<u>NNV</u>
(Neural Network Viewer)

EE265 - Independent Study Report
Instructor: Professor Francesco Palmieri
Student: David J. Ferrero

Abstract

The following report gives a brief description of the project objective in the Introduction, and a detailed explanation of the steps taken to design and build the NNV (Neural Network Viewer) application (app) for the NeXT Computer in the Design section. It also includes a discussion of the algorithms tried and used as well as the difficulties faced during the design and coding stages of the app. The conclusion contains some ideas, which if implemented would improve the NNV app. Overall, the NNV app proved to be a challenging project to design and build, and greatly increased the author's appreciation for the NeXTstep programming environment and improved his skill with the Objective C language.

Introduction

In studying neural network algorithms and behavior, visualization of the neural network activity and connectivity can prove very useful. With that notion, the project was conceived and design begun. The goal being, to design a software system on the NeXT computer to graphically display an arbitrary feed forward neural network. Each neuron was to be represented in a window with connections to associated neurons within the network. The strength of the connections between neurons as well as the activity level within each neuron was to be represented graphically. The NNV app was to propagate input values through the network and compute the outputs as well as the activity level according to some feed forward rules and a network definition. The input values as well as the network definition were to be supplied by the user in user created ascii files.

Design

In planning and designing this project, I first set out to decide what inputs the program should expect, and the format for each input. Since the program was to calculate values based on sets of inputs, the best format for the input sets would be rows of numbers stored in a file where each number was to be separated by a space, and each row was to be terminated with a carriage-return/line-feed. The format expected by NNV for a .input file is as follows: each row in the .input file must contain as many numbers as inputs to the .network file opened. For example, we might have test.input as

0.34 0.78 0.45
0.24 0.56 0.87
0.79 0.88 0.12
test.input file

where we have 3 input numbers (1 for each input neuron) and 3 input sets to cycle through. The network definition was also to be a file. It was to describe the network and the strength of the connections between neurons. The other information needed be the program was the number of neurons, and the number of inputs and outputs. It was determined that the network description would contain this information by observing some rules about a network. Also, in order to store the network description into memory dynamically, the number of neurons would need to be known before reading in the network description. From the network file, the number of outputs of the network can be derived since the matrix columns on the right, corresponding to the outputs, will contain 0's. The number of inputs to the network can also be derived by counting the top rows with all 0's. Therefore, the format expected by NNV for a .network file is as follows: the first line contains one number (n) representing the number of neurons in the network. The next (n) lines contain the neuron connection matrix of numbers corresponding to the strength of the connections between neurons. Call this matrix C with elements referenced as Cij (i = row, j = col). Cij is the strength of the connection from the jth to the ith neuron. Therefore, the network file might be called test.network and be stored as follows:

```
6
0  0  0  0  0  0
0  0  0  0  0  0
0  0  0  0  0  0
.4  0  .2  0  0  0
0  .7  .6  0  0  0
```
test.network file

This example file contains a network description of an 6x6 matrix which contains 3 inputs and 2 outputs with 1 neuron in the middle. The numeric output format of the NNV app was also determined to be a row of values corresponding to the number of outputs defined in the network description. The main output of the program was to be the graphical display of the network which would show the neurons and the connectivity strength between them, as well as the level of activity within each neuron according to the input set supplied.

After the input and output formats were initially established, I set out to develop the functionality of the program. I began by prototyping the main body on an IBM XT using Turbo C. This could easily have been done on the NeXT using ANSI C, but the IBM system was more accessible. The prototype was to do all that the final application would do with the exception of the graphical interface. The graphical interface could have been done using Turbo C graphics libraries, but this would defeat the purpose of using a graphical environment like the NeXT and would have required much more time and effort. The prototype program would read in the .network and .input files, and perform the neuron calculations, propagating the results to the output nodes which were printed.

The calculations for a feed forward neural network are performed as follows: each node j feeds only the nodes with index i > j. X1, X2, ... Xn will be the input values read from the .input file and this input will propagate through the neurons according to matrix C (read from the .network file) and function f(value). The output of neuron Cj will equal the result of the sum of the input values to neuron Cj multiplied by the connection strength from j to i after calculating f(x). For example, using the test.network in the above example, and the first set of inputs from the above test.input file, X1 = .34, X2 = .78, and X3 = .45, the following calculations would be performed:

Value of node 1 (X1) = .34
Value of node 2 (X2) = .78
Value of node 3 (X3) = .45
Value of node 4 = $X1*C_{41} + X3*C_{43}$ => f(.226) = .556
Value of node 5 (output 1) = $X3*C_{53}$ + f(.226)*C_{54} => f(.749) = .679
Value of node 6 (output 2) =>$X2*C_{62} + X3*C_{63}$ => f(.816)= .693

where $f(x) = 1/(1 + e^{-x})$. Once the input and output file I/O was coded and memory allocated dynamically for the neural network and the intermediate values, all that was left was to code the propagation functions. These would take the input values and calculate the activity in the neurons through the output nodes according to the description above. The propogation function consists of two nested "for loops". The outer loop counts from i=#inputs to i=#neurons, and the inner loop counts from j=0 to j <i. Thus, x (intermediate neuron value) equals the sum of network[i][j]*value[j] while j iterates from 0 to j < i. Then the next value[i] equals f(x) as given above (see NeuronCntrl.m for more detail). When this was complete, the prototype read in both the "test.net" and "test.inp" files and calculated the interior and output values for the neural network. The output was then printed to the screen.

Since the NNV app was to run on a NeXT Dimension computer and display the neural network and activity graphically, the language of choice to code the app was Objective C, the primary language for the NeXT computer. Interface Builder (IB) was used to build and design the graphical interface and serve as the development environment for the project. I initially decided that the input and output nodes would be filled circles on the left and right side respectively of the main window. The input circles would be connected to the interior circles and these would be connected to the output circles via lines. All the graphics were to be displayed in a window using display postscript (the graphical language of the NeXT) and consist of colored circles (neurons) and lines with corresponding thicknesses (nerve connections) ie: thicker nerve connections represent stronger connections between neurons. The neurons themselves were to be colored according to their "activity" or value of the number calculated from the input nerve connections. If NNV is run on a NeXTstation color or NeXT Dimension system, the connection strengths and neuron activity will be represented with color. If NNV is run on a NeXTstation or NeXT Cube equipped with a 4-gray monitor, activity and strengths will be represented with dark and light grays and dithered for better numeric representation. On a color equipped system, red neurons represent very active neurons (activity = 1), blue neurons represent no activity in the neuron (activity = 0), and various shades of red/blue depending on the activity (somewhere between 0 and 1). Red connections represent (+) connections. Blue connections represent (-) connections. Connection thickness varies with connection strength. In addition to the scrollable graphical view, there will be a single "Next Input Set" button which when pressed will draw the neural network according to the next input set and activity calculated.

So far I have omitted the description of just how the circles were to be arranged in the graphics view, other than the input and output neurons. This proved to be the most difficult part of the design for various reasons. The main difficulty was derived from the fact that any neuron (i) could be connected to any neuron i+1, i+2,...i+n. I wanted to have little or no lines (nerve connections) cross through the body of another neuron. The two solutions I initially used were based on a parabolic interior layout. The input and output neurons would be spaced evenly along their respective sides, and the interior neurons would be draw along a positive parabolic branch. That way, any neuron along the path could be connected to any other neuron without having the nerve connection cross through the body of of another neuron. The second idea was to use both sides of the parabola but turn it so that the base of the parabola would be at the output neuron side, and the widening end of the parabola would be at the input side. The idea was to alternate the interior neurons on either side of the parabolic branches. Initially, I chose the first method since it was easier to implement. I also added a slider control to the interface which would "adjust" the parabolic path by changing the steepness of the curve to allow different neurons to be examined. This design met the solution, however, the parabola branch "grew" too high too quickly to be viewed easily on the display even with a scrollable view. Also, the goal was to plot the neurons in a manor similar to the "typical" way a neural network might be plotted. In the end, the neural network was divided into "layers" from the beginning so that the neurons which don't connect to the next neuron in the sequence would be on the same Y axis. The algorithm which divided the network matrix into the layers is too complex to describe in detail here (see NeuronView.m), however the essence of it was as follows: starting at node 1, search for the largest sub-matrix of zeros around the diagonal. Each of the nodes associated with the sub-matrix of zeros was placed on the same vertical layer. Then this would continue, until the entire matrix was searched. The input neurons are all on the same layer since no input neuron can be connected to another input neuron. The interior neurons are divided up according to the algorithm, and the output neurons usually are all on the same layer. If the neuron before the 1st output neuron is not connected to the 1st output neuron then these neurons will be on the same layer, and the other output neurons will be on the next (final) layer. The output neurons are easily distinguished from the interior or input neurons since they have NO outgoing lines (nerve connections). So although some nerve connections cross the body of other neurons, the arrangement of the neurons is closer to the actual arrangement than either of the parabolic methods. The Conclusion section will describe an improvement on the layer method.

So far, I have described the algorithms which contribute to the NNV app. Now I will describe the steps taken to construct the Objective C classes used in NNV. Only two new classes were designed for the NNV, and the others were inherited or included from the NeXTstep Appkit library.

The main new class used for the NNV app is the NeuronCntrl class which includes the following methods:

```
- appDidInit:sender;
- clearTemp:sender;
- openInputFile:sender;
- openNetworkFile:sender;
- writeOutput:sender;
- nextInputSet:sender;
- computeActivity;
- getNumOutputs;
- getNumInputs;
```

<u>NeuronCntrl methods</u>

These methods work to achieve the functionality of the prototype as well as control all the menu and interface options except the neuron size slider which will be described in the NeuronView class description.

Following, I will describe the purpose of each of NeuronCntrl's methods. The appDidInit method is called when the application starts. In it, all app initialization code should be put. The clearTemp method clears the outputstream buffer (if not already clear) so that the new output values can be buffered. This is a menu option under File. The openInputFile method opens the user specified .input file and stores the first input set into NeuronValues[]. The number of inputs in a row must be the same number as expected by the .network file opened. The openNetworkFile method opens the user specified .network file, reads the size, allocates memory for the network and input values, loads the network matrix into memory, counts the inputs and outputs, resizes the view according to the number of nodes in the network, calls a NeuronView method to display the network, and finally closes the .network file. The writeOutput method saves the outputstream to a file specified by the user, either appending to old an old file, or saving to a new file. Then it closes the file after and clears the output buffer. The nextInputSet method responds to the button in the interface. This method reads the next input set from the .input file and stores the values into NeuronValues[]. If the end of .input file is reached, the user can either do nothing by selecting NO! (don't restart at beginning), or OK, to restart at beginning of .input file. The computeActivity method is called by the nextInputSet method and uses the new input values to calculate the neuron activity and propagate the output values. The output values are then stored to the outputstream buffer, and finally, the displayActivity method is called within NeuronView object, pointed to by the outlet variable, theNeuronView. The getNumOutputs method counts the columns at the right which have all zeros. This number is the number of outputs. The getNumInputs method likewise counts the number of rows at the top which have all zeros. The number of inputs is that number of rows.

The second class which I designed for NNV is NeuronView. This class interfaced the graphical view with the NeuronCntrl object, and was responsible for updating the view according to the neuron activity and selections by the user. The following are methods written for the NeuronView class:

- initFrame:(const NXRect *)frameRect;
- displayActivity;
- initNetworkDisplay;
- adjustNeuronSize:sender;
- drawSelf: (const NXRect*)r :(int)c

<u>NeuronView methods</u>

As with the NeuronCntrl class, the initFrame method was called to initialize the new NeuronView object. The displayActivity method (called by NeuronCntrl method computeActivity) fills in a circle with color for the activity rate (output value) within each neuron at the location determined in initNetworkDisplay method. The initNetworkDisplay method finds the layers within the network matrix, determines a location within the view for each neuron, and draws lines to represent the network with color and thickness varying according to + and - values and the size of the value. The adjustNeuronSize method responds to the user adjustable slider which adjusts the size of the displayed neurons. The drawSelf method is coded in this class to supersede the View class drawSelf method. This allows the NeuronView class to implement its own "redisplay" code. This method is called by the display method which is called by either the openNetworkFile method of NeuronCntrl class, or called by the Workspace Manager if any part of the window needs refreshing.

All of the file I/O is contained within the NeuronCntrl object, and all the display postscript code necessary to draw on the screen is contained within the NeuronView object. As stated earlier, although both classes need significant code to implement, the code needed would have been much greater if the NeXTstep environment did not provide the Appkit libraries. The entire Help window required minimal effort compared with what other computer environments would require. The main difficulty in the actual coding of the app was the implementation of the ScrollView object. Whenever a network of over nine (#defined in NeuronCntrl.m) neurons is opened, the openNetworkFile method must resize the graphics view used by NeuronView so that the neurons won't get too crowded. The view in essence gets larger even thought the window containing the view doesn't. In order to see the rest of the view, the user must adjust either or both of the horizontal or vertical sliders, or resize the window. The Scrollview object and resizeable view was implemented by first defining a customview from the IB Palette, and then selecting Group In Scrollview within the Layout menu option in IB. Then I needed to specify the new size of the custom view and also specify that the scrollview object would contain both horizontal and vertical sliders. Some of this was done right within IB in the Inspector window. Other features of NNV such as the online help window, and program Icon were not added until the rest of the program functioned properly. Both of these where developed using other helpful applications like Edit, and Icon.app (a bitmap editor).

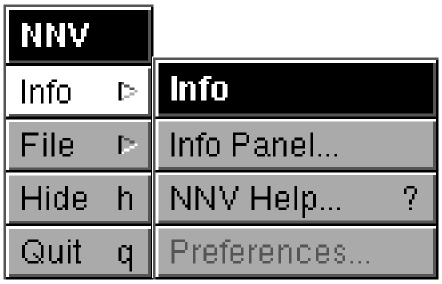
Conclusion

Much was learned through the design and development of this program including Object Oriented Design and Programming with Objective C, Display Postscript, and the steps required to complete a project specified by another. (This proved to be one of the most challenging parts of the project since I needed to understand and implement what the user wanted. Sometimes, what I understood as their need turned out to be different than they wanted, and thus, the modify and present sequence).

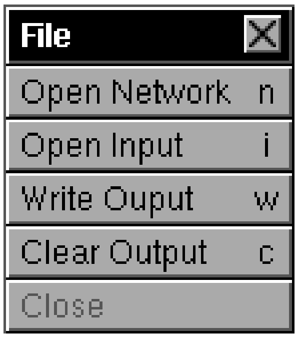
As mentioned above, although the layer method of displaying neurons is closer in reality to a typical neural network, the nerve connections are sometimes found to intersect other neuron bodies. This is due to the fact that for every nerve connection leaving or entering a neuron, there is an independent line. In reality, nerve connections follow a different approach where one main nerve might enter/leave a neuron, and where subsequent micro-nerves would branch out from this main nerve and attach to the other neuron's main incoming nerve, much like a root/branch system of a tree. The algorithms to implement such a display were beyond the scope of this project. If implemented though, the NNV app would take on a new level of neural network display technology.

The main source of information concerning Object Oriented Design and Programming with Objective C, and Display Postscript was taken from a free NeXT manual called The NeXTstep Advantage: Application Development with NeXTstep. Additional help was referenced from the NeXT Technical Documentation and online Technical Documentation as needed. For an explanation of terms used in this report such as Class, Object, Outlet, and Method, please see The NeXTstep Advantage... obtained at your local NeXT reseller.

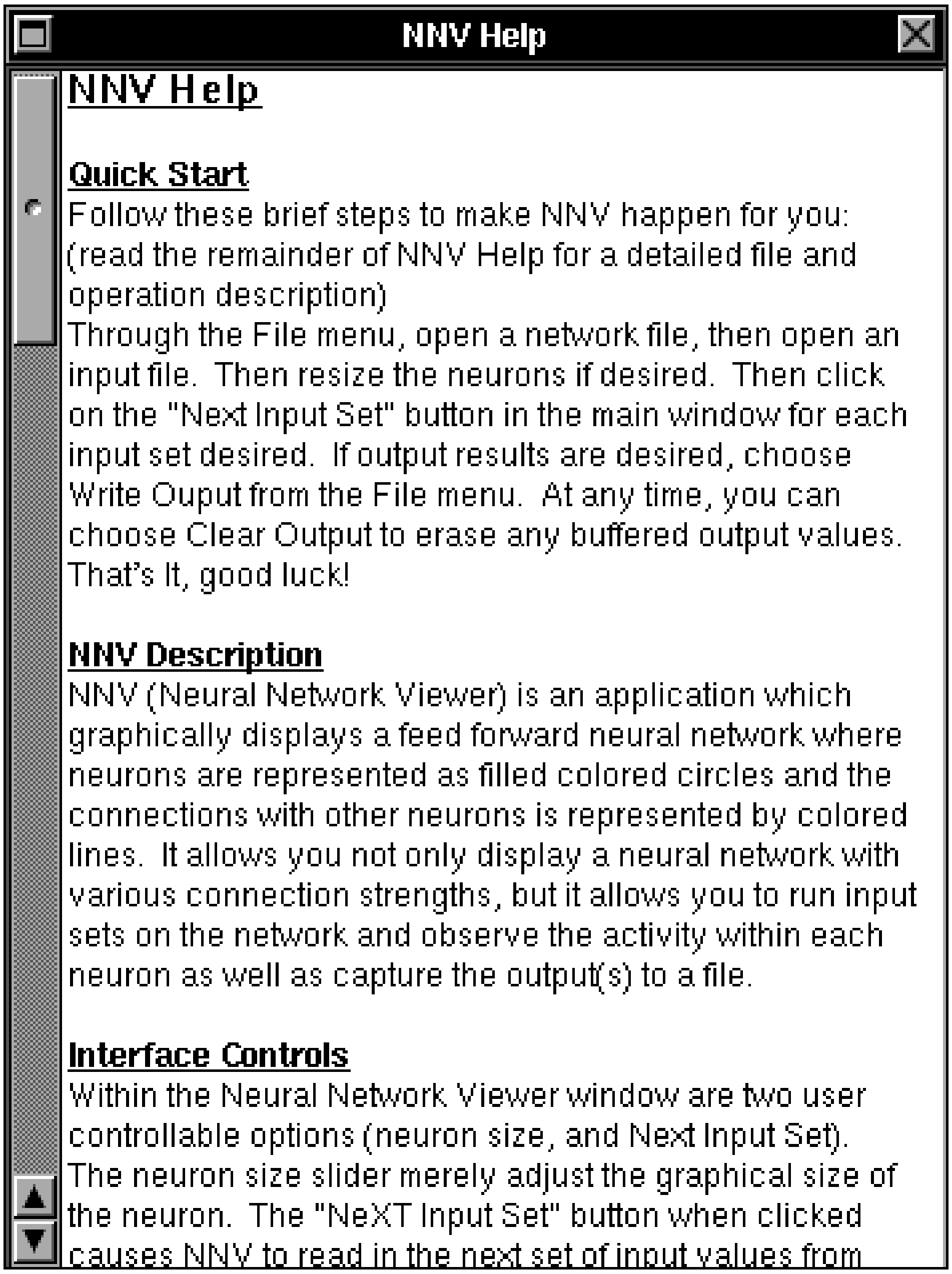
Below are some pictures of the NNV application menus, help window, and main graphics interface.



Main menu of NNV with Info menu



File Menu of NNV


NNV Help

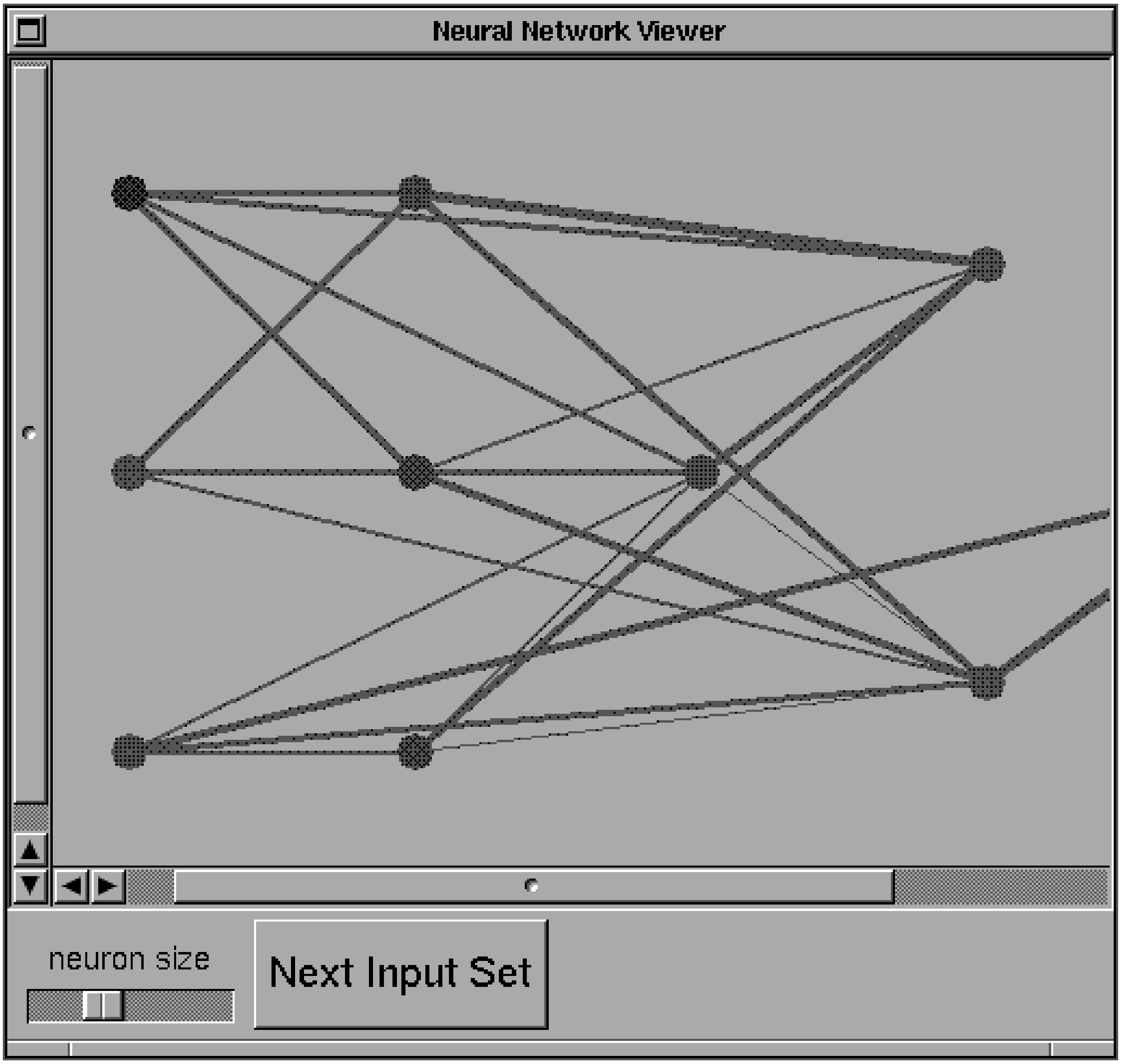
NNV Help

Quick Start
Follow these brief steps to make NNV happen for you:
(read the remainder of NNV Help for a detailed file and operation description)
Through the File menu, open a network file, then open an input file. Then resize the neurons if desired. Then click on the "Next Input Set" button in the main window for each input set desired. If output results are desired, choose Write Ouput from the File menu. At any time, you can choose Clear Output to erase any buffered output values.
That's It, good luck!

NNV Description
NNV (Neural Network Viewer) is an application which graphically displays a feed forward neural network where neurons are represented as filled colored circles and the connections with other neurons is represented by colored lines. It allows you not only display a neural network with various connection strengths, but it allows you to run input sets on the network and observe the activity within each neuron as well as capture the output(s) to a file.

Interface Controls
Within the Neural Network Viewer window are two user controllable options (neuron size, and Next Input Set).
The neuron size slider merely adjust the graphical size of the neuron. The "NeXT Input Set" button when clicked causes NNV to read in the next set of input values from

Help menu of NNV



Interface for NNV