Appendix A

Publications

Caelli, T. M., Squire, D. M. and Wild, T. P. (1993). Model-based neural networks, *Neural Networks* **6**: 613–625.

Squire, D. M. and Caelli, T. M. (1995). Shift, rotation and scale invariant signatures for two-dimensional contours, in a neural network architecture, *to appear in the Proceedings of the 1st International Conference on the Mathematics of Neural Networks and Applications (MANNA 95), Lady Margaret Hall, Oxford, published as a special edition of Annals of Mathematics and Artificial Intelligence*, J.C. Baltzer Scientific Publishing Company, Basel - Switzerland.

Appendix B

Introduction to the **Xnet** Neural Network Simulator

The **Xnet** Neural Network Simulator was written as a flexible environment in which to develop and test the Model-Based Neural Networks used in this thesis. It is written in C, for an X windows and Unix environment, using the Motif widget set. There is a graphical interface which allows the user to create networks of various architectures and connection paradigms interactively. Training and test sets can also be created using the program, but are more frequently generated by other systems.

Xnet provides powerful diagnostic output during the training of a network. Graphs of the training and test set errors and classification performances are updated after a user-specified number of iterations. The system can be made to "animate" the network's performance on a given input pattern as training proceeds, by assigning grey-levels to the nodes corresponding to their outputs after each iteration. These data allow the rapid detection of problems such as unit saturation, and allow the network architecture and training parameters for a given problem to be tuned quickly.

There is a "batch" version of the network training system which contains none of the overhead required for the monitoring of and interaction with the system required by the graphical user interface. This is optimized for fast training, and is used to train sets of networks once the architecture and training parameters have been decided upon using **Xnet**. This version still logs data during training, so that the dynamics of the training process may be investigated later, if necessary.

In all, the system consists of 11819 lines of code. It has been successfully compiled and used on both Silicon Graphics and Sun workstations. The code is available upon request via email to `squizz@cs.curtin.edu.au`.

The remainder of this appendix takes the form of three tutorials, which demonstrate the "look and feel" of the **Xnet** system, and the way it can be used to develop and analyse neural network solutions to problems. It is aimed at the novice user.

B.1 Familiarization with **Xnet**

B.1.1 Introduction

The aim of this tutorial is to become familiar with the use of the **Xnet** Neural Network Simulator. By the end, you should know how to load a neural network using **Xnet**, how to load sets of data for training neural networks with **Xnet**, and how to train networks. You will learn about the parameters used in training a feed-forward neural network using the backpropagation algorithm.

Next, you will learn how to construct your own networks using **Xnet**, and how to specify your own training sets. You will discover some of the capabilities of feed-forward networks trained with the backpropagation algorithm, and also some of their limitations.

B.1.2 Starting **Xnet**

Go to the directory where you have compiled the **Xnet** system, and simply type `Xnet` to launch the **Xnet** system. If the background colour of the **Xnet** window is blue when it starts up, it means that the **Xnet** resource database file has not been loaded for the display at which you are working. Typing `XN` in the directory where `Xnet` is located should fix this.

When **Xnet** has loaded, you should see the window shown in Figure B.1. As you can see, it is empty, except for a row of buttons on the right hand side of the window. In the course of this tutorial, you will learn what all of these buttons do.

B.1.3 Loading a Network

In order quickly to become familiar with the **Xnet** interface, it will be easiest to load a network that has been created earlier. To do this, click on the "Load Network" button. This will cause a dialog box to pop up, as shown in Figure B.2. To load a network file, you can click once on the filename, and then click "OK", or you can simply double-click on the filename. Load the network "ab.net".

When you have loaded the network, a diagram of its structure should appear in the left hand side of the **Xnet** main window. It should look like Figure B.3. In the **Xnet** program, networks are drawn with their input layer as the top layer of the network. The output layer is the bottom layer in the diagram. The network "ab.net", therefore, has a 15×15 input layer, a 3×3 hidden layer, and a 1×2 output layer.

B.1.4 Specifying an Input Vector For The Network

Now that you have a network loaded into the program, you need to specify an *input vector* so that you can run the network. The input vector specifies the outputs of

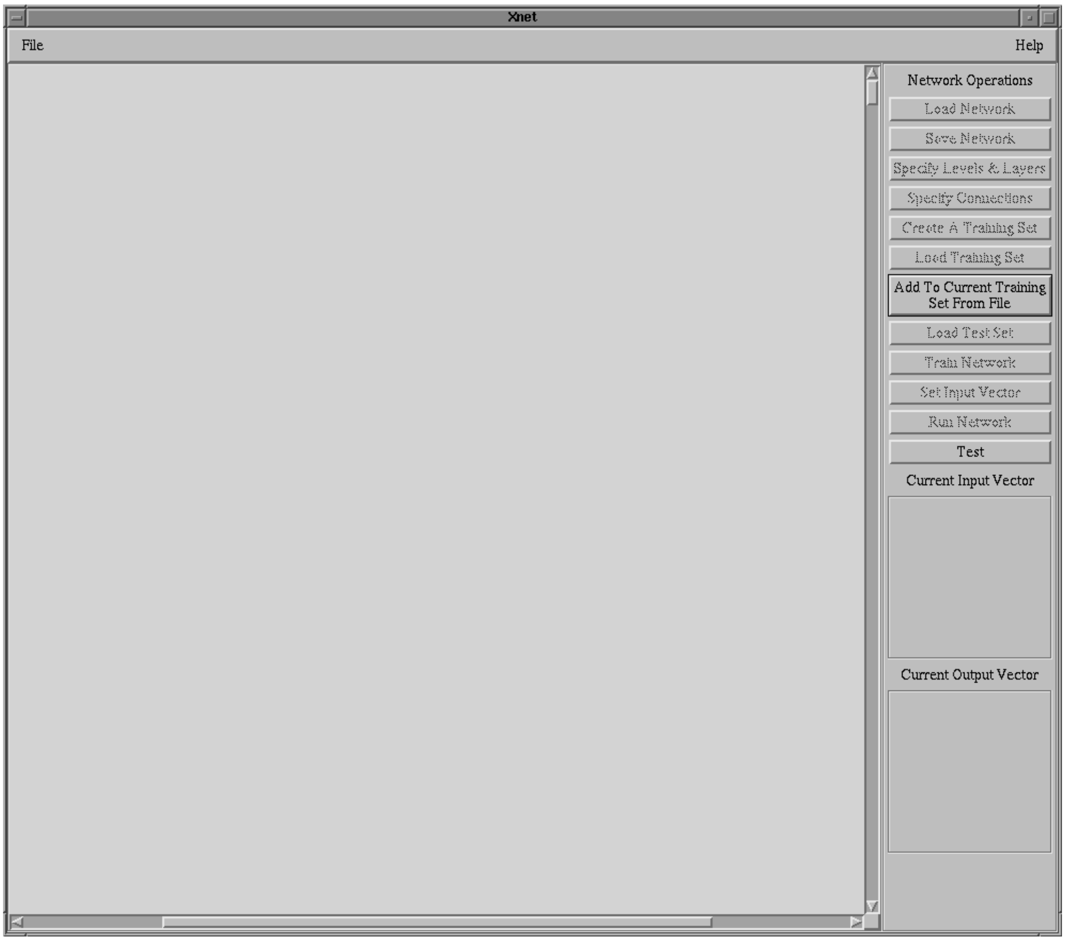


Figure B.1: **Xnet** main window on startup.

the nodes (artificial neurons) in the input layer of the network. These values are then propagated through the network to determine the output values of the nodes in the output layer.

To specify the input vector for the network, click on the button labeled "Set Input Vector". This will cause a window titled "Get Vector" to pop up, as shown in Figure B.4. This window contains tools. Most of the window (containing a hand-drawn "a" in Figure B.4) is a grid of squares of the same dimensions as the input layer of the network. Clicking in a square causes it to change to the current colour. You can draw in this area by holding down any mouse button in this area and dragging.

On the right hand side of the window is a column of rectangles of different grey levels. This is the palette from which you select the current colour, by clicking on the colour you want. Below this column is a square which displays the current colour. The various colours represent values in the vector in the range 0 to 1. Black represents 0 and white represents 1. Black and white are often the only values that you need. Along the bottom of the window are some buttons. Hopefully their functions are obvious from

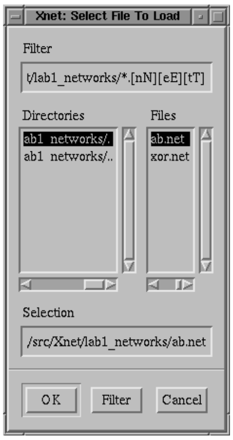


Figure B.2: **Xnet** file load dialog box.

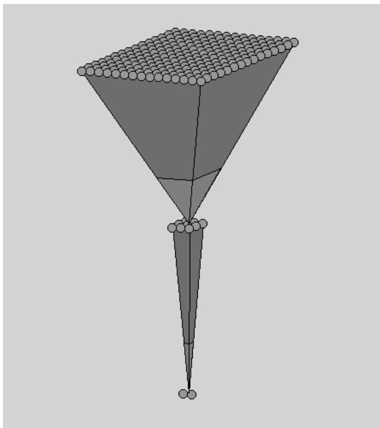


Figure B.3: **Xnet** network "ab.net".

their labels.

Practice drawing various different input vectors. When you have tried out the features available, click on "Clear to Zeros" to reset all the elements of the vector to zero. Then try drawing an "a" like the one in Figure B.4. When you are happy with your effort (it doesn't need to be a copy), click on the "Done" button.

Notice that the input vector you have just specified appears in the main **Xnet** window, below the column of buttons on the right. It should look like Figure B.5. Below the input vector, the current output vector is displayed. This is updated every time you modify the input vector, by running the input vector through the network and displaying the result. At the moment both the output nodes should have output values close to 0.5, because the weights in the network are initialized to ensure that this is the case before training. They should both be displayed as a mid-grey colour.

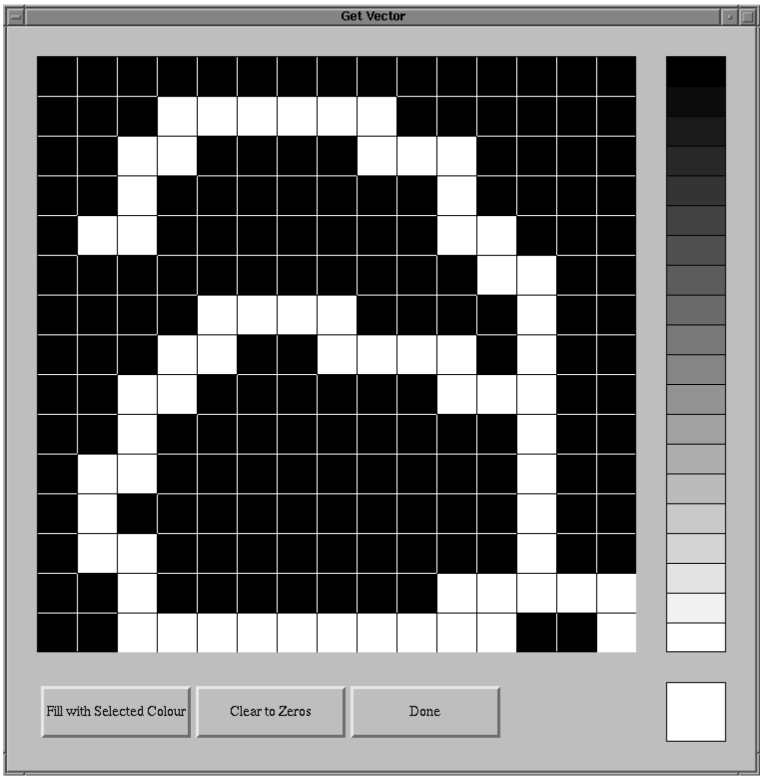


Figure B.4: Window for specifying a vector in **Xnet**.

B.1.5 Training Sets

In order for a neural network to be useful, values must be found for the connection weights so that the output of the network is correct (or approximately so) for a specified set of input vectors. This is usually achieved by presenting the network with a set of input vectors and the desired corresponding output vectors. This set of vectors is called a *training set*. The weights are then modified to minimize some cost function which indicates how well the network is performing at reproducing the behaviour specified by the training set.

In **Xnet**, training sets consist of a number of *training patterns*. Each training pattern consists of an output vector, and a set of input vectors. This format is used because it is very common to want many input vectors to produce the same output vector, especially in classification problems.

Loading A Training Set

A training set has been created to allow you to see how the format works. To load this training set into the program, click on the "Load Training Set" button in the main window. This will cause the dialog box in Figure B.2 to pop up. This time, you want to load the training pattern file "ab.pat".

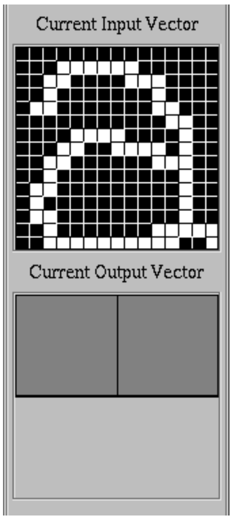


Figure B.5: Input and output vectors as displayed in the **Xnet** main window.

Viewing A Training Set

Note that the is a button in the main window labeled "Create, View or Edit A Training Set". After you have loaded "ab.pat", click on this button so that you can view the training set you have loaded. A window titled "Specify Training Set" will pop up, as shown in Figure B.6.

As you can see, Training Pattern 1 consists of an output vector for which the left node is 1 and the right node is 0, and nine input vectors, all of which are examples of the letter "a". There is another training pattern too. To see this, click in the darker grey area at the right of the button labeled "View/Edit Training Pattern Number:". The background should change to white, and a cursor will appear. Type "2", since you wish to view Training Pattern 2. Type "Enter" to cause this action to take place. A training pattern consisting of "b"s should appear. Notice that the upper group of buttons in the "Specify Training Set" window becomes greyed-out when you do this, and the lower group become active. At this stage you do not wish to modify the training set, so click on "Training Pattern Done", and then click on "Done" to return control to the main window.

B.1.6 The Training Control And Monitoring Window

Now that you have specified a network and a training set, you are in a position to train the network. To do this, click on the button labeled "Train Network" in the main window. This will cause a window labeled "Train Network" to pop up, as shown in Figure B.7.

This window allows you to set the parameters for the backpropagation algorithm and to monitor the progress of training. In the top left of the window are seven

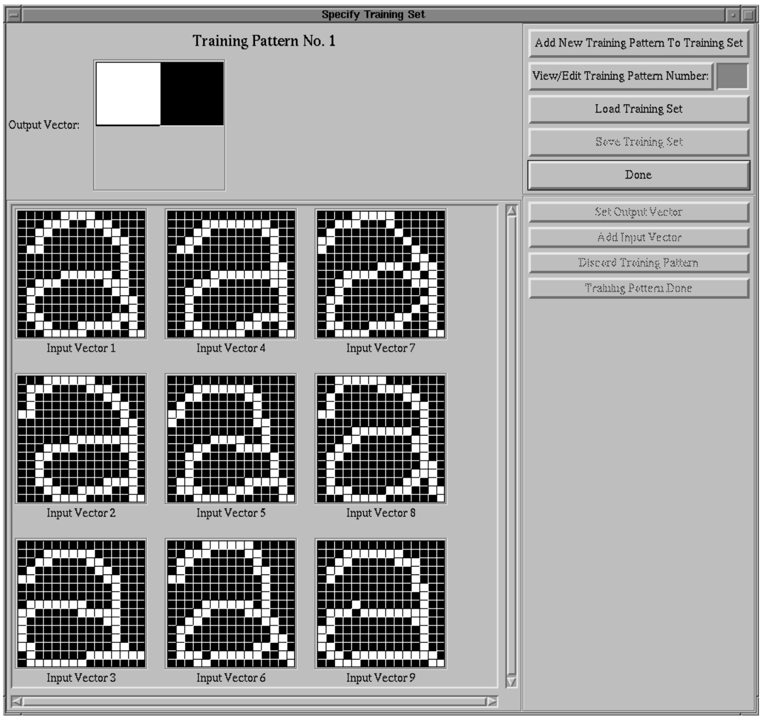


Figure B.6: Window for creating, viewing or editing a training set.

fields where you can enter numbers that control training, and the display of training performance.

The Weight Update Equation

To understand these numbers, you need to be familiar with the weight update equation used in the backpropagation training algorithm. It is given in equation B.1.

$$w_{ij_{t+1}} = w_{ij_t} - \epsilon \frac{\partial E}{\partial w_{ij}}\bigg|_t + \alpha \Delta_{t-1} - \lambda w_{ij_t} \tag{B.1}$$

The Δ refers to the *total* change in w_{ij} at the previous iteration. Therefore,

$$\Delta_t = -\epsilon \frac{\partial E}{\partial w_{ij}}\bigg|_t + \alpha \Delta_{t-1} - \lambda w_{ij_t} \tag{B.2}$$

The symbols in equation B.1 correspond to the parameters in Table B.1

These parameters are all initialized to values that should work, though not necessarily optimally, in most situations. Do not change the Weight Decay Rate λ from zero, since it is not intended for use in this tutorial.

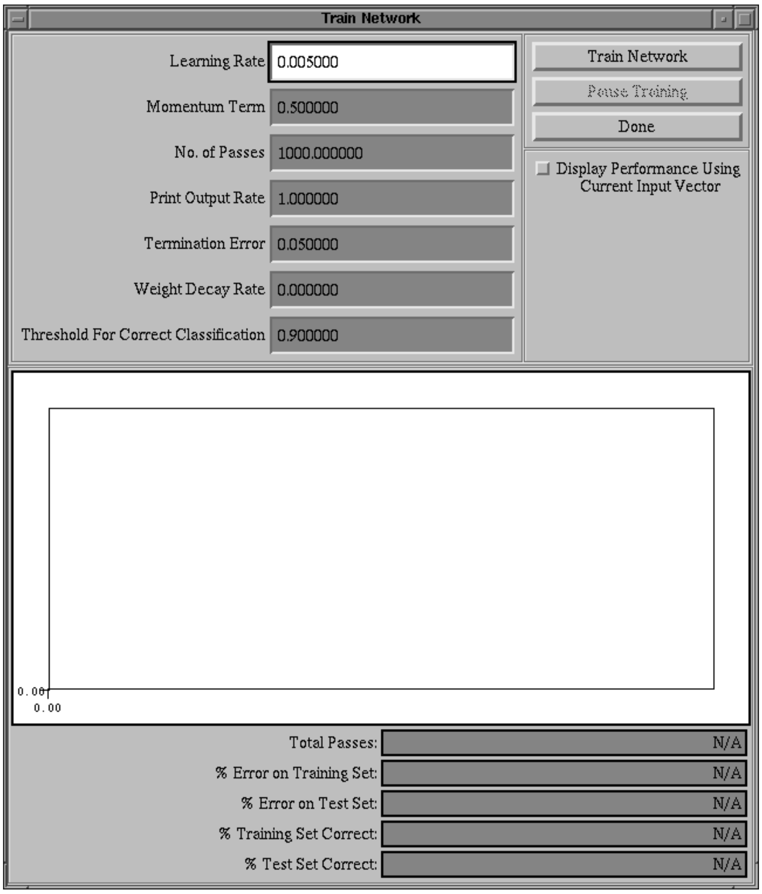


Figure B.7: Window for training a network.

Other Parameters

The other parameters which you can change control how long training goes on for, and how the progress is reported to you.

No. of Passes, as you would expect, controls the maximum number of passes over the training set that are to be made during training. The changes required in all the weights are accumulated for each input-output pair in the training set, and then the weights are updated at the end of each pass. A pass over the entire training set is often called an *epoch*.

Print Output Rate controls how often you are informed of the progress of training. Graphs and text values will be updated after every n passes, where n is the value typed in for **Print Output Rate**. You should not need to change it from 1 in this tutorial.

Termination Error provides a way to specify a condition whereby training will stop before the specified number of passes have been made. If the Sum Squared Error on the entire training set falls below the value specified here, training will stop. This value is given as a percentage, and allows you to specify a condition that is "good

Symbol	Parameter
$w_{ij\,t}$	Weight between nodes i and j at iteration t
$\left.\frac{\partial E}{\partial w_{ij}}\right\|_t$	Partial derivative of the error function E
	with respect to weight w_{ij} at iteration t
ϵ	Learning Rate
α	Momentum Term
λ	Weight Decay Rate

Table B.1: Training parameters

enough".

Threshold For Correct Classification determines when a given input vector will be deemed to have been classified correctly. It assumes that, for classification problems, there is one node in the output layer for every class. Each node is to have the value 1 for the class it corresponds to, and 0 for all other classes. There should, therefore, be only one node "on" for any given input pattern. Networks can only approximate this behaviour, the approximation improving as training progresses. The value here determines what is considered correct classification. It is initialized to 0.9. This means that the node which is supposed to be "on" must have an output value greater than 0.9, and all the others must have outputs less than (1 - 0.9) = 0.1 in order for the pattern to be considered correctly classified. This is not the only possible means of deciding upon classification, but is the one which will be used in this tutorial.

The "Display Performance Using Current Input Vector" Button

In the upper right of the "Train Network" window is a toggle button labeled "Display Performance Using Current Input Vector". When this toggle is on, it appears red. When on, this causes the input vector specified from the main window to be run through the network after each training pass, and the result displayed in the "Current Output Vector" region of the main window. Check this box, as watching this can be very useful in answering some of the questions below.

Training A Network

The initial values of all the parameters are suitable for the network ("ab.net") and training pattern ("ab.pat") which you have loaded. Click on the "Train Network" button to see what happens during the training of a network.

You should see that the axes on the graph in the middle of the window (see figure B.7) are labeled to reflect the desired number of passes. There will be up to four graphs plotted on these axes during training. The legend for their colours is given in Table B.2.

Graph Colour	Quantity Plotted
red	% Error on Training Set
black	% Error on Test Set
blue	% of Training Set Correctly Classified
green	% of Test Set Correctly Classified

Table B.2: Graph colours

The numerical values of these quantities after the most recent pass are displayed below the graph.

Now click “Done” to get back to the main window. Load the network “ab.net” from the file again, so that you can get it back to the state it was in before training. Change the control parameters to the values given in Table B.1.6.

Parameter	Value
Learning Rate	4
Momentum Term	0.5
No. of Passes	20
Print Output Rate	1
Termination Error	0
Weight Decay Rate	0

Table B.3: New values

Exercises

> **Aside:** These exercises are designed to help the user to understand the ways in which the **Learning Rate** and **Momentum** parameters influence the dynamics of training. Specifically the values have been chosen to demonstrate how the **Momentum** parameter can control oscillations, and how large values of the **Learning Rate** can lead to saturation.

Train the network with these parameters. Describe what you observe. Why do you think that this has happened?

Reload “ab.net”. Now increase the **Momentum** to 0.7. Train the network with this new parameter. Describe what happened and why.

Reload ‘ab.net”. Change the **Momentum** back to 0.5, and increase the **Learning Rate** to 5. Train the network. Describe what happened and why.

Reload ‘ab.net”. Change the **Learning Rate** to 4.2. Train the network. Describe what happened and why.

B.2 Creating Networks And Training Sets with **Xnet**

B.2.1 Introduction

The aim of this tutorial is to build upon the work done in tutorial B.1, in order to complete your familiarization with **Xnet**. In doing this, you will study one of the classic problems from the history of neural networks, the XOR problem. This will give you some insights into the fundamental limitations of the simplest neural networks, and how to get around them.

B.2.2 The XOR problem

The first researchers in the field of neural networks were inspired by two main motivations. It had been discovered that brain cells, neurons, seemed to operated "summing" a collection of inputs, and then firing if a threshold was exceeded. The output of such a neuron would contribute to the input of many other neurons. Some researchers were interested in discovering how the brain might accomplish rational thought using such simple elements. Other researchers were interested in building devices (for tasks such as pattern recognition) based on such simple units – if it was good enough for the brain, why not for machines?

One of the first tasks researchers undertook was to demonstrate that networks of such simple units could perform all the basic logic functions (AND, OR, NOT *etc.*). One of the most interesting turned out to be the Exclusive OR function, or XOR. The exclusive OR of two logical variables is true if either of the variables, but not both, is true. It is false otherwise. The truth table is shown in Table B.4.

	0	1
0	0	1
1	1	0

Table B.4: XOR truth table.

This problem is easily mapped onto a neural network. The network must have an input layer with two nodes, and an output layer with one node. The values of these nodes should correspond to the values in the truth table.

B.2.3 Creating A Network

Specifying The Number Of Layers And Their Dimensions

Start the **Xnet** simulator, as described in tutorial B.1. Once **Xnet** has started, click on the button labeled "Specify Levels and Layers". This will cause a window titled

"Specify Network Dimensions" to pop up (see Figure B.8).

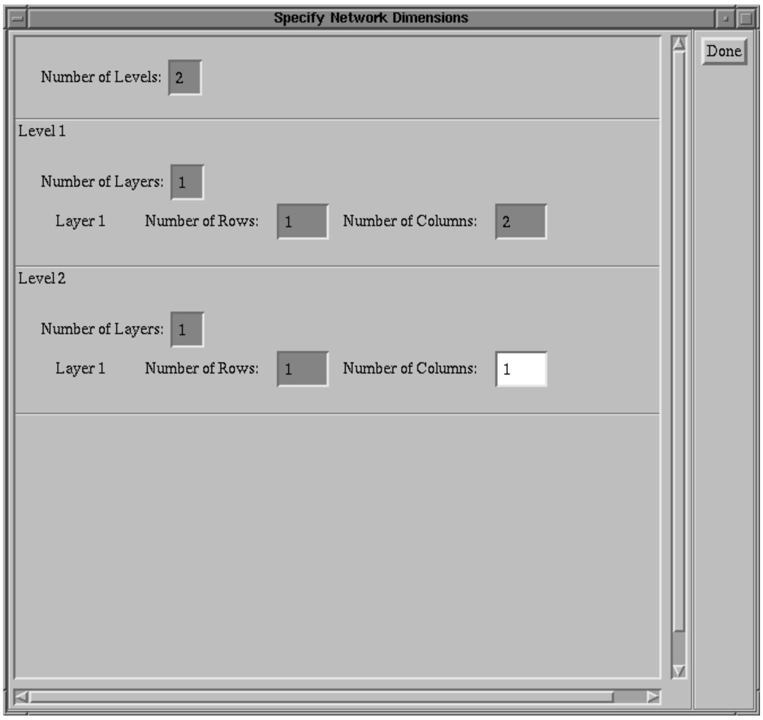


Figure B.8: **Xnet** window for specifying network dimensions.

Initially, only the topmost prompt, "Number of Levels", will be visible.[1] The other prompts will appear in response to the values you type in.

When designing a neural network to solve a problem, it is usually best to start with the simplest possible architecture. The simplest architecture for an attempt to solve the XOR problem has a 2 node input layer, a 1 node output layer, and nothing else. To construct this network, enter the numbers at each prompt as shown in Figure B.8. You need two levels, with one layer at each level. Level 1 is the input level, and the input layer should have one row of two columns. The last level, Level 2, is the output level. The output layer should have one row and one column.

You can navigate around the window by clicking in the dark grey data entry area you wish to edit. You can also move to the next data entry area by hitting `<tab>`. You can go back to the previous one using `<Shift>+<Tab>`.

Note: The value you type into a data entry area for a number of levels or a number of layers will not have any effect unless you hit return after typing it. The effect of changing one of these values is immediately

[1]The **Xnet** program allows the user to create networks other than the standard backpropagation networks used in these tutorials. These networks can have more than one *layer* of nodes at each *level* of the network. For the purposes of these tutorials, you will only ever need one layer per level.

visible, because it affects the number of other data entry areas in the scrolled window.

Once you have specified all the required values, click on "Done" to return to the main window. You should see a diagram of the network structure you have just specified, as shown in Figure B.9.

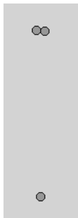


Figure B.9: **Xnet** two layer XOR network (still unconnected).

If the network which you have just created does not look like Figure B.9, click on "Specify Levels and Layers" and try again.

Specifying The Connections

Once you have created a system of layers of nodes, you then need to specify how those nodes are to be connected. The most common connection model for feed-forward neural networks is to have the outputs of all the nodes in the input layer connected to the inputs of all the nodes in the hidden layer, with each connection having an independent weight, and then for all the outputs of all the nodes in the hidden layer to be connected to all the inputs of all the nodes in the output layer, again with each connection having an independent weight.

To bring up the window for specifying how the layers you have specified should be connected, click on the button labeled "Specify Connections". This will bring up a window titled "Xnet: Connect Layers", as shown in Figure B.10.

This window, like the main window, consists of two main areas:

- A large area on the left containing a diagram of the network.
- An area of buttons on the right. These are in three groups:
  1. Buttons for setting the *Connection Model* for the connections between two layers.
  2. Buttons for setting the *Connection Function* for the connections between two layers.
  3. Buttons for connecting or disconnecting layers, and for indicating that you have finished making connections.

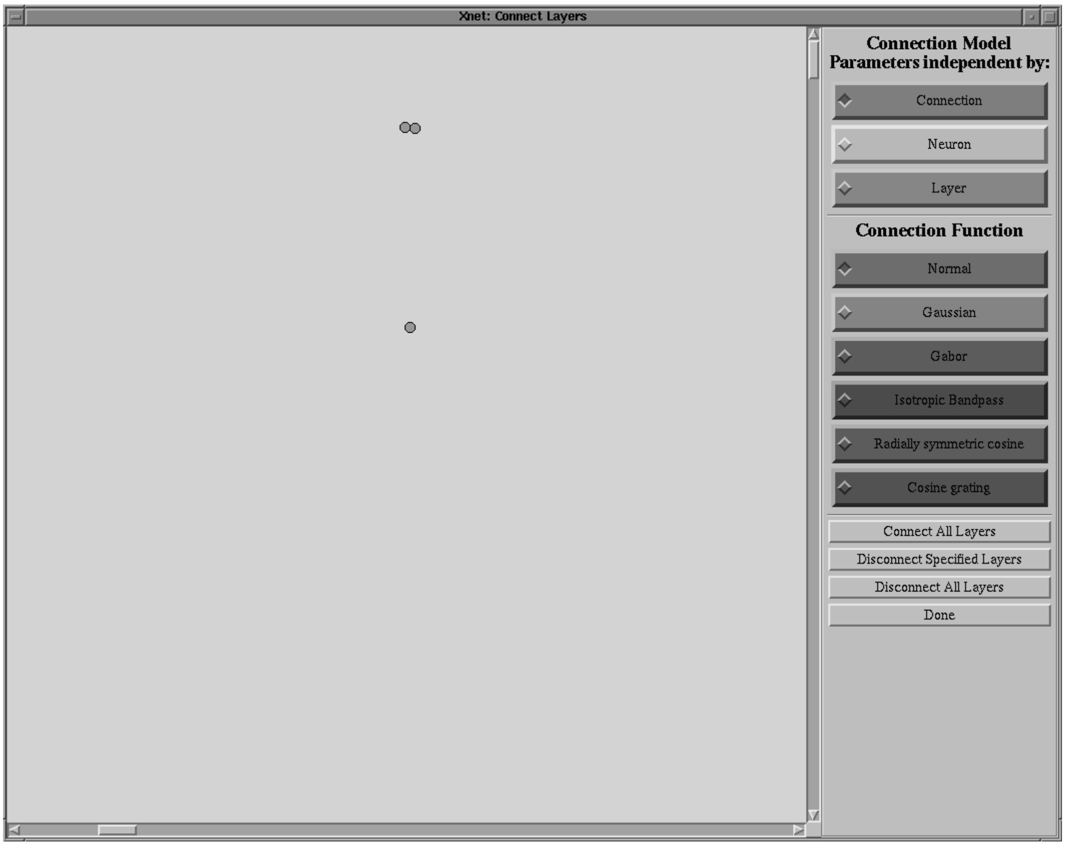


Figure B.10: **Xnet** window for specifying the way layers should be connected.

In **Xnet**, whenever a layer is a *source* for another layer (the *destination* layer), all the outputs of the nodes in the *source* layer are connected to all the inputs of all the nodes in the *destination* layer.

Xnet allows the user to create *Model-Based Neural Networks*, in which each connection is associated with a function rather than just with a number (the weight). Functions can dependent on the relative positions of the source and destination nodes, and on several parameters. These parameters may be shared between all nodes in a destination layer, between all connections entering a node, or they may be independent for each connection (in which case it behaves exactly like a normal neural network). You will not need to use any of these features.

The default *Connection Model* is "Connection". This is indicated by the diamond-shaped button on the left of the coloured label appearing depressed, and being red. The default *Connection Function* is "Normal", which means that each connection simply has an independent weight associated with it. This is indicated in the same way. Ensure that the "Connection Model" and "Connection Function" buttons are both set to their default values.

The quickest and easiest way to connect up the layers to form a standard neural network is simply to click on the button labeled "Connect All Layers". This will cause each layer to be connected to the layer immediately below it. The connections are indicated by a "wedge" which starts out the same size as the base of the source layer, and finishes at a point at the centre of the destination layer.

Connect the network by clicking on "Connect All Layers", to see what the connected network looks like. This is the way you will usually connect networks. Then disconnect the network by clicking on "Disconnect All Layers".

Note: The wedges which indicate connections between layers in **Xnet** are colour-coded. The colour of the main body of the wedge indicates the *Connection Function* of those connections, and the colour of the tip indicates the *Connection Model*. The colours used match the colours of the buttons on the right of the "Connect Layers" window.

You can also specify connections by specifying a source layer and a destination layer, and then connecting them. To specify a layer as the source layer, click on the layer with the **left** mouse button. It will change colour, from orange to mustard-yellow (lighter than the default node colour). A layer is selected as the destination layer by clicking on it with the **middle** mouse button. It will change colour, to brown (darker than the default colour). Once a source and a destination layer have been specified, they can be connected by clicking on either layer with the **right** mouse button. Select the top (input) layer as a source layer, and the bottom (output) layer as a destination layer. Connect these layers by clicking on one of them with the right mouse button. Now that you have connected up your network, click on the "Done" button.

You will now be back at the main window, which now shows a diagram of your connected network. You will need to use this (untrained) network several times in the course of this tutorial, so you should save it. Click on the "Save Network" button. You will be prompted for a file name. Save the network as "xor1.net".

B.2.4 Creating a Training Set

You now have a network ready to train. Before you can train it though, you need a training set with which to do so. Click on the button labeled "Create, View, or Edit A Training Set" to pop up the "Specify Training Set Window".

You now need to specify the input and output vectors that correspond to the XOR problem. Remember that an **Xnet** training set consists of a number of *training patterns*. Each training pattern consists of an output vector, and an arbitrary number of input vectors for which the network should have that output. For the XOR problem, you have only two possible output vectors, since there is only one output node, which can have a desired output value of 0 or 1.[2] There are four possible input vectors. All the

[2]In a classification problem such as this, it is usual to specify the desired values of all output nodes

required values are specified in Table B.4.

At this stage there are no patterns in the training set. The first pattern you will add will be that for which the single output node has an output of 1. The corresponding input vectors are $\begin{bmatrix} 0 & 1 \end{bmatrix}$ and $\begin{bmatrix} 1 & 0 \end{bmatrix}$.

Click on the "Add New Training Pattern To Training Set" button, and then click on the "Set Output Vector" button. This will cause the "Get Vector" window to pop up (as described in tutorial B.1). The grid in which you draw the output vector has, in this case, only one square, occupying the entire drawing area. You must set its value to 1. Select the colour white (corresponding to 1) from the palette. Click in the drawing area. The drawing area should change colour to white. Click on the "Done" button.

The output vector you have just specified will now appear in the output vector area at the top left of the window, below the label "Training Pattern No. 1". You now need to specify the two corresponding input vectors. This is done by clicking on the "Add Input Vector" button, and using the "Get Vector" window to draw the desired input vector. You will need to do this twice. Remember that black (at the very top of the palette) corresponds to 0, and white to 1. Add the two input vectors to Training Pattern No. 1. When you have done this, the "Specify Training Set" window should look like that shown in Figure B.11, on the left.

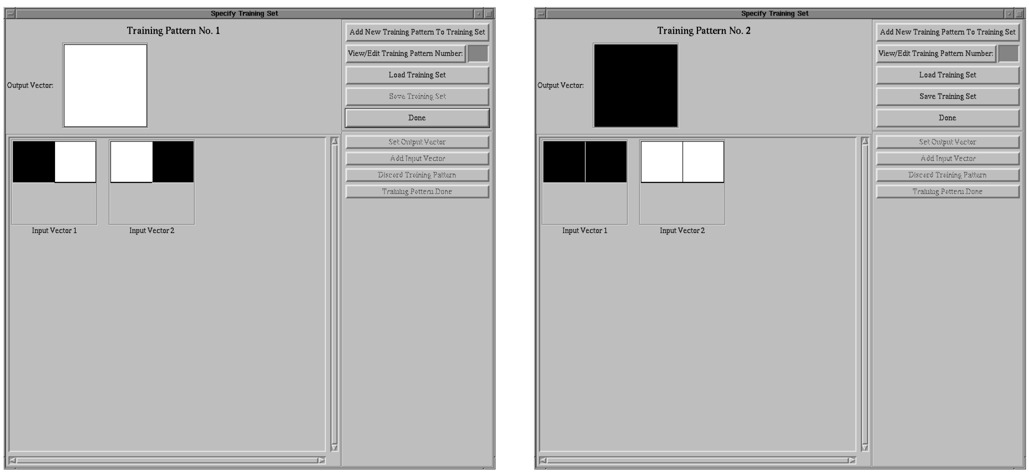


Figure B.11: Training patterns for the XOR problem.

as exactly 0 or 1. This is because there is usually a node corresponding to each class, and when an input vector of a given class is run through the network, the desired behaviour is for the output node corresponding to that class to be "on" (*i.e.* have an output of 1), and for all other nodes to be off. In some other sorts of neural network applications, such as engineering plant control or function estimation, this is not the case.

Click on the "Training Pattern Done" button to indicate that this training pattern is complete. This will return control to the upper group of buttons. Now you need to add a second training pattern to the training set. In this pattern, the desired value for the output node is 0, and the corresponding input vectors are $\begin{bmatrix} 0 & 0 \end{bmatrix}$ and $\begin{bmatrix} 1 & 1 \end{bmatrix}$. Add the required second training pattern to the training set, by repeating the procedure used to add the first pattern. When done, it should look like that shown on the right in Figure B.11.

You have now created a training set specifying the the XOR problem. You will need this training set later as well, so now you need to save it. Make sure that you have clicked on "Training Pattern Done" to return control to the upper set of buttons. Click on the "Save Training Set" button. You will be prompted for a file name. The default is "new.pat". Change this to "xor.pat", and then click on "OK". Then click 'Done" to return to the main window.

B.2.5 Training A Network To Solve The XOR Problem

The Two Layer Network

If you have followed the instructions provided, you now have in **Xnet** a two layer network that has not yet been trained, and a training set specifying the XOR problem. You will now attempt to train this network to solve the problem[3]. Click on "Train Network" to bring up the training window. Select whatever training parameters you think appropriate. Train the network, making no more than 10,000 passes over the training set. You can reload the saved version of the network, "xor1.net", to make another attempt at training. Make no more than three attempts.

No doubt you have discovered by now that the two-layer network fails to find a set of weights which solves the XOR problem. It is worth considering why, because it is to do with a fundamental property of the operation of any two layers of a neural network.

The Equations For Calculating The Output Of A Node In A Neural Network

The two-layer network used above is simple enough for us to be able to write down the equation describing its operation, and this will lead you to understand why it can't solve the XOR problem.

Every node in a traditional neural network (except those in the input layer) has a weighted input from the output of each of its source nodes, and one from the *bias*

[3]The various parameters used in training a neural network, and the graphs and other information that **Xnet** provides during the progress of training are described in tutorial B.1.

node.[4] The equation for the activation of node j, x_j, is thus:

$$x_j = \sum_{i=1}^{n} w_{ij} y_i + w_{bias} \tag{B.3}$$

where n is the number of source nodes for this node, y_i is the output of the ith node, and w_{ij} is the weight on the connection between nodes i and j. The activation, x_j, is used to calculate the actual output of the node by using it as the argument of a nonlinear function, here the usual sigmoid:

$$\begin{aligned} y_j &= f(x_j) \\ &= \frac{1}{1+e^{-x_j}} \end{aligned} \tag{B.4}$$

The shape of the sigmoid is shown in Figure B.12.

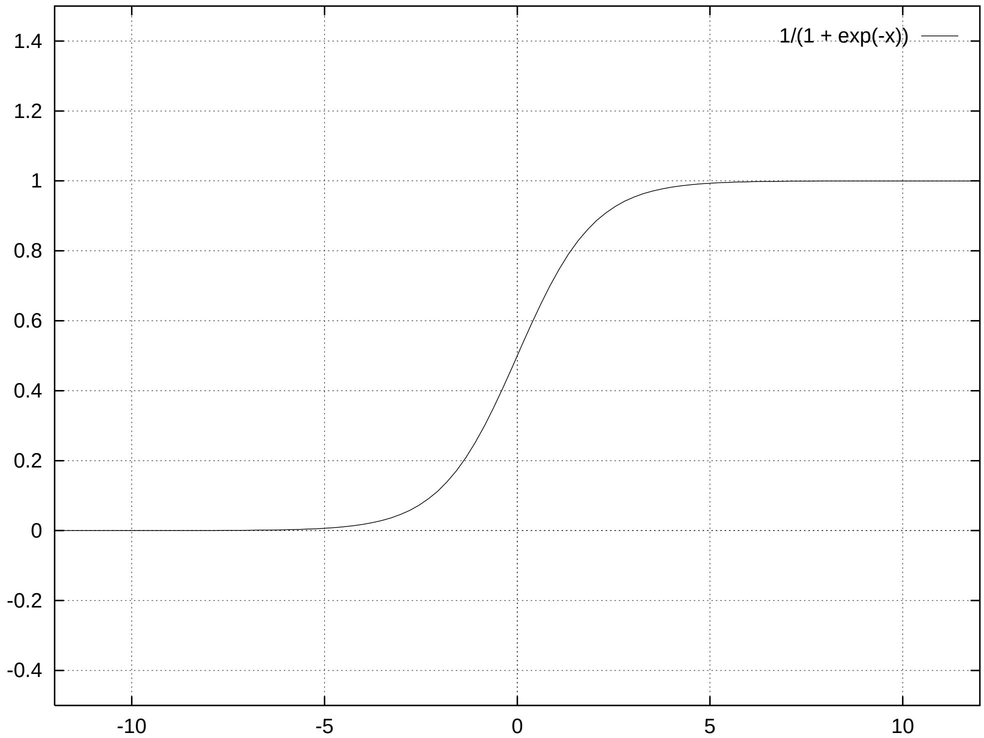


Figure B.12: The sigmoid function.

The equation for the output of the two-layer network used above can be written down in expanded form as shown in Equation B.5 (note that the index j has been

[4]Every node in a neural network for which the output is calculated (*i.e.* all except the input nodes) has a weighted connection to the *bias node*. This is a node which has a fixed output of 1. These thus provide an offset in the activation equation for each node. The negative of the weight on the connection to the bias node for a given node is analogous to a a threshold for that node.

dropped since there is only one output node).

$$x = w_1 y_1 + w_2 y_2 + w_{bias} \tag{B.5}$$

This is a very simple equation. Recall that for correct classification, we require that value of the output to be either above or below a threshold, which we will call θ. The fact that the output is "squeezed" by the sigmoid nonlinearity does not alter this, because the sigmoid is monotonic.

The variables y_1 and y_2 are constrained to only take on the values 0 or 1. For the network to solve the XOR problem, therefore, we require that the output be "on" (greater than the threshold) for two of the four possible combinations, and "off" for the the others. For successful performance then, we would need to find values of w_1, w_2 and w_{bias} so that the inequality in Equation B.6 is satisfied for inputs requiring a 1 output, and that in Equation B.7 for those requiring a 0 output. Note that the values of w_1, w_2, w_{bias} and θ must be the same in both equations.

$$w_1 y_1 + w_2 y_2 + w_{bias} > \theta \tag{B.6}$$

$$w_1 y_1 + w_2 y_2 + w_{bias} < \theta \tag{B.7}$$

Exercises

> **Aside:**This exercise is aimed at introducing the user to the notion of linearly separable problems.

Show that the two-layer network described by Equation B.5 can not solve the XOR problem. A graph using y_1 and y_2 as axes will be useful.

How does the above result apply to two-layer neural networks with more than two input nodes? Can you come up with a statement of the general class of problems that this kind of network can solve?

A Three-Layer Network We have seen that the XOR problem cannot be solved by a two layer network. We will now try a three layer network. Go to the **Xnet** main window. Click on the "Specify Levels & Layers" button to bring up the window allowing you to specify a new network. Create a network with:

- 3 levels, 1 layer per level
- An input layer with two nodes; 1 row, 2 columns
- A hidden layer with two nodes; 1 row, 2 columns
- An output layer with one node; 1 row, 1 column

Connect the network so that the input layer is a source for the hidden layer, and the hidden layer is a source for the output layer. Save the untrained network as "xor2.net".

Train the three-layer network using the training set created earlier. Record the parameters you used, and the performance obtained.

- Find the set of parameters that allow you to train the network in the least number of passes. Remember that you must reload the network from the file "xor2.net" before each attempt at training.
- Comment on the way the choice of parameters influences the % Error on Training Set (red graph) and the % Training Set Correct. It may be helpful to check the response of the trained network to various input patterns using the "Set Input Vector" button in the main window.

Why do you think that the three-layer network was able to solve this problem, whilst the two-layer network could not? What are the implications of this for larger networks?

There exists a network which can solve the XOR problem which has only four nodes. Can you find it? If so, describe it. (Hint **Xnet** allows you to connect a layer to *any* layer below it).

B.3 Generalization and Repeatability

B.3.1 Introduction

One of the claims often made about neural networks is that they have the ability to *generalize*. This means that they have the ability to classify correctly patterns which were not present in the training set. In order to do this, a neural network would have to somehow extract the "essence" of a class of patterns.

In this tutorial, you will investigate the ability of neural networks to generalize. You will see that neural networks certainly can generalize to some extent, but this capacity is limited. You will also investigate some training strategies aimed at improving generalization.

Another issue that is often overlooked in the study of neural networks is repeatability. When a neural network is created, the connection weights are initialized to random values. This means that the training procedure can proceed in differing ways for separate instances of the same network architecture even when using exactly the same training data. You will see how this can affect the final performance of networks after training.

B.3.2 Generalization

The claim that neural networks have the ability to *generalize* is one of the main reasons for their popularity. A pattern classification system that is able to classify correctly patterns which were not present in the training set used to produce it is clearly desirable. Such a system would be able to operate in the "real world", where input patterns might be noisy, or incomplete. A system based, for instance, on table lookup would fail in such an environment.

The Test Set

In order to evaluate the ability of a neural network to generalize, it is necessary to have a set of patterns other than the training set on which the network's performance can be tested. The **Xnet** simulator allows you to specify a *test set* for network. This is a set of patterns created in exactly the same way as those used for training the network. During training, if a test set is specified, the mean squared error and the classification performance of the network on the test set are evaluated after each pass over the training data. These results are displayed in the graph in the training window. In order to see how this works, we will do an example.

Start **Xnet**. Reload the network "ab.pat" that was used in tutorial B.1. Click on the "Specify Connections" button in the main window, and selected "OK" when warned about parameters being reinitialised. The "Connect Layers" window will pop up. Click on "Connect All Layers", and then click on "Done" to return to the main window. This procedure will ensure that you have an untrained network, with connection weights randomly initialized.

Now click on "Load Training Set", and load the pattern file "ab.pat". Click on "Create, View or Edit A Training Set". This will bring up the "Specify Training Set" window. Have a look at the basic style of the characters in each of the training patterns. To do this, type the number "1" in the grey box beside "View/Edit Training Pattern Number:", and then hit return. This will cause the lower set of buttons to become active. When you think you are familiar with the style of the characters, click on "Discard Training Pattern". This will cause the "a" pattern to be removed from the set. Then click on "'View/Edit Training Pattern Number:" again (since the "b" pattern is now pattern number 1). When you are familiar with the style of the "b"s, click on "Discard Training Pattern". There are now no patterns in the training set.

Now you must create a set of patterns to be used to evaluate the generalization performance of a network trained using the pattern file "ab.pat". You should be familiar with the procedure for creating a training set from tutorial B.2.

Create a set of patterns containing three examples of the letter "a", and three examples of "b". An example pattern is shown in Figure B.13. Do *not* copy this example. Draw your letters freehand. Save these patterns in a file called “ab_test.pat”.

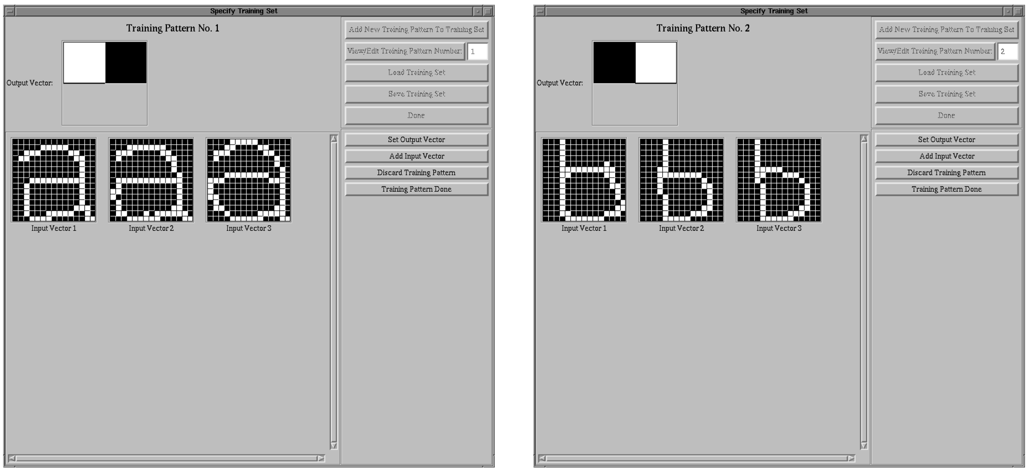


Figure B.13: Example test patterns for network trained using “ab.pat”.

Then click “Done” to return to the main window.

Click “Load Training Set”, and load the file “ab.pat” as the training set. Click “Load Test Set”, and load the file you have just created, “ab_test.pat”, as the test set for the network. Then train the network with the default training parameters.

Exercises

> **Aside:** These exercises are intended to introduce the user to the notion of over-training.

Record all the performance parameters of the network after 1000 passes over the training data. Describe the way the **% Error on Test Set** varied compared to the **% Error on Training Set**. What was the generalization performance of your network at the completion of training? Do you think that this performance is good? How might it be improved?

Train the network for another 1000 passes. Did the generalization improve with further training? If so, why do you think this was so?

B.3.3 Repeatability

The greater the number of patterns in the training and test sets, the longer it takes to train the network. Even on extremely fast computers it can take hours to train a network if a realistically-sized training set is used. Since you may not have this amount of time available, some simulations have been done for you.

The task was to classify an input pattern as one of the digits from 0 to 9. The network architecture used is shown in Figure B.14.

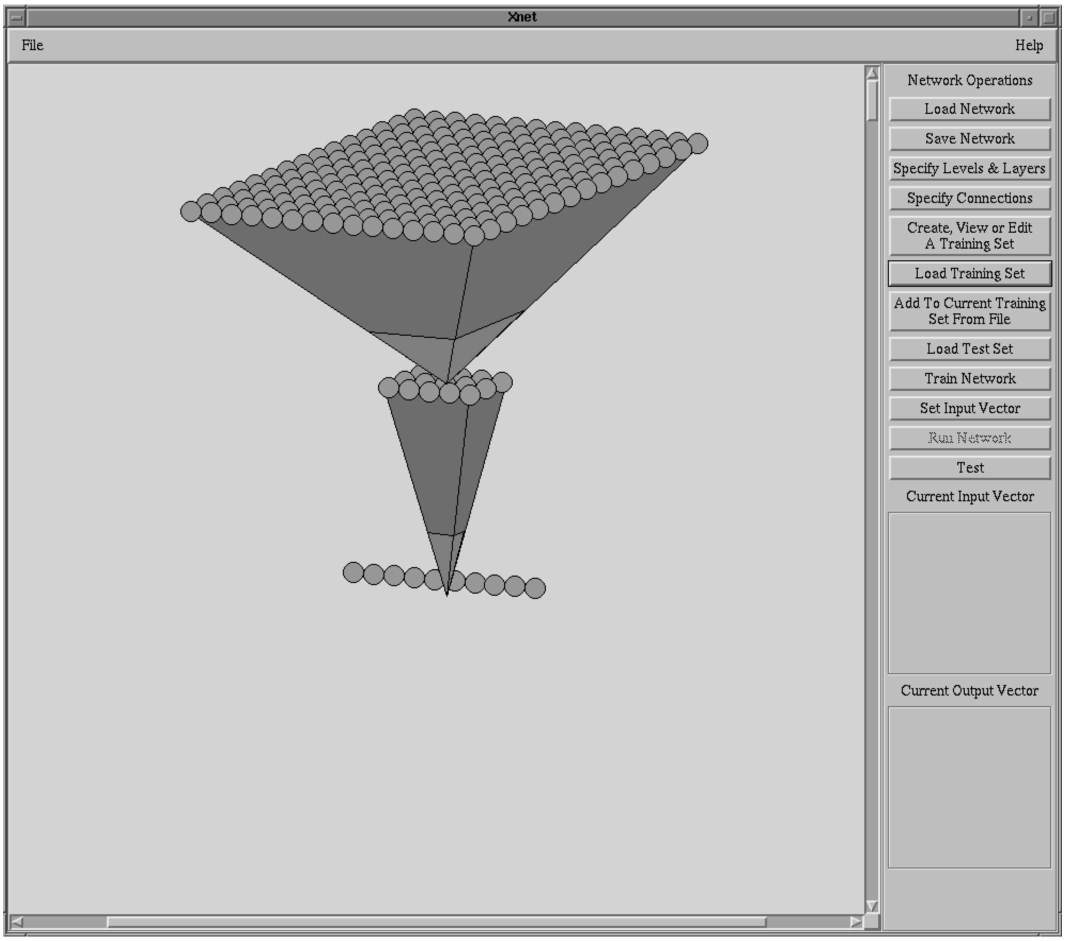


Figure B.14: Network used for classify patterns as a digit.

The training pattern consisted of 90 example patterns, 9 of each digit. Each input pattern had noise added to it. An example, for the digit "2", is shown in Figure B.15. These patterns can be found in the file "digits_train.pat". The test set also contained 90 patterns, each also with additive noise. These patterns can be found in the file "digits_test.pat".

Five networks of the architecture shown in Figure B.14 were created. Although they all had the same architecture, the connection weights of each network were randomly initialized. This meant that the initial conditions for training using the backpropagation algorithm were different for each network. Each network was trained for 4000 passes over the training data. Some of the graphs of network performance during training are shown in Figure B.16

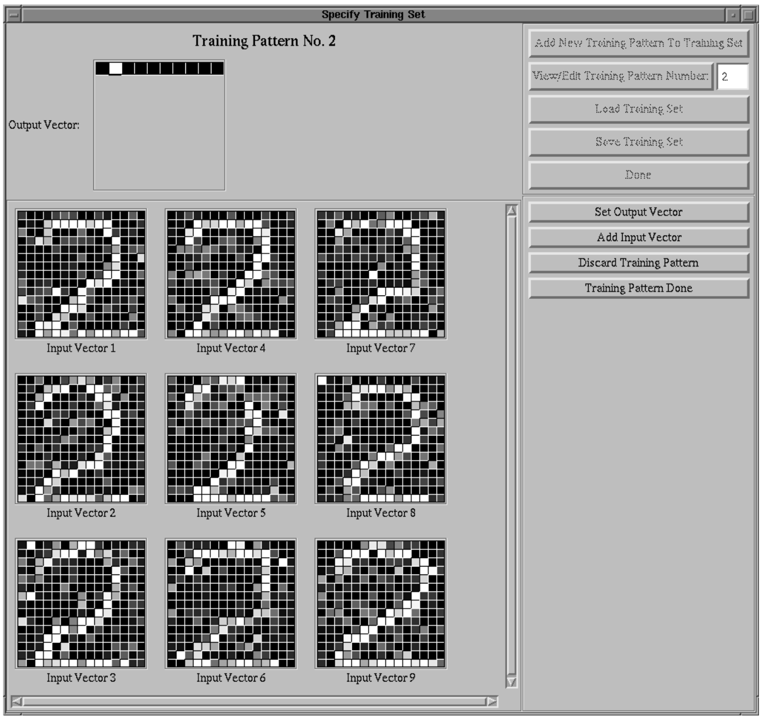


Figure B.15: Training pattern for the digit “2”.

The final performances of the networks on the test and training data are summarized in Table B.5.

	Percent Correct		Percent Error	
	Training Set	Test Set	Training Set	Test Set
digits1.net	100	72.8	0.03	2.06
digits2.net	100	75.0	0.03	2.05
digits3.net	100	73.9	0.03	1.94
digits4.net	100	79.4	0.04	1.85
digits5.net	100	77.2	0.04	1.78
average	100	75.7	0.03	1.94

Table B.5: Final performance of networks trained on digit classification data.

Exercises

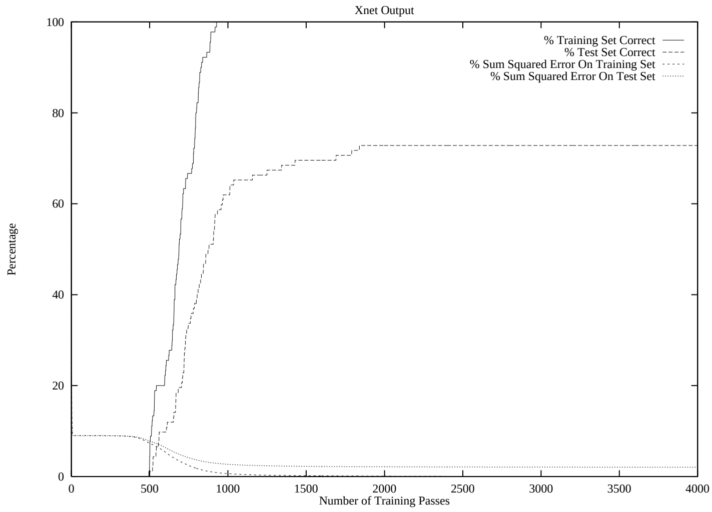
> **Aside:** These exercises are intended to encourage the user to think about the expected value of a networks classification performance, measures for quantifying generalization, and the importance of ensuring that results obtained from a randomly-initialized system are repeatable.

What would be the expected % Correct on the Test Set if the network was tested before training, and was *forced* to classify each pattern (*i.e.* no "undecided" results allowed)?

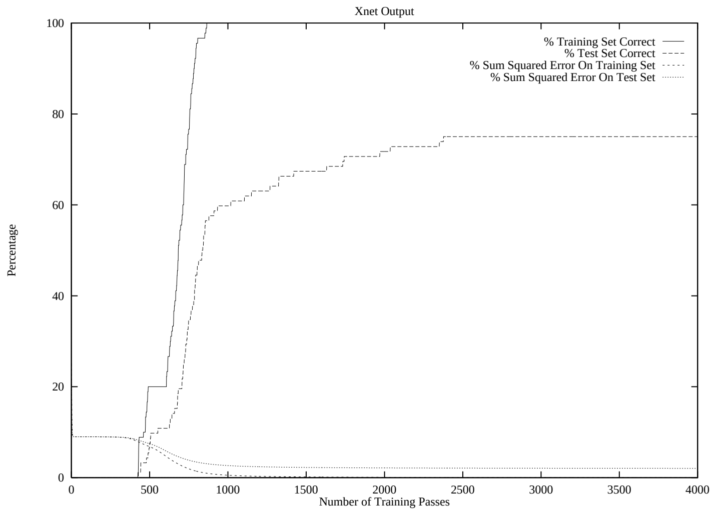
Do you consider the generalization performance of these networks to be good? What criterion could be used to judge the generalization performance of these networks?

It is clear that the performances of each of these networks after training with identical training sets and training parameters are different. Why is this so? Could this be avoided? What are the implications of this variability of performance for real-world applications of neural networks?

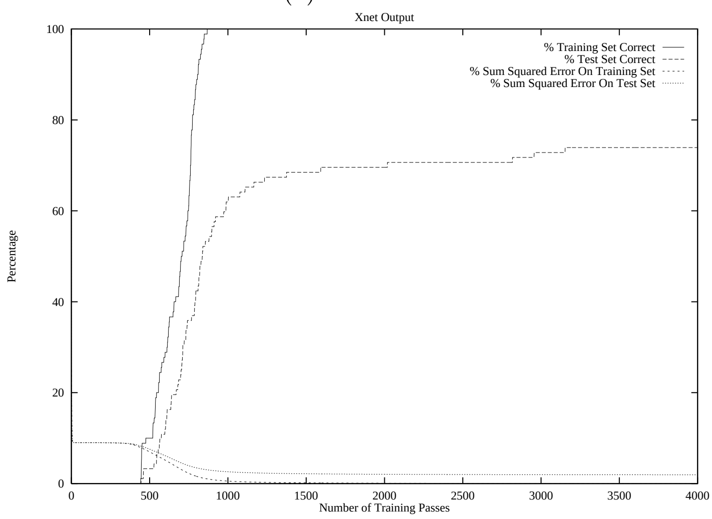
Finally, you might like to see how one of these pre-trained networks performs at recognizing your own hand-drawn digits. Load the network "digits1.net" into the **Xnet** simulator. You can qualitatively evaluate the performance of the network at classifying your own hand-drawn digits by clicking on "Set Input Vector" in the main window. This causes the "Get Vector" window to appear. Anything you draw in this window immediately gets run through the network, and the resultant output vector is displayed the the "Current Output Vector" section of the main window. Try to determine which features of each digit are important for correct classification. What was the performance of the network on your patterns? Do you consider this good? Can you suggest a more robust system for classifying hand-written characters?



(a) Network 1



(b) Network 2



(c) Network 3

Figure B.16: Performance of networks trained on digit classification data.