# NeuralGraphics:
# An Artificial Neural Network Simulator
# Users Information Guide

Gregory L. Tarr

April 16, 1991

# Contents


**1 Introduction** **3**
- 1.1 Background . . . 4
  - 1.1.1 Building A Bug . . . 4
  - 1.1.2 Artificial Perception . . . 4

**2 The Neural Graphics System** **6**
- 2.1 Getting Started . . . 6
- 2.2 Feedforward Tools . . . 7
  - 2.2.1 Multi-layer Backpropagation . . . 7
  - 2.2.2 Hybrid Propagation . . . 11
  - 2.2.3 Special Features . . . 12
  - 2.2.4 Notes on the Unix No-Graphics Version . . . 13
  - 2.2.5 Kohonen Self-Organizing Maps . . . 14
  - 2.2.6 The Traveling Salesman Problem . . . 15
  - 2.2.7 Hopfield Associative Memory . . . 15
- 2.3 Summary . . . 15

**3 Neural Network Modeling** **17**
- 3.1 Software Design . . . 18
  - 3.1.1 The Top Level Design . . . 20
  - 3.1.2 Objects and Methods . . . 21
  - 3.1.3 Memory Management . . . 22
  - 3.1.4 2D Dynamic Allocation . . . 23
  - 3.1.5 Table Dispatch . . . 25
- 3.2 Initialization Module . . . 25
- 3.3 The Main Program Loop . . . 26
  - 3.3.1 Making an Input Vector and Desired Output Vector . . . 26
  - 3.3.2 Propagating the Input Vector . . . 27
  - 3.3.3 Updating the Weights . . . 27
  - 3.3.4 Measuring the Error . . . 29
  - 3.3.5 Displaying the Progress . . . 29
  - 3.3.6 Interactive Program Control . . . 30

3.4 Adding Your Own Paradigm . . . . . 31
3.5 Conclusion . . . . . 31

# Chapter 1

# Introduction

The software described here is a development and tutorial system for the study of artificial neural networks. The system has been implemented on a number of machines but the most developed versions run on Silicon Graphics IRIS 3130's, 4D's and an IBM with EGA. Less graphical versions run on any Unix machine.

The environment includes demonstrations and applications of Kohonen mappings, multi-layer feedforward networks using backpropagation, hybrid (joint supervised/unsupervised) training paradigms, radial basis functions, Hopfield associative memory, error surface analysis and network topology analysis. Several pattern recognition problems are included as test sets with some versions of the program.

This program was written for a number of reason. The original intention was to provide engineers new to the neural network field a of graphic demonstration to illustrate some of the concepts in neural network problem solving. As such, the program has been used at the Air Force Institute of Technology for a couple of years. Several other institutions have used the software as well. Together with our introductory tutorial text, and articles on the subject, neural neophytes can come up to speed quickly in this rapidly changing discipline.

This booklet is organized as follows. The first section provide background information related to pattern recognition and the research efforts the software was intended to support. The purpose and capabilities of the environment are discussed to illustrate how a classification problem can be presented to the system for evaluation and selection of network parameters.

The second chapter describes the models and demonstrations in the program, how to use them and what they are for. Included is a description of the user interface and file formats necessary to construct user models. The screen display shows the network node activity and tracks the progress of the network in an interactive manner. The network software system can be

tested using several recognition problems common to military applications.

## 1.1 Background

Neural Networks may offer a new approach to many problems which have proven intractable for most conventional algorithms. With the increased interest in finding neural network solutions to common problems, an environment is developed to examine which types of problem may be amenable to neural network solutions.

Anyone who has tried to build intelligent system can't fail to be impressed by the simplest biological creature. The actions of a common house fly are extraordinary compared with the best computer systems. A house fly or any bug in the pursuit of its daily survival performs functions beyond the reach of our best mechanical and electronic systems. Bugs certainly possess the elements of intelligence, and researchers around the world are trying to mimic that type of intelligence with something called artificial neural networks.

### 1.1.1 Building A Bug

Suggesting that one square centimeter protoplasm is intelligent may be generous with the term, but no one can build anything with the capabilities of a house fly. If you don't believe that, try building an airplane that can find its own gas.

The first few steps involved in building a bug is to integrate sensor systems with classification systems. The problem can be considered a two part problem, to see and to understand.

Neural networks can work in both parts of the problem. Computer vision systems (the original design function of the Neural Graphics system) require a means of taking measurements from pixel imagery and converting them into symbols used to describe the environment in which the sensor finds itself.

### 1.1.2 Artificial Perception

Artificial perception, as opposed to artificial intelligence, is the function of converting sensor measurement into symbols used by the intelligence system. Artificial perception allows a sensor to understand its environment.

Identification and classification of targets from electronic imagery and signal intelligence is a difficult problem due to the vast amounts of data involved. A single image can contain millions of bits of information, all of which need to be processed. Processing images for pattern recognition is

a threefold problem. First, the targets must be separated from the background or segmented. Second, the data must be reduced to a manageable size, commonly called vector quantization or feature selection. This reduction in data can be accomplished by selecting specific features of a pattern and using only these features for classification. Good pattern recognition requires good features. The final task is classification of the vectors. In most cases the final classification is the easiest part of a neural network problem.

Determining which features of an image form the best description of an object is a difficult problem. In addition to selecting the best features, the data must sometimes undergo significant preprocessing.

Success of a particular classification problem depends on a number of factors. First, consider the validity of the segmentation of the data. Has the actual target been separated from the background data and noise? Is the feature extraction legitimate? Do the features selected for the input vector represent a good description of the target? Once the target has been extracted from the background, is the vector quantized description unique enough to allow classification ? Finally, is the neural network topology sufficient for the size of the decision region and can it accurately classify input patterns ? Special tools may be needed to answer these questions.

An environment to examine the internal operations of a neural networks as it train could help determine the efficiency and accuracy of different topologies. Evaluation of the internal constants and variables as the network trains may offer insight into which values may be best suited for a particular set of circumstances.

# Chapter 2

# The Neural Graphics System

The Neural Graphics system is a collection of software tools and demonstration to provide graphical presentation and interaction of neural network concepts. Actually, the only difference between a demonstration and a tool in this package, is that a demonstration has a fixed (hard-wired in the software) data set.

## 2.1 Getting Started

The system will come up by typing the program "net". SG machines need to be in the window server (type "mex"). Older SG machines allow for running programs with a single window environment. For multiple windows, the program **mex** must be run before, the net program. Neural Graphics uses lots of windows. The first thing you need to do is set up all the network parameters. A good set of defaults is provided, so in most case a simple return will do (IBM).

For the SG pick the configure item using the mouse and the right button. The default values in the boxes can be changed by clicking on the small white circles. Clicking on an item opens a dialog window to type in the new values.

The system contains several models and demonstrations. The primary systems to evaluate user data sets are contained in the feedforward models. These includes a multi-layer backpropagation and a hybrid propagation network.

The demonstrations include Kohonen mapping, a neural net solution for the traveling salesman problem and a Hopfield associative memory.

While all models are not supported for both implementation, (i.e. Sili-

con Graphics and IBM) the user interface works about the same for both. Currently the Silicon Graphics tool does not support Hopfield demonstrations and the IBM does not support Error Analysis. These add-ons may be included in a future update.

Much of the information specific to the IBM implementation is contained in a help tutorial contained on the disk. Simply respond to the menu selection noted as tutorial.

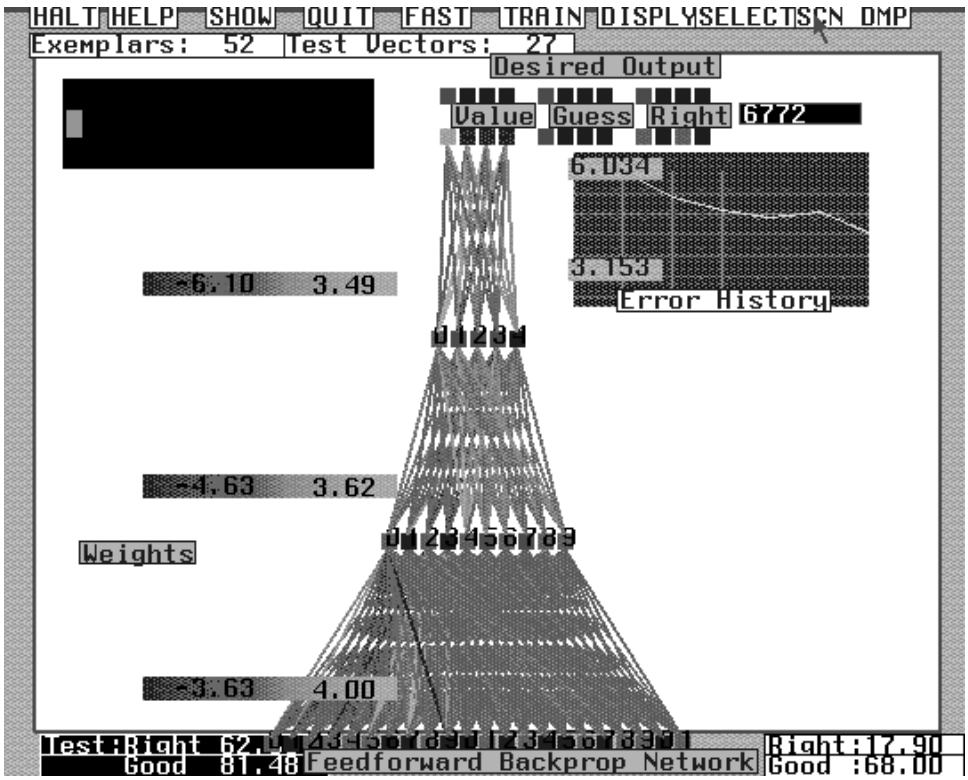


Figure 2.1: A Three Layer Network Display

## 2.2 Feedforward Tools

### 2.2.1 Multi-layer Backpropagation

A multi-layer backpropagation model is provided which allows one, two, or three layers of weights. The user can select up to 100 or more nodes per layer. The network generally uses dynamic allocation, so everything is

a trade off. Bigger nets, mean less data storage and so on. If you try to allocate to much memory after the data is stored, the program will exit, and usually tell you why.

**Step 1: Select function**

The first step in using the simulator is to set up the desired network. Several types are available, which include Multi-layer Feedforward Back propagation of Errors, Hybrid propagation, Kohonen Mapping, Error Surface Analysis.

Both systems prompt the user for setup data, the workstation models require a mouse and the IBM version does not. The selection is made with the mouse or arrow keys on start up. Select the net type, then in response to the prompts, type in the number of nodes (computational units) in each hidden layer.

Request for a weight file can be entered at start-up. If there are no previous files, enter "r" for random weights to be generated by the program. Typing an "n" will allow noise to be added to each input vector. This allows each input exemplar to represent a cluster of data.

**Step 2: The training file**

The training file contains all the data for testing and training the neural network. The training file name is entered at the prompt requesting data.

A training file is created as a standard ascii file using a standard text editor.

The basic format is as follows:

The first line defines the size of the training set, test set and the input and output vectors.

After that, each exemplar is listed in order. The first number is an arbitrary integer and is ignored by the program. The only purpose is to identify the exemplar number. Next list each element of the exemplar vector x0,x1,x2 . . . x(input-1). The last element is the exemplar class type.

Exemplar class types must be sequential, i.e. 1,2,3,... etc. The first class type must be one and no numbers can be skipped. Classes can be randomly mixed. To allow flexibility, very little error checking is performed on the input file. An incorrect file can cause the program to appear dead. If this happens use control-C to get to the extended menu. Example:

```
20 20 3 2

1 3.4 5.4 2.1 1
2 5.6 2.3 7.1 0
```

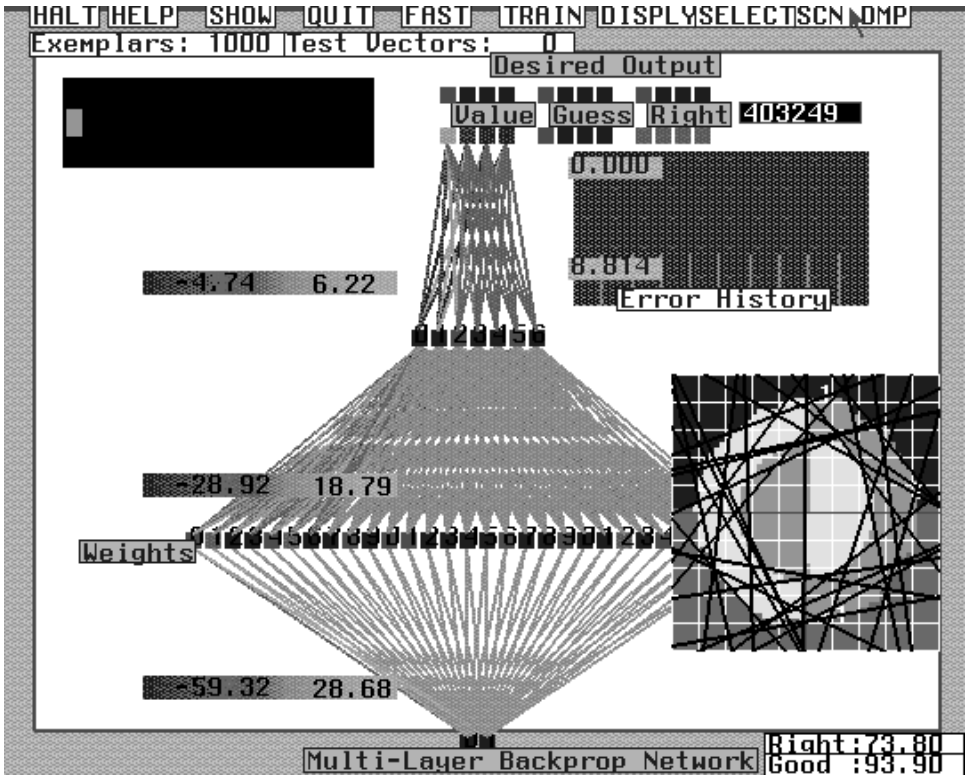


Figure 2.2: Neural Net O-Scope on a Silicon Graphics

```
3 7.2 8.1 6.5 1
4 3.3 4.1 9.1 0
   .
   .
40 4.1 8.5 3.2 0
```

In the example above there are 20 training exemplars and 20 test exemplars.

The first class must be labeled zero, the second labeled one etc.

The test exemplars are those set aside for testing the generalization capability of the net and are not used in training. The input vector is of length three and there are two distinct output classes.

*Two files for testing and demonstration are included in the IBM package. These are : xor.dat and test.dat. The silicon graphics version offers more in the data directory.* The system uses dynamic memory allocation, so the size of input and output vector, number of hidden nodes is not strictly limited. The screen display shows the amount of memory left at run time. Should you try to allocated more memory than is available, the program will tell you so and exit.

**Step 3: Interaction with the network**

The mainframe version interacts with the user in two ways: a mouse and a pull down menu that is engaged by hitting the control -C.

The three mouse buttons are used as follows:

- Right: Selection of top menu items.
- Left : Removes/replaces selected Nodes.
- Middle: Magnification of the screen.

The screen displays information about the progress of the network as it trains. The top line allows menu selections by clicking the right mouse button as follows:

- HALT: Toggles a halt action (freezes display).
- HELP: Displays a help file.
- SHOW: Displays node values.
- QUIT: Exits program.
- FAST: Toggles display update between one to preset value
- TRAIN: Toggles training on and off.

- DISPLAY: Displays error surfaces for selected nodes.
- SELECT: Selects a node for special displays.
- SCN DMP: Dumps screen display to a file. (Utah RLE format);

The IBM version menus are activated by pressing the first letter of the menu item listed on the top line of the display.

**Error Measurements (IRIS only)**

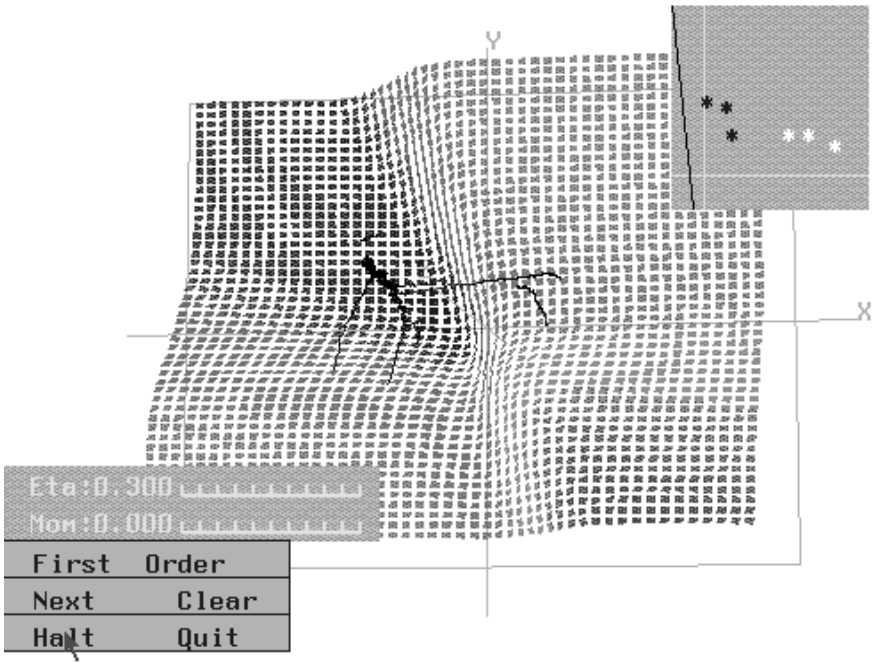


Figure 2.3: Error Surface Analysis

Three error measurements are built into the system. The error history is a graph which displays the progress of the network as training progresses. Also, after each display cycle a tabulation is made to determine the percent correct for both the training set and the test set. If the output of the network is within 20 percent of the desired value for every node, a counter

is incremented for the *Right* indicator. If the highest output corresponds to the correct node for that test case the *Good* counter is incremented. These values are displayed for the training and test sets as a percentage correct.

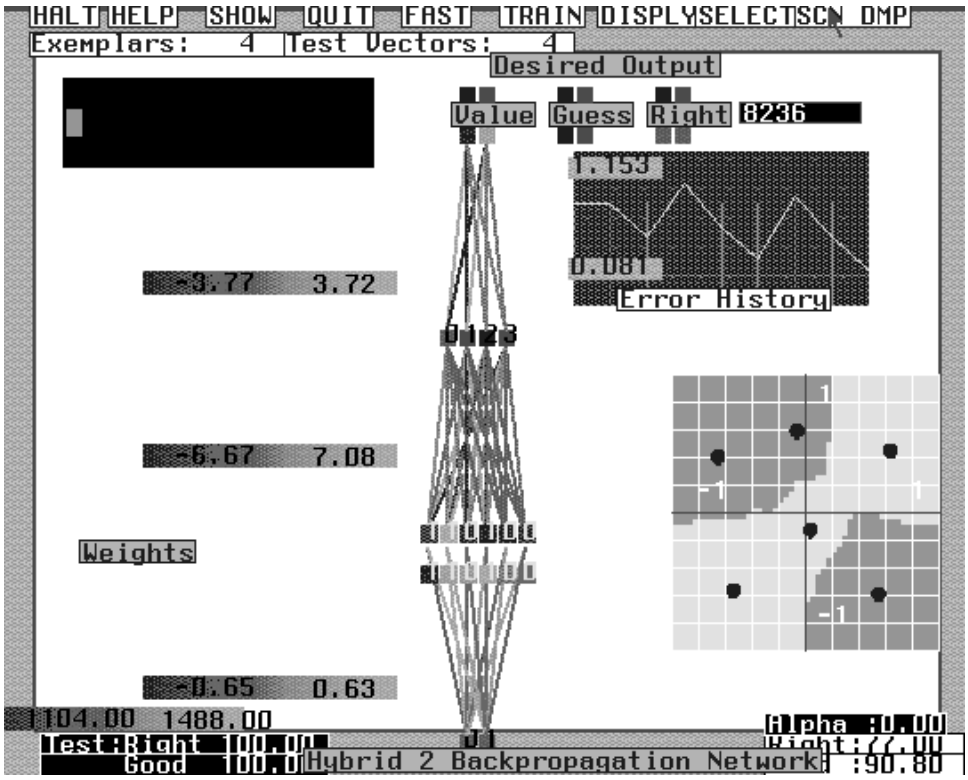


Figure 2.4: A Hybrid Propagation Network

### 2.2.2 Hybrid Propagation

This paradigm works exactly the same as the Backprop model from the users point of view. The difference is in the convergence time for various types of problems. Backprop has difficulty training when decision regions are very close together, but works well for many distinct and disjoint decision regions. The hybrid propagation network works well for several disjoint regions which are spaced closely together as long as the number of first layer nodes is a few more than the number of distinct decision regions. Hybrid propagation uses Kohonen self-organization on the input layer followed by a two layer backprop for classification.

Hybrid propagation is used in this package as a conglomeration of hybrid type networks such as ART2 and Counterpropagation. In most cases, this network is more appropriate to testing data sets than either ART or Counterpropagation. Hybrid propagation requires the conscience parameter to be adjusted while ART and Counter propagation require several parameters which must be set exactly.

### 2.2.3 Special Features

The feedforward models offer a few special features.

- Extended Menu. Using the control-C, addition parameters can be adjusted by using the extended menu. Weight files can be saved or restored, displays can be adjusted and internal values can be displayed.
- Neural Net Probe. When the input is exactly two entries wide, a sort of neural network O-scope kicks in which displays the response of a node over the total range of the input.
- Node Elimination. Placing the cursor over a node and pressing the Left Mouse button, removes the node and all connected weights from the network.
- Magnification Window. Placing the cursor on any point in the screen can blow up that region for examination. This capability is used to examine the value of the weight near the node terminals.
- Hard Copy. Network progress is also tracked in a file called datastats. Any orderly termination of the program will allow the file to print training graphs using GNUplot.

### 2.2.4 Notes on the Unix No-Graphics Version

For use in batch job mode and machines with only termial graphics a setup file called setup.fil allow automatic reading of the net configuration. For example:

```
3
20 5
random
0.0
2
parity.dat
0
```

The first line is the number of layers Next come the number of nodes in each layer The next line is the weight file to be loaded. Normally weights are randomly set by using "random"

Next is the amount of noise added to each vector. This is a random amount of which varies between specified number.

Next is the training paradigm number.

- 1 - Vanilla Backprop
- 2 - Backprop with momentum
- 3 - Radial Basis functions
- 4 - Suder/Oxley (unusual squashing function proof of concept only)
- 5 - Higher order, add all higher order cross correlation products.
- 6 - Batch
- 7 - Semi-batch

Unless you really know what the above paradigm do, use number 2.

The final line specifies the data file used.

The final item is something of an after though. Sometimes data is better handle if it is preprocessed. The program allow for 8 different methods. For the specific numbers, seen normalize() in the file normal.c. The algorithms are explained there as well.

To leave data alone, use 0, to statistically normalize use 1.

### 2.2.5 Kohonen Self-Organizing Maps

Kohonen maps organize data according the distribution of the training data. The program here is a demonstration which graphically illustrates the distribution of the Kohonen nodes with different distributions of input data.

The first selection allows the data to be distributed across a square, a triangle or a cross. The second allows the input in the x and y directions to be distributed according to a constant, a Gaussian or a chi-square probability function.

The display has three maps: a map of the input data, a Kohonen map and a graphical display of the distribution of the Kohonen nodes. The input data is colored red and yellow to form two arbitrary input classes. The class information is not used in the training of the net, it is only provided to demonstrate the grouping of the Kohonen nodes for data with similar characteristics.

### 2.2.6 The Traveling Salesman Problem

The traveling salesman problem is a classic example of an NP complete type problem. This demonstration solves a traveling salesman problem from either a randomly generated list of cities or a specified file.

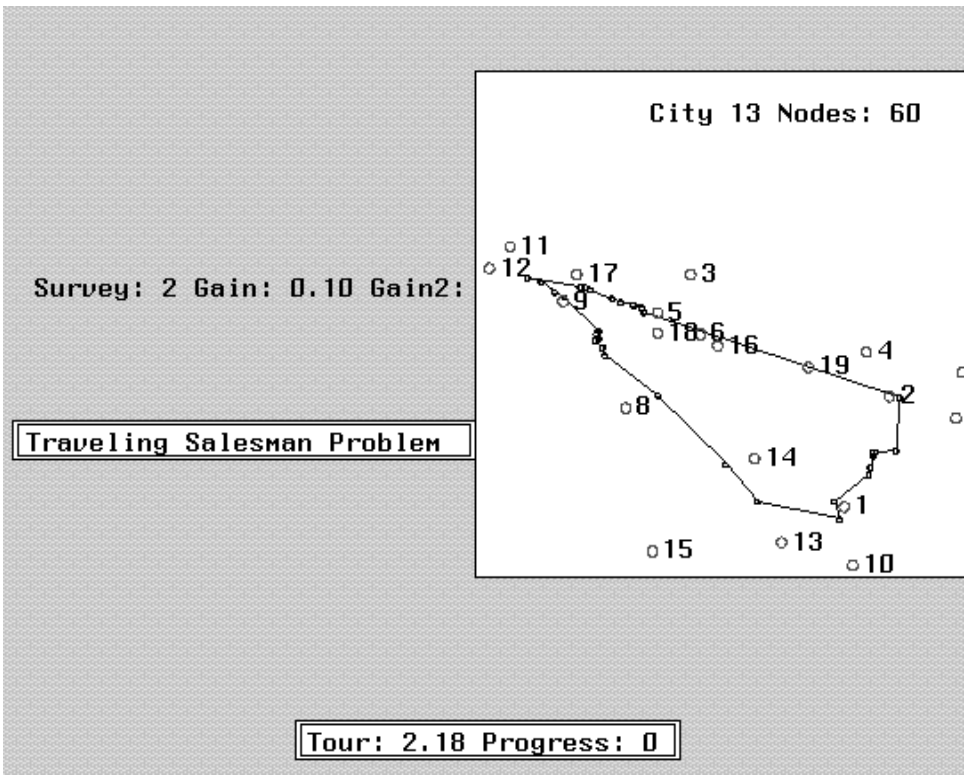


Figure 2.5: The Traveling Salesman Problem

The input file defaults to sales.dat. The file consists of a number of cities, the word 'list' or 'random', a problem number and a list of x,y locations of the cities.

The display shows the current location of the nodes with interconnecting lines, the total length of the tour, the current number of nodes, and the current city being evaluated.

### 2.2.7 Hopfield Associative Memory

The Hopfield associative memory model accepts a user file. The first line of the file contains three pieces of information. The first value, as with the other models is the number of exemplars. the second and third relate to the display only. These are the width of each exemplar followed by the height respectively. On the next line the patterns follow. First comes an unused pattern number. After that each bit of the exemplar pattern is listed 0 for pixel off, 1 for pixel on.

The Hopfield net treats each exemplar as a single binary word, the height and width are for the benefit of the user only. The package provides a sample input file called "hop3.dat".

The Hopfield network is an associative memory. Several patterns are stored in a set of weights. A pattern is selected, then corrupted with noise randomly inverting bits in the image plane. A Hopfield uses only binary patterns. The image is iterated through the network feeding the output of the net back on itself. The result is hopefully, that the pattern will converge to a noise free pattern of the closest match of the input exemplar. Two things can affect convergence. One is the number of patterns stored in the network and the second is the amount of noise added to the exemplar pattern. A third consideration is the hamming distance between exemplars, obviously if two exemplars are very close to each other small amounts of noise can confuse the network. The rule of thumb is .15 stored images per bit in the pattern. These routines seem to work only for much smaller numbers of exemplar sets.

## 2.3 Summary

The neural graphic software package provides a simple interface for testing the viability of using neural network. Applications and problems which arise in other areas of study may be tested quickly without having to depend on primitive I/O and display routines.

These neural network models have been tested against a number of problems including identification of tactical targets(Ruck,1987) (Troxel,1987) (Tarr,1988), speech recognition(Barmore,1988), radar signal analysis, and spread spectrum communications identification (DeBerry,1989) and a host of other problems.

In addition to the basic execution code, the software system is built from a set of reusable components. For investigation of new training paradigms only one module would need to be replaced.

# Chapter 3

# Neural Network Modeling

Writing good neural network models can be ensured by using coding techniques that are expandable, portable, and readable. Implementing these goals can be made easier by using three coding techniques called dynamic allocation, table dispatch, and event-driven programming.

Expanding the models to include more paradigms and interactive user controls can be a problem. Most neural network models start out small, but as one gets more finesse in coding and learns new paradigms,[1] the code seem to grow to the point of being unmanageable. Spaghetti code is likely and poor memory management is inevitable.

Portability can be an even bigger problem. Contrary to popular belief, there is no such thing as portable software. Something in your code always behaves just a little differently on other types of machines. Windows don't close quite the same way, or worse yet, one compiler can ignore mistakes that next machine won't. Without proper care, adapting the code for one type of machine always ruins it for another. The solution is to isolate as much as possible with separate modules, so when a compiler or runtime error is encountered during the porting process, it can be found quickly.

Finally, code should be readable. Besides the usual suggestions[2], code can be improved by using data structures. Data structures can act as symbols for large concepts. Instead of many lines at the top of the page allocating all the various elements of a neural network, only one allocation statement is needed to communicate with all modules. Dynamic allocation is the best way to implement compact and logical data structures.

For neural networks, the solution to these constraints lie in these three important coding techniques.

[1] A paradigm is an algorithm which includes the training and propagation rules for a particular neural network architecture.

[2] Include lots of white space and in-line comments.

- Dynamic Allocation
- Table Dispatch
- Event Driven Loops

Dynamic allocation allows runtime tradeoff between each of the large memory items. Table dispatch improves flow of control by reducing the number of **if,then**, and **switch,case** type statements from the code. Dynamic allocation allow data structure element to consume only the amount of memory required for the problem at hand.

Event driven interruption take advantage of hardware speedups for the user interface. Menus and user control functions can be called from any points in the program. Normal interrupt vectors[3] can be replaced by your own.

Application of these techniques requires a good initial design and techniques of software engineering.

To some, the term *software engineering* carries the same internal contradiction as, military intelligence, portable software, marital bliss, and jumbo shrimp. Still, one can't dismiss basic elements of clean software design.

The following section is a guide to constructing neural network models using an object-oriented design (OOD). While not the only approach, the software design shown below is based on many simulation models constructed for digital machines. The examples assume you are comfortable programming in **C**; if not, **C** is probably the easiest language to learn quickly.

## 3.1 Software Design

Simulator design for artificial neural networks involves many common elements, data structures, and procedures, even though the propagation and training procedures may vary greatly. Because of the similarities, a procedure will be discussed to reduce storage requirements and improve the program flow of control.

A neural network model is composed of a number of independent software modules. Changing the topology should require only small local changes within a particular module. Changing the old network to include new training rules or display features should effect as few modules as possible. Although neural networks have been applied to many classes of problems with varying requirements, each design uses about the same types of subroutines, so the first level of design decomposition is usually the

[3] An interrupt vector is the address of a routine that is called when a hardware interrupt is encountered. For example, when the control-C is pressed a routine called **kill** is called exiting the program, cleaning up the loose ends and returning control to the operating system.

same. Object oriented design can be used to identify the primary objects and methods[4].

At the highest level of abstraction, a neural network will consist of: a routine to fetch input vectors, a propagation algorithm to feed the vector through the network, a training routine for modification of weights, a graphic-display package, and an analysis routine for periodic testing. Important lower-level modules include the initialization procedure and an event-driven menu to control the user interaction with the network.

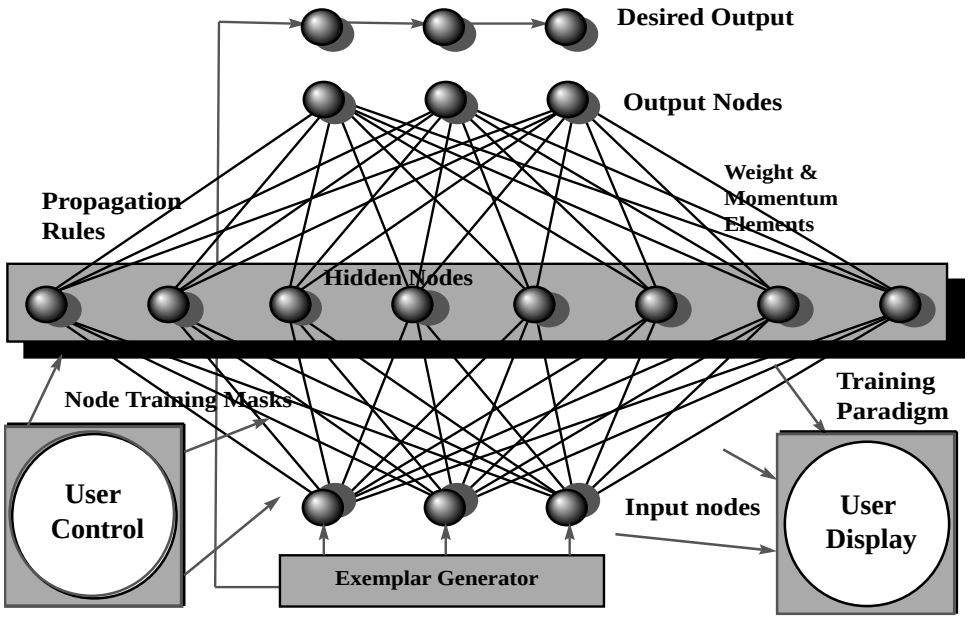


Figure 3.1: Elements of a Neural Network Simulator

Object-oriented design, as applied here, means memory management by means of logical data structures (objects) and limited operation on those structure (methods). The goal of good software design is to minimize the interaction between modules (coupling), and maximize the logical grouping of function within each module (cohesion). In practice, high cohesion and low coupling tend to be contradictory goals.

As applied to neural networks, good design can be facilitated by use of three programming techniques: 1) memory management using data structures and dynamic allocation, 2) logical flow of control using table dispatch, and 3) user interfaces built on event driven interrupt procedures. These

[4]A method is OOD's name for a procedure which can only operate on one object.

methods involve the use of variable and function pointers.[5]

Pointer programing allows dynamic rerouting of the program flow by substituting one function pointer for another rather than a direct test of a flag which must be evaluated everytime the decision is encountered. Instead an assignment statement in the initialization process replace **if,then** statements in the training loops.

### 3.1.1 The Top Level Design

Top down design[6] starts with the largest most abstract requirements, then recursively refines each level of design to include closer and closer attention to detailed requirements.

At the top level, a neural network simulation consists of two loops: the network loop which is controlled by the programmer, and the event loop which is usually associated with the hardware. The event loop is the hardware and software structure which watch the input devices, i.e. keyboard and mouse for activity.

The neural network loop consists of: a routine to select the input vectors, a propagation function to feed the vector through the network and compute the output of all the network nodes, a training routine for changing the weights, a graphic display function and an analysis routine for periodic testing.

The event loop is based in the hardware and monitors the keyboard for keys being pressed or the mouse being moved. This loop allows the user to change the flow of control in the training process to for such functions as saving the weights, changing the network topology, or eliminating nodes.

Outside both loops is an important lower level module which provides for the initialization of the graphics hardware and net topology. In pseudo-code the general flow of the network is shown below:

---

[5] **C** allows the user to work not only with a variable but the *address* of a variable without resorting to assembly language. Thus, two type of variable are define. Variables whose contents is a value, and variables, called variable pointer, whose content is not a value but a memory address where the value can be found. Pointer can be very powerful because they can be defined for functions and complex data structures as well.

[6] The Hackers Guide defines top down design as a process which can only be implemented bottom up, much as accurate flow charts are possible only *after* the code is written.

```
begin
INITIALIZE
loop {
      MAKE_INPUT
      PROPAGATE
      TEST
      TRAIN_NET
      DISPLAY_NET
      Event Handler}
end loop
end
```

### 3.1.2 Objects and Methods

Object oriented design requires that logically related software components be grouped into single program modules. The effect of OOD is to force good design on a software system by ensuring small changes in the code won't cause bugs to ripple through the system causing unexpected side effects.

This is done by creating two type of software elements: objects, and methods. An objects something which methods do things to and a method is something that does things to objects. Sort of like a noun and a verb. While not a hard and fast rule, an object is usually a data structure, and a method is usually a function call.

The principle object involved with a neural network is some type of data structure for storing the network configuration, weights, nodes etc. The net is made up of a number subobjects, some obvious, like layers, nodes, and weights, and some not so obvious, like pointers to propagation and training rules, and connection information.

Parallel to the objects are the methods, or procedures that act upon the objects. In this case, the net can be *displayed* or a layer can be *propagated*, *updated*, *tested* etc.

Objects and subobjects can best be represented in the form of data structures: a network and a set of layers.

```
typedef struct{
            int layers;
            float *input;
            float *output;
            float *desired_output;
            short *out_mask;
            layer *layer[10];
```

```
} NET;
```

NET contains variables to declare its input, output, desired output and an array of layers. Layers are a substructure to NET, and are described below.

```
typedef struct {
        int size_input,size_output;
        float *input;
        float **weights;
        float *mask,*out_mask;
        float *output;
        float *theta;
        float *delta;
        float **momentum;
        void (* update)();
        float (* propagate)();
        } LAYER;
```

The first object, the **net**, is made up almost entirely of pointers. Each of the subelements of the network that make up the structure may require only a few bytes of memory to direct the program under the event loop. The only element that is actually stored in the net is the actual number of layers.

Those methods which act on the net have all the information they need to process all the elements. Propagating and training of layers can be treated in a systematic way, a single loop, index on the layer number. Several simulator carry this type of generalization even further. Pointer and indexes are used for every weight

Pointers are filled in using dynamic allocation for memory storage and function pointer for table dispatch.

### 3.1.3 Memory Management

The key to handling large neural network problems is to use dynamic allocation for all large data structures. Here all layers, weights, nodes, and masks ( which give the ability to turn nodes on or off) are allocated at run time. This allows automatic trade-off of exemplar space for weight and node storage between layers.

The first step in dynamic allocation is to allocate the network pointer. By including the **typedef** structures in a definitions package included by each module the network can be allocated as follows.

```
NET *net; /* allocate a pointer to the net;*/
```

net = (NET *)malloc(sizeof(NET)) /* fills the pointer; /

The process is recursive, filling in each of the other items in the structure. The structure includes an array of layers. The initialization of the net must allocate the layers as well then fill in the separate layers, using malloc for each item.

Adding to the flexibility, is the use of two pointers to functions in the data structure. This allows the program, during run time, to locate the propagation and updating rules. The flow of control follows a thread of pointers rather than make a decision every time based on the network topology.

This allows the layers to be self-contained and alternate algorithms are possible by changing only one line of code. The connectivity scheme is also stored in the data structure. Only the storage for the output is allocated during initialization. The layer input pointer net->layer[n]->input is filled in with the output pointer of some other layers. These features allow complex connection schemes with variable training rules.

### 3.1.4 2D Dynamic Allocation

Here is a simple, yet very powerful **C** trick.

There are three ways to represent weights matrices in memory: static arrays declared at compile time, single pointer array dynamically allocated at run time, and double pointer arrays declared at compile time. Declaration statements for each are show below.

```
float a[5][6];
float *a;
float **a;
```

Double pointer arrays seem to be the best way to declare a weight matrix. Double pointers solve the problems of passing two-dimension arrays to subroutines without a lot of overhead.

The most important tricks used in building neural nets is the use of the double pointer arrays. There is a special feature [7] in **C** which can allow array elements of the form a[i][j] to be treated in two different ways.

**C** can treat array pointers in two different ways. The way most programmers are use to seeing two dimensional array is in the form of the static declaration:

$$float\ a[5][6]; \tag{3.1}$$

While perfectly adequate for small nets, technically this is a single pointer array (*a) . The address of the value is calculated by taking the

[7] Software engineering defines a *special feature* as a documented bug.

first index multiplying it by five (as per the example) then adding the value of the second index, finally adding the value of $a$ (which is an address) to determine the address of $a[i][j]$.

If one uses this type of arrays pointers for neural net weights, it can get pretty messy. Single pointer arrays cannot be passed to function except as (*a) or (a[][5]) Getting a single function to handle a variable sized array requires a lot of patch-up work, and the code becomes a little unreadable. A better method is to use double pointer array.

Double pointer array (**a) find the the value of a[i][j] by going to the address found at a[i] then adding $j$ to find the address of the value a[i][j]. Notice, the compiler can handle the same statement, a[i][j], two different ways depending on whether $a$ was declared with as a[5][5] or **a. With the second method, two-dimensional arrays can be passed and manipulated with impunity. Here's a function to allocate memory runtime. 1

```
float **matrix(x,y)
int x,y;
{ float **a; /* Pointer to x by y matrix                        */
             /* first allocate for columns  pointers           */
a = (float **)malloc(sizeof(float *) * x);
             /* Now allocate rows by filling column pointers. */
for(i=0;i<x;i++) a[i] = (float *)malloc(sizeof(float)*y);
return a; }
```

This technique will greatly ease expansion of your system.

### 3.1.5 Table Dispatch

Another common trick is called table dispatch. Just like a variable can be specified as a value or an address, a function can too. Consider the code below.

```
extern void backprop(),radial_basis(),counterprop();
                     /* Tell compiler
                        functions exists
                        elsewhere.           */

void *train();                  /* declare a function pointer       */
initialize_function()           /* Initialize program to backprop   */
{ train = backprop; }
```

Now after the initialize routine has been called, every time the train routine is called it will always do the backprop routine.

## 3.2 Initialization Module

The initialization routine has two functions. The most important is the establishment of the network in memory. In addition, the weights and thresholds must be filled either by a random number generator or by a stored file from a previously trained net.

After the layers are allocated and interlayer connections are made, the address of the update and propagate functions can be added to the structure.

The second function is the equipment check to determine the nature of the graphic displays and to initialize the video drivers as necessary. The size of the screen, color planes, and graphic capabilities will determine exactly how much data can actually be displayed.

As might be expected, the graphical displays are the most difficult part of the problem, especially when rehosting a system. In general, the design should consider using three files. Isolation of the geometric primitives and

placing windowing opening function in one module while the actual display routines are in a third module.

The greatest single software invention is probably the **C** printf group of display routines; however, it has no place in complex graphical displays. The alternative is to replace all printf's with sprintf's and use graphics to display the strings. This simplifies porting by allowing common display functions between windows and character displays.

When using displays such as dumb terminals, use the curses packages (instead of *printf* (#include <curses.h> and the flags -lcurses, and -ltermcap) to allow portability over a wide range of terminal types. It allows optimal screen updates (only screen changes are sent over the communication link) and the package looks up the control codes for cursor movement based on the type of terminal stored in the termcap definitions.

## 3.3 The Main Program Loop

The main program loop consists of the MAKEINPUT module, the PROPAGATE module, the TRAINNET module, the TESTNET and the DISPLAY and SHOW modules. A counter is used to inhibit the calling of some modules through the loop to reduce computations. For example, the screen may be updated only every few hundred cycles. However, each pass through the loop represents one training cycle, so MAKEINPUT, PROPAGATE, and TRAINNET will always be called.

### 3.3.1 Making an Input Vector and Desired Output Vector

A set of input patterns and a defined classification, are essential to the training of a neural net. In other words, an input vector and a desired output (doft) are needed for every update cycle.

This MAKEINPUT() function fills the net data structure with a randomly selected exemplar from the exemplar or test set. The module allows for calculating the vectors or taking them from a stored data set read in during initialization. Also, the data preprocessor may add Gaussian noise to vectors. The work of Sietsma and Dow indicates that adding noise distributed across the actual statistics of the data seems to improve the performance of the network.[1].

#### File Input for Exemplar Sets

Most problems can be described in terms of a set of input vectors and a prescribed classification. An exemplar is selected randomly from the pool whenever the makeinput routine is called. Also, the routine should insure

that in addition to random selection on exemplar number there is also random selection based on class type. This prevents excessive training on a single class when the classes are not evenly distributed in the input file.

The MAKEINPUT package can contain other modules necessary for handling the data files. The three principle functions contained in the package allow for initializing the data storage, returning a random exemplar to the main routine, and statistical preprocessing of the data.

### 3.3.2 Propagating the Input Vector

With the organization of the net in memory, the net can begin to learn and classify data. To use the net, the rules for propagating the data from the input to the output must be specified. This is the purpose of the Propagate package.

The primary design goal for simulators is to reduce the amount of software that would need to be altered to suit a new experiment.

The feedforward routine is really only a hook to the real feedforward method, pointed to in the *layer* data structure. The function pointed to in the *layer* data structure will only update one node, so the feedforward routine will loop though all the nodes.

### 3.3.3 Updating the Weights

The TRAINNET module specifies the training algorithm for the network. Again it is only a hook to the real update routine. Because the update of a particular node may depend on all nodes in a layer, the update routine will update an entire layers weights. The source code below is taken from a paradigm package to update the weights using standard backprop with momentum. [8]

```
#include "definitions.h"
#include <math.h>
extern nnet *net;

extern setup parameters; /* Setup is a data type for the
                            size of the net.  It include
                            numbers of node in the hidden
                            layers ect.          */

extern float calcy() ;  /*  Propation rule for backprop  */

void TRAIN_LOWER_LAYER(),   /* Training rules for backprop */
```

[8] Because neural networks involve a lot of training loops, a useful macro to replace the **for,next** structure is provided by the defintion # define loopi(A) for(i=0;i<A;i++)

```
     TRAIN_OUTPUT_LAYER();

init_train_BACKPROP()

{int i;
 init_net(parameters); /* Allocate net according to
                          sizes in a parameter structure. */
 loopi(parameters.layers){
                net->layer1[i]->updater = TRAIN_LOWER_LAYER;
                net->layer1[i]->eta[0] = 0.3;
                net->layer1[i]->eta[1] = 0.7;
                                        }
                net->layer1[parameters.layers-1]->updater
                                    = TRAIN_OUTPUT_LAYER;
                loopi(parameters.layers)
                net->layer1[i]->propagate = calcy;
}


void TRAIN_OUTPUT_LAYER(l_on,l_up)
layer *l_on,*l_up;
{ int i,j,k,lower,upper;
  float neta,momentum;
  lower = l_on->size_input;
  upper = l_on->size_output;

  neta     = l_on->eta[0];
  momentum = l_on->eta[1];

dely(l_on->output,net->doft,l_on->delta,upper);

loopj(upper) {
    l_on->theta[j] =  l_on->theta[j] + neta * l_on->delta[j] ;
    loopi(l_on->size_input) {
      l_on->momentum[j][i] = (neta * l_on->delta[j] * l_on->input[i]
                                  + momentum * l_on->momentum[j][i])
                                  * l_on->mask[i];
      l_on->weights[j][i] += l_on->momentum[j][i];

               }}
}
void TRAIN_LOWER_LAYER(l_on,l_up)
layer *l_on,*l_up;
```

```
{ int i,j,k,lower,upper;
  float neta,momentum;
  lower = l_on->size_input;
  upper = l_on->size_output;
  neta = l_on->eta[0];
  momentum = l_on->eta[1];
  delx(l_on->output,l_up->delta,l_on->delta,
      l_up->weights,l_up->size_output,upper,j);
  loopj(upper) {
     l_on->theta[j] =  l_on->theta[j] + neta * l_on->delta[j] ;
         loopi(l_on->size_input) {l_on->momentum[j][i]
                                   = (neta * l_on->delta[j] *
                                     l_on->input[i]
                                   + momentum * l_on->momentum[j][i])
                                   * l_on->mask[i];
          l_on->weights[j][i] += l_on->momentum[j][i];
               }}
}
```

### 3.3.4 Measuring the Error

The TEST routines allows this type of checking mid-process by running a quick test set through the net, then measuring the performance. The test set is specified in the initialization routine. When the data is read into memory during initialization, the first line of the file specifies the number of training exemplars followed by the number of test exemplars. This partitioning of the data allows the TEST routines to test the net with a set of vectors the net hasn't seen before.

### 3.3.5 Displaying the Progress

The display routines refer only to the graphics portions of the display. These include such functions as drawing the network weights, drawing the nodes, setting the colors, finding data ranges, and drawing color bars. The package is split into two types of graphics routines. The basic set contains those functions that are machine dependent. In general, a macro is used when possible to allow for redefinition to other machines. Those which are not machine dependent are combinations of the basic routines that are machine dependent. An example would be the color bar routine. These two types of routines are included in the graphics package. The group of routines that are specific to a particular topology are contained in the display package. In general, when the problem under consideration changes from something like a Kohonen map to a counterpropagation model, the entire

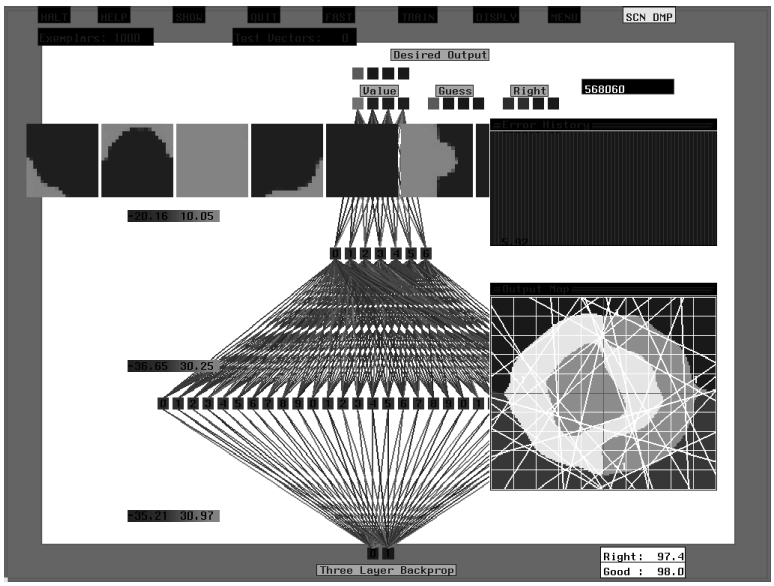


Figure 3.2: Exampler:Display for the Neural O-scope

display package is replaced. The graphics package is used as an include file so the process of building a new display is somewhat reduced.

### 3.3.6 Interactive Program Control

An event-driven menu provides control for housekeeping functions of the network. Event-driven routines are more efficient than using a keyboard poll.

Main frame computers under UNIX allow trapping come keyboard events like the control break. Event-driven menus require a hardware event to call the menu subroutine. No device polling is necessary. The event, in this case, is typing a control **C** on the keyboard. A control-C activates a hardware interrupt to the kill-address vector. The program has substituted the normal kill-address vector with the menu-address vector. Personal computers allow the devices (keyboard or mouse) to be checked for events.

While the main purpose of the menu is to allow the user to save and restore weights, the menu also allows control parameters to be changed while training is in progress. A menu display in a text window offers a series of selections. Selection of a particular item will then prompt the user of the parameters associated with the particular call.

Adapting a neural network development system to the wide range of problems requires a flexible design that allows rapid prototyping. The

design shouldn't make restrictions on the size of the network (other than hardware limits) or the size of the input and output vectors. Such decision should be left as much as possible to runtime.

## 3.4 Adding Your Own Paradigm

Almost all the work in constructing a neural network is putting together the graphical displays. If an investigator wants to add a paradigm his code the code, and retain the same displays the procedure is as follows:

- Add function name to menu with a function like add_menu_item(); Make sure external declaration is included for initialization routine.
- Copy a paradigm package to a new name for your paradigm.
- The new package must contain an initialization procedure, A training procedure, and a propagation procedure.

A common difficulty in building large neural network models is allowing for all the different configurations possible. The problem is compounded by addition of variable numbers of layers, weights, and training parameters. Unless the design requirement are tolerant of endless **if .. then ... else** and **switch,case:** statements, a better means should be found to direct the flow of control through the program using different propagation and training rules.

The function of the paradigms package is to facilitate the addition of new training rules. The procedure to add a new paradigm to the software is to build three routines: an initialization routine, a training or update routine, and a propagation routine. The add_a_paradigm() routine places the address of the routines in the appropriate dispatch table. The paradigm rules are each placed in a separate software package.

## 3.5 Conclusion

While the suggestions above are not the only way to write a neural network, the presented here can allow a flexibility not possible with many other methods.

Source code for the Neural Graphics software is available by anonymous FTP from AFIT using the arpanet. The address is blackbird.afit.af.mil or node number:129.92.1.2. The user name is *ftp* and password is *ftp*. Transfer the file tarrtar.Z which is a compress, tar file.

# Bibliography

[1] Jocelyn Sietsma. Neural net pruning -why and how. *Proceedings of the IEEE International Conference on Neural Networks*, 1988.

[2] Gregory L. Tarr. AFIT neural network development tools and modeling artificial neural networks. In *SPIE Symposium on Applications of Artificial Neural Networks*, 1989.

[3] Gregory L. Tarr. Neural network development tools at the Air Force Institute of Technology. In *Proceeding of the 6th Annual AAAIC*. ACM/SIGART, 1989.

[4] Gregory L. Tarr. Artificial neural network implementation of the karhuenen-loeve transforms for feature vector reduction. In *Proceeding of the 7th Annual AAAIC*. ACM/SIGART, 1990.

**Biography**
**Gregory L. Tarr**

Captain Gregory L. Tarr was born on September 18, 1952 in Roseburg, Oregon. Capt Tarr entered the Air Force in December of 1978 as an Aerospace Ground Equipment Technician. After serving two years at Castle AFB, California with the 84th Fighter-Interceptor Squadron, he entered the Airman's Education and Commissioning Program completing a Bachelor of Science in Electrical Engineering at the University of Arizona in December 1983. Capt Tarr served three year with the Aeronautical Systems Division at Wright-Patterson AFB, Ohio as an Electro-Optics Systems Engineer for several intelligence collection platforms. He completed a Master of Science degree in Digital Systems at the School of Engineering, Air Force Institute of Technology (AFIT) in December 1988.

Capt Tarr is currently pursuing a doctoral degree at the Air Force Institute of Technology in Optical Information Processing and Artificial Neural Networks.