Msg           : String;
  ch            : char;
  s1, s2, s3    : string[8];
  Arrow_coords  : coord_ptr;

{------------------------------------------}

{Draws the arrows between the layers indicating connections
between the layers}

Procedure Draw_FF_Arrows (j : integer);

var
  width1 : integer;

begin   {Draw_FF_Arrows}
  line (arrow_coords^[j, 'w'], arrow_coords^[j, 'x'],
        arrow_coords^[j, 'y'], arrow_coords^[j, 'z']);
  line (arrow_coords^[j, 'y'], arrow_coords^[j, 'z'],
        arrow_coords^[j, 'y']-6, arrow_coords^[j, 'z']- 4);
  line (arrow_coords^[j, 'y'], arrow_coords^[j, 'z'],
        arrow_coords^[j, 'y']-6, arrow_coords^[j, 'z']+ 4);
  if (net_description.net_arch = recurrent) and (j =
                                            num_layers - 1)
    then        {Draw Recurrent arrow back to input layer}
      begin
        line (arrow_coords^[j, 'y']+ width, arrow_coords^[j,
                                                   'z']+ 30,
              arrow_coords^[j, 'y'] + width + 20,
                                    arrow_coords^[j, 'z']+ 30);
        line (arrow_coords^[j, 'y'] + width + 20,
                                     arrow_coords^[j, 'z']+ 30,
              arrow_coords^[j, 'y'] + width + 20,
            arrow_coords^[j, 'x']- (8 *  TextHeight ('M')));
        if layers [1].arch = matrices
          then width1 := center_width        {matrix width}
          else width1 := center_width div 2; {Line
                                         Architecture width}
          line (arrow_coords^[j, 'y'] + width + 20,
                arrow_coords^[j, 'x']- (8 * TextHeight ('M')),
                arrow_coords^[1, 'w']- width1 - 20,
               arrow_coords^[j, 'x']- (8 * Textheight ('M')));
          line (arrow_coords^[1, 'w']- width1 - 20,
                arrow_coords^[j, 'x']- (8 * Textheight ('M')),
                arrow_coords^[1, 'w']- width1 - 20,
                arrow_coords^[1, 'x']+ 10);
          line (arrow_coords^[1, 'w']- width1 - 20,
                arrow_coords^[1, 'x']+ 10,
                arrow_coords^[1, 'w']- width1,
                arrow_coords^[1, 'x']+ 10);
          line (arrow_coords^[1, 'w']- width1,
                arrow_coords^[1, 'x']+ 10,
                arrow_coords^[1, 'w']- width1 - 6,
                arrow_coords^[1, 'x']+ 6);
          line (arrow_coords^[1, 'w']- width1,
                arrow_coords^[1, 'x']+ 10,
                arrow_coords^[1, 'w']- width1- 6,
                arrow_coords^[1, 'x']+ 16);
      end;
end;    {Draw_FF_Arrows}

{-----------------------------------------------}

{Draws the layer representations of each layer in the
network}

Procedure Draw_Layer_Box (x, y : integer);

begin   {Draw_Layer_Box}
  SetFillStyle (SolidFill, LightBlue);
  Bar (x, y, x + width, y + height);
  SetColor (lightBlue);
  Rectangle (x, y, x + width, y + height);
  SetColor (White);
end;    {Draw_Layer_Box}

{------------------------------------------------}

begin    {Net_Display}
  FullPort;
  ClearDevice;
  New (Arrow_Coords);
  Write_Header ('CURRENT NETWORK DEFINITION', 45);
  SetColor (white);
  SetTextStyle (SmallFont, HorizDir, 5);
  OutTextXY (MaxX Div 2, 70,
                         '('+net_description.net_name+')');
```

```
SetColor (yellow);
Write_Command_Line ('<ESC>  ', i);
SetTextJustify (CenterText, CenterText);
GetViewSettings (ViewInfo);
with ViewInfo do
  begin
    center_width := (X2 Div Num_Layers) Div 2;
    height := (Y2 Div 3) - 10;
  end;
y := height + 10;
x := 0;
for j := 1 to Num_layers do
  begin
    if layers[j].arch = matrices
      then width := center_width           {Matrix Width}
      else width := center_width Div 2;    {Line width}
    Border := x;
    x := x + center_width - (width div 2);
    Draw_Layer_Box (x, y);
    if j = 1
      then msg := 'Input'    {Write layer type over box}
      else if layers[j].out_layer
        then msg := 'Output'
      else msg := 'Hidden';
    OutTextXY (x + (width Div 2), y - TextHeight ('H') -
                                                 4, Msg);
    if layers[j].arch = matrices    {Write layer size
                                              inside box}
      then
        begin
          str (layers[j].arch_row_length, s1);
          str (layers[j].arch_col_length, s2);
          msg := concat (s1, ' Rows');
          OutTextXY ((Border + center_width), (y + (height
                                        Div 2) -15), Msg);
          OutTextXY ((Border + center_width), (y + (height
                                            Div 2)), 'X');
          Msg := concat(s2, ' Cols');
          OutTextXY ((Border + center_width), (y + (height
                                        Div 2) +15), Msg);
        end
      else
        begin
         str (layers[j].neurons, s1);
         msg := s1;
          OutTextXY ((Border + center_width), (y + (height
                                            Div 2)), Msg);
        end;
    str (j, s1);              {Write Layer # under box}
```

```
      msg := concat ('(', s1, ')');
      OutTextXY (x + (width Div 2), y + Height + TextHeight
                                          ('H') + 4, msg);
      str (layers[j].neural_set.threshold[1]:2:2, s1);
      msg := concat ('(th = ', s1, ')');
      if j > 1     {Write layer thrshld under box}
        then OutTextXY (x + (width Div 2), y + Height +
                      2*(TextHeight ('H') + 4), msg);
      arrow_coords^[j, 'w'] := x + width;
      arrow_coords^[j, 'x'] := y + 4*TextHeight ('M');
      if j > 1
        then
          begin    {Finish Arrow Coords for previous layer}
            arrow_coords^[j-1, 'y'] := x;
            arrow_coords^[j-1, 'z'] := y + 4*TextHeight
                                                  ('M');
          end;
      x := x + (width div 2) + center_width;
    end;
  str (net_description.threshold:2:2, s1);
  Msg := concat ('Activation  -  ', Activation_kind[ord
              (net_description.activation)], '   (thrshld
                                          = ', s1, ')');
  OutTextXY (MaxX Div 2, ((2 * Height) + 70), msg);
  if (net_description.learning = delta) or
                        (net_description.learning = backp)
    then
      begin
        str (net_description.factor:4:2, s1);
        str (net_description.momentum:2:2, s2);
        msg := concat ('Learning  -  ', Learning_kind[ord
                    (net_description.learning)]+' (n =
                     '+s1+')', '  (α = ', s2, ')');
      end
    else msg := concat ('Learning  -  ', Learning_kind[ord
                    (net_description.learning)]);
  OutTextXY (MaxX Div 2, ((2 * Height) + 90), msg);
  str (net_description.factor1:4:3, s1);
  str (net_description.factor2:4:4, s2);
  str (net_description.factor3:4:4, s3);
  msg := concat ('(t = ', s1, ')', '  (s = ', s2, ')',
                                   '  (p = ', s3, ')');
  OutTextXY (MaxX Div 2, ((2 * Height) + 110), msg);
  if length (infilename) > 0
    then
      begin
        msg := ('Input File : '+infilename);
        OutTextXY (MaxX Div 2, ((2 * Height) + 130), msg);
      end;
  for j := 1 to (num_layers - 1) do    {Draw Arrows}
    begin
      if layers[j].FFConnection_kind = fully
        then Msg := '(Full)'
      else if layers[j].FFConnection_kind = Partial
        then Msg := '(Partial)';
      Draw_FF_Arrows (j);
      OutTextXY (arrow_coords^ [j, 'w'] + ((arrow_coords^[j,
               'y'] - arrow_coords^[j, 'w']) Div 2),
               (arrow_coords^[j, 'x'] - TextHeight ('M') -
                                                   4), Msg);
     end;
  repeat
    ch := readkey;
  until Ch = #27;
  dispose (arrow_coords);
end;     {Net_Display}

{-----------------------------------------------------------}

{Sets graphmode (graphics) and calls net_display}

Procedure Display_Network (i : integer);

begin    {Display_Network}
  SetGraphMode (graphmode);  {Return to graphics mode}
  Net_Display (i);
  RestoreCrtMode;  {back to text}
end;     {Display_Network}

{-----------------------------------------------------------}

{Draws the square neuron representation and the memory
square representation for the I/O display}

Procedure Draw_Memory_Box (x, y, width, height : integer;
                           Color, Bord : word);

var
  center_width  : integer;
  center_height : integer;
  j             : integer;
  Border        : integer;
  ViewInfo      : ViewPortType;

begin   {Draw_Memory_Box}
  SetFillStyle (SolidFill, color);
  Bar (x, y, x + width, y + height);
  SetColor (Bord);
  Rectangle (x, y, x + width, y + height);
  SetColor (yellow);
end;    {Draw_Memory_Box}

{-----------------------------------------------------------}

{Provides the shell menu for removal and restoration of
individual neurons within each layer.}

Procedure Modify_Neurons (var ch : char);

  var
    layer, int, i, j : integer;
    saved            : boolean;
    prob             : real;

begin    { Modify_Neurons}
  layer := 2;
  prob := 0.0;
  RestoreCrtMode;
  repeat
    clrscr;
    gotoxy (20, 8);
    write ('Modify Neurons');
    gotoxy (22, 9);
    write ('(Layer ', layer, ')');
    gotoxy (20, 11);
    write ('1.  Remove Neuron');
    gotoxy (20, 12);
    write ('2.  Restore Neuron');
    gotoxy (20, 13);
    write ('3.  Restore Trained Configuration');
    gotoxy (20, 14);
    write ('4.  Random Neuron Removal (Prob = ', prob:2:2,
                                                          ')');
    gotoxy (20, 15);
    write ('5.  Save Weight State');
    gotoxy (20, 16);
    write ('6.  Display Network');
    gotoxy (1, 24);
    write ('<ESC> Run Network                     <RTN> for
                                            next layer');
    gotoxy (1, 21);
    clreol;
    write ('=> ');
    repeat
      ch := readkey;
    until ch in ['1'..'6', #27, #13];   {nums, <ESC>, <RTN>}
```

```
case ch of
  '1' : begin                        {remove neuron}
          {alert if no saved_weights}
          repeat
            clrscr;
            gotoxy (1, 20);
            write ('Select Neuron for Deletion (Layer ',
                                              layer, ')');
            gotoxy (1, 21);
            write ('=> ');
            gotoxy (1, 24);
            write ('Enter Number or <RTN> for No
                                                  Entry');
            gotoxy (4, 21);   {place cursor after =>}
            clreol;
            getline;                   {mathutil unit}
            if (ColsOnLine > 0) and (Nextchar in
                                              ['0'..'9'])
              then
                begin
                  process_number (int); {mathutil unit}
                  if int > layers[layer].neurons
                    then
                      begin
                        gotoxy (10, 2);
                        reverse;
                        write ('Exceeds Layer Neurons
                        (', layers[layer].neurons, ')');
                        normvid;
                        gotoxy (1, 21);
                        clreol;         {Erase =>}
                        gotoxy (1, 24);
                        clreol;
                        gotoxy (1, 23);
                        write ('<RTN> to continue');
                        repeat
                          ch := readkey
                        until ch = #13;
                        gotoxy (1, 23);
                        clreol;         {Erase <RTN>..}
                        gotoxy (10, 2);   {Clear error
                                                message}
                        clreol;
                        gotoxy (1, 21);
                        write ('=> ');
                      end
                    else
                      begin
                        if layer <
```

```
                              net_description.num_layers
                            then
                              for i := 1 to layers[layer +
                                             1].neurons do
                               layers[layer + 1].weights^.
                                     value[i, int] := 0.0;
                                    {all connections := 0}
                              for i := 1 to layers[layer -
                                             1].neurons do
                                layers[layer].weights^.
                                     value[int, i] := 0.0;
                                       {all input := 0}
                        end;
                  end;
              gotoxy (1, 24);
              clreol;
              write ('<^C> To Accept        <RTN> to
                                 Select Another Neuron');
              ch := readkey;
            until ch = #3;          {<^C> to exit}
            clrscr;
          end;
   '2' : begin                     {restore Neuron}
            if ((layer = net_description.num_layers) and
                (layers[layer].saved_weights = nil)) or
               ((layer < net_description.num_layers) and
                ((layers[layer].saved_weights = nil) or
                 (layers[layer + 1].saved_weights = nil)))
              then
                begin
                  clrscr;
                  gotoxy (1, 1);
                  reverse;
                  write ('No saved state exists! <RTN> to
                                              Continue');
                  repeat
                    ch := readkey;
                  until ch = #13;
                end
              else
                repeat
                  clrscr;
                  gotoxy (1, 20);
                  write ('Select Neuron for Resoration
                                    (Layer ', layer, ')');
                  gotoxy (1, 24);
                  write ('Enter Number or <RTN> for No
                                                Entry');
                  gotoxy (1, 21);
```

```
write ('=> ');
clreol;
getline;              {mathutil unit}
if (ColsOnLine > 0) and (Nextchar in
                              ['0'..'9'])
  then
    begin
      process_number (int);   {mathutil
                                  unit}
      if int > layers[layer].neurons
        then
          begin
            gotoxy (10, 2);
            reverse;
            write ('Exceeds Layer
             Neurons (', layers[layer].
                        neurons, ')');
            normvid;
            gotoxy (1, 21);
            clreol;          {Erase =>}
            gotoxy (1, 24);
            clreol;
            gotoxy (1, 23);
            write ('<RTN> to continue');
            repeat
              ch := readkey;
            until ch = #13;
            gotoxy (1, 23);
            clreol;    {Erase <RTN>..}
            gotoxy (10, 2);   {Clear
                        error message}
            clreol;
            gotoxy (1, 21);
            write ('=> ');
          end
        else
          begin
            if layer <
             net_description.num_layers
              then
                for i := 1 to
                     layers[layer + 1].
                              neurons do
                  layers[layer + 1].
                  weights^.value[i, int]
                   := layers[layer + 1].
                       saved_weights^.
                          value[i, int];
                for i := 1 to layers
```

```
                                   [layer - 1].neurons do
                                  layers[layer].weights^.
                                    value[int, i] :=
                                    layers[layer].
                                     saved_weights^.
                                           value[int, i];
                          end;
                    end;
                gotoxy (1, 24);
                clreol;
                write ('<^C> To Accept     RTN> to
                                  Select Another Neuron');
                ch := readkey;
              until ch = #3;   {<^C> to exit}
          clrscr;
        end;
 '3' : begin          {restore trained configuration}
          saved := true;
          for i := 2 to net_description.num_layers do
            if layers[i].saved_weights <> nil
              then layers[i].weights^ :=
                layers[i].saved_weights^
              else saved := false;    {single layer with
                                            no S_W poss?}
          if not saved
            then
              begin
                clrscr;
                gotoxy (1, 1);
                reverse;
                write ('At least one saved state
                                       doesn''t exist!');
                normvid;
                gotoxy (1, 2);
                write ('Others May Have Been Saved.
                                     <RTN> to Continue');
                repeat
                  ch := readkey;
                until ch = #13;
              end
           else
             begin
               gotoxy (1, 1);
               reverse;
               write ('Weights Restored!  <RTN>');
               repeat
                 ch := readkey;
               until ch = #13;
               normvid;
            ch := '>'
          end;
      end;
'4' : begin                  {random neuron removal}
        get_removal_probability (prob);  {math unit}
        remove_random_neurons (prob, layer);  {math
                                               unit}
        gotoxy (1, 1);
        reverse;
        write ('Neurons Removed! <RTN>');
        repeat
          ch := readkey;
        until ch = #13;
        normvid;
        ch := '<';
      end;
'5' : begin                  {save weight state}
        clrscr;
        gotoxy (1, 1);
        reverse;
        write ('Previous saved state will be destroyed
                                  - Continue? Y/N');
        repeat
          ch := readkey
        until ch in ['Y', 'y', 'n', 'N'];
        normvid;
        if ch in ['Y', 'y']
          then
            begin
              for i := 2 to net_description.num_layers
                                                   do
                begin
                  if layers[i].saved_weights = nil
                    then new(layers[i].saved_weights);
                  layers[i].saved_weights^ :=
                                 layers[i].weights^;
                end;
              clrscr;
              gotoxy (1, 1);
              reverse;
              write ('Weights Saved! <RTN>');
              repeat
                ch := readkey;
              until ch = #13;
              normvid;
            end
          else
            begin
              clrscr;
```

```
                    gotoxy (1, 1);
                    reverse;
                    write ('Weights Not Saved! <RTN>');
                    repeat
                      ch := readkey;
                    until ch = #13;
                    normvid;
                  end;
            end;
      '6' : Display_Network (11);  {display network}
      #13 : begin                  {<RTN>}
              if layer = net_description.num_layers
                then layer := 2
                else layer := layer + 1;
            end;
    end;               {case}
  until ch in [#27];   {<ESC>}
  SetGraphMode (graphmode);
end;     { Modify_Neurons}

{-----------------------------------------------------------}

{Include file for Procedure Modify_Connections.  File too}
{large for Pascal editor.  Provides menu for modifications}
{to connections, removal, restoration, etc.}

{$I connect.pas}

{-----------------------------------------------------------}

{Main procedure for displaying the network computations,
input, output, memory state, etc.}

Procedure Display_Net_Output (var ch : char; var semaphore :
                                integer);

var
  comline                 : string;
  viewport                : ViewPortType;
  ViewPort1, ViewPort2    : ViewPortType;
  ColSpace, RowSpace      : integer;
  ColSpace1, RowSpace1    : integer;
  x, y, x3, y3            : integer;  {Square and Number
                                             coordinates}
  i, j, k, z, it          : integer;  {Counters}
  R, C, A                 : integer;  {For finding minimum
                                                  number}
  outlayer                : integer;
  s                       : string[8];
  hue                       : byte;
  converge                  : boolean;

{-------------------------------------------}

{Enables the manual modification of input values to the
network.  Also allows for the random corruption of the input
values}

Procedure Modify_Input (viewport : ViewPortType; var ch :
                        char);

var
  x, y   : integer;
  x4, y4 : integer;
  k, j   : integer;

begin    {Modify_Input}
  ClearViewPort;
  comline := '<ESC> Run Mode   <^C> Accept/Calculate   <RTN>
                                                  Change   <';
  comline := comline+#27#24#25#26+'> Next Neuron';
  with viewport do
    OutTextXY (20, (y2-y1) Div 2, comline);
  with ViewPort1 do
    SetViewPort (x1, y1, x2, y2, true);
  SetTextStyle (SmallFont, HorizDir, 4);
  SetTextJustify (CenterText, CenterText);
  repeat
    with viewport1 do
      begin
        if layers[1].arch = lines
          then
            begin
              x := x1 + ((x2-x1) Div 2);
              if layers[1].neurons > 2
                then y := TextHeight ('8') + RowSpace Div 2
                else if layers[1].neurons = 1
                  then y := (y2-y1) Div 2
                  else y := (y2-y1) Div 2 - (RowSpace Div
                                                        2);
              SetColor (Yellow);
              OutTextXY (x, y, 'X');
              i := 1;
              repeat
                repeat
                  ch := readkey;
                until ch in [#27, #13, #3, #0];
                case ch of
```

```
#0 : begin
       ch := readkey;
       case ch of
         #80 : begin   {Down Arrow}
                 if layers[1].neural_set.binary_value[i] = 1
                   then SetColor (blue)
                   else SetColor (LightGray);
                 OutTextXY (X, Y, 'X');  {Erase 'X'}
                 SetColor (Yellow);
                 if i < layers[1].neurons
                   then
                     begin
                       y := y + RowSpace;     {next square}
                       i := i + 1;
                       OutTextXY (x, y, 'X');
                     end
                   else  {Was last neuron in line layer}
                     begin
                       i := 1;
                       y := y -
                             (layers[1].neurons-1)*RowSpace;
                                              {reset to first}
                       OutTextXY (x, y, 'X');
                     end;
               end;
         #72 : begin      {Up Arrow}
                 if layers[1].neural_set.binary_value[i] = 1
                   then SetColor (blue)
                   else SetColor (LightGray);
                 OutTextXY (X, Y, 'X');          {Erase 'X'}
                 SetColor (Yellow);
                 if i > 1  {Not first neuron}
                   then
                     begin
                       y := y - RowSpace;     {next square}
                       i := i - 1;
                       OutTextXY (x, y, 'X');
                     end
                   else  {Was first neuron in line layer}
                     begin
                       i := Layers[1].neurons;
                       y := y +
                             (layers[1].neurons-1)*RowSpace;
                                             {reset to last}
                       OutTextXY (x, y, 'X');
                     end;
               end;
         end;  {case}
      end;
```

```
    #13 : begin    {<RTN>}
            if layers[1].neural_set.binary_value[i] = 1
              then    {Was blue and set}
                begin
                  hue := lightgray;
                  layers[1].neural_set.binary_value[i] := 0;
                end
              else   {Was lightgray and not set}
                begin
                  hue := blue;
                  layers[1].neural_set.binary_value[i] := 1;
                end;
            x3 := x - (ColSpace Div 2);
            y3 := y - (RowSpace Div 2);
            Draw_Memory_Box (x3, y3, ColSpace, RowSpace,
                                                 hue, white);
            SetColor (Yellow);
            OutTextXY (x, y, 'X');  {Replace 'X'}
          end;
    #3  : begin   {Return to Modify_Parameters}
            if layers[1].neural_set.binary_value[i] = 1
              then SetColor (Blue)
              else SetColor (LightGray);
            OutTextXY (x, y, 'X');   {Erase 'X'}
            SetColor (yellow);
            case net_description.training of
              supervised   : calculate_output; {math unit}
              unsupervised : if net_description.learning <>
                             Hopfield then calculate_output;
              zip          : calculate_output  {math unit}
            end;   {case}
          end
   end;  {case}
 until ch in [#27, #3]; {ESC to exit, ^C to compute}
end
  else    {Layer Architecture = matrices}
    begin
      x := x1 + ((x2-x1) Div 2);
      if layers[1].arch_col_length Mod 2 = 0  {Even # of
                                                    Cols}
        then x := x - ((layers[1].arch_col_length Div
                       2)*ColSpace) + ColSpace Div 2
        else x := x - ((layers[1].arch_col_length Div
                       2)*ColSpace);
      y := y1 + ((y2-y1) Div 2);       {Center of viewport}
      if layers[1].arch_row_length Mod 2 = 0  {Even # of
                                                     Rows}
        then y := y - ((layers[1].arch_row_length Div
                       2)*RowSpace) + RowSpace Div 2
  else y := y - ((layers[1].arch_row_length Div
                  2)*RowSpace);
y4 := y;     {Save y value for future use}
x4 := x;     {Save x for future use}
SetColor (Yellow);
OutTextXY (x, y, 'X');
i := 1;      {i counts neurons in layer}
j := 1;      {j counts columns in matrix architecture}
k := 1;      {k counts rows in matrix architecture}
repeat
  repeat
    ch := readkey;
  until ch in [#27, #13, #3, #0];
  case ch of
    #0 : begin
           ch := readkey;
           case ch of
             #80 : begin        {down arrow}
                     if layers[1].neural_set.
                                 binary_value[i] = 1
                       then SetColor (blue)
                       else SetColor (LightGray);
                     OutTextXY (X, Y, 'X');  {Erase
                                             the 'X'}
                     SetColor (Yellow);
                     if k < layers[1].arch_row_length
                       then
                         begin
                           y := y + RowSpace;  {Next
                                                square}
                           i := i + 1;
                           k := k + 1;
                           OutTextXY (x, y, 'X');
                         end
                       else   {Last neuron in row}
                         begin
                           k := 1;
                           i := i - (layers[1].
                                 arch_row_length-1);
                           y := y4;
                           OutTextXY (x, y, 'X');
                         end;
                   end;
             #72 : begin   {Up Arrow}
                     if layers[1].neural_set.
                                 binary_value[i] = 1
                       then SetColor (blue)
                       else SetColor (LightGray);
                     OutTextXY (x, y, 'X'); {Erase'X'}
```

```
         SetColor (Yellow);
         if k > 1
           then
             begin
               i := i - 1;
               k := k - 1;
               y := y - RowSpace;
               OutTextXY (x, y, 'X');
             end
           else  {First neuron in column}
             begin
               i := i + (layers[1].
                       arch_row_length-1);
               k := layers[1].
                          arch_row_length;
               y := y + (RowSpace *
           (layers[1].arch_row_length-1));
               OutTextXY (x, y, 'X');
             end
       end;
#75 : begin   {Left Arrow}
         if layers[1].neural_set.
                      binary_value[i] = 1
           then SetColor (blue)
           else SetColor (LightGray);
         OutTextXY (x, y, 'X');  {Erase
                                 the 'X'}
         SetColor (Yellow);
         if j > 1
           then
             begin
               j := j - 1;
               i := i - layers[1].
                      arch_row_length;
               x := x - ColSpace;
               OutTextXY (x, y, 'X');
             end
           else  {First column in matrix}
             begin
               j := layers[1].
                         arch_col_length;
               i := i+((layers[1].
                     arch_col_length-1)*
              layers[1].arch_row_length);
               x := x + (ColSpace *
                    (layers[1].
                      arch_col_length-1));
               OutTextXY (x, y, 'X');
            end
```

```
                      end;
    #77 : begin   {Right Arrow}
                 if layers[1].neural_set.binary_value[i] = 1
                   then SetColor (blue)
                   else SetColor (LightGray);
                 OutTextXY (x, y, 'X');  {Erase the 'X'}
                 SetColor (Yellow);
                 if j < layers[1].arch_col_length
                   then
                     begin
                       j := j + 1;
                       i := i + layers[1].arch_row_length;
                       x := x + ColSpace;
                       OutTextXY (x, y, 'X');
                     end
                   else  {Last column in matrix}
                     begin
                       j := 1;
                       i := i-((layers[1].arch_col_length
                                -1)*layers[1].
                                           arch_row_length);
                        x := x4;
                        OutTextXY (x, y, 'X');
                     end;
              end;
          end;     {case}
       end;
#13 : begin    {<RTN>}
          if layers[1].neural_set.binary_value[i] = 1
            then    {Was blue and set}
              begin
                hue := lightgray;
                layers[1].neural_set.binary_value[i] := 0;
                layers[1].neural_set.real_value[i] :=
                  net_description.zero_value;
              end
            else   {Was lightgray and not set}
              begin
                hue := blue;
                layers[1].neural_set.binary_value[i] := 1;
                layers[1].neural_set.real_value[i] :=
                  net_description.one_value;
              end;
          x3 := x - (ColSpace Div 2);
          y3 := y - (RowSpace Div 2);
          Draw_Memory_Box (x3, y3, ColSpace, RowSpace,
                                   hue, white);
          SetColor (Yellow);
          OutTextXY (x, y, 'X');  {Replace 'X'}
          end;
      #3  : begin   {Return to Modify_Parameters and
                                         calculate net}
            if layers[1].neural_set.binary_value[i] = 1
              then SetColor (Blue)
              else SetColor (LightGray);
            OutTextXY (x, y, 'X');   {Erase 'X'}
            SetColor (yellow);
            case net_description.training of
              supervised    : calculate_output; {math unit}
              unsupervised : if net_description.learning
                               <> Hopfield
                                 then calculate_output;
              zip           : calculate_output  {math unit}
            end;   {case}
          end;
      end;  {case}
    until ch in [#27, #3];
   end;
  end;
 until ch in [#27, #3];
end;     {Modify_Input}

{-----------------------------------------}

{Main procedure used to call procedures for changing
neurons, connections, varying input, etc.}

Procedure Modify_Parameters;

var
  ch : char;
  viewport : ViewPortType;

begin   {Modify_Parameters}
  repeat
    SetTextStyle (DefaultFont, HorizDir, 1);
    SetTextJustify (LeftText, BottomText);
    SetViewPort (0, MaxY-TextHeight ('M')-11, MaxX, MaxY,
                 true);
    ClearViewPort;
    comline := '<ESC> Accept <F1> Mod Input <F2> Mod
                                         Neurons';  ;
    comline := comline+'<F3> Mod Connections <F4> Corrupt';
    GetViewSettings (Viewport);
    with viewport do
      OutTextXY (20, (y2-y1) Div 2, comline);
    repeat
      ch := readkey;
    until ch in [#0, #27];
    case ch of
      #0  : begin
              ch := readkey;
              case ch of
                #59 : Modify_Input (viewport, ch);      {F1}
                #60 : Modify_Neurons (ch);              {F2}
                #61 : begin                             {F3}
                        RestoreCrtMode;
                        Modify_Connections (ch, 1, 'Run
                                              Network');
                        SetGraphMode (graphmode);
                      end;
                #62 : begin                             {F4}
                        Corrupt_input (net_description.
                                   factor3);  {Math unit}
                        case net_description.training of
                          supervised    : calculate_output;
                                              {math unit}
                          unsupervised : if
                                 net_description.learning <>
                                   Hopfield
                                     then calculate_output;
                                              {math unit}
                          zip           : calculate_output;
                                              {math unit}
                        end;   {case}
                      end;
              end;  {case}
            end;
    end;   {case}
  until ch in [#27, #62, #3];          {<ESC>, <F4>, <^C>}
end;    {Modify_Parameters}

{-------------------------------------------}

begin   {Display_Net_Output}
  repeat
    FullPort;
    SetColor (yellow);
    ClearDevice;
    Write_Header ('INPUT                             OUTPUT',
                                                         5);

    SetColor (white);
    if train
      then write_header ('Training Mode', 4)
      else write_header ('Run Mode', 4);
    SetColor (yellow);
```

```
Comline := '<ESC> Quit  <SPACE> Mod Params  <RTN> Next
                                               Input';
Comline := concat (ComLine, '  <F1> Network   <^C>
                                             Memory');
Write_Command_Line (ComLine, 0);
SetViewPort (0, 20+(TextHeight('M')), MaxX,
             MaxY-20-(TextHeight('M')), ClipOn);
GetViewSettings (viewport);
SetColor (LightGray);
SetTextStyle (SmallFont, HorizDir, 5);
with ViewPort do
  begin                   {Draw Border around screen}
    line (0, 0, 0, y2-28);
    line (0, 0, x2, 0);
    line ((x2-x1) Div 2, 0, (x2-x1) Div 2, y2-28);
    line (0, y2-28, x2, y2-28);
    line (x2, y2-28, x2, 0);
    SetViewPort (x1+1, y1+1, (x2-x1) Div 2 - 1, y2-1,
                                               true);
    GetViewSettings (ViewPort1);
    SetViewPort ((x2-x1) Div 2 + 1, y1+1, x2 - 1, y2-1,
                                               true);
    GetViewSettings (ViewPort2);
  end;
SetTextStyle (SmallFont, HorizDir, 4);
SetTextJustify (CenterText, CenterText);
with layers[1] do                         {Draw Input}
  begin
    if arch = lines then     {Line Layer Architecture}
      begin
        with viewport1 do
          begin
            SetViewPort (x1, y1, x2, y2, true);
            RowSpace :=
           ((y2-TextHeight('8'))-(y1+TextHeight('8')))
             Div layers[1].neurons;
            if (neurons = 1) or (neurons = 2)
              then RowSpace := RowSpace Div 3;
                                  {Reduce size for one}
            ColSpace := RowSpace;  {Will display a
                                              square}
            x := x1 + ((x2-x1) Div 2);
            x := x - (ColSpace Div 2);    {first square
                                         x coordinate}
            if neurons > 2
              then y := TextHeight('8')      {First
                                 square y coordinate}
              else
                if neurons = 1
              then y := ((y2 - y1) Div 2) - RowSpace
                                                 Div 2
              else y := ((y2 - y1) Div 2) -
                            RowSpace;
          end;
        x3 := x - 3*TextWidth('15');  {Number position}
        y3 := y + (RowSpace Div 2);
        SetColor (white);
        for i := 1 to neurons do   {Write neuron numbers
                                             vertically}
          begin
            str (i, s);
            OutTextXY (x3, y3, s);
            y3 := y3 + RowSpace;
          end;
        x3 := x;  {Reset to first square coordinates}
        y3 := y;
        for i := 1 to neurons do
          begin
            case neural_set.binary_value [i] of
              1 : hue := Blue;
              0 : hue := lightgray;
            end;   {case}
            Draw_Memory_Box (x3, y3, ColSpace, RowSpace,
                                          hue, white);
            y3 := y3 + RowSpace;
          end;
      end
    else if arch = matrices then   {Matrix Layer
                                       Architecture}
      begin
        with viewport1 do
          begin
            SetViewPort (x1, y1, x2, y2, true);
            R :=
        ((y2-5*TextHeight('8'))-(y1+5*TextHeight('8')))
                 Div layers[1].arch_row_length;
            C :=
        ((x2-5*TextWidth('25'))-(x1+5*TextWidth('25')))
                 Div layers[1].arch_col_length;
            RowSpace := Minimum (R, C);  {Find smallest
                                           dimension}
            ColSpace := RowSpace;  {Will display a
                                                square}
            x := x1 + ((x2-x1) Div 2);      {Center of
                                              viewport}
            if layers[1].arch_col_length Mod 2 = 0
                                        {Even # of Cols}
              then x := x - ((layers[1].arch_col_length
                              Div 2)*ColSpace)
              else x := x - ((layers[1].arch_col_length
                              Div 2)*ColSpace)
                                      - ColSpace Div 2;
            y := y1 + ((y2-y1) Div 2);    {Center of
                                               viewport}
            if layers[1].arch_row_length Mod 2 = 0
                                          {Even # of Cols}
              then y := y - ((layers[1].arch_row_length
                              Div 2)*RowSpace)
              else y := y - ((layers[1].arch_row_length
                              Div 2)*RowSpace)
                                      - RowSpace Div 2;
          end;
        x3 := x - 3*TextWidth('25');  {Row Number
                                               position}
        y3 := y + (RowSpace Div 2);
        SetColor (white);
        for i := 1 to layers[1].arch_row_length do
                                   {Vertical row numbers}
          begin
            str (i, s);
            OutTextXY (x3, y3, s);
            y3 := y3 + RowSpace;
          end;
        x3 := x + (ColSpace Div 2);
        y3 := y - (RowSpace Div 2);
        for i := 1 to layers[1].arch_col_length do
                                       {Horiz Col numbers}
          begin
            str (i, s);
            OutTextXY (x3, y3, s);
            x3 := x3 + ColSpace;
          end;
        x3 := x;  {Reset to first square coordinates}
        y3 := y;
        i := 1;
        for j := 1 to layers[1].arch_col_length do
          begin
            for k := 1 to layers[1].arch_row_length do
              begin
                case neural_set.binary_value [i] of
                  1 : hue := Blue;
                  0 : hue := lightgray;
                end;   {case}
                Draw_Memory_Box (x3, y3, ColSpace,
                                RowSpace, hue, white);
                y3 := y3 + RowSpace;
                i := i + 1;
            end;
          y3 := y;              {Reset to top row}
          x3 := x3 + ColSpace   {Move over one column}
        end;
    end;
  end;
outlayer := net_description.num_layers;
with layers[outlayer] do        {Draw Output}
  begin
    if arch = lines then  {Line Layer Architecture}
      begin
        with viewport2 do
          begin
            if net_description.training = supervised
              then A := 3    {divide out viewport for
                                      display of tgt}
              else A := 2;   {vector also}
            SetViewPort (x1, y1, x2, y2, true);
            RowSpace1 :=
          ((y2-TextHeight('8'))-(y1+TextHeight('8')))
                          Div layers[outlayer].neurons;
            if (neurons = 1) or (neurons = 2)
              then RowSpace1 := RowSpace1 Div 4;
                                  {Reduce square size}
            ColSpace1 := RowSpace1;  {Will display a
                                              square}
            x := (x2-x1) Div A;   {set center of output
                                                line}
            x := x - (ColSpace1 Div 2);    {first square
                                         x coordinate}
            if neurons > 2
              then y := TextHeight('8')      {First
                                    square y coordinate}
              else
                if neurons = 1
                  then y := ((y2 - y1) Div 2) -
                                RowSpace1 Div 2
                  else y := ((y2 - y1) Div 2) -
                                          RowSpace1;
          end;
        x3 := x - 3*TextWidth('15');  {Number position}
        y3 := y + (RowSpace1 Div 2);
        SetColor (white);
        for i := 1 to neurons do   {Write neuron numbers
                                          vertically}
          begin
            str (i, s);
            OutTextXY (x3, y3, s);
            y3 := y3 + RowSpace1;
      end;
    x3 := x;  {Reset to first square coordinates}
    y3 := y;
    for i := 1 to neurons do
      begin
        case neural_set.binary_value [i] of
          1 : hue := Blue;
          0 : hue := lightgray;
        end;   {case}
        Draw_Memory_Box (x3, y3, ColSpace1,
                          RowSpace1, hue, white);
        SetTextJustify (LeftText, CenterText);
        str (neural_set.real_value[i]:3:4, s);
        SetColor (black);  {write weight values in
                                          memory box}
        OutTextXY (x3 + 3, y3 + RowSpace Div 2 -
                    TextHeight ('8'), s);
        if A = 3
          then
            begin
              str (r_out_buffer[i]:3:4, s);
              OutTextXY (x3 + 3, y3 +(RowSpace Div
                        2) + TextHeight ('8'), s);
            end;
        SetColor (yellow);
        SetTextJustify (CenterText, BottomText);
        y3 := y3 + RowSpace1;
      end;    {for i := 1 to neurons}
    if A = 3 then
      begin
        x3 := 2 * x;
        x3 := x3 + (ColSpace1 div 2); {reset to
                               first sqr coordinates}
        y3 := y;
        for i := 1 to neurons do
          begin
            if r_out_buffer[i] =
                           net_description.one_value
             then hue := Blue
             else hue := lightgray;
             Draw_Memory_Box (x3, y3, ColSpace1,
                             RowSpace1, hue, white);
             SetTextJustify (LeftText, CenterText);
             y3 := y3 + RowSpace1;
          end;    {for i := 1 to neurons}
      end;   {if A = 3}
end
```

```
else if arch = matrices then   {Matrix Layer
                                      Arctitecture}
  begin
    with viewport2 do
      begin
        SetViewPort (x1, y1, x2, y2, true);
        R := ((y2-5*TextHeight('8'))
              -(y1+5*TextHeight('8')))
               Div layers[outlayer].arch_row_length;
        C := ((x2-5*TextWidth('25'))
              -(x1+5*TextWidth('25')))
               Div layers[outlayer].arch_col_length;
        RowSpace1 := Minimum (R, C);  {Find smallest
                                           dimension}
        ColSpace1 := RowSpace1;  {Will display a
                                              square}
        x := (x2-x1) Div 2;     {Center of viewport}
        if layers[outlayer].arch_col_length Mod 2 =
                                   0  {Even # of Cols}
          then x := x - ((layers[outlayer].
                   arch_col_length Div 2)*ColSpace1)
          else x := x - ((layers[outlayer].
            arch_col_length Div 2)*ColSpace1) -
                                    ColSpace1 Div 2;
        y := y1 + ((y2-y1) Div 2);   {Center of
                                           viewport}
        if layers[outlayer].arch_row_length Mod 2 =
                                   0  {Even # of Cols}
          then y := y - ((layers[outlayer].
                   arch_row_length Div 2)*RowSpace1)
          else y := y - ((layers[outlayer].
              arch_row_length Div 2)*RowSpace1) -
                                  RowSpace1 Div 2;
      end;
    x3 := x - 3*TextWidth('25');  {Row Number
                                           position}
    y3 := y + (RowSpace1 Div 2);
    SetColor (white);
    for i := 1 to layers[outlayer].arch_row_length
                                   do {Vert row #}
      begin
        str (i, s);
        OutTextXY (x3, y3, s);
        y3 := y3 + RowSpace1;
      end;
    x3 := x + (ColSpace1 Div 2);
    y3 := y - (RowSpace1 Div 2);
```

```
        for i := 1 to layers[outlayer].arch_col_length
                                         do  {Horiz Col #}
          begin
            str (i, s);
            OutTextXY (x3, y3, s);
            x3 := x3 + ColSpace1;
          end;
        x3 := x;    {Reset to first square coordinates}
        y3 := y;
        i := 1;
        for j := 1 to layers[outlayer].arch_col_length
                                                      do
          begin
            for k := 1 to layers[outlayer].
                                   arch_row_length do
              begin
                case neural_set.binary_value [i] of
                  1 : hue := Blue;
                  0 : hue := lightgray;
                end;   {case}
                Draw_Memory_Box (x3, y3, ColSpace1,
                                 RowSpace1, hue, white);
                SetTextJustify (leftText, CenterText);
                str (neural_set.real_value[i]:3:4, s);
                SetColor (black);  {write weight values
                                        in memory box}
                OutTextXY (x3 + 3, y3 + RowSpace Div 2,
                                                    s);
                str (r_out_buffer[i]:3:4, s);
                OutTextXY (x3 + 3, y3 +(RowSpace Div 2)
                           + 2 * TextHeight ('8'), s);
                SetColor (yellow);
                SetTextJustify (CenterText, BottomText);
                y3 := y3 + RowSpace1;
                i := i + 1;
              end;
            y3 := y;             {Rest to top row}
            x3 := x3 + ColSpace1  {Move over one column}
          end;
      end;
  end;
repeat
  ch := readkey;
until (ch in [#27, #3, #13, #32, #0, 'T', 'S']);
```

```
case ch of
  #0  : begin
          ch := readkey;
          if ch = #59     {F1}
            then semaphore := 3;          {Display_Network}
        end;
  #13 : begin            {<RTN>}
          case net_description.training of
            supervised    : begin
                              if train
                                then train_sup_net
                                               {math unit}
                                else calc_sup_net;
                                               {math unit}
                            end;
            unsupervised : begin
                              if train
                                then train_unsup_net (ch)
                                               {math unit}
                                else calc_unsup_net;
                                               {math unit}
                            end;
            zip           : begin
                              next_input;
                              calculate_output;
                                               {math unit}
                            end;
          end;   {case}
        end;
  'T' : begin            {continual training}
          z := 0;
          gotoxy (44, 23);
          write ('Enter number of iterations ');
          readln (it);
          gotoxy (44, 23);
          textcolor (black);
          write ('                                  ');
          textcolor (yellow);
          gotoxy (44, 22);
          write (it);
          repeat
            case net_description.training of
              supervised    : begin
                                if train
                                  then train_sup_net
                                               {math unit}
                                  else calc_sup_net;
                                               {math unit}
                              end;
```

```
                    unsupervised : begin
                                     if train
                                       then train_unsup_net
                                              (ch)  {math unit}
                                       else calc_unsup_net;
                                                    {math unit}
                                   end;
                    zip          : begin
                                     next_input;
                                     calculate_output;
                                                    {math unit}
                                   end;
                  end;   {case}
                  gotoxy (44, 23);
                  z := z + 1;
                  write ('Z = ', z);
                  if net_description.iterate > 0   {Don't
                                                 divide by 0}
                    then if z mod net_description.iterate = 0
                      then save_state (z);     {ioutil unit}
                until z = it;
              end;
       'S' : Save_State (0);     {ioutil unit}
       #32 : Modify_Parameters;  {<SPACE>}
       #3  : semaphore := 1;   {Switch to Display_Memory
                                                    <ESC>}
    end;   {case}
  until (ch in [#27, #59, #3]);   {<ESC>, <F1>, or <^C>}
end;    {Display_Net_Output}

{-----------------------------------------------------------}

{Provides the memory display while in the display mode}

Procedure Display_Memory (var ch : char; var semaphore :
                          integer);

var
  ComLine, s             : string;
  viewport1, viewport2 : ViewPortType;
  i, j, k, x, y          : integer;
  x3, y3                 : integer;
  rows, cols, hue        : integer;
  RowSpace, ColSpace     : integer;
  check                  : real;

begin   {Display_Memory}
  FullPort;
  SetColor (yellow);
```

```
ClearDevice;
Write_Header ('RELATIVE WEIGHT VALUES', 5);
Comline := '<ESC> Quit   <Space> Next Layer   <RTN> Next
                                                      Input';
Comline := concat (ComLine, '   <F1> Network   <^C>
                                                    Output');
Write_Command_Line (ComLine, 0);
SetViewPort (0, 20+(TextHeight('M')), MaxX,
              MaxY-20-(TextHeight('M')), ClipOn);
GetViewSettings (viewport1);
j := 2;   {Layer 2}
repeat             {Display Layer Headers and Labels}
  SetTextStyle (SmallFont, HorizDir, 4);
  with viewport1 do
    begin
      ClearViewPort;
      SetViewPort (x1, y1, x2, y2, true);
      RowSpace := ((y2-5*TextHeight('8'))-
                    (y1+5*TextHeight('8')))
                       Div layers[j-1].neurons;
      ColSpace := (x2-(x1+TextWidth('Layer 5  25  ')))
                    Div layers[j].neurons;
      x := TextWidth ('Layer 5  25  ');
      x :=  x + ((x2-x) Div 2);    {Center of viewport}
      x := x - ((layers[j].neurons Div 2)*ColSpace);
                                            {x is top left}
      if layers[j].neurons Mod 2 <> 0      {Odd # of Cols
        then x := x - (ColSpace Div 2);  {first square }
      y := y1 + ((y2-y1) Div 2);     {Center of viewport}
      y := y - ((layers[j-1].neurons Div 2)*RowSpace);
                                              {y is top left}
      if layers[j-1].neurons Mod 2 <> 0  {Odd # of Rows
                                          coordinate of}
        then y := y - (RowSpace Div 2);
                                          {first square}
      SetColor (white);
      SetTextJustify (LeftText, CenterText);
      str (j-1, S);
      OutTextXY (0, (y2-y1) Div 2, 'Layer '+S);
      OutTextXy (0, ((y2-y1) Div 2) + TextHeight ('L') +
                                          4, 'Neurons');
      x3 := x - TextWidth('  2'); {Row Number position}
      y3 := y + (RowSpace Div 2);
      SetTextJustify (CenterText, CenterText);
      for i := 1 to layers[j-1].neurons do  {Vertical row
                                                  numbers}
        begin
          str (i, s);
          OutTextXY (x3, y3, s);
        y3 := y3 + RowSpace;
      end;
    str (j, s);
    SetTextJustify (LeftText, BottomText);
    OutTextXY ((x+(x2-x)) Div 2, y-(4*TextHeight('8')),
                   'Layer '+s+ ' Neurons');
    x3 := x + (ColSpace Div 2);
    y3 := y - (TextHeight ('8'));
    SetTextJustify (CenterText, BottomText);
    for i := 1 to layers[j].neurons do{Horiz Colnumbers}
      begin
        str (i, s);
        OutTextXY (x3, y3, s);
        x3 := x3 + ColSpace;
      end;
  end;
repeat{Show memory weights in color within the viewport}
  x3 := x;       {Reset to first square coordinates}
  y3 := y;
  for rows := 1 to layers[j-1].neurons do
    begin        {Draw square for each entry in weight
                                    matrix for layer j}
      for cols := 1 to layers[j].neurons do
        begin
          check := layers[j].weights^.value[cols, rows];
          if check = 0        {Revise the colors!!!!}
            then hue := Black
          else if (check <= -6.0)
            then hue := Darkgray
          else if (check > -6.0) and (check <= -5.0)
            then hue := Blue
          else if (check > -5.0) and (check <= -4.0)
            then hue := Brown
          else if (check > -4.0) and (check <= -3.0)
            then hue := red
          else if (check > -3.0) and (check <= -2.0)
            then hue := Green
          else if (check > -2.0) and (check <= -1.0)
            then hue := Magenta
          else if (check > -1.0) and (check <= -0.0)
            then hue := Lightblue
          else if (check > 0.0) and (check <= 1.0)
            then hue := Cyan
          else if (check > 1.0) and (check <= 2.0)
            then hue := Lightgray
          else if (check > 2.0) and (check <= 3.0)
            then hue := Lightcyan
          else if (check > 3.0) and (check <= 4.0)
            then hue := Lightred
          else if (check > 4.0) and (check <= 5.0)
            then hue := Lightmagenta
          else if (check > 5.0) and (check <= 6.0)
            then hue := Lightgreen
          else if (check > 6.0) and (check <= 7.0)
            then hue := Yellow
          else if (check > 7.0)
            then hue := White;
          Draw_Memory_Box (x3, y3, ColSpace, RowSpace,
                                            hue, white);
          SetTextJustify (CenterText, CenterText);
          str (layers[j].weights^.value[cols, rows]:2:2,
                                                      s);
          SetColor (black);  {write weight values in
                                               memory box}
          if layers[j-1].neurons < 25 then
            OutTextXY (x3 + (ColSpace Div 2), y3 +
                                       RowSpace Div 2, s);
          SetColor (yellow);
          SetTextJustify (CenterText, BottomText);
          x3 := x3 + ColSpace;      {Move square over
                                               one column}
        end;
      x3 := x;                  {Reset to Original}
      y3 := y3 + RowSpace;      {Increment by one row}
    end;
  repeat
    ch := readkey;
  until (ch in [#27, #3, #13, #32, #0]);
  case ch of
    #0  : begin
            ch := readkey;
            if ch = #59    {F1}
              then semaphore := 4; {Switch to
                                         Display_Network}
          end;
    #13 : begin  {<RTN>} {fix for recurrent networks!!!}
            case net_description.training of
              supervised   : begin
                               if train
                                 then train_sup_net
                                             {math unit}
                                 else calc_sup_net;
                                             {math unit}
                             end;
```

```
                        unsupervised : begin
                                         if train
                                           then train_unsup_net
                                                 (ch)   {math unit}
                                           else calc_unsup_net;
                                                       {math unit}
                                       end;
                        zip          : begin
                                         next_input;
                                         calculate_output;
                                                       {math unit}
                                       end;
                      end;   {case}
                    end;
              #32 : begin        {<SPACE>}
                      if j = net_description.num_layers
                        then j := 2   {reset to first iteration}
                        else j := j + 1;   {increment layer}
                    end;
              'S' : Save_State (0);
              #3  : semaphore := 2;  {Switch to
                                            Display_Net_Output}
          end;   {case}
        until (ch in [#27, #3, #32, #59]);  {<^C>, <ESC>,
                                              <SPACE>, or <F1>}
      until (ch in [#27, #3, #59]);   {<^C>, <ESC>, or <F1>}
    end;    {Display_Memory}

    end.   {GrafStuff Unit}
```

APPENDIX F

Development System Display Unit Listing
(Connection Modification Include File)

Appendix B through Appendix J constitute the complete development system program listing

```
{---------------------------------------------------------}

{This is an include file to the grafstuf unit.  Allows}
{modification of connectivity between the layers of the}
{network.  This procedures can be accessed through both the}
{menu system and the display system for modifications to}
{the connectivity definitions.                          }

{------------------------------------------------------------}

{Allows for removal/restoration of individual connections}
{or for the random removal of connections}

Procedure Modify_Connections (var ch : char; layer :
                                      integer; str :string);

  var
    int, i, j, n : integer;
    GResult      : integer;  {Store results of GraphResults}
    saved, modify : boolean;
    prob          : real;
    connections   : packed array[1..max_bodies] of boolean;
                                           {connected?}
    disconnected  : boolean;

begin    {Modify_Connections}
  prob := 0.0;
  int := 0;
  repeat
    clrscr;
    gotoxy (20, 8);
    write ('Modify Connections');
    gotoxy (22, 9);
    write ('(Layer ');
    reverse;
    write(' ', layer, ' ');
    normvid;
    write (' to Layer ');
    reverse;
    write (' ', layer + 1, ' ');
    normvid;
    write (')');
    gotoxy (20, 11);
    write ('1.  Remove Connection');
    gotoxy (20, 12);
    write ('2.  Restore Connection');
    gotoxy (20, 13);
    write ('3.  Restore Full Layer Connectivity');
    gotoxy (20, 14);
```

```
write ('4.  Random Connection Removal (Prob = ',
                                       prob:2:2, ')');
gotoxy (20, 15);
write ('5.  Save Weight State');
gotoxy (20, 16);
write ('6.  Display Network');
gotoxy (1, 24);
write ('<ESC> ', str, '                         <RTN> for
                                         next layer');
gotoxy (1, 21);
clreol;
write ('=> ');
repeat
  ch := readkey;
until ch in ['1'..'6', #27, #13];   {nums, <ESC>, <RTN>}
case ch of
  '1' : begin                  {remove connection}
          if layers[layer + 1].saved_weights = nil
            then
              begin
                clrscr;
                gotoxy (1, 1);
                reverse;
                write ('Saved state doesn''t exist!');
                write ('  Will not be able to
                                          restore!');
                normvid;
                gotoxy (1, 3);
                write ('<RTN> to Continue  <ESC> to
                                               Quit');
                repeat
                  ch := readkey;
                until ch in [#13, #27];
              end;
           if (ch = #13) or (layers[layer +
                             1].saved_weights <> nil)
             then
               begin
                 ch := '1';
                 modify := false;
                 i := 0;
                 clrscr;
                 gotoxy (1, 15);
                 write ('Select Connection for Deletion
                                        (Layer ', layer);
                 writeln (' to Layer ', layer + 1,')');
```

```
repeat
  repeat
    gotoxy (1, 21);
    clreol;
    write ('=> ');
    gotoxy (1, 24);
    clreol;
    write ('Enter Numbers for Layer ', layer, ' or
                                                <RTN> ');
    writeln ('for No Entry');
    write ('(Neurons in layer = ',
                       layers[layer].neurons, ')');
    gotoxy (4, 21);   {place cursor after =>}
    getline;          {mathutil unit}
    if (ColsOnLine > 0) and (Nextchar in ['0'..'9'])
      then
        begin
          process_number (int);      {mathutil unit}
          if int > layers[layer].neurons
            then
               begin
                 gotoxy (10, 2);
                 reverse;
                 write ('Exceeds Layer Neurons (',
                          layers[layer].neurons, ')');
                 normvid;
                 gotoxy (1, 21);
                 clreol;              {Erase =>}
                 gotoxy (1, 24);
                 clreol;
                 gotoxy (1, 23);
                 write ('<RTN> to continue');
                 repeat
                   ch := readkey
                 until ch = #13;
                 gotoxy (1, 23);
                 clreol;             {Erase <RTN>..}
                 gotoxy (10, 2);   {Clear error message}
                 clreol;
                 gotoxy (1, 21);
                 write ('=> ');
               end
            else
              if ch = #13{<RTN>- 2nd layer designation}
                then
                  begin
                    j := int;
                    connections[j] := false;
                    modify := true;
```

```
                        end
                      else    {1st layer neuron designation}
                        begin
                          i := int;
                          if layer =
                                   net_description.num_layers
                            then layer := 1
                            else layer := layer + 1;
                          for j := 1 to
                                     layers[layer].neurons do
                            if layers[layer].weights^.
                                           value[j, i] = 0.0
                              then connections[j] := false
                              else connections[j] := true;
                        end;
              end;
            gotoxy (1, 24);
            clreol;
            write ('<^C> To Accept & Continue
                                        <ESC> to Exit');
            write ('        <RTN> to Accept Neuron');
            gotoxy (1, 25);
            clreol;
            ch := readkey;
        until ch in [#3, #13, #27];       {<^C> to exit}
      until ch in [#3, #27];     {<^C> or <ESC> to exit}
      if ch = #3   {^C to accept/ESC exit with no change}
        then
          if (i > 0) and modify
            then
              begin
                for n := 1 to layers[layer].neurons do
                  if connections[n] = false
                    then
                     layers[layer].weights^.value[n, i] := 0.0;
                  layers[layer -
                             1].FFConnection_kind := partial;
              end;
      clrscr;
      if layer >= net_description.num_layers
        then layer := 1;
    end;
  end;
'2' : begin                      {restore connection}
        disconnected := false;
        if (layer < net_description.num_layers) and
            (layers[layer + 1].saved_weights = nil)
          then
            begin
        clrscr;
        gotoxy (1, 1);
         reverse;
          write ('No saved state exists! <RTN> to
                                               Continue');
          repeat
            ch := readkey;
          until ch = #13;
          normvid;
      end
    else
      begin
        modify := false;
        i := 0;
        clrscr;
        gotoxy (1, 15);
        write ('Select Connection to Restore
                                    (Layer ', layer);
        write (' to Layer ', layer + 1,')');
        gotoxy (1, 5);
        reverse;
        write ('Neurons in layer ', layer);
        write (' with Disconnections');
        normvid;
        gotoxy (3, 7);
        for j := 1 to layers[layer].neurons do
          begin
            for n := 1 to layers[layer +
                                      1].neurons do
              if layers[layer +
                           1].weights^.value[n, j] = 0.0
                then disconnected := true;
              if disconnected then write (j:3);
                          {write disconnected neuron}
               disconnected := false;
           end;
        repeat
          gotoxy (1, 21);
          clreol;
          write ('=> ');
          gotoxy (1, 24);
          clreol;
          write ('Enter Numbers for Layer ', layer);
          write (' or <RTN> for No Entry');
          gotoxy (4, 21);{place cursor after =>}
          getline;              {mathutil unit}
          if (ColsOnLine > 0) and (Nextchar in
                                          ['0'..'9'])
            then
```

```
begin
  process_number (int);  {mathutil unit}
  if int > layers[layer].neurons
    then
      begin
        gotoxy (10, 2);
        reverse;
        write ('Exceeds Layer Neurons
          (',layers[layer].neurons,')');
        normvid;
        gotoxy (1, 21);
        clreol;         {Erase =>}
        gotoxy (1, 24);
        clreol;
        gotoxy (1, 23);
        write ('<RTN> to continue');
        repeat
          ch := readkey
        until ch = #13;
        gotoxy (1, 23);
        clreol;    {Erase <RTN>..}
        gotoxy (10, 2);   {Clear}
                         {error message}
        clreol;
        gotoxy (1, 21);
        write ('=> ');
      end
    else
      if ch = #13  {layer + 1}
        then
          begin
            j := int;
            connections[j] := true;
            modify := true;
          end
        else       {layer}
          begin
            i := int;
            if layer =
             net_description.num_layers
            then layer := 1
            else layer := layer + 1;
            for j := 1 to
                layers[layer].neurons do
              if
            layers[layer].weights^.value
                            [j, i] = 0.0
               then
                 connections[j] := false
```

```
                                    else
                                      connections[j] := true;
                                  gotoxy (1, 9);
                                  reverse;
                                  write ('Neurons in
                                             layer ', layer);
                                  write (' with
                                       Disconnections from');
                                  write (' Layer ',
                                                 layer - 1);
                                  normvid;
                                  write (' (Neuron ', i,')');
                                  gotoxy (3, 11);
                                  for j := 1 to
                                     layers[layer].neurons do
                                  if layers[layer].weights^.
                                        value[j, i] = 0.0
                                    then write (j:3);
                                                {disconnected}
                               end;
                    end;
                gotoxy (1, 24);
                clreol;
                write ('<^C> To Accept & Continue
                                          <ESC> to Exit');
                write ('     <RTN> to Accept Neuron');
                ch := readkey;
              until ch in [#3, #27];{<^C> or <ESC> to exit}
              if ch <> #27   {ESC exit with no change}
                then
                  if (i > 0) and modify
                    then
                      for n := 1 to
                                  layers[layer].neurons do
                        if (connections[n] = true) and
                  (layers[layer].weights^.value[n, i] = 0.0)
                          then
                      layers[layer].weights^.value[n, i] :=
                   layers[layer].saved_weights^.value[n, i];
              clrscr;
              gotoxy (1, 1);
              reverse;
              if modify
                then write ('Selected Connections
                                         Restored!  <RTN>')
                else write ('No Connections
                                        Restored!  <RTN>');
              repeat
              until keypressed;
```

```
          normvid;
          if layer >= net_description.num_layers
            then layer := 1;
          for n := 2 to net_description.num_layers do
            begin
              disconnected := false;
              for j := 1 to layers[n].neurons do
                 for i := 1 to layers[n - 1].neurons do
                   if
                    layers[n].weights^.value[i, j] = 0.0
                    then disconnected := true;
              if disconnected
                then layers[n -
                       1].FFConnection_kind := partial
                else layers[n -
                          1].FFConnection_kind := fully;
            end;
         end;
        end;
'3' : begin {restore full connectivity for the layer}
          saved := true;
          if layers[layer + 1].saved_weights <> nil
            then layers[layer+1].weights^ :=
                         layers[layer+1].saved_weights^
            else saved := false;{single layer with no}
                                      {S_W poss?}
          if not saved
            then
              begin
                clrscr;
                gotoxy (1, 1);
                reverse;
                write ('Saved state doesn''t exist!');
                normvid;
                gotoxy (1, 2);
                write ('Connectivity Not Restored.
                                    <RTN> to Continue');
                repeat
                  ch := readkey;
                until ch = #13;
              end
           else
             begin
               gotoxy (1, 1);
               reverse;
               write ('Connections Restored!  <RTN>');
               repeat
                 ch := readkey;
               until ch = #13;
              normvid;
              ch := '>'
            end;
         for n := 2 to net_description.num_layers do
           begin
             disconnected := false;
             for j := 1 to layers[n].neurons do
               for i := 1 to layers[n - 1].neurons do
                 if layers[n].weights^.value[i, j] =
                                                     0.0
                   then disconnected := true;
             if disconnected
               then layers[n - 1].FFConnection_kind :=
                                                  partial
               else layers[n - 1].FFConnection_kind :=
                                                   fully;
           end;
       end;
 '4' : begin                 {random connection removal}
         get_removal_probability (prob);     {math unit}
         remove_random_connections (prob, layer);
                                             {math unit}
         gotoxy (1, 1);
         reverse;
         write ('Connections Removed! <RTN>');
         repeat
           ch := readkey;
         until ch = #13;
         normvid;
         ch := '<';
       end;
 '5' : begin                     {save weight state}
         clrscr;
         gotoxy (1, 1);
         reverse;
         write ('Previous saved state will be destroyed
                                   - Continue? Y/N');
         repeat
           ch := readkey
         until ch in ['Y', 'y', 'n', 'N'];
         normvid;
         if ch in ['Y', 'y']
           then
             begin
               for i := 2 to net_description.num_layers
                                                     do
                 begin
                   if layers[i].saved_weights = nil
                     then new
```

```
                                  (layers[i].saved_weights);
                        layers[i].saved_weights^ :=
                                        layers[i].weights^;
                      end;
                    clrscr;
                    gotoxy (1, 1);
                    reverse;
                    write ('Weights Saved! <RTN>');
                    repeat
                      ch := readkey;
                    until ch = #13;
                    normvid;
                  end
                else
                  begin
                    clrscr;
                    gotoxy (1, 1);
                    reverse;
                    write ('Weights Not Saved! <RTN>');
                    repeat
                      ch := readkey;
                    until ch = #13;
                    normvid;
                  end;
            end;
      '6' : Display_Network (12);  {display network}
      #13 : begin                  {<RTN>}
              if str = 'Run Network'  {From Display Mode}
                then if layer >= (net_description.num_layers
                                                       - 1)
                  then layer := 1
                  else layer := layer + 1;
            end;
    end;               {case}
  until ch in [#27];   {<ESC>}
end;     {Modify_Connections}

{--------------------------------------------------------}
```

APPENDIX G

Development System Input/Output Utility Unit Listing (Unit ioutil)

Appendix B through Appendix J constitute the complete development system program listing

```
{-----------------------------------------------------------}

{Unit ioutil provides file read/write routines and the}
{input of the next data input line and its proper}
{formatting for the algorithm in use}

{-----------------------------------------------------------}

Unit ioutil;

Interface

uses nnglobal, mathutil, crt, printer;

Procedure Reverse;
Procedure Normvid;
Procedure Get_Input_File;
Procedure Save_Network;
Procedure Load_File;
Procedure Compile;
Procedure Print_Files;
Procedure Save_State (z : integer);
Procedure Save_Net_Option;
Procedure Save_Freq_Option;
Procedure readfile;
Procedure ReadTrainFile;
Procedure Next_Input;

Implementation

{-----------------------------------------------------------}

Procedure Reverse;

begin
  textbackground (white);
  textcolor (black);
end;

{-----------------------------------------------------------}

Procedure NormVid;

begin
  normvideo;
  textcolor (yellow);
end;

{-----------------------------------------------------------}
```

```
Procedure Get_File_Name (var Name : string; var ch : char);

var
  File_OK : boolean;
  f       : integer;
  s       : string[80];

begin   {Get_File_Name}
  clrscr;
  repeat
    File_Ok := false;
    gotoxy (1, 20);
    write ('Enter File Name');
    gotoxy (1, 21);
    write ('"*"  Indicates c:\pascal\infiles\');
    gotoxy (1, 22);
    write ('=> ');
    Readln (name);
    if name[1] = '*'
      then
        begin
          delete (name, 1, 1);
          s := 'c:\pascal\infiles\'+name;
          name := s;
        end;
    reverse;
    gotoxy (1, 23);
    write (name);
    gotoxy (1, 24);
    write ('File Name OK?  Y/N   <ESC>');
    normvid;
    repeat
      ch := readkey;
      gotoxy (27, 24);
    until ch in ['Y', 'y', 'N', 'n', #27];
    clrscr;
    if ch in ['Y', 'y']
      then File_OK := true;
  until File_OK or (ch = #27);
end;   {Get_File_Name}

{----------------------------------------------------------}
```

```
Procedure Get_Input_File;

var
  name : string;
  ch   : Char;
  f    : integer;

begin   {Get_Input_File}
  ch := '6';  {Set to some value other than <ESC>}
  Get_File_Name (name, ch);
  if ch = #27
    then exit;   {<ESC> set from get_file_name}
  if (infilename <> 'NONE SELECTED') and (length
                                          (InFileName) > 0)
    then
      begin
        close (infile);  {close old file if open}
        infilename := 'NONE SELECTED';
      end;
  {$I-}
    assign (infile, name);
    reset (infile);
  {$I+}
  f := ioresult;
  if f <> 0   {not a valid name}
    then
      begin
        gotoxy (1, 1);
        reverse;
        write ('File Not Found!  Hit any key to continue');
        normvid;
        repeat
        until keypressed;
      end
    else InFileName := name;
end;    {Get_Input_File}

{---------------------------------------------------------}

Procedure Save_Network;

  var
    name   : string;
    ch     : char;
    i, f   : integer;
    w, ws  : weight_ptr;
```

```
begin     {Save_Network}
  repeat
    ch := '6';  {set to some value other than <ESC>)}
    Get_File_Name (name, ch);
    if ch = #27
      then exit;   {<ESC> set from get_file_name}
    {$I-}
      assign (savefile1, Name+'.1');
      reset (savefile1);
    {$I+}
    f := ioresult;
    if f = 0   {file already exists}
      then
        repeat
          gotoxy (1, 1);
          reverse;
          write ('File Exists!  Overwrite?  Y/N');
          normvid;
          read (ch);
        until ch in ['Y', 'y', 'N', 'n'];
  until ch in ['Y', 'y'];
  assign (savefile1, name+'.1');
  assign (savefile2, name+'.2');
  assign (savefile3, name+'.3');
  assign (savefile4, name+'.4');
  rewrite (savefile1);
  rewrite (savefile2);
  rewrite (savefile3);
  rewrite (savefile4);
  write (savefile1, net_description);
  close (savefile1);
  for i := 2 to net_description.num_layers do
    begin
      write (savefile4, layers[i].weights^);
      write (savefile4, layers[i].saved_weights^);
    end;
  close (savefile4);
  for i := 1 to net_description.num_layers do
    begin
      w := layers[i].weights; {Temporary pointer to weights}
      layers[i].weights := nil;  {to be saved set to nil}
      ws := layers[i].saved_weights;   {Temporary pointer}
      layers[i].saved_weights := nil;  {saved set to nil}
      write (savefile2, layers[i]);{Save layer data to file}
      layers[i].weights := w;          {reset to weights}
      layers[i].saved_weights := ws;{reset to saved weights}
{     write (savefile3, layers[i].connections^); }
    end;
  close (savefile2);
  close (savefile3);
end;    {Save_Network}

{-----------------------------------------------------------}

Procedure Load_File;

  var
    name     : string;
    ch       : char;
    i, f     : integer;

begin
  repeat
    ch := '6';    {set ch to something other than <ESC>}
    Get_File_Name (name, ch);
    if ch = #27
      then exit;    {<ESC> set from get_file_name}
    {$I-}
      assign (savefile1, Name+'.1');
      reset (savefile1);
    {$I+}
    f := ioresult;
    if f <> 0    {file does not exists}
      then
        begin
          gotoxy (1, 1);
          reverse;
          write ('File Not Found!  Press <RTN> to
                                               continue');
          normvid;
          readln;
        end;
  until f = 0;
  assign (savefile1, name+'.1');
  assign (savefile2, name+'.2');
  assign (savefile3, name+'.3');
  assign (savefile4, name+'.4');
  reset (savefile1);
  reset (savefile2);
  reset (savefile3);
  reset (savefile4);
  for i := 1 to net_description.num_layers do {Initialize}
                                          {pointers to nil}
    begin
      if layers[i].weights <> nil
        then
          begin
```

```
            dispose (layers[i].weights);
            layers[i].weights := nil;
          end;
      if layers[i].saved_weights <> nil
        then
          begin
            dispose (layers[i].saved_weights);
            layers[i].saved_weights := nil;
          end;
      if layers[i].connections <> nil
        then
          begin
            dispose (layers[i].connections);
            layers[i].connections := nil;
          end;
    end;
  read (savefile1, net_description);
  net_description.save_state := false; {For File integrity}
                                       {always load false}
  net_description.iterate := 0; {0 when save_state is false}
  num_layers := net_description.num_layers;
  close (savefile1);
  for i := 1 to net_description.num_layers do
    begin
      read (savefile2, layers[i]);
      {read (savefile3, layers[i].connections^);}  {Not}
                                              {defined yet}
    end;
  close (savefile2);
  close (savefile3);
  Initialize_Weights;
  for i := 2 to net_description.num_layers do
    begin
      read (savefile4, layers[i].weights^);
      if layers[i].saved_weights = nil
        then new (layers[i].saved_weights);
      read (savefile4, layers[i].saved_weights^);
    end;
  close (savefile4);
end;    {Load_File}

{-----------------------------------------------------------}

Procedure Compile;

begin   {Compile}
  {Not defined yet}
end;    {Compile}
```

```
{-------------------------------------------------------------}

Procedure Print_Files;

var
 i, f, x, y, z  : integer;
 count          : integer;
 ch             : char;

begin  {Print_Files}
  clrscr;
  gotoxy (1, 1);
  reverse;
  write ('WARNING! Printing will destroy the current
           Network!  Proceed?  Y/N');
  normvid;
  repeat
    ch := readkey;
  until ch in ['Y', 'y', 'N', 'n'];
  if ch in ['N', 'n']
    then exit;
  if net_description.save_state = false
    then
      begin
        assign (savefile1, 'c:\pascal\infiles\netstate.1');
        assign (savefile2, 'c:\pascal\infiles\netstate.2');
        assign (savefile3, 'c:\pascal\infiles\netstate.3');
        assign (savefile4, 'c:\pascal\infiles\netstate.4');
      end;
  reset (savefile1);
  reset (savefile2);
  reset (savefile3);
  reset (savefile4);
  for i := 1 to max_layers do   {Initialize pointers to nil}
    begin
      if layers[i].weights <> nil
        then
          begin
            dispose (layers[i].weights);
            layers[i].weights := nil;
          end;
      if layers[i].saved_weights <> nil
        then
          begin
            dispose (layers[i].saved_weights);
            layers[i].saved_weights := nil;
          end;
    if layers[i].connections <> nil
      then
        begin
          dispose (layers[i].connections);
          layers[i].connections := nil;
        end;
  end;
read (savefile1, net_description);  {net description data}
num_layers := net_description.num_layers;
close (savefile1);
for i := 1 to net_description.num_layers do
  begin
    read (savefile2, layers[i]);   {layer data}
    {read (savefile3, layers[i].connections^);}  {Not}
                                                {defined yet}
  end;
writeln (lst, 'NET DESCRIPTION');
writeln (lst);
with net_description do
  begin
    writeln (lst, net_name);
    writeln (lst, 'Net Architecture ', ord (net_arch));
    writeln (lst, num_layers, '  layers');
    writeln (lst, 'Activation Type ', ord (activation));
    writeln (lst, 'Threshold ', threshold:4:2);
    writeln (lst, 'Learning Type ', ord (learning));
    writeln (lst, 'Factor - ', factor:4:3, ' Factor1 - ',
             factor1:4:3, ' Factor2  - ', factor2:4:3, ',
             Factor3 - ', factor3:4:3);
    writeln (lst, 'Momentum ', momentum:4:2);
    writeln (lst, 'Input values ', one_value:3:1, ' / ',
             zero_value:3:1);
    writeln (lst);
  end;
for i := 1 to num_layers do
  begin
    writeln (lst, 'LAYER ', i);
    writeln (lst, '  Connection type ', ord
                        (layers[i].FFconnection_kind));
    writeln (lst, '  Number of Neurons ',
                                     layers[i].neurons);
    writeln (lst, '  Layer threshold ',
                 layers[i].neural_set.threshold[1]:4:2);
  end;
writeln (lst);
count := 1;
x := net_description.iterate;
reset (savefile2);   {prime pump for repeat loop - set to}
                        {first record}
```

```
repeat
  for i := 1 to net_description.num_layers do
    begin
      read (savefile2, layers[i]);   {layer data}
      {read (savefile3, layers[i].connections^);}  {Not}
                                              {defined yet}
    end;
  initialize_weights;  {layers data had weight ptr = nil}
  writeln (lst, 'ITERATION NUMBER ', count * x);
  writeln (lst, 'Order is - Binary value, Real value,
                                         Threshold, Rate');
  writeln (lst);
  for i := 1 to num_layers do
    begin
      writeln (lst, 'LAYER ', i);
      writeln (lst);
      for y := 1 to layers[i].neurons do
        begin
          write (lst, '  ');
          write (lst, y, '.  ',
            layers[i].neural_set.binary_value[y], ' ');
          write (lst,
             layers[i].neural_set.real_value[y]:4:4, ' ');
          write (lst,
              layers[i].neural_set.threshold[y]:4:2, ' ');
          writeln (lst, layers[i].neural_set.rate);
        end;
      if i > 1     {layer 2 and above for weights}
        then
          begin
            read (savefile4, layers[i].weights^);
            writeln (lst);
            writeln (lst, 'Weight values for layer ', i);
            writeln (lst);
            for y := 1 to layers[i].weights^.row_length do
              begin
                write (lst, y, '.  ');
                for z := 1 to
                        layers[i].weights^.col_length do
                  write (lst, layers[i].weights^.value[z,
                         y]:4:2, ' ');
                  writeln (lst);
              end;
            writeln (lst);
          end;
    end;
  count := count + 1;
until eof (savefile2);
close (savefile2);
```

```
  close (savefile3);
  close (savefile4);
  net_description.save_state := false;
  net_description.iterate := 0;
end;   {Print_Files}

{------------------------------------------------------------}

Procedure Save_State (z : integer);

  var
    w, ws : weight_ptr;
    ch    : char;
    i     : integer;

begin  {Save_State}
  if (net_description.save_state = true) and (z = 0)
                    {manually selected for save}
    then
      begin
        gotoxy (1, 1);
        write ('Do you want to purge previous Savefiles?
                                             Y/N/<ESC>');
        repeat
          ch := readkey;
        until ch in ['Y', 'y', 'N', 'n', #27];
        if ch in ['Y', 'y']
          then
            begin
              rewrite (savefile1);
              rewrite (savefile2);
              rewrite (savefile3);
              rewrite (savefile4);
            end
          else if ch = #27
            then exit;  {if 'N' then continue with no}
                                              {rewrite}
      end
    else if net_description.save_state = false
      then
        begin
          assign (savefile1,'c:\pascal\infiles\netstate.1');
          assign (savefile2,'c:\pascal\infiles\netstate.2');
          assign (savefile3,'c:\pascal\infiles\netstate.3');
          assign (savefile4,'c:\pascal\infiles\netstate.4');
          rewrite (savefile1);
          rewrite (savefile2);
          rewrite (savefile3);
          rewrite (savefile4);
          net_description.save_state := true;
        end;  {if save_state true than don't rewrite - just}
                                                     {continue}
  if (net_description.iterate = z) or (z = 0)
    then write (savefile1, net_description);  {Only save}
                                                  {once}
  for i := 2 to net_description.num_layers do
    write (savefile4, layers[i].weights^);
  for i := 1 to net_description.num_layers do
    begin
      w := layers[i].weights; {Temporary pointer to weights}
      layers[i].weights := nil;  {to be saved set to nil}
      ws := layers[i].saved_weights;   {Temporary pointer}
      layers[i].saved_weights := nil;  {saved set to nil}
      write (savefile2, layers[i]);{Save layer data to file}
      layers[i].weights := w;          {reset to weights}
      layers[i].saved_weights := ws;{reset to saved weights}
{     write (savefile3, layers[i].connections^); }
    end;
end;   {Save_State}

{---------------------------------------------------------}

Procedure Save_net_Option;

  var
    ch   : char;
    f  : integer;

begin  {Save_Net_Option}
  clrscr;
  if net_description.save_state = true
    then
      begin
        gotoxy (1, 1);
        reverse;
        write ('Overwrite Existing NetState File?
                                                  Y/<ESC>');
        gotoxy (1, 2);
        normvid;
        write ('               OR'               );
        gotoxy (1, 3);
        reverse;
        write ('Toggle Net Save Option to OFF? (Close files)
                                                        -T-');
        normvid;
        repeat
          ch := readkey;
        until ch in ['Y', 'y', 'T', 't', #27];
        if ch in ['T', 't']
          then
            begin
              net_description.iterate := 0;
              net_description.save_state := false;
              close (savefile1);
              close (savefile2);
              close (savefile3);
              close (savefile4);
              exit;
            end
         else if ch = #27
           then exit;
      end
    else    {net_description.save_state = false}
      begin
        assign (savefile1, 'c:\pascal\infiles\netstate.1');
        assign (savefile2, 'c:\pascal\infiles\netstate.2');
        assign (savefile3, 'c:\pascal\infiles\netstate.3');
        assign (savefile4, 'c:\pascal\infiles\netstate.4');
        if net_description.iterate = 0
          then net_description.iterate := 1;
      end;
  rewrite (savefile1);
  rewrite (savefile2);
  rewrite (savefile3);
  rewrite (savefile4);
  net_description.save_state := true;
end;  {Save_Net_Option}

{---------------------------------------------------------}

Procedure Save_Freq_Option;

  var
    n, f : integer;

begin    {Save_Freq_Option}
  repeat
    clrscr;
    gotoxy (1, 20);
    writeln ('Enter Number of Iterations Between Saves (0 -
                                                    1000)');
    write ('Enter 0 for no saves');
    gotoxy (1, 22);
    write ('=> ');
    {$I-}
    readln (n);
    {$I+}
    f := ioresult;
  until (f = 0) and (n >= 0) and (n <= 1000);
  if (n > 0) and (net_description.save_state = false)
    then
      begin
        gotoxy (1, 1);
        reverse;
        write ('Ensure Save State it toggled to "ON" - <RTN>
                                              to continue');
        normvid;
        readln;
      end;
  net_description.iterate := n;
  if n = 0
    then net_description.save_state := false;
end;   {Save_Freq_Option}

{-----------------------------------------------------------}

Procedure Set_Input_Layer;

  var
   i : integer;

begin    {Set_Input_Layer}
  with layers[1] do
    begin
      for i := 1 to neurons do
        begin
          neural_set.binary_value[i] := In_Buffer[i];
          if neural_set.binary_value[i] = 1
            then neural_set.real_value[i] :=
                                   net_description.one_value
            else neural_set.real_value[i] :=
                                   net_description.zero_value;
        end;
    end;   {with layers[1]}
end;     {Set_Input_Layer}

{-----------------------------------------------------------}

Procedure ReadTrainFile;

var
  i, x : integer;

begin   {ReadTrainFile}
  if eof (infile)
    then reset (infile);   {If at the end then start over}
  for i := 1 to layers[1].neurons do
    begin
      read (infile, x);
      In_Buffer [i] := x;
    end;
  readln (infile); {Advance to next line - training values}
  for i := 1 to layers [net_description.num_layers].neurons
                                                          do
    begin
      read (infile, x);
      Out_Buffer [i] := x;
      if x = 1
        then r_out_buffer[i] := net_description.one_value
        else r_out_buffer[i] := net_description.zero_value;
    end;
  readln (infile);   {Advance to next input line}
end;    {ReadTrainFile}

{----------------------------------------------------------}

Procedure ReadFile;

var
  i, x : integer;

begin   {readfile}
  if eof (infile)
    then reset (infile);   {If at the end then start over}
  for i := 1 to layers[1].neurons do
    begin
      read (infile, x);
      In_Buffer [i] := x;
    end;
  readln (infile);   {Advance to next line}
  readln (infile);   {Skip training vector pair}
end;    {readfile}

{-----------------------------------------------------------}

Procedure Next_Input;{Sets buffers to training set values}
          {if not a training scenario - out_buffer values}
                        {will be ignored}
var
  i : integer;

begin   {Next_Input}
  if Train                                    {Training Mode}
    then
      case net_description.training of
```

```
        unsupervised : ReadFile;
        supervised   : ReadTrainFile;
        zip          : ReadFile;
      end  {case}
    else                                                {Run Mode}
      if net_description.learning = hopfield
        then
          begin
            if not eof (infile)    {Do not read input except}
              then readfile        {on first iteration with}
                                   {Hofield Net}
              else exit;           {Do not set input layer}
          end
        else ReadFile; {Run mode and not Hopfield - Sup or}
                          {Unsup training}
  Set_Input_Layer;           {What about recurrent nets????}
end;    {Next_Input}

{--------------------------------------------------------}

end.  {ioutil}
```

APPENDIX H

Development System Math Unit Listing
(Unit Math)

Appendix B through Appendix J constitute the complete development system program listing

```
{-----------------------------------------------------------}

{This unit (Math) includes all the matrix and vector}
{algebra routines, neuron activation, and learning}
{algorithms.}

{-----------------------------------------------------------}

unit math;

interface

uses nnglobal, crt, ioutil;

Procedure Calculate_Output;
Procedure Learning_Rule;
Procedure Corrupt_input (prob : real);
Procedure Remove_Random_Neurons (prob : real; layer :
                                                    integer);
Procedure Remove_Random_Connections (prob : real; layer :
                                                    integer);
Procedure Get_Removal_Probability (var prob : real);
Procedure Train_Sup_Net;
Procedure Train_Unsup_Net (var ch : char);
Procedure Calc_Sup_Net;
Procedure Calc_Unsup_Net;

Implementation

{-----------------------------------------------------------}

Procedure Error_Routine;

begin   {Error_Routine}
    begin
      writeln ('Error occurred'); {Needs more functionality}
      readln;
    end;
end;    {Error_Routine}

{----------------------------------------------------------}

Procedure Delta_Calculation (n : real);

var
  delta      : real;
  correction : real;
  i, k, j    : integer;
  x, y       : integer;
```

```
begin   {Delta_Calculation}             {Make work for
multiple layers!!!!}
  with layers[net_description.num_layers] do  {output layer}
    begin
      for i := 1 to neurons do
        begin
          delta := r_out_buffer[i] -
                                   neural_set.real_value[i];
          for j := 1 to layers[1].neurons do
            begin
              Correction := n * delta *
                          layers[1].neural_set.real_value[j];
              weights^.value[i, j] := weights^.value[i, j] +
                                                 correction;
            end;
        end;
    end;
end;    {Delta_Calculation}

{-----------------------------------------------------------}

Procedure Corrupt_input (prob : real);

var
  i, x : integer;

begin {Corrupt_input}
  randomize;
  with layers[1] do
    begin
      if net_description.learning = Hopfield
        then
          begin
            reset (infile);
            for i := 1 to neurons do
              begin
                read (infile, x);
                neural_set.binary_value[i] := x;
              end;
            readln (infile);
          end;
      for i := 1 to neurons do
        begin
          x := neural_set.binary_value[i];
          if random < prob    {reverse values}
            then
              if x = 1
                then x := 0
                else x := 1;
          neural_set.binary_value[i] := x;  {set value to
                                             input file value}
          if x = 1
            then neural_set.real_value[i] :=
                                     net_description.one_value
            else neural_set.real_value[i] :=
                                   net_description.zero_value;
          if net_description.learning = hopfield
            then
              begin   {Force output to input values for
                       initialization}
                layers[2].neural_set.binary_value[i] :=
                          neural_set.binary_value[i];
                layers[2].neural_set.real_value[i] :=
                            neural_set.real_value[i];
              end;
        end;
    end;   {with layers..}
end;  {Corrupt_input}

{----------------------------------------------------------}

Procedure Remove_Random_Neurons (prob : real; layer :
                                                  integer);

var
  i, j : integer;

begin      {Remove_Random_Neurons}
  randomize;
  for j := 1 to layers[layer].neurons do
    begin
      if random < prob    {remove neuron}
        then
          begin
            if layer < net_description.num_layers
              then
                for i := 1 to layers[layer + 1].neurons do
                  layers[layer + 1].weights^.value[i, j] :=
                                                         0.0;
            for i := 1 to layers[layer - 1].neurons do
              layers[layer].weights^.value[j, i] := 0.0;
          end;
    end;
end;     {Remove_Random_Neurons}

{---------------------------------------------------------}
```

```
Procedure Remove_Random_Connections (prob : real; layer :
                                                  integer);

var
  i, j      : integer;
  connected : boolean;

begin   {Remove_Random_Connections}
  connected := true;
  if layers[layer + 1].saved_weights <> nil  {restore
                                        weights if possible}
    then layers[layer + 1].weights^:= layers[layer +
                                          1].saved_weights^;
  randomize;
  for j := 1 to layers[layer].neurons do
    for i := 1 to layers[layer + 1].neurons do
      if random < prob    {remove connection}
        then
          begin
            layers[layer + 1].weights^.value[i, j] := 0.0;
            connected := false;
          end;
  if not connected   {For network display}
    then layers[layer].FFConnection_kind := partial
    else layers[layer].FFConnection_kind := fully;
end;    {Remove_Random_Connections}

{-----------------------------------------------------------}

Procedure Get_Removal_Probability (var prob : real);

  var
    result : integer;

begin    {Get_Removal_Probability}
  repeat
    clrscr;
    gotoxy (20, 8);
    write ('Enter Neuron Removal Probability (', prob:2:2,
                                                     ')');
    gotoxy (1, 21);
    write ('=> ');
    {$I-};
    readln (prob);   {Fix for rtn or esc!!!!}
    result := ioresult;
    {$I+}
    {Check for real number and correct probability}
    if (result <> 0) or ((Prob > 1) or (Prob < 0)) {Not
                                                correct value}
      then
        begin
          gotoxy (1, 1);
          reverse;
          write ('Illegal real number or probability');
          normvid;
          gotoxy (4, 23);
          write ('<RTN> to Continue');
          repeat
          until keypressed;
          gotoxy (1, 23);
          clreol;
        end;
  until result = 0;
end;    {Get_Removal_Probability}

{-----------------------------------------------------------}

Procedure Back_Propogation (n : real);

var
  delta, delta1, delta2, delta3 : real;
  correction, m_correction      : real;
  i, x, y, j, k                 : integer;
  {M                             : weight_matrix;}
  vector                        : DeltaVectorType;
  origmode                      : word;

{------------------------------------------}

{Matrix_Transpose changes rows of matrix to columns of the}
{output matrix     }

Procedure Weight_Matrix_Transpose (k : layerange);

var
  c    : integer;
  r    : integer;

begin   {Weight_Matrix_Transpose}
  for r := 1 to layers[k].saved_weights^.row_length do
                                                {Transpose rows}
    for c := 1 to layers[k].saved_weights^.col_length do
                                                {to columns}
      M.value [r, c] := layers[k].saved_weights^.value
                                                        [c, r];
  M.col_length := layers[k].saved_weights^.row_length;
  M.row_length := layers[k].saved_weights^.col_length;
end;    {Weight_Matrix_Transpose}

{--------------------------------------------}

Function Delta_Dot_Product (Vec : DeltaVectorType; c, layer
                                            : integer) : real;

var
  hold : real;
  j    : integer;

begin  {Delta_Dot_Product}
  hold := 0.0;
  for j := 1 to layers[layer].neurons do
    hold := hold + (vec[j] * M.value[c, j]);
  Delta_Dot_Product := hold;
end;   {Delta_Dot_Product}

{------------------------------------------}

Procedure Matrix_Vector_Multiply (vec : DeltaVectorType;
       var vector : DeltaVectorType; layer : layerange);
var
  c : integer;

begin   {Weight_Matrix_Multiply}
  for c := 1 to layers[layer-1].neurons do
    vector[c] := Delta_Dot_Product (vec, c, layer);
end;    {Weight_Matrix_Multiply}

{-----------------------------------------}

begin    {Back_Propogation}
  with layers[net_description.num_layers] do {output Layer}
    begin
      M := saved_weights^; {M = weights at time T-1};
      saved_weights^ := weights^;  {set saved_weights to
                                               time T}
      for i := 1 to neurons do
        if abs (r_out_buffer[i] - neural_set.real_value[i])
           >= net_description.factor1  {Train only if out of
                                               tolerance}
          then
            begin
              delta3 := r_out_buffer[i] -
                       neural_set.real_value[i];   {tgt-out}
            if net_description.activation = mod_sigmoid
              then delta1 := 0.5 +
                          neural_set.real_value[i]    {out}
              else delta1 := neural_set.real_value[i];
            delta2 := 1 - delta1;  {1 - out}
            delta := delta1 * delta2 * delta3; {out(1 -
                                         out)(tgt - out)}
            layers[net_description.num_layers].
                                  delta_vector[i] := delta;
            for j := 1 to layers[net_description.
                                 num_layers - 1].neurons do
              begin
                if weights^.value[i, j] = 0.0    {not
                                                 connected}
                  then
                    begin    {do nothing}
                    end
                  else
                    if (delta1 * delta2) >=
                                  net_description.factor2
                      then    {neuron not saturated}
                        begin
                          m_correction :=
                                net_description.momentum *
                                    (weights^.value[i, j]
                                         - M.value[i, j]);
{-  momentum correction = α (weight (t) - weight (t-1))-}
                          Correction := (n * delta *
                                  layers[net_description.
   num_layers-1].neural_set.real_value[j]) + m_correction;
                          weights^.value[i, j] :=
                        weights^.value[i, j] + correction;
                        end
                      else weights^.value[i, j] := (-5 +
                        10/(1 + exp (-weights^.value[i,
                                                j]/5)));
      {Out approaching saturation - desaturate the neuron}
               end;
           end
         else
           layers[net_description.num_layers].
                                  delta_vector[i] := 0.0;
  end;     {with}
for k := (net_description.num_layers) downto 3 do
  begin
    Weight_Matrix_Transpose (k);
    matrix_vector_multiply (layers[k].delta_vector,
                                              vector, k);
    M := layers[k-1].saved_weights^;{weights at time t-1}
```

```
      layers[k-1].saved_weights^ := layers[k-1].weights^;
                                                   {weights at t}
      for i := 1 to layers[k-1].neurons do
        begin
          if net_description.activation = mod_sigmoid
            then delta1 :=  0.5 + layers[k-1].neural_set.
                                                  real_value[i]
            else delta1 := layers[k-1].neural_set.
                                            real_value[i]; {out}
          delta2 := 1 - delta1;    {1- out}
          delta := delta1 * delta2 * vector[i];
                                             {(out)(1-out)(Σδw)}
          layers[k-1].delta_vector[i] := delta;
          for j := 1 to layers[k-2].neurons do
            begin
              if layers[k-1].weights^.value[i, j] = 0.0
                                                 {not connected}
                then
                  begin   {do nothing}
                  end
                else
                  if (delta1 * delta2) >=
                                      net_description.factor2
                    then
                      begin
                         m_correction :=
                               net_description.momentum *
                            (layers[k-1].weights^.value[i,
                                        j] - M.value[i, j]);
                         correction := (n * delta *
                               layers[k-2].neural_set.
                               real_value[j])+ m_correction;
                         layers[k-1].weights^.value[i, j] :=
                               layers[k-1].weights^.
                                   value[i, j] + correction;
                      end
                    else layers[k-1].weights^.value[i, j] :=
                          (-5 + 10/(1 + exp (-layers[k-1].
                                  weights^.value[i, j]/5)));
    {Out approaching saturation - desaturate the neuron}
            end;
        end;
    end;
end;     {Back_Propogation}

{------------------------------------------------------------}
```

```
Procedure Signal_Hebb;

var
  i, j, k          : integer;
  delta1, delta2 : real;
  saturation       : real;

begin    {signal_hebb}
  for i := 2 to net_description.num_layers do
    begin
      for j := 1 to layers[i].neurons do
        begin
          if net_description.activation = mod_sigmoid
            then delta1 :=  0.5 +
                       layers[i].neural_set.real_value[j]
            else delta1 := layers[i].neural_set.
                                    real_value[j];{out}
          delta2 := 1 - delta1;  {1- out}
          Saturation := (delta1 * delta2);
          if saturation < net_description.factor2
                                       {de-saturate neuron}
            then
              for k := 1 to layers[i-1].neurons do
                layers[i].weights^.value[j, k] := (-5 +
                          10/(1 + exp (-layers[i].weights^.
                                           value[j, k]/5)))
            else for k := 1 to layers[i-1].neurons do
              begin
                if ((layers[i].neural_set.real_value[j] >=
                        net_description.threshold) and
                   (layers[i-1].neural_set.real_value[k] >=
                        net_description.threshold))
                  then
                    begin
                      if (layers[i].neural_set.real_value[j]
                         < 0) and (layers[i-1].neural_set.
                                  real_value[k] < 0)
                        then
                          layers[i].weights^.value[j, k] :=
                           layers[i].weights^. value[j, k] -
                           (layers[i].neural_set.
                                real_value[j] * layers[i-1].
                                    neural_set.real_value[k])
                        else
                          layers[i].weights^.value[j, k] :=
                           layers[i].weights^.value[j, k] +
                           (layers[i].neural_set.
                                real_value[j] * layers[i-1].
                                  neural_set.real_value[k]);
                    end
                  else
                    layers[i].weights^.value[j, k] :=
                          layers[i].weights^.value[j, k] -
                     (layers[i].neural_set.real_value[j] *
                     layers[i-1].neural_set.real_value[k])
              end;
       end;
    end;
end;     {signal_hebb}

{-----------------------------------------------------------}

Procedure user_defined;

begin      {other}
end;       {other}

{-----------------------------------------------------------}

Procedure Madline;

begin     {Madline}
end;      {Madline}

{-----------------------------------------------------------}

Procedure Adline;

begin      {Adline}
end;       {Adline}

{-----------------------------------------------------------}

Procedure Learning_Rule;

begin  {Learning_Rule}
  case net_description.learning of
    delta: Delta_Calculation (net_description.factor);
                                        {gain factor param}
    backp: Back_Propogation (net_description.factor);
    Hebb : Signal_Hebb;
    Other: User_defined;
    madaline : madline;
    adaline : adline;
  end;   {case}
end;   {Learning_Rule}

{-----------------------------------------------------------}
```

```
Procedure Calculate_Output;

type
  SumType = record
              value  : packed array [1..max_bodies] of real;
              length : integer;
            end;

var
  layer, i, n, y    : integer;
  vector, sum, out : SumType;
  origmode          : word;

{-------------------------------------------}

Function Weight_Dot_Product (Vector : SumType; c, i :
integer) : real;

var
  hold : real;
  j    : integer;

begin  {Weight_Dot_Product}
  hold := 0.0;
  for j := 1 to vector.length do
    begin
      hold := hold + (vector.value[j] *
                      layers[i].weights^.value[c, j]);
    end;
  Weight_Dot_Product := hold;
end;   {Weight_Dot_Product}

{------------------------------------------}

Procedure Weight_Matrix_Multiply (vector : SumType; layer :
           layerange; var sum : SumType);
var
  c : integer;

begin   {Weight_Matrix_Multiply}
  for c := 1 to layers[layer].neurons do
    sum.value[c] := Weight_Dot_Product (vector, c, layer);
  sum.length := layers[layer].neurons;
end;    {Weight_Matrix_Multiply}

{-----------------------------------------}
```

```
Procedure Calculate_Activation (sum : SumType; layer :
                              layerange; var out : SumType);

var
  i : integer;

begin    {Calculate_Activation}
  case net_description.activation of
    Threshold  : begin
                   for i := 1 to sum.length do
                     begin
                       if sum.value[i] >
                               net_description.threshold
                         then
                           begin  {fires if above threshold}
                             out.value[i] :=
                                  net_description.one_value;
                             layers[layer].neural_set.
                                       binary_value[i] := 1;
                             layers[layer].neural_set.
                                 real_value[i] :=
                                  net_description.one_value;
                           end
                         else
                           begin
                             out.value[i] :=
                                 net_description.zero_value;
                             layers[layer].neural_set.
                                       binary_value[i] := 0;
                             layers[layer].neural_set.
                                         real_value[i] :=
                                 net_description.zero_value;
                           end;
                     end;
                 end;
    Sigmoid    : begin
                   for i := 1 to sum.length do
                     begin
                       out.value[i] := (1/(1 + exp
                                    (-sum.value[i])));
                       if out.value[i] >=
                                net_description.threshold
                         then
                           layers[layer].neural_set.
                                        binary_value[i] := 1
                         else
                           layers[layer].neural_set.
                                       binary_value[i] := 0;
                       layers[layer].neural_set.
```

```
                                    real_value[i] := out.value[i];
                           end;
                       end;
    Mod_Sigmoid : begin
                        for i := 1 to sum.length do
                          begin
                           out.value[i] := -0.5 + (1/(1 + exp
                                                   (-sum.value[i])));
                           if out.value[i] >=
                                       net_description.threshold
                             then
                               layers[layer].neural_set.
                                            binary_value[i] := 1
                             else
                               layers[layer].neural_set.
                                           binary_value[i] := 0;
                           layers[layer].neural_set.
                                  real_value[i] := out.value[i];
                          end;
                      end;
    HypTangent : begin
                     end;
  end;   {case}
  out.length := sum.length;
end;     {Calculate_Activation}

{-----------------------------------------}

begin   {Calculate_Output}
  with layers[1] do
    begin
      for i := 1 to neurons do
        vector.value[i] := neural_set.real_value[i];
        {vector.value[neurons + 1] := 1;  {bias input}
      vector.length := neurons;
    end;   {with layers[1]}
  for layer := 2 to net_description.num_layers do
    begin
      Weight_Matrix_Multiply (vector, layer, sum);
      Calculate_Activation (sum, layer, out);
      vector := out;   {Out becomes input for next layer
                                              calculations}
    end;
end;    {Calculate_Output}

{-----------------------------------------------------------}
```

```
Procedure Calc_Sup_Recur_Net;

begin {Calc_Recur_Net}
end;  {Calc_Recur_Net}

{----------------------------------------------------------}

Procedure Calc_Sup_Net;

begin    {Calc_Sup_Net}
  if net_description.net_arch = feedforward
    then
      begin
        next_input;           {ioutil unit}
        calculate_output;     {math unit}
      end
    else calc_sup_recur_net;
end;     {Calc_Sup_Net}

{---------------------------------------------------------}

Procedure Calc_Unsup_Recur_Net;

var
  converge : boolean;
  i        : integer;

begin   {Calc_Unsup_Recur_Net}
  if net_description.learning = Hopfield
    then
      begin
        next_input;    {ioutil}
        Converge := true;
        Calculate_Output;   {math Unit}
        for i := 1 to
              layers[net_description.num_layers].neurons do
          begin
            if layers[net_description.num_layers].
               neural_set.binary_value[i]
               <> layers[1].neural_set.binary_value[i]
              then converge := false;
            layers[1].neural_set.real_value[i] :=
              layers[net_description.num_layers].
                                   neural_set.real_value[i];
            layers[1].neural_set.binary_value[i] :=
              layers[net_description.num_layers].
                                 neural_set.binary_value[i];
          end;   {for i := 1 to ...}
        if converge
          then
            begin
              gotoxy (1, 1);
              write ('Net Convergence');
              delay (1000);
            end;
      end;
end;    {Calc_Unsup_Recur_Net}

{------------------------------------------------------------}

Procedure Calc_Unsup_Net;

begin   {Calc_Unsup_Net}
  if net_description.net_arch = feedforward
    then
      begin
        next_input;          {ioutil unit}
        calculate_output;    {math unit}
      end
    else calc_unsup_recur_net;
end;    {Calc_Unsup_Net}

{-----------------------------------------------------------}

Procedure Train_Unsup_Recur_Net (var ch : char);

var
  i, j : integer;

begin    {Train_Unsup_Recur_Net}
  case net_description.learning of
    Hopfield : begin
                 for i := 1 to layers[2].neurons do
                   for j := 1 to layers[1].neurons do
                     layers[2].weights^.value[i, j] := 0;
                   layers[2].weights^.col_length :=
                                         layers[2].neurons;
                 layers[2].weights^.row_length :=
                                         layers[1].neurons;
                 reset (infile);
                 gotoxy (1, 1);
                 write ('Weights Initializing..');
                 while not eof (infile) do
                   begin
                     next_input;          {ioutil unit}
                     for i := 1 to layers[2].neurons do
                       for j := 1 to layers[1].neurons do
```

```
                          if i <> j
                            then layers[2].weights^.value[i,
                              j] := layers[2].weights^.
                               value[i, j] + (layers[1].
                                 neural_set.real_value[i] *
                                  layers[1].neural_set.
                                              real_value[j]);
                      write ('*');
                    end;
                  close (infile);
                  infilename := 'NONE SELECTED';
                  gotoxy (1, 2);
                  write ('Hopfield Net Initialized');
                  repeat
                  until keypressed;
                  gotoxy (1, 1);
                  clreol;
                  gotoxy (1, 2);
                  clreol;
                  ch := #27;   {<ESC> out of net output}
                end;
    other     : begin    {Provisions for user definition}
                end;
  end;                   {case}
end;     {Train_Unsup_Recur_Net}

{--------------------------------------------------------}

Procedure Train_Sup_Recur_Net;

begin    { Train_Sup_Recur_Net}
  case net_description.learning of
    other : begin
            end;
  end;    {case}
end;     { Train_Sup_Recur_Net}

{--------------------------------------------------------}

Procedure Train_Unsup_Net (var ch : char);

var
  i, j, num, x : integer;

begin    {Train_Unsup_Net}
  if net_description.net_arch = feedforward
    then
      begin
        Calc_Unsup_Net;
        Learning_Rule;     {Math unit}
      end
     else train_unsup_recur_net (ch);
end;     {Train_Unsup_Net}

{---------------------------------------------------------}

Procedure Train_Sup_Net;

begin   {Train_Sup_Net}
  if net_description.net_arch = feedforward
    then
      begin
        Calc_Sup_Net;
        Learning_Rule;     {Math unit}
      end
     else train_sup_recur_net;
end;    {Train_Sup_Net}

{----------------------------------------------------------}

end.   {Math Unit}
```

APPENDIX I

Development System Network Operations Unit Listing (Unit Ops_Stuf)

Appendix B through Appendix J constitute the complete development system program listing

```
{------------------------------------------------------------}

{Unit Op_stuff provides the transition entry point from the}
{menu system to the display (graphics) mode via either the}
{Training Mode or the Run Mode.}

{-----------------------------------------------------------}

Unit Ops_Stuf;

Interface

uses crt, graph, nnglobal, grafstuf, mathutil;

  Procedure Training (var Flag : Char);
  Procedure Run_Program (var Flag : Char);

Implementation

{------------------------------------------------------}

{Entry to display via training}

Procedure Training (var Flag : Char);

var
  ch        : char;
  semaphore : integer;
  i         : integer;
  f         : integer;
  check     : boolean;

begin   {Training}
  {$I-}
  check := eof (infile);                {check for open file}
  {$I+}
  f := ioresult;
  if f <> 0
    then
      begin
        clrscr;
        gotoxy (1, 1);
        reverse;
        write ('Input File Not Open!! - Hit any key to
                                               continue');
        normvid;
        repeat
        until keypressed;
        exit;   {return to program operations menu}
      end;
  SetGraphMode (graphmode);
  ch := '1';                                             {Initialize}
  if net_description.learning = backp
    then Init_Saved_Weights;                          {mathutil unit}
  semaphore := 2;
  repeat
    case semaphore of
      1 : Display_Memory (ch, Semaphore);          {In grafstuf}
      2 : Display_Net_Output (ch, Semaphore);      {In grafstuf}
      3 : begin
            Net_Display (9);                        {In grafstuf}
            semaphore := 2;    {Came from Display_Net_Output}
          end;
      4 : begin
            Net_Display (8);                        {In grafstuf}
            semaphore := 1;        {Came from Display_Memory}
          end;
    end;   {case}
  until ch = #27;                          {<ESC> for Terminate}
  RestoreCrtMode;
end;    {Training}

{-----------------------------------------------------------}

{Entry to display via run mode}

Procedure Run_Program (var Flag : Char);

var
  ch          : char;
  Semaphore : integer;
  f, i, x    : integer;
  check       : boolean;

begin   {Run_Program}
  {$I-}
  check := eof (infile);                   {check for open file}
  {$I+}
  f := ioresult;
  if f <> 0
    then
      begin
        clrscr;
        gotoxy (1, 1);
        reverse;
        write ('Input File Not Open!! - Hit any key to
                                                   continue');
        normvid;
        repeat
        until keypressed;
        exit;     {return to program operations menu}
      end;
  if net_description.learning = hopfield
    then reset (infile);  {reset for hopfield special case}
  SetGraphMode (graphmode);
  ch := '1';                                   {Initialize}
  semaphore := 2;
  repeat
    case semaphore of
      1 : Display_Memory (ch, Semaphore);     {In grafstuf}
      2 : Display_Net_Output (ch, Semaphore); {In grafstuf}
      3 : begin
            Net_Display (9);                  {In grafstuf}
            semaphore := 2;  {Came from Display_Net_Output}
          end;
      4 : begin
            Net_Display (8);                  {In grafstuf}
            semaphore := 1;      {Came from Display_Memory}
          end;
    end;   {case}
  until ch = #27;                     {<ESC> for Terminate}
  RestoreCrtMode;
end;    {Run_Program}

{-----------------------------------------------------------}

end.   {Ops_Stuf}
```

APPENDIX J

Development System Math Utility Unit Listing (Unit MathUtil)

Appendix B through Appendix J constitute the complete development system program listing

```
{--------------------------------------------------------------}

{Unit mathutil provides various initialization routines and}
{various other utilities}

{--------------------------------------------------------------}

Unit MathUtil;

Interface

uses
  nnglobal, crt, graph;

Procedure Initialize_Weights;
Procedure Init_Saved_Weights;
Procedure Initialize_stuff;
Procedure Initialize_Layers;
Procedure Repeat_Input (var ch : char);
Procedure GetChar;
Procedure GetLine;
Procedure Process_Number (var int: integer);
Procedure Get_Network_Name (var S : string);
Function HeapFunc (size : word) : integer;

Implementation

{--------------------------------------------------------------}

{F+}
Function HeapFunc (size : word) : integer;

begin
  gotoxy (1, 12);
  write ('Error in heap allocation of ', size, ' Hit any key
                                                   to continue');
  HeapFunc := 1;
  repeat
  until keypressed;
end;
{$F-}

{---------------------------------------------------------------}
```

```
Procedure Initialize_Weights;

var
  i, j, k : integer;

begin   {Initialize_Weights}
  heapError := @heapfunc;
  for i := 2 to net_description.num_layers do
    begin
      if layers[i].weights = nil
        then new (layers [i].weights);
      if layers[i].weights = nil
        then exit;    {Heap overflow}
      for j := 1 to max_bodies do
        for k := 1 to max_bodies do
         layers[i].weights^.value[j, k] := 0.0;
      layers[i].weights^.col_length := layers[i].neurons;
                                                   {Reset}
      layers[i].weights^.row_length := layers[i-1].neurons;
    end;
  if max_layers > net_description.num_layers
    then for i := max_layers downto
                        (net_description.num_layers + 1) do
      begin
        if layers[i].weights <> nil
          then
            begin
              dispose (layers[i].weights);
              layers[i].weights := nil;
            end;
        if layers[i].saved_weights <> nil
          then
            begin
              dispose (layers[i].saved_weights);
              layers[i].saved_weights := nil;
            end;
      end;
  net_description.set_weights := true;
  net_description.random_weights := false;
end;    {Initialize_Weights}

{------------------------------------------------------------}
```

```
Procedure Init_Saved_Weights;

var
  i, j, k : integer;

begin    {Init_Saved_Weights}
  heapError := @heapfunc;
  for i := 2 to net_description.num_layers do
    begin
      if layers[i].saved_weights = nil
        then new (layers [i].saved_weights);
      if layers[i].saved_weights = nil
        then exit;    {Heap overflow}
      for j := 1 to max_bodies do
        for k := 1 to max_bodies do
          layers[i].saved_weights^.value[j, k] := 0.0;
      layers[i].saved_weights^ := layers[i].weights^;
      end;
  if max_layers > net_description.num_layers
    then for i := max_layers downto
                    (net_description.num_layers + 1) do
      begin
        if layers[i].saved_weights <> nil
          then
            begin
              dispose (layers[i].saved_weights);
              layers[i].saved_weights := nil;
            end;
      end;
end;     {Init_Saved_Weights}

{------------------------------------------------------------}

Procedure Initialize_Layers;

var
  i, j : integer;

begin  {Initialize_Layers}
  num_layers := 3;       {Default value}
  arch_set := num_layers;
  neuron_set := num_layers;
  for i := 1 to max_layers do
    begin
      layers[i].Arch := lines;
      layers[i].neurons := 5;
      layers[i].FFconnection_kind := fully;  {Connection to}
                                             {next layer}
      layers[i].out_layer := false;
      layers[i].arch_row_length := 1;
      layers[i].arch_col_length := 1;
      layers[i].weights := nil;
      layers[i].saved_weights := nil;
      layers[i].connections := nil;
      for j := 1 to max_bodies do
        begin
          layers[i].neural_set.binary_value[j] := 0;
                                    {neurons set to binary 0}
          layers[i].neural_set.threshold[j] := 0.0; {Neuron}
                                                    {threshold}
          layers[i].neural_set.rate := 0;
        end;
    end;
  Net_Description.num_layers := num_layers;  {Set default}
                                               {values}
  Net_Description.net_name := 'Name ?';
  Net_Description.Learning := backp;
  Net_Description.Training := supervised;
  Net_Description.factor := 0.7;       {Gain factor}
  Net_Description.factor1 := 0.0;      {Tolerance}
  Net_Description.factor2 := 0.0;      {Saturation}
  Net_description.factor3 := 0.0;   {Corruption Probability}
  Net_Description.Activation := Sigmoid;
  Net_Description.threshold := 0.5;
  Net_Description.momentum := 0.7;
  Net_Description.net_arch := feedforward;
  Net_Description.random_weights := false;
  Net_Description.set_weights := false;
  Net_description.iterate := 0;
  Net_description.save_state := false;
  Net_Description.one_value := 1.0;
  Net_Description.zero_value := 0.0;
  layers[num_layers].out_layer := true;   {Set final layer}
                                            {to true}
end;   {Initialize_Layers}

{-----------------------------------------------------------}

Procedure Reverse;

begin
  textbackground (white);
  textcolor (black);
end;

{-----------------------------------------------------------}
```

```
Procedure Repeat_Input (var ch : char);

begin
  write (#7);  {Ring bell - illegal entry}
  gotoxy (4, 21);
  clreol;    {Clear from cursor to end of line}
  ch := readkey;
end;

{-----------------------------------------------------------}

Procedure Initialize_stuff;

begin {Initialize_stuff}
  Infilename := 'NONE SELECTED';
end;  {Initialize_stuff}

{-----------------------------------------------------------}

Procedure FullPort;

begin  {FullPort}
  SetViewPort (0, 0, MaxX, MaxY, ClipOn);
end;   {FullPort}

{-----------------------------------------------------------}

Procedure GetChar;

begin      {GetChar}
  if ColNo > ColsOnLine
    then EndOfLine := true
    else
      begin
        NextChar := Lin [ColNo];
        ColNo := ColNo + 1;
      end;
end;       {GetChar}

{-------------------------------------------------------}

Procedure GetLine;

var
  i : integer;

begin   {GetLine}
  i := 1;
  ColNo := 1;
  ColsOnLine := 0;
  readln (S);    {Read input from console.  Place in S}
  if length (s) <> 0 then    {if no input will not return}
                                            {value}
    begin
      while i <= length (S) do
        begin
          ColsOnLine := ColsOnLine + 1;
          Lin [ColsOnLine] := S[i];   {Transfer S to buffer}
          i := i + 1;
        end;
      NextChar := ' ';   {Prime the pump if valid line}
      EndOfLine := false;
      while (nextchar = ' ') and (not EndofLine) do
        GetChar;     {Skip initial blanks - make nextchar}
                              {1st valid int}
    end;
end;    {GetLine}

{-----------------------------------------------------------}

{Process_Number reads characters from the input buffer and}
 {converts them to integers.}

Procedure Process_Number (var int : integer);

begin    {Process_Number}
  int := ord (nextchar) - ord ('0');
  getchar;
  while ((nextchar in ['0'..'9']) and (not EndofLine)) do
    begin
      int := int * 10 + (ord (nextchar) - ord ('0'));
      getchar;
    end;
end;     {Process_Number}

{---------------------------------------------------------}

Procedure Get_Network_Name (var S : string);

begin    {Get_Network_Name;} {Put some more checks in this!!}
  S := '';
  clrscr;
  gotoxy (20, 15);
  write ('Enter Network MsDos File or Path Name');
  gotoxy (37-(length (net_description.net_name) Div 2), 17);
  write ('(', net_description.net_name, ')');
  gotoxy (5, 24);
  write ('<RTN> Accept');
  gotoxy (1, 21);
  write ('=> ');
  Getline;
  while (not EndofLine) do
    begin
      if nextchar in ['a'..'z']
        then nextchar := chr (ord (nextchar) - 32);
      S := concat (S, NextChar);
      getchar;
    end;
  if length (S) = 0
    then S := net_description.net_name;
end;    {Get_Network_Name;}

{------------------------------------------------------------}
end.
```

APPENDIX K

Network Saved State Data Printout

```
NET DESCRIPTION

ALFA1
Net Architecture 0
3  layers
Activation Type 1
Threshold 0.00
Learning Type 4
Factor - 0.700 Factor1 - 0.495 Factor2  - 0.001,
  Factor3 - 0.000
Momentum 0.70
Input values 0.5 / -0.5

LAYER 1
  Connection type 0
  Number of Neurons 35
  Layer threshold 0.50
LAYER 2
  Connection type 0
  Number of Neurons 80
  Layer threshold 0.00
LAYER 3
  Connection type 0
  Number of Neurons 8
  Layer threshold 0.00

ITERATION NUMBER 0
Order is - Binary value, Real value, Threshold, Rate

LAYER 1

    1.  1 0.5000 0.50 0
    2.  1 0.5000 0.50 0
    3.  1 0.5000 0.50 0
    4.  1 0.5000 0.50 0
    5.  1 0.5000 0.50 0
    6.  1 0.5000 0.50 0
    7.  1 0.5000 0.50 0
    8.  1 0.5000 0.50 0
    9.  0 -0.5000 0.50 0
   10.  0 -0.5000 0.50 0
   11.  1 0.5000 0.50 0
   12.  0 -0.5000 0.50 0
   13.  0 -0.5000 0.50 0
   14.  0 -0.5000 0.50 0
   15.  1 0.5000 0.50 0
   16.  0 -0.5000 0.50 0
   17.  0 -0.5000 0.50 0
   18.  1 0.5000 0.50 0
   19.  0 -0.5000 0.50 0
   20.  0 -0.5000 0.50 0
   21.  0 -0.5000 0.50 0
   22.  1 0.5000 0.50 0
   23.  0 -0.5000 0.50 0
   24.  0 -0.5000 0.50 0
   25.  1 0.5000 0.50 0
   26.  0 -0.5000 0.50 0
   27.  0 -0.5000 0.50 0
   28.  0 -0.5000 0.50 0
```

```
 29.  1 0.5000 0.50 0
 30.  1 0.5000 0.50 0
 31.  1 0.5000 0.50 0
 32.  1 0.5000 0.50 0
 33.  1 0.5000 0.50 0
 34.  1 0.5000 0.50 0
 35.  1 0.5000 0.50 0

LAYER 2

  1.  0 -0.2733 0.00 0
  2.  0 -0.2979 0.00 0
  3.  1 0.2790 0.00 0
  4.  1 0.4607 0.00 0
  5.  0 -0.0970 0.00 0
  6.  0 -0.4094 0.00 0
  7.  0 -0.2226 0.00 0
  8.  1 0.3160 0.00 0
  9.  0 -0.4954 0.00 0
 10.  0 -0.3544 0.00 0
 11.  0 -0.0316 0.00 0
 12.  1 0.4040 0.00 0
 13.  1 0.2550 0.00 0
 14.  0 -0.1614 0.00 0
 15.  0 -0.4256 0.00 0
 16.  0 -0.3173 0.00 0
 17.  1 0.4000 0.00 0
 18.  1 0.4446 0.00 0
 19.  1 0.2395 0.00 0
 20.  0 -0.4045 0.00 0
 21.  1 0.1182 0.00 0
 22.  1 0.1346 0.00 0
 23.  1 0.3773 0.00 0
 24.  0 -0.0204 0.00 0
 25.  1 0.3873 0.00 0
 26.  0 -0.0687 0.00 0
 27.  1 0.0609 0.00 0
 28.  1 0.4437 0.00 0
 29.  0 -0.1460 0.00 0
 30.  0 -0.4898 0.00 0
 31.  1 0.2339 0.00 0
 32.  1 0.0963 0.00 0
 33.  0 -0.2328 0.00 0
 34.  0 -0.4957 0.00 0
 35.  0 -0.1171 0.00 0
 36.  0 -0.1508 0.00 0
 37.  0 -0.4916 0.00 0
 38.  0 -0.3655 0.00 0
 39.  0 -0.4689 0.00 0
 40.  0 -0.4610 0.00 0
 41.  1 0.3558 0.00 0
 42.  1 0.1155 0.00 0
 43.  0 -0.4356 0.00 0
 44.  0 -0.4300 0.00 0
 45.  1 0.3077 0.00 0
 46.  0 -0.3825 0.00 0
 47.  1 0.2515 0.00 0
 48.  0 -0.2148 0.00 0
 49.  0 -0.4249 0.00 0
```

```
50. 1 0.2047 0.00 0
51. 1 0.0111 0.00 0
52. 1 0.4039 0.00 0
53. 0 -0.3801 0.00 0
54. 0 -0.2510 0.00 0
55. 0 -0.0914 0.00 0
56. 1 0.1600 0.00 0
57. 1 0.4629 0.00 0
58. 0 -0.1925 0.00 0
59. 0 -0.2996 0.00 0
60. 1 0.2822 0.00 0
61. 1 0.0084 0.00 0
62. 1 0.3217 0.00 0
63. 0 -0.0481 0.00 0
64. 1 0.4272 0.00 0
65. 0 -0.0371 0.00 0
66. 1 0.3771 0.00 0
67. 0 -0.4101 0.00 0
68. 0 -0.1211 0.00 0
69. 0 -0.3587 0.00 0
70. 1 0.3286 0.00 0
71. 1 0.1769 0.00 0
72. 0 -0.4924 0.00 0
73. 1 0.2806 0.00 0
74. 1 0.4852 0.00 0
75. 0 -0.2162 0.00 0
76. 0 -0.2830 0.00 0
77. 0 -0.2927 0.00 0
78. 0 -0.2851 0.00 0
79. 0 -0.4625 0.00 0
80. 0 -0.0879 0.00 0
```

Weight values for layer 2

```
1.  -1.01 -1.30 -0.17 1.34 0.29 -1.29 -0.14 -0.86 -1.29 -1.74 0.91 -0.86 0.13 -0.50 -1.68 -0.33 0.32 0.06 0.42
 -0.42 -0.04 0.05 0.53 -0.37 0.84 -0.85 -1.49 0.07 0.20 -1.83 1.24 -0.91 0.71 -1.07 -0.62 -0.06 -0.11 -0.72 -0.03
-0.02 0.26 -1.43 -0.31 -0.80 -0.47 -1.72 -1.54 -0.60 -0.20 -0.72 0.58 0.63 -0.62 -0.08 -0.53 -1.99 -0.71 -0.89
-1.19 0.58 -1.63 2.71 0.44 0.08 0.48 1.35 -0.01 -1.02 -1.05 -0.02 0.56 -0.75 -0.87 -0.13 -1.04 -1.72 0.76 -1.45
2.04 -1.43
2. -0.42 -0.77 0.27 1.37 -1.04 -1.17 0.58 -0.84 -1.24 0.64 0.41 0.52 -0.27 -1.21 -0.27 -0.48 0.98 0.89 1.37 1.00
0.68 0.57 0.53 0.10 0.93 -0.65 0.07 1.74 0.02 -0.47 -0.33 -0.58 0.84 -1.23 -0.64 0.14 -1.17 -1.31 -1.13 -0.06
0.73 -0.06 -0.66 0.68 -0.22 -1.34 0.97 -1.71 -0.82 0.36 -0.18 1.04 -0.27 0.63 -0.13 -1.41 0.72 -0.64 -0.41 -0.22
-0.40 0.20 -0.32 0.18 -0.08 1.22 -0.98 -0.75 -0.39 -0.59 -0.83 -0.31 -0.67 -0.82 -1.41 -0.97 0.14 -0.81 -0.95 -0.96
3.  0.39 0.49 -0.38 0.25 0.72 0.56 0.51 0.73 -0.92 0.01 0.59 -0.76 0.85 0.21 -0.09 -1.04 0.80 0.42 0.07 -0.75
0.14 0.33 0.06 0.58 0.57 -0.78 0.02 1.96 -0.10 -1.47 -0.92 -0.44 -0.45 -0.12 0.20 -0.06 -0.59 -0.27 -0.27 -0.31
1.61 -0.75 -0.75 1.74 -0.05 -0.51 -0.46 -1.05 -1.00 -0.02 0.91 -0.56 -0.33 -0.32 -0.34 -2.05 0.25 0.93 -1.25 1.81
-1.38 0.54 -0.08 0.01 -0.70 0.04 -0.52 1.24 -0.57 -0.25 0.43 -0.56 -0.57 0.40 -0.56 0.12 1.71 -0.97 1.57 1.01
4.  -1.71 -0.25 -0.33 0.76 -0.79 0.96 -0.18 -1.16 -1.22 -0.69 0.29 0.27 0.31 -0.55 -0.52 -0.73 0.67 0.16 1.59
-0.01 0.81 1.31 0.24 0.24 -0.21 -0.37 1.37 0.32 -0.42 0.14 0.33 0.75 -0.22 -0.58 0.15 1.06 -1.22 -1.07 -0.80
0.09 1.26 0.12 1.40 -1.02 0.05 -1.45 -0.24 0.47 -0.15 0.44 -0.12 -0.39 -0.15 0.12 -0.63 -2.01 0.98 0.25 -1.73
-0.18 -0.45 -1.26 -0.82 0.05 -0.45 0.91 -0.34 -0.96 0.15 -1.11 1.02 -0.54 -0.24 -0.97 -0.19 -0.98 0.62 -1.06 1.38
0.97
5. -0.79 -1.32 -1.69 0.56 0.63 -0.53 0.13 1.21 -1.10 -1.08 0.19 0.96 0.38 0.65 -1.71 0.92 -0.46 0.32 -0.52 0.56
0.28 0.01 0.21 -0.17 1.05 -0.44 -0.14 1.31 -0.23 -0.91 0.20 0.37 -0.52 -1.29 -0.23 -0.29 0.71 -1.23 -0.63 0.44
0.93 -0.35 -1.68 -0.15 -0.90 -1.22 -0.46 -0.44 -0.61 0.18 0.19 0.18 0.77 0.17 -0.13 -1.69 0.07 0.10 -0.62 -1.16
-0.30 -0.23 -0.93 0.29 -0.20 1.19 -0.19 1.83 0.28 -0.98 1.93 -1.55 1.73 1.65 -1.28 -0.76 0.37 -1.63 -2.96 -1.54
6. -0.01 -0.10 -1.64 -0.01 0.07 -1.08 -0.76 -0.44 -0.26 -0.34 0.04 0.28 0.73 -0.35 0.81 -0.43 0.26 0.29 0.36 0.40
0.20 0.84 0.21 0.36 1.23 -0.69 -1.59 1.31 0.67 -0.97 -0.60 1.18 -1.02 0.20 0.14 0.02 -0.40 -1.11 -0.72 -1.46
0.52 -0.28 -0.62 -1.32 -0.90 0.98 -0.55 1.16 -0.71 0.57 -1.20 0.08 -0.24 -0.42 -0.74 1.36 0.18 0.07 -1.17 -0.36
```

0.29 -0.11 -0.32 -0.71 -0.47 -0.69 0.49 -0.22 0.05 -0.74 -1.40 0.26 -0.31 -0.32 -0.99 1.19 0.64 0.16 -0.50 0.81
7. -1.49 -0.35 -1.12 0.11 0.33 -2.00 0.15 -0.51 -0.36 -0.02 -0.05 0.58 -0.32 -0.65 0.02 -0.82 0.79 0.39 0.60 0.30
0.63 0.31 0.46 0.27 0.23 -0.47 -0.32 0.95 0.54 -0.52 -0.67 -0.36 -1.80 -0.34 0.56 0.31 -0.13 -1.06 -1.04 0.18
1.84 0.01 0.76 -1.91 -1.03 0.00 0.63 0.19 -0.41 0.39 -0.23 -0.02 -0.53 0.15 0.13 1.42 0.89 0.04 -0.72 -1.35
-0.91 0.35 1.07 0.38 0.58 0.34 0.15 -0.86 -1.00 -0.31 0.72 0.21 0.93 -0.03 -0.35 -0.83 1.93 -0.70 0.73 -0.70
8. 1.49 -0.54 -0.65 -0.32 0.83 0.50 1.09 0.35 -0.60 -0.16 0.94 0.81 0.50 -0.19 1.32 -0.01 -0.77 0.60 0.94 -0.05
-0.33 0.13 0.48 1.07 -0.10 -0.62 -0.64 -0.67 0.18 -0.97 -0.46 -0.53 -1.47 -0.14 0.55 0.57 -1.21 -1.07 1.01 -1.62
0.45 -0.07 0.16 0.88 -0.72 0.31 -1.24 -0.71 -0.65 0.61 -1.03 1.24 -0.12 -0.04 -0.72 -0.09 0.12 0.37 1.49 -0.61
-1.52 0.04 0.65 0.63 -0.57 0.23 0.32 0.50 0.94 -1.21 0.45 -1.75 0.48 2.11 0.11 1.87 -2.98 0.99 2.15 -0.38
9. -0.23 -0.44 -1.41 -0.43 1.01 0.02 -0.12 -1.13 -1.00 -0.26 -0.70 0.85 0.74 -0.87 -0.08 -1.18 0.25 -0.51 0.74
0.85 0.40 0.62 0.58 0.67 0.01 -1.10 -1.25 0.60 -0.57 -0.71 -0.80 -1.09 -0.70 -0.70 0.46 0.53 -0.08 0.07 -0.62
-0.50 0.26 0.53 -1.67 -0.07 -0.87 1.01 -1.32 0.68 -0.25 -0.09 -0.18 0.30 -0.62 -0.27 -0.48 -1.27 -0.06 0.77 -0.22
-0.56 -0.57 -1.01 -0.99 -0.78 -0.22 0.01 0.84 -0.41 0.01 -1.25 1.12 -0.35 -0.24 0.92 -0.80 -0.76 0.12 -1.26 0.44
-1.31
10. -0.22 -1.05 0.38 -0.78 0.30 -0.52 0.51 0.09 0.10 -0.85 0.21 0.46 0.41 -1.01 -0.43 -0.82 0.23 0.26 0.16 0.31
0.31 1.17 -0.57 0.61 0.47 -0.76 -0.02 1.89 -0.34 -1.16 -0.92 0.35 1.51 -0.56 0.18 0.46 -0.40 -0.08 -0.07 -0.50
1.96 -0.21 1.09 -0.87 -0.73 -0.57 -1.01 -0.31 0.81 0.97 -0.95 0.12 -0.05 0.93 -0.56 -0.24 -0.13 0.87 -1.39 -0.73
0.29 1.65 -0.51 -0.77 -0.19 0.64 0.48 -0.11 -0.17 -1.04 0.09 0.43 -0.72 0.74 -0.53 0.83 2.05 -1.31 -1.79 0.06
11. -1.66 1.21 0.90 -0.68 -0.87 -0.19 0.62 -0.78 -1.29 -0.22 0.30 -0.24 -0.02 -0.46 -1.74 0.18 0.84 0.72 -0.54
0.62 0.61 1.01 0.77 0.04 0.75 -0.40 -0.38 0.60 -0.56 -0.94 -0.07 -0.42 1.66 -0.39 -0.17 0.76 -0.57 -0.89 -0.19
-0.31 0.98 0.00 -1.83 -0.80 0.91 -0.90 -0.26 -0.33 -1.27 1.02 1.40 0.18 -0.37 -0.74 -0.20 0.45 1.12 0.61 -1.27
0.97 0.41 0.44 -0.57 0.07 -0.50 0.81 0.49 1.21 1.10 -0.26 0.88 -0.80 -0.00 -1.25 -1.03 -0.35 -1.51 -1.24 0.48
-0.65
12. -0.22 -0.53 -0.60 1.08 0.33 0.23 0.93 -1.68 -0.97 -0.98 0.19 0.61 -0.10 -0.77 0.63 -0.17 0.97 0.45 0.35 0.70
0.08 1.10 0.98 0.38 -0.04 -0.38 0.56 0.65 -0.22 -0.33 -0.43 -0.14 -0.07 -0.62 0.16 0.83 -0.77 -0.23 -1.10 -1.10
1.53 -0.53 -1.55 -0.28 -0.40 -1.01 0.16 -0.97 -0.27 0.20 0.36 0.60 0.36 0.79 -0.81 -0.35 -0.17 0.58 -0.27 -1.23
-1.07 -2.55 0.72 0.18 -0.73 0.65 0.50 0.99 0.56 -0.98 -0.08 -0.26 -0.07 -0.29 -0.95 -0.06 1.33 -0.65 0.52 -1.08
13. -0.10 -0.56 -1.03 -1.05 0.40 -0.03 0.75 -0.89 0.25 -0.05 0.81 0.67 1.07 -0.30 -1.66 -1.11 0.50 0.51 1.40
-0.30 0.52 0.39 1.19 0.00 -1.17 -0.40 -1.55 1.91 -0.11 -1.32 -0.24 1.27 -0.68 -0.51 -0.81 0.35 -0.42 -0.16 -0.17
-1.13 0.49 -1.22 -0.17 0.72 -0.89 -0.38 0.07 0.00 -0.91 0.23 -1.12 0.56 0.22 0.46 0.66 -0.79 0.33 -0.01 -0.74
-1.31 -0.27 -0.11 -0.41 0.28 0.01 0.55 0.29 -0.40 0.85 -1.19 0.10 0.81 0.98 -0.27 -0.92 -0.67 2.01 -0.90 1.16
-0.32
14. -0.85 0.02 -1.70 0.38 0.72 -0.22 0.60 -0.78 1.24 0.70 0.59 0.04 -0.14 -0.83 -1.91 0.13 -0.64 0.54 -1.07 0.22
0.54 1.06 -0.49 -0.28 0.53 -0.59 0.91 -0.72 0.45 -0.10 -0.31 -0.07 -0.71 -0.32 -1.08 0.59 0.57 0.35 -0.36 -0.51
0.97 -0.73 -1.10 -1.19 -1.20 -0.35 -0.32 -0.96 -0.45 0.67 -0.27 0.18 -0.59 0.06 -0.05 -1.15 -0.46 0.40 -0.18 -0.63
-0.00 0.49 -0.67 -0.19 0.02 0.68 0.72 -0.96 -0.76 -0.10 1.68 -1.30 0.72 -0.52 -0.20 -0.65 0.31 -0.36 -0.21 -0.42
15. 0.32 -1.55 -0.18 0.05 0.39 -0.25 0.39 0.16 -1.36 -1.59 0.58 0.75 0.88 -1.21 -1.60 -0.32 0.83 0.15 0.79 -0.41
0.30 0.91 0.17 0.61 -0.17 0.90 -1.62 0.65 -0.70 -1.00 -0.43 0.13 -1.37 -0.51 0.24 0.90 -0.50 -0.64 -0.49 -1.24
0.24 -0.87 -0.25 -1.64 -0.74 -0.02 -0.38 -0.72 0.65 -1.07 -0.97 1.38 -0.34 0.59 -1.13 -0.92 1.11 0.31 1.59 -0.89
0.90 1.97 -0.55 0.49 -0.07 -0.00 0.65 0.35 -0.19 -0.41 1.17 -1.36 1.15 0.41 -0.38 -0.42 0.19 1.32 -3.30 -0.51
16. -1.24 -0.67 -0.96 1.90 0.51 -0.20 0.04 -1.08 -0.57 -1.40 0.68 0.19 1.05 0.30 -1.76 0.69 0.37 -0.67 0.60 0.81
0.33 0.92 0.36 0.77 0.45 -1.42 0.06 0.91 -0.25 0.08 0.37 -0.77 -1.14 0.58 0.75 0.49 0.61 0.18 0.03 0.12 0.15
-0.90 -0.46 0.06 -1.64 -0.53 -0.39 -0.17 -1.85 0.87 0.07 0.56 0.34 -0.22 -1.02 -0.02 0.40 0.01 -0.56 -1.01 1.21
2.19 -0.75 -0.02 -0.61 0.12 0.80 -0.31 0.60 -0.91 -1.62 -0.02 -0.87 -1.31 -0.99 0.16 -0.08 -1.29 -2.16 0.49
17. -1.72 0.89 -0.44 0.60 -0.14 -0.55 0.88 -0.68 1.43 -1.51 0.99 -0.46 0.48 0.29 -1.23 -0.25 -0.10 -0.92 -0.59
0.52 0.67 0.75 0.50 0.64 0.92 -0.59 -0.58 -0.30 0.12 -0.35 -0.27 -1.12 -0.16 -0.68 1.24 0.34 -0.29 0.05 -0.10
0.05 -1.28 1.20 -0.33 -0.97 -1.42 -1.36 -0.96 0.18 1.41 -0.04 -1.12 0.87 -0.89 -0.31 -0.50 -0.43 0.11 0.46 -0.54
1.31 -0.50 -1.10 -0.90 -0.43 -0.70 -0.31 0.69 -0.67 0.60 -1.04 1.34 -0.76 0.43 -1.01 0.03 -1.04 0.59 -0.45 1.67
-1.47
18. -1.40 -0.10 1.38 1.47 0.15 -0.49 0.44 -0.38 -1.11 -1.61 -0.26 1.03 0.16 0.57 -1.12 -0.68 0.31 0.82 -0.89
-0.03 0.33 0.74 1.37 -0.29 0.37 0.94 -0.67 1.45 -0.27 -1.06 -0.28 -0.65 0.35 -0.34 -0.04 -0.67 -0.46 -1.61 -0.44
-1.43 1.01 1.44 -1.21 -1.62 -0.62 -0.68 -0.88 -0.35 -0.73 1.04 -0.39 1.04 -1.41 -0.15 -0.31 -0.09 1.19 -0.37 0.11
0.10 -0.50 1.18 0.49 0.21 -1.09 1.35 0.11 -0.66 0.22 -0.39 -0.13 0.01 -0.80 0.06 -0.87 1.09 1.91 -1.40 2.86
-1.54
19. -0.15 0.36 -1.61 -0.04 0.13 -0.11 1.66 -1.08 0.76 -1.11 0.49 -0.34 -0.18 0.01 -1.13 0.78 0.88 0.63 0.51 1.02
-0.55 0.08 1.06 0.74 1.15 -0.62 -0.51 1.51 -0.03 0.14 -0.67 -0.41 -0.49 0.04 0.80 0.57 -0.27 -1.39 0.53 0.39
1.28 -0.53 -1.17 0.98 1.00 -0.86 -0.23 -0.98 0.01 0.51 -0.92 -0.08 -1.04 1.24 0.85 -0.01 0.66 0.44 -0.58 -0.65
-0.67 0.59 0.11 0.22 -0.39 0.56 -0.67 0.03 0.36 -0.70 -0.00 -0.15 -0.47 1.27 1.28 -0.04 0.32 -1.17 2.18 -1.11
20. -0.32 0.13 -0.42 1.50 0.42 -0.50 0.17 1.04 0.18 -0.67 0.98 0.75 0.75 -0.46 -0.93 -0.94 0.13 0.42 -1.63 0.09

0.24 0.34 0.30 0.41 0.02 -0.09 -0.59 1.14 -0.15 0.93 1.23 -0.19 -0.96 1.42 1.00 0.18 1.27 -1.19 0.17 1.75 1.56
-0.08 0.16 -0.53 -1.01 0.42 0.08 -0.04 -0.88 0.49 -0.44 0.32 0.57 0.67 -0.48 -0.82 0.87 0.66 -0.80 -2.01 -0.40
0.53 -1.06 0.16 -0.58 0.46 0.46 0.03 0.98 -1.61 0.34 0.02 -0.78 -0.21 -0.27 -0.56 0.59 -0.66 1.27 -0.72
21. -0.59 -1.43 -1.54 -0.81 0.31 -2.38 0.19 -1.09 -0.36 -0.84 1.32 0.00 0.86 -0.50 -1.24 -0.85 -0.70 0.16 0.79
-0.29 -0.29 1.45 0.41 -0.26 0.14 -1.28 -0.61 1.70 -0.54 0.36 -1.45 -0.63 -0.62 -0.63 -0.25 -0.22 -0.78 -1.59 -0.65
0.01 0.37 -1.23 -1.18 -0.58 0.23 0.13 -0.28 0.47 -0.21 -0.53 -1.09 0.25 -1.11 0.63 -0.90 -1.25 0.19 -0.40 -0.31
0.04 -0.37 1.29 -0.90 0.70 -0.72 0.26 0.54 -0.48 0.31 -0.09 1.19 2.04 -0.91 -1.64 -1.86 -0.46 1.11 -0.75 2.52
-1.15
22. -0.48 -0.09 0.38 -0.25 0.38 -1.20 0.44 -0.63 -0.74 -1.40 0.60 0.70 0.72 -0.11 -1.96 -1.17 0.53 -0.02 0.30
-0.92 0.69 -0.12 0.90 0.41 0.29 -0.49 -0.60 -1.98 -0.43 -0.90 -1.54 -0.04 -1.61 -1.10 0.12 0.03 -0.11 0.26 -1.05
-0.69 1.85 -1.15 -1.69 0.61 0.47 -0.54 -0.00 -0.19 -0.33 0.44 -1.14 0.21 -0.85 0.11 -0.30 -0.07 1.16 0.15 0.71
-1.11 -1.14 -1.37 -1.46 0.06 -1.25 -1.00 0.38 0.52 -0.04 -0.37 1.49 1.01 -0.91 0.10 0.00 0.30 -0.40 -0.58 -0.18
0.99
23. 0.62 0.17 -1.63 -1.30 -0.37 -1.68 0.94 -0.63 -0.78 -0.77 0.58 0.40 0.84 0.54 -1.51 -0.38 0.58 0.42 1.40 1.05
1.29 0.36 -0.87 -0.39 0.11 -0.84 -1.71 0.27 0.43 0.04 -0.61 -1.44 0.07 -0.33 0.79 0.89 0.62 -0.91 -0.15 -0.55
0.92 -0.18 -0.27 -0.22 -0.53 0.96 -0.88 -0.83 -0.54 0.17 0.89 0.14 -1.03 -0.57 -0.14 1.20 0.08 -0.17 0.95 0.04
-0.44 2.79 -0.64 -0.76 0.44 0.47 0.84 -1.57 1.24 -0.06 1.33 -0.23 -1.09 0.63 -0.85 -0.01 0.87 1.10 -1.75 -0.34
24. -0.92 -0.01 -0.72 1.81 0.37 1.44 0.19 -0.54 0.11 -0.04 0.90 0.12 -1.11 -1.06 0.49 -0.81 0.85 0.73 0.09 0.45
1.39 -0.68 0.07 0.46 -0.22 1.91 -1.36 1.03 -0.92 -1.00 -0.44 -0.74 -0.55 -0.01 0.27 0.01 -0.19 -1.90 -0.07 -0.66
-1.65 -1.44 -0.62 -0.21 0.93 -0.59 -0.92 -0.25 0.26 0.77 -0.34 -0.08 -0.41 0.82 1.03 0.08 0.51 0.11 0.80 1.15
-1.52 0.58 -0.91 -0.31 0.22 0.31 -0.64 -1.09 1.24 -1.50 1.02 -0.25 -0.76 -1.97 -0.32 1.69 -1.49 0.83 2.06 0.71
25. -0.98 -1.08 -0.98 1.71 0.71 -1.32 1.17 -0.52 -1.37 0.94 0.13 0.87 1.04 -0.92 0.12 -1.06 0.31 -0.02 1.09 0.44
0.20 1.17 0.29 0.41 0.46 -0.13 -0.26 1.90 -0.94 -1.36 0.04 -0.99 -0.56 -0.93 0.08 1.04 -0.64 -0.63 -0.28 -1.32
-0.10 -0.16 -0.88 -1.19 0.06 1.94 0.14 -0.19 -0.46 0.81 -0.33 0.37 0.18 -0.83 0.43 0.22 0.98 -0.48 0.05 0.80
0.35 1.51 -0.92 0.07 -0.96 0.94 0.38 -0.62 1.01 -1.36 0.56 0.77 -1.60 -0.10 0.48 -0.88 0.04 -0.61 -1.62 -0.85
26. -0.77 1.22 -0.45 1.70 0.46 -1.17 0.79 -0.82 -1.61 -0.90 0.47 0.57 -0.25 0.18 -0.40 -0.36 0.13 -0.74 0.13
0.10 0.31 0.84 0.93 -0.60 0.21 -0.03 -0.05 -1.26 -0.71 -0.90 -1.21 0.14 -1.05 -0.14 -0.37 0.66 0.78 -0.75 -0.56
-1.32 1.57 -0.99 -0.09 -0.63 -0.64 -0.44 0.36 -0.76 -0.64 -0.02 -0.85 1.14 1.97 0.64 -0.70 -0.46 0.27 -0.09 0.73
-0.98 -0.23 0.35 -0.95 0.13 -0.71 -0.30 1.00 1.29 0.55 -0.57 0.17 -0.91 -0.81 -1.81 -1.85 -0.52 0.63 -0.60 1.91
0.12
27. -0.91 -1.51 -0.68 1.34 0.27 0.91 0.79 -1.36 0.21 -0.48 0.07 -0.06 -0.26 -1.38 1.51 -0.68 0.36 0.21 0.36 0.11
0.63 -0.27 0.35 0.06 0.61 0.17 -0.88 -1.85 -0.72 -0.35 -1.69 -0.12 -0.05 0.24 -0.09 0.80 0.93 -0.46 -1.14 2.14
1.98 -1.25 0.08 -0.04 -0.06 -0.87 -0.81 0.12 -1.06 0.68 -1.00 -0.07 0.87 0.32 -0.22 -0.74 0.47 1.12 0.28 -0.92
-1.24 -0.02 -0.16 0.92 -0.36 -0.27 0.01 -1.03 -0.29 -1.27 0.89 0.40 -0.98 0.59 -0.89 -0.20 -2.16 0.22 1.22 -0.42
28. -1.15 -0.80 -0.44 0.92 0.06 -0.87 0.84 -0.63 -0.65 -0.80 0.23 0.47 1.17 -0.34 0.07 0.26 -0.45 0.50 1.53 0.15
0.83 0.37 1.50 0.44 1.29 -1.59 -1.41 0.15 -0.11 -0.24 -1.62 1.22 -0.71 -0.54 0.35 0.41 -1.03 -0.03 -0.10 -0.77
1.94 -0.73 -1.24 -0.14 0.18 1.19 -1.06 0.67 0.60 0.10 -0.47 -0.43 0.24 0.80 -0.91 -0.70 0.04 0.71 -1.38 -0.77
-0.15 1.41 0.18 0.30 -0.07 0.15 -0.71 -0.45 0.74 -0.46 1.05 -1.21 -1.39 -0.72 -0.85 -0.24 1.57 -1.14 1.00 -0.49
29. 0.46 -0.05 -1.38 0.95 0.23 0.76 -0.21 -0.84 0.34 -0.73 0.02 0.35 -0.67 -0.22 0.84 0.42 0.25 0.02 0.25 0.32
0.67 0.13 -0.00 -0.61 0.20 -0.98 -0.29 1.20 -0.56 0.32 -0.41 -0.96 -1.65 -1.72 0.19 1.06 0.15 -0.46 -1.70 -0.89
2.01 -0.58 -0.51 -0.02 1.21 -0.28 -0.49 -0.74 -0.91 -0.80 -0.84 1.21 -0.03 0.93 -1.00 -0.36 0.37 0.98 -1.55 -1.09
0.08 -0.01 -0.06 0.32 -0.45 -0.99 -0.64 -0.71 0.11 -1.44 0.75 -0.77 0.18 0.80 -0.29 0.50 0.12 -0.56 1.00 -0.46
30. -0.15 0.38 -1.47 0.72 -0.56 0.39 0.61 -0.01 1.41 -1.33 0.94 1.07 -0.02 -0.08 -1.65 -1.28 0.72 0.81 0.32 0.37
0.28 0.14 1.12 -0.66 0.04 1.24 -0.31 0.48 -0.58 -0.04 -0.93 1.22 -1.38 -1.32 0.16 0.76 -0.00 -0.59 -0.59 -0.75
0.70 -1.55 -1.45 -1.21 -0.08 -0.95 -0.74 -0.22 -0.75 0.80 -0.01 0.17 -1.05 0.57 0.45 1.98 -0.22 -0.08 1.52 -1.59
0.54 0.01 -0.72 0.18 -0.11 0.96 -0.78 -2.31 -0.40 0.86 1.49 -1.38 0.20 -0.89 -0.22 -1.68 1.65 -1.24 -0.03 -0.89
31. -1.25 -1.54 -1.62 0.29 0.99 0.22 0.75 -1.07 -0.80 -0.70 0.69 0.04 0.74 -0.81 -0.79 -0.57 -0.20 0.61 0.50
-0.18 0.62 0.21 0.42 -0.31 -0.63 -0.04 -0.85 0.36 -0.33 -0.91 -0.09 0.09 0.81 0.00 0.09 -0.00 -0.48 -1.23 0.02
-0.28 0.55 -1.34 -1.50 -0.77 0.03 -0.45 -0.36 1.00 -1.39 0.53 -0.39 0.90 -0.07 0.56 1.72 0.08 0.43 0.70 -0.01
0.83 -0.55 2.27 -0.62 0.03 0.17 0.02 0.46 -0.57 0.85 -1.93 -0.09 -0.38 -1.03 1.16 -0.59 -0.82 0.41 1.44 1.88
-1.23
32. 0.74 -0.92 -0.78 1.99 0.54 -0.51 0.35 -0.95 -0.29 -1.27 0.06 -0.63 0.65 -0.20 -1.34 -0.15 1.00 0.66 0.80
0.66 0.31 0.29 0.41 0.22 0.93 -0.02 -0.34 0.72 -0.31 -0.41 -0.11 -0.66 0.80 -1.17 -0.08 0.40 -1.21 -0.14 -0.77
-0.86 -0.13 -0.91 -1.39 0.20 -0.45 -1.19 1.13 0.96 -1.00 0.31 -1.34 0.54 0.71 0.61 -0.49 0.22 0.20 0.65 0.14
-1.21 1.64 2.11 -1.08 0.61 -0.58 0.42 0.31 -0.31 0.50 -0.33 -0.70 -0.59 -0.31 -0.64 -1.11 -0.52 1.40 -1.02 -1.06
-0.88
33. -0.60 -0.57 -0.29 0.87 1.11 -1.71 0.74 -0.83 0.33 -1.21 0.84 1.23 1.18 0.02 -0.11 -0.42 -0.24 0.22 0.23
-0.73 0.92 0.09 0.80 0.33 0.72 -1.36 -0.11 0.94 -0.61 -0.39 -0.30 -0.41 -0.85 -0.62 0.94 0.02 -0.63 0.56 -1.02
1.63 0.20 0.60 -0.59 0.78 0.42 0.01 0.77 -0.52 -0.46 1.25 -0.42 0.73 -0.66 0.59 -0.69 -0.67 0.02 0.40 -1.57 1.49

```
-0.74 1.00 -0.28 0.07 1.19 -0.37 0.21 0.07 0.70 0.02 1.14 -0.50 -0.78 0.59 -0.70 1.07 0.32 -0.89 0.60 -0.29
34.  -1.75 0.10 -1.73 1.35 -0.34 -1.55 1.15 -0.27 0.43 0.13 -0.46 1.00 0.26 -0.86 -1.64 -0.34 0.71 0.30 0.24 0.57
0.11 0.54 1.17 0.92 -0.21 -1.28 -0.69 -0.94 -0.41 -0.28 -0.97 0.40 0.08 -0.57 0.12 -0.51 -0.42 1.54 -0.29 0.61
-0.45 0.53 -0.61 -0.66 -1.12 1.68 -1.02 -0.21 1.21 0.67 -0.79 -0.30 0.74 0.51 -0.60 -0.94 0.59 0.38 -0.92 -1.58
0.32 0.86 -1.04 0.85 -0.08 0.69 0.22 -0.83 0.34 1.53 0.01 -0.94 -1.04 0.66 0.39 -1.03 -2.48 -1.47 0.60 0.74
35.  -0.90 1.38 0.72 0.66 0.24 -0.34 -0.61 -0.64 -0.98 -1.12 0.81 0.83 0.35 -0.66 -1.55 -0.38 0.08 0.27 -1.07
-0.44 0.25 0.92 0.10 0.32 1.32 -1.71 0.35 0.90 -0.02 -0.09 -0.72 -0.15 -0.70 -0.41 0.68 0.18 0.00 -0.08 -0.83
-0.74 1.14 -0.57 -0.26 -0.92 -0.04 -0.98 -0.29 -0.77 0.99 -0.08 -1.06 0.25 -0.53 0.80 0.28 0.97 0.14 0.34 -0.18
-0.94 -0.88 -2.07 -1.11 0.83 0.27 0.50 -0.22 -2.05 0.57 -0.36 0.64 -1.57 0.03 -0.02 -1.82 -0.27 0.21 0.73 -1.13
-0.38
```

LAYER 3

```
 1.  0 -0.3485 0.00 0
 2.  1 0.2570 0.00 0
 3.  0 -0.2273 0.00 0
 4.  0 -0.3218 0.00 0
 5.  0 -0.1159 0.00 0
 6.  0 -0.2202 0.00 0
 7.  0 -0.2353 0.00 0
 8.  1 0.1861 0.00 0
```

Weight values for layer 3

```
 1.  0.97 0.02 0.06 0.45 0.27 0.19 0.17 0.43
 2.  -0.67 -0.07 0.22 -0.31 0.67 -0.31 -0.28 0.07
 3.  0.15 -0.01 -0.26 0.08 0.10 -0.89 -0.21 0.09
 4.  1.26 0.21 0.58 0.35 -0.33 0.20 0.18 0.13
 5.  -1.24 -0.09 -0.20 0.34 -0.18 -0.07 0.08 -0.11
 6.  0.69 0.13 0.07 0.79 0.19 0.00 0.66 -0.20
 7.  1.55 -0.63 0.86 0.33 -0.45 -0.72 -0.17 0.11
 8.  -0.06 0.12 -0.15 -0.93 0.12 0.01 0.08 -0.10
 9.  -0.10 0.14 0.26 0.73 -0.21 0.31 -0.24 0.63
10.  0.39 -0.05 0.12 0.24 -0.04 -0.57 0.45 -0.03
11.  -0.74 -0.39 0.37 -0.29 0.90 -0.00 0.46 -0.16
12.  0.28 0.33 0.03 -0.28 0.65 -0.12 -0.09 0.21
13.  0.41 -0.20 -0.28 -0.22 0.01 0.26 -0.48 0.41
14.  0.61 0.23 -0.58 -0.36 -0.08 -0.16 0.06 0.10
15.  -0.01 -0.01 0.07 0.05 -0.00 0.17 1.06 -0.71
16.  0.30 -0.35 0.23 0.28 -0.14 0.17 -0.07 0.12
17.  0.43 0.03 0.07 0.32 -0.13 -0.20 -0.05 -0.01
18.  0.28 0.15 -0.15 0.49 0.20 0.35 -0.28 0.47
19.  0.41 0.29 0.27 0.41 -0.46 0.13 0.17 0.43
20.  0.23 0.16 -0.38 -0.50 0.15 -0.31 0.51 0.37
21.  0.94 0.16 0.25 0.02 0.25 0.25 -0.09 0.18
22.  0.53 0.13 -0.04 0.23 0.03 0.03 0.18 0.26
23.  -1.04 0.40 -0.61 -0.18 0.10 -0.07 0.01 0.21
24.  0.58 -0.22 0.08 -0.04 -0.16 -0.04 -0.16 0.29
25.  1.42 -0.27 0.48 -0.23 0.01 -0.05 0.30 0.02
26.  0.40 -0.06 0.18 0.32 -0.31 -0.37 0.19 -0.63
27.  0.03 0.10 -0.12 -0.30 -0.15 0.08 0.87 0.32
28.  -0.04 0.29 0.17 0.31 -0.33 0.82 -0.20 0.94
29.  0.43 -0.08 -0.04 0.08 -0.27 -0.68 -0.25 0.05
30.  0.43 -0.06 -0.04 0.20 -0.01 -0.06 -0.05 0.38
31.  -0.09 0.32 -0.16 -0.09 -0.11 -0.05 -0.77 0.17
32.  0.81 0.33 0.29 0.15 0.21 -0.17 -0.43 -0.39
33.  0.54 -0.07 0.11 0.33 0.56 0.18 -0.20 0.07
34.  0.86 0.01 0.37 -0.32 0.32 0.41 -0.52 -0.17
35.  0.52 0.09 0.50 0.09 0.13 0.21 -0.10 0.08
```

36. -0.47 0.24 -0.27 -0.62 0.37 -0.17 0.13 -0.19
37. -0.09 -0.00 -0.30 0.12 0.64 -0.27 0.10 0.26
38. 0.04 0.01 0.07 0.12 -0.06 0.53 0.28 -0.29
39. 0.98 0.02 0.12 -0.36 0.18 0.13 -0.54 0.02
40. -1.03 0.22 -0.33 0.21 0.05 0.28 -1.10 0.16
41. 0.60 0.26 0.38 0.16 -0.38 -0.44 0.61 0.87
42. -1.20 0.34 -0.06 -0.46 0.45 0.44 -0.35 0.25
43. -0.36 0.07 -0.03 -0.61 -0.10 0.14 0.36 0.17
44. 0.08 0.23 -0.03 0.29 -0.67 0.50 -0.07 1.05
45. -1.52 0.43 -0.62 1.03 -0.60 -0.63 0.01 0.19
46. 0.36 -0.00 0.47 0.09 0.16 -0.26 0.35 -0.71
47. 0.76 -0.11 0.32 0.05 0.06 0.32 0.26 -0.03
48. 0.36 -0.03 0.44 0.45 0.24 0.56 -0.53 -0.27
49. 0.38 0.21 0.13 0.28 0.27 -0.27 0.32 0.65
50. 0.96 -0.18 0.40 -0.45 0.16 -0.56 -0.05 0.13
51. -0.29 -0.10 0.15 0.13 0.11 -0.19 -0.37 0.18
52. 0.47 0.10 -0.10 0.24 -0.03 0.57 0.00 -0.02
53. 0.95 -0.24 0.29 -0.28 -0.15 0.71 0.77 -0.37
54. 0.99 0.17 0.03 0.19 0.11 0.19 0.66 0.84
55. 0.86 -0.09 -0.08 0.49 0.28 -0.92 0.07 -0.11
56. 0.56 -0.29 0.50 0.13 0.61 -1.23 -0.06 -0.19
57. -0.19 0.13 0.04 -0.05 -0.19 -0.41 0.03 -0.65
58. -0.72 0.06 -0.63 0.23 0.12 -0.19 0.01 -0.21
59. 0.93 0.07 0.01 0.44 -0.42 -0.28 -0.11 -0.36
60. -0.25 0.04 0.49 0.76 0.05 -0.00 -0.62 0.11
61. 0.77 -0.22 0.25 -0.25 -0.52 0.26 0.00 -0.03
62. 0.33 0.27 -0.23 0.46 -0.17 0.25 -0.82 0.15
63. -0.54 0.50 -0.14 0.01 -0.45 -0.21 0.49 -0.47
64. -0.48 -0.22 0.61 -0.06 0.62 -0.06 -0.26 0.18
65. 0.77 -0.34 0.30 -0.08 0.83 -0.56 0.02 0.60
66. 0.22 0.11 0.20 -0.02 -0.01 -0.21 0.06 0.27
67. -1.28 0.11 0.07 0.28 -0.34 0.36 -0.05 -0.23
68. 0.88 -0.27 0.36 -0.03 -0.82 0.46 -0.18 -0.13
69. 1.56 -0.61 0.81 0.13 0.19 0.17 0.11 0.34
70. 0.01 0.19 -0.43 -0.70 0.26 -0.05 -0.24 0.17
71. 0.19 -0.15 0.21 -0.20 -0.17 -0.28 0.15 0.72
72. 0.78 -0.03 0.61 -0.09 0.04 0.47 -0.34 -0.32
73. 0.25 0.53 -0.29 -0.42 -0.08 0.22 -0.33 -0.01
74. -0.34 0.47 -0.04 -0.20 0.33 -0.10 0.23 -0.19
75. 0.72 -0.03 -0.40 0.56 0.34 -0.35 0.21 0.60
76. 0.61 -0.05 0.51 -0.27 -0.01 0.20 0.04 0.55
77. 0.02 0.71 0.15 -0.01 0.67 0.71 -0.54 -0.20
78. -0.46 0.34 0.15 -0.05 0.36 -0.42 0.08 -0.18

Vita

David Southworth was born on June 17, 1946 in Los Angeles, California. He served for twenty-one years in the United States Navy as a Naval Flight Officer and also held several positions in weapon and computer systems acquisition and in computer technology research and development. He is currently working for RBC, Inc., a defense contracting firm, as a Program Manager providing support to the Navy in the acquisition of various weapon systems. Mr. Southworth holds a Bachelor of Arts in Psychology from Northwestern University and a Master of Science in Systems Management from the University of Southern California.

[illegible]