DEVELOPING NEURAL NETWORK APPLICATIONS USING LABVIEW

A Thesis presented to the faculty of the Graduate School

University of Missouri-Columbia

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

---

by

POGULA SRIDHAR, SRIRAM

Dr. Robert W. McLaren, Thesis Supervisor

JULY 2005

The undersigned, appointed by the Dean of the Graduate School,

have examined the thesis entitled.

DEVELOPING NEURAL NETWORK APPLICATIONS USING LABVIEW

Presented by Pogula Sridhar, Sriram

A candidate for the degree of Master of Science

And hereby certify that in their opinion it is worthy of acceptance.

Robert W. McLaren

Robert W. McLaren, Professor Emeritus, Dept. of Electrical and Computer Engineering

[illegible]

Michael J. Devaney, Professor, Dept. of Electrical and Computer Engineering

[illegible]

Jeffrey Uhlmann, Assistant Professor, Dept. of Computer Science

Dedicated to my parents

ACKNOWLEDGEMENTS

First of all, I thank my parents who have always showered unconditional love on me, encouraged, and supported me in every aspect. It was my mom’s dream that I continue with graduate studies. She has always been my inspiration and I thank one and all who have helped me fulfill her dream. I thank my dad Sridhar, who, during difficult times stood by me and encouraged me to complete my graduate studies. I take this opportunity to thank my advisor Dr. Robert W. McLaren, who has been so kind, supportive and inspiring. His guidance and advice during my course of study is invaluable. I also thank Dr. Michael Devaney and Dr. Jeffrey Uhlmann for accepting my request to be members of my Master’s thesis committee.

I thank Dharani, Rajkumar and all my friends who have been so helpful and supportive during my course of study and made life cheerful at Mizzou.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

CHAPTER 1

INTRODUCTION

1.1 Artificial Neural Networks and LabVIEW

Artificial Intelligence is a branch of study which enhances the capability of computers by giving them human-like intelligence. The brain architecture has been extensively studied and attempts have been made to emulate it. ‘Artificial Neural Networks’ (ANN) represent an area of AI. They are basically an attempt to simulate the brain. The study of neural networks is also called neuroengineering or neural computing.

There is a variety of neural net architectures that have been developed over the past few decades. Beginning from the feed-forward networks and the multi-layer perceptrons (MLP) to the highly dynamic recurrent ANN, there are many different types of neural nets. Also, various algorithms have been developed for training these networks to perform the necessary functions.

In this thesis, a new approach has been applied to build neural nets. LabVIEW, powerful graphical programming software developed by National Instruments, which has, so far been successfully used for data acquisition and control, has been used here for building neural nets. Both supervised and the unsupervised neural nets have been successfully developed in this thesis using LabVIEW, and it has been proved that LabVIEW is a very powerful software tool for building neural nets.

Neural nets are parallel processors. They have data flowing in parallel lines simultaneously. LabVIEW has the unique ability to develop data flow diagrams that are highly parallel in structure. So LabVIEW seems to be a very effective approach for building neural nets.

Some of the artificial neural nets discussed in this thesis are multi-layered feed-forward ANN, Radial Basis Function ANN (RBF), Principal Component ANN (PCNN), and Recurrent ANN. The training algorithms applied to these nets include error back propagation, multi-dimensional clustering algorithms such as the Self Organized Mapping technique (SOM), crisp K-means and fuzzy K-means algorithms. The remainder of this chapter discusses the LabVIEW graphical programming environment and its features, which makes it a better tool in building artificial neural networks. Also, the journals that were used as references throughout this thesis are reviewed at the end of this chapter in Section 1.4.

1.2 LabVIEW and its features

National Instruments LabVIEW is a highly productive graphical programming environment that combines easy to use graphical developments with the flexibility of a powerful programming language [22]. LabVIEW is a general purpose programming language used for developing projects graphically. It can also be called an Application-specific Development Environment (ADE). It is a highly interactive environment for rapid prototyping and incremental development of applications, from measurement and automation to real-time embedded and general purpose applications. National Instruments LabVIEW is a revolutionary programming language that depicts program code graphically rather than textually. LabVIEW excels in data acquisition and control, data analysis, and data presentation, while delivering the complete capabilities of a traditional programming language such as Microsoft Visual C. One major benefit of using graphical programming rather than text-based languages is that one writes program codes simply by connecting icons. In addition, graphical programming solutions offer the performance and flexibility of text-based programming environments, but conceal many programming intricacies such as memory allocation and syntax. LabVIEW involves structured dataflow diagramming. It is, in fact, a much richer computational model than the control flow of popular text-based languages because it is inherently parallel, while C/C++ and VBA are not. They have to rely on library calls to operating system functions to achieve parallelism. Even a newcomer can design a highly parallel application.

Data flow diagrams in general are so useful for building general purpose applications, that, instead of working with text, one can directly modify and edit data-flow diagrams. This is briefly being done at same level in LabVIEW. Because it is the flow of data between objects on a block diagram, and not sequential lines of text, that determines execution order in LabVIEW, one can create diagrams that simultaneously execute multiple operations. Consequently, LabVIEW is a multitasking system capable of concurrently running multiple execution threads and multiple VI's (Virtual Instruments).

Some features of popular languages are missing in LabVIEW, but some elaborate general purpose applications can be built using LabVIEW even without data acquisition or analysis. LabVIEW is most suitable for measurement and automation applications.

Basically, in any programming language there are editors and compilers. In text based programming languages, the editor edits the program in its own style and finally produces ASCII characters which are passed onto the compiler. But in graphical and pictorial programming, an image is produced by the editor, which is parsed by the compiler. But LabVIEW has a better approach than the above mentioned two methods when it comes to editing and compiling. The editor parses the image as it is being constructed so it is highly interactive. As a result, a higher and richer version of graphics is visible on screen. Also, another significant advantage of LabVIEW is the relatively high speed of the compiler.

The editor has a rich set of operations to quickly create elaborate user interfaces by direct manipulation. The fact that every module, or VI, has a user interface means that interactive testing is simple to do at each step, without writing any extra code. The fraction of an application that has to be completed before meaningful testing can take place is much smaller in LabVIEW than in traditional programming tools, making design much faster.

Even the data types on the diagram are easy to use. Strings and arrays can be routed and operated on without worrying about the details of memory allocation, which means that errors, such as losing or overwriting memory, just do not exist.

The net result of all of these capabilities in LabVIEW greatly increases its potential. Block diagrams are assembled - a natural design notation for scientists and engineers. LabVIEW is cross-platform compatible and has all the same development tools and language capabilities of a standard programming language. LabVIEW accelerates development over traditional programming by a significant factor. With the modularity and hierarchical structure of LabVIEW, one can prototype, design, and modify systems in a relatively short amount of time.

LabVIEW has many interesting features that make it a very useful tool for building neural nets. LabVIEW programs are called Virtual Instruments (VI), because they appear very similar to actual laboratory instruments. There are two parts to a VI – the 'Front Panel' and the 'Block Diagram'. The front panel constitutes one part of the program in which the GUI is developed. Controls, constants and indicators form an integral part of the front panel. Examples for controls and indicators include knobs, horizontal and vertical sliders, buttons, input devices, graphs and charts. LabVIEW utilizes a powerful Graphical User Interface (GUI). The Front panel acts as the user interface. The front panel, as the name suggests acts as the front-end of the virtual instrument, while the block diagram is the background code. Thus, the block diagram constitutes the actual program part of LabVIEW. It is highly graphical, and no text codes are involved. The block diagram follows a similar idea as the 'data flow diagram' concept, in which the logic is presented as a diagram rather than as text code.

After building the front panel, codes could be added by using graphical representations of functions to control the front panel objects. The block diagram contains this graphical source code. Front panel objects appear as terminals on the block diagram.

The terminals represent the data type of the control (input) or indicator (output). The front panel controls or indicators can be configured to appear as icons or data type terminals on the block diagram. By default, front panel objects appear as icon terminals. The terminals in the block diagram represent the data type as double precision, floating point or any other format of that type.

Terminals are entry and exit ports that exchange information between the front panel and the block diagram. The data that a user enters into the front panel controls', enter the block diagram through the control terminals. After the functions complete their calculations in the block diagram, the data flow to the indicator terminals, where they exit the block diagram, reenter the front panel, and appear as front panel indicators. Nodes are objects on the block diagram that have inputs and/or outputs and perform operations when a VI runs. They are analogous to statements, operators, functions, and subroutines in text-based programming languages. Some node examples follow,

Summation Node - Σ   Square Root Node - √   Subtract Node - −

Wires

Data transfer between block diagram objects is done through wires. In Figure 1.1, wires connect the control and indicator terminals to the multiply node. Each wire has a single data source, but one can wire it to many VI's and functions that read the data. Wires are different colors, styles, and thicknesses, depending on their data types. A broken wire appears as a dashed black line with a red X in the middle.

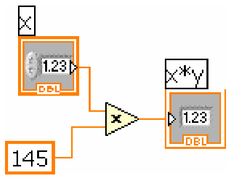


Figure 1.1 Block diagram example to illustrate simple multiplication

Structures

Structures are graphical representations of loops and case statements in text-based programming languages. Structures are used on the block diagram to repeat blocks of code and to execute code conditionally or in a specific order.

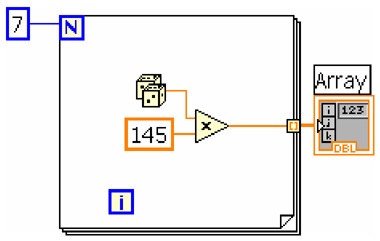


Figure 1.2 Block diagram example to illustrate using 'for' loop with indexing

Figure 1.2 shows a simple "for loop" structure that repeats 7 times. Each time, the output value is stored in memory. After the loop repeats 7 times, the stored values form an output array.

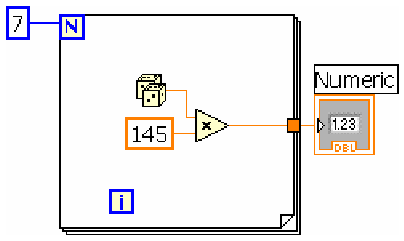
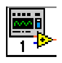


Figure 1.3 Block diagram example to illustrate using 'for' loop without indexing

In figure 1.3, the same loop as in figure 1.2 is executed, but the only difference is that in figure 1.3, the output of the loop is a single value each time the loop executes instead of an array. Loop indexing is disabled in figure 1.3.

After building a VI front panel and block diagram, one needs to build the icon and the connector pane so that this VI could be used as a sub-VI. Every VI displays an icon, such as the one shown in figure 1.4, in the upper right corner of the front panel and block diagram windows. An icon is a graphical representation of a VI. It can contain text, images, or a combination of both. If one uses a VI as a sub-VI, the icon identifies the sub-VI on the block diagram of the VI. One can double-click the icon to customize or edit it.

One also needs to build a connector pane, to use the VI as a sub-VI. The connector pane is a set of terminals that correspond to the controls and indicators of that VI, similar to the parameter list of a function-call in text-based programming languages. The connector pane defines the inputs and outputs one can wire to the VI so that it could be used as a sub-VI. A connector pane receives data at its input terminals and passes the data to the block diagram code through the front panel controls and receives the results at its output terminals from the front panel indicators. When the connector pane is viewed for the first time, one can see a connector pattern. A different pattern could be selected if necessary. The connector pane generally has one terminal for each control or indicator on the front panel. Upto 28 terminals could be assigned to a connector pane. If any changes are anticipated to the VI that would require a new input or output, one can leave extra terminals unassigned.

Icon - 

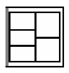
Connector Pane - 

Figure 1.4 Example of Icons and Connector panes

After building a VI and creating its icon and connector pane, one can use it as a sub-VI. One can save VI's as individual files, or group several VI's together and save them in a VI library. One can customize the appearance and behavior of a VI and also create custom menus for every VI that is built.

The **controls** palette is available only on the front panel. The Controls palette contains the controls and indicators used to create the front panel. The controls and indicators are located on sub-palettes based on the types of controls and indicators.

The **functions** palette is available only on the block diagram. The Functions palette contains the VI's and functions one can use to build the block diagram. The VI's and functions are located on sub-palettes based on the types of VI's and functions.

The **tools** palette is available on the front panel and the block diagram. A tool is a special operating mode of the cursor. The cursor selects an icon corresponding to a tool in the palette. One can use the tools to operate and modify front panel and block diagram objects.

Indicators

Graphs and charts are used to display data in a graphical form. Graphs and charts differ in the way they display and update data. VI's with graphs usually collect the data in an array and then plot the data to the graph, which is similar to a spreadsheet that first stores the data, then generates a plot of it. In contrast, a chart appends new data points to those already in the display to create a history. On a chart, one can see the current reading or measurement in the context of previously acquired data.

Arrays

An array consists of elements and dimensions. Elements are the data that make up the array. A dimension is the length, height, or depth of an array. An array can have one or more dimensions and as many as 230 elements per dimension, memory permitting. One can build arrays of numeric, Boolean, string, waveform, and cluster data types. One should consider using arrays when working with a collection of similar data or when one performs repetitive computations. Arrays are ideal for storing data collected from waveforms or data generated in loops, where each iteration of a loop produces one element of the array. One cannot create arrays of arrays. However, one can use a multidimensional array or create an array of clusters, where each cluster contains one or more arrays. To locate a particular element in an array requires one index per dimension. In LabVIEW, indices let one navigate through an array and retrieve elements, rows, columns, and pages from an array on the block diagram. The user create an array control or indicator on the front panel by placing an array shell on the front panel, and dragging a data object or element into the array shell, which can be a numeric, Boolean, string, or cluster control or indicator. The array shell automatically resizes to accommodate the new object, whether a Boolean control or a 3D graph. To create a multidimensional array on the front panel, one would right-click the index display and select *Add Dimension* from the shortcut menu. One also can resize the index display until one has as many dimensions as needed. To delete dimensions one at a time, the user would right-click the index display and select *Remove Dimension* from the shortcut menu. One also can resize the index display to delete dimensions. To display a particular element on the front panel, the user can either type the index number in the index display or use the arrows on the index display to navigate to that number.

Array functions can be used to create and manipulate arrays, such as in the following tasks:

- Extract individual data elements from an array.
- Insert, delete, or replace data elements in an array.
- Split arrays.

The *Index Array*, *Replace Array Subset*, *Insert into Array*, *Delete from Array*, and *Array Subset* functions automatically resize to match the dimensions of the input array that are wired. For example, if one wires a 1D array to one of these functions, the function shows a single index input. If one wires a 2D array to the same function, it shows two indexed inputs—one for the row and one for the column. One can access more than one element, or sub-array (row, column, or page) with these functions by using the positioning tool to manually resize the function. When one expands one of these functions, the function expands in increments determined by the dimension of the array wired to that function. If one wired a 1D array to one of these functions, the function expands by a single indexed input. If one wired a 2D array to the same function, the function expands by two indexed inputs—one for the row and one for the column. The indexed inputs that are wired determine the shape of the sub-array one wants to access or modify. For example, if the input to an *Index Array* function is a 2D array, and if only the *row* input is wired, a complete 1D row of the array is extracted. If only the *column* input is wired, a complete 1D column of the array is extracted. If both the *row* input and the *column* input are wired, a single element of the array is extracted. Each input group is independent and can access any portion of any dimension of the array.

Clusters

Clusters group data elements of mixed types, such as a bundle of wires, as in a telephone cable, where each wire in the cable represents a different element of the cluster. A cluster is similar to a record or a structure in text-based programming languages. Bundling several data elements into a cluster eliminates wire clutter on the block diagram and reduces the number of connector pane terminals required by the corresponding sub-

vi. The connector pane can have, at most, 28 terminals. If the front panel contains more than 28 controls and indicators required by a program, then some of them could be grouped into a cluster, which in turn can be assigned to a terminal on the connector pane.

Although cluster and array elements are both ordered, one must unbundle all cluster elements at once rather than index one element at a time. One also can use the *Unbundle by Name* function to access specific cluster elements. Clusters also differ from arrays in that they have a fixed size. Similar to an array, a cluster is either a control or an indicator. A cluster cannot contain a mixture of controls and indicators. Most clusters on the block diagram use a pink wire pattern and a data type terminal. Clusters of the numeric type, sometimes referred to as points, use a brown wire pattern and a data type terminal. One can wire brown numeric clusters to 'Numeric functions', such as 'Add' or 'Square Root', to perform the same operation simultaneously on all elements of the cluster.

Cluster elements have a logical order unrelated to their position in the shell. The first object placed in the cluster is element 0; the second will be element 1, and so on. If an element is deleted, the order adjusts automatically. The cluster order determines the order in which the elements appear as terminals in the *Bundle* and *Unbundle* functions placed on the block diagram. One can view and modify the cluster order by right-clicking the cluster border and selecting *Reorder Controls in Cluster* from the shortcut menu. To connect two clusters, both clusters must have the same number of elements. Corresponding elements, determined by the cluster order, must have compatible data types. For example, if double-precision floating-point numeric data in one cluster corresponds in cluster order to a string in another cluster, the connecting wire on the block diagram will appear broken, and the VI will not run. If the numeric data are of different representations, LabVIEW forces them to have the same representation. In addition, one can perform the following tasks:

- Extract individual data elements from a cluster.
- Add individual data elements to a cluster.
- Break a cluster out into its individual data elements.

1.3 Problem Statement Summary

Artificial Neural Networks have gained wide-spread popularity over the last few decades. Their application is considered as a substitute for many classical techniques that have been used for many years. Some of the most common applications of an ANN would be pattern recognition, plant modeling, plant control, and image compression. A large variety of neural network architectures have been developed. These ANN models have the capability to use more than one learning algorithm for training purposes. Different aspects of ANN such as efficiency, speed, accuracy, dependability and the like have been studied extensively in the past. Many approaches have been explored to improve the performance of neural nets.

In this thesis, a new approach is proposed to build neural net architectures. LabVIEW is a graphical programming software developed by National Instruments. Using LabVIEW, ready-made Virtual Instruments (VI) can be developed for various applications. LabVIEW is basically an *Application Development Environment* (ADE) for building user friendly applications. This thesis concentrates on a LabVIEW approach to build various neural net structures.

1.4 Literature Review

The backpropagation algorithm for a feed-forward ANN has been discussed in [1] [3] [4] [5]. The error between the desired and the target output values is used to adjust the parameters (weights) of the network. The equations for updating the weight parameters are derived and explained. In this thesis, the backpropagation algorithm has been applied successfully for pattern classification, image compression and the control system modeling applications using LabVIEW.

Radial Basis Function ANNs (RBF ANN) have been discussed extensively in [1] [6] [7]. The evolution of the RBF ANNs and the techniques for training these neural nets have been discussed in [1] [7]. The reason why the RBF is a better neural net than the standard feed-forward ANN has been explained in [1]. In this thesis the RBF ANN has been used for a pattern classification application.

[1] discusses the K-Means clustering algorithm which is very popular for data clustering in pattern classification. This thesis applies the K-Means clustering technique for classifying the IRIS flower data.

Kohonen [8] proposed the idea of self-organizing maps (SOM), which has evolved into one of the most popular algorithms for unsupervised data clustering. This reference discusses how the input feature vectors organized to form groups that are dissimilar from the other groups using the SOM algorithm. The evolution of the unsupervised algorithms and their recent developments are discussed. In [8] the Learning Vector Quantization technique, which is an advancement of the SOM, has been explained and its application in speech coding has been discussed in detail. In this thesis work, the SOM algorithm has been applied for a pattern classification application.

Also a hybrid algorithm which utilizes both the K-Means and the SOM algorithms has been applied here for the classification of the IRIS flower data.

References [9] [10] discuss the development and the techniques for applying the fuzzy K-Means algorithm which is an extension of the hard k-means clustering algorithm. In the standard K-Means algorithm, the input feature vector belongs to one of ‘K’ possible clusters or classes. But in the fuzzy K-Means algorithm, it is assumed that the input feature vector simultaneously belongs to more than one cluster, but with different membership values. The crisp K-Means and the fuzzy K-Means algorithms have been built using LabVIEW VI’s that can classify the IRIS flower data [11].

The history of pattern classification and its statistical background have been discussed in [3] and [12]. Also, the Bayesian framework for classification, different approaches for pattern classification and their comparison are covered in detail. Reference [3] discusses the neural network approach for classification.

The second neural net application developed in this thesis is “image compression”. References [13] [14] [15] discuss the various approaches for image compression using a neural network approach. [1] and [15] discuss a simple application which uses the concept of Principal Component Analysis (PCA) and extend it to an image coding application using a Principal Component Neural Net (PCNN). The mathematical concepts of PCA have been explained in detail in [20]. The third application addressed in this thesis is ‘Control System Modeling’. It is also referred to as ‘Plant Identification’. [16] and [17] discuss the background of control system modeling and how a neural network can be developed into a control system model. The standard feed-forward neural network and the recurrent neural network have been used for modeling control systems. Reference [19] discusses a new type of recurrent neural networks, called the memory neuron network, which has memory neurons attached to network neurons in both the hidden layer and in the output layer and utilizies these memory neuron nets for plant identification and control. The dynamic nonlinear control plant used for modeling in this thesis is based on that addressed in [19].

Chapter 2 discusses the problem solution and the different types of neural network architectures implemented in this thesis. Chapter 3 discusses various learning algorithms utilized to train artificial neural nets. Chapter 4 discusses the various areas in which neural networks are generally applied. The various results obtained in this thesis are discussed in Chapter 5.

CHAPTER 2

PROPOSED SOLUTION

2.1 What are Artificial Neurons?

Neurons are basic building blocks of a neural network. They are processing elements that accept signal inputs, process them using a function and then form an output signal. They are also called basic nodes or units that accept inputs from an external source. Each input to a neuron has a weight or gain associated with it. In Figure 2.1, the neuron output Y_i is a function of the weighted input that can be expressed by,

$$f\left(\sum_{j=1}^{3} W_{ij} * Y_j\right) \qquad \text{… (2.1)}$$

In Eq. 2.1, W_{ij} 's are weights associated with the i^{th} neuron where i=1, 2….n; j=1, 2, 3; n is the number of neurons. The Y_j's are the inputs to the neuron. $\mathbf{Y}_i$ is the output of the i^{th} neuron. Figure 2.1 shows a simple neuron.

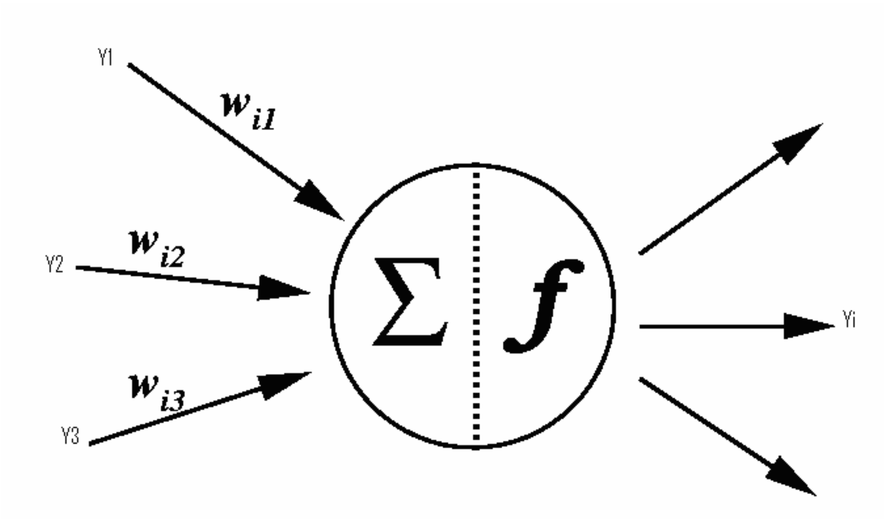


Figure 2.1 A simple neuron

A function is applied to the dot product of the weight vectors and input vectors. The dot product $W_{ij} * Y_j$ (for j = 1,2…n) is the *net input* to the ith neuron. Also the output of the neuron could be an input to some other neuron through another set of weights. This single neuron configuration was constructed using LabVIEW to yield the following block diagram in LabVIEW.

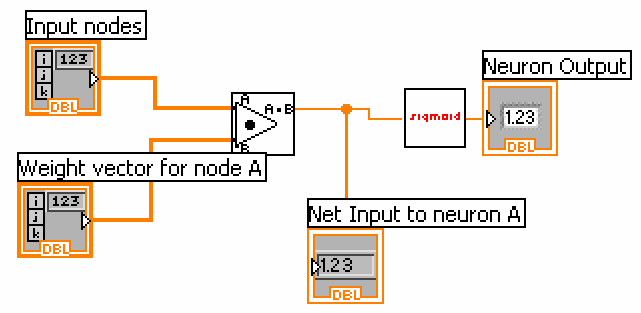


Figure 2.2 Block diagram example to illustrate dot product

In Figure 2.2, the dot product of the weight vector and the input vector for a particular node ‘A’ is performed. This is the *net input* to the neuron in which an activation function (‘Sigmoid.VI’ as shown in Figure 2.2) is applied to the *net input* of the neuron. The “Sigmoid” function is a sub-VI, which is similar to function-call or sub-routine in text based programming languages. The main library function that is used here is the ‘Dotproduct.VI’, which performs the dot product of two vectors.

2.2 What are Artificial Neural Networks?

An Artificial Neural network (ANN) is a collection of neurons in a particular arrangement or configuration. It is a parallel processor that can compute or estimate any function. Basically in an ANN, knowledge is stored in memory as experience and is available for use at a future time. The weights associated with the output of each individual neuron represent the memory which stores the knowledge. These weights are also called inter-neuron connection strengths. Each neuron can function locally by itself, but when many neurons act together they can participate in approximating some function. The knowledge that is being stored will also be referred to as “numeric data”, which is transferred between neurons through weights.

ANNs learn through training. There are various training algorithms for different types of ANN, based on the specific application. Based on the training, the weights associated with each neuron adjust themselves. By training, it is meant that a set of inputs is presented repeatedly to the ANN and the weights adjusted so that the weights reach an optimum value, where optimum means that weights either tend to minimize or maximize. An ANN is trained repeatedly and at one point it reaches a stage where it has ‘learned’ a particular desired function. With proper training it can generalize, so that even new data, which was not part of the training data, will yield the desired output.

2.3 Inputs to an ANN

The inputs to any ANN should be assigned numerical values. For example, if colors are inputs to an ANN, then each color should be assigned a number. Thus, symbols are encoded into numerical values and then applied as inputs to an ANN. Also other than symbols, parameters such as length, width and the like which are quantitative, can also be inputs to an ANN.

Inputs to an ANN are also called ‘patterns’. Each pattern could be a single numeric value or a vector. The number of ‘features’ of input data is determined by the number of components in this input vector.

2.4 Weight Vector of an ANN

As mentioned earlier, the weights are the connection strengths between neurons in adjacent layers. In Figure 2.1, w_{i1}, w_{i2}, w_{i3} are the weights associated with each of the three connections between the inputs to a neuron and the neuron. The inputs to a neuron are weighted according to the type of the neural net architecture as discussed in Section 2.5. The neural net architectures and the different types of *net inputs* used are discussed in Section 2.7.

Suppose there are 5 features applied as an input data vector and that there are 6 hidden neurons in the neural net; then the weights between the input layer and the hidden layer are represented by a 6×5 weight matrix.

2.5 *Net Input* to a neuron

It has been mentioned earlier that the *net input* to a neuron is the dot product of the weight vector and the input data vector. In addition to using the dot product norm, there is another type of *net input* to a neuron which is discussed in this section. The two types of *net inputs* to a neural net are as follows:

1. *Linear Basis Input*: This is a linear combination of the inputs. Suppose that the LBF is denoted as $U(\mathbf{x}, w)$, where

x – Input vector where **x** = (x_1,x_2,….x_m) an m-dimensional vector

w – Weight matrix

Suppose that '**x**' is assumed to be the input vector and 'w' is the weight matrix which connects the input layer to the hidden layer with n hidden neurons; then, the $U(\mathbf{x}, w)$ function (dot product norm) can be defined as

$$U_i\ (w, \mathbf{x}) = \sum_{j=1}^{n} w_{ij}\ x_j \qquad j = 1,2,\ldots m;\ \ i = 1,2,\ldots n;$$

Thus $U(\mathbf{x}, w)$ defines a set of hyper planes that linearly separate input patterns.

2. *Radial Basis Input*: This is a nonlinear basis function in which the net distance (Euclidean norm) between the input patterns and the weights is defined as

$$(W, X) = \sqrt{\sum_{j=1}^{n} (x_j - w_{ij})^2} \qquad j = 1,2,\ldots m;\ \ i = 1,2,\ldots n;$$

This is a hyper spherical function.

2.6 Activation Functions

A non-linear function (activation function) is applied to the *net input* (either Linear Basis or Radial Basis) of the neuron to produce an output. This activation function output is called the 'response' of a given neuron. This nonlinear activation function of the neuron is also simply called the "neuron function".

Some of the generally used activation functions are

1. sigmoid function:

$$F(u_i) = \frac{1}{1+\exp(\frac{-u_i}{\sigma})}$$

2. Gaussian function:

$$F(u_i) = \exp(-\frac{u_i}{\sigma^2})$$

Figure 2.3 is a LabVIEW block diagram that applies the sigmoid activation to the *net input* of a neuron. It uses an exponential function call from the library.

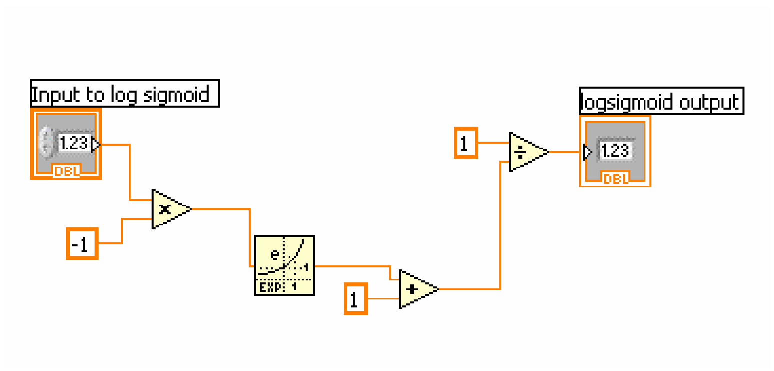


Figure 2.3 Block diagram to evaluate sigmoid function

2.7 How 'Neuron Responses' are calculated in an ANN

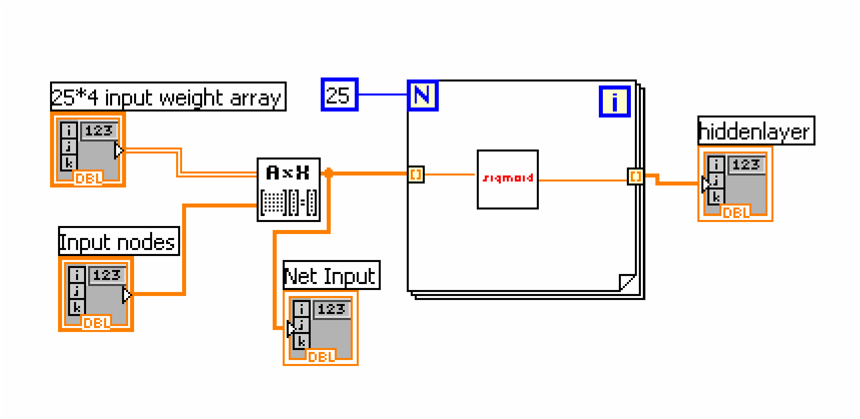


Figure 2.4 Block diagram to evaluate response of a neuron

In the LabVIEW block diagram shown in Figure 2.4, it is assumed that there are 25 neurons in the hidden layer. The *net inputs* to the 25 neurons in the hidden layer are obtained by using the "A* Vector.VI" which performs the dot product of each weight vector in the weight matrix and the input data vector. Following that, a "for loop" is used to apply the non-linear "sigmoid" activation function (Sigmoid.VI) to these *net inputs.* The "enable indexing" option available to the "for loop" could be used to apply the sigmoid function to each of those 25 *net inputs*. Even the output of the "for loop" could be indexed so that one obtains 25 hidden layer outputs in the form of a 25×1 vector.

2.8 Types of ANN

Neural nets can be broadly classified into two types.

- Supervised
- Unsupervised

Supervised ANN: This is the most popular type of ANN in which a "teacher" provides information to the network to drive it to emulate the desired function. Suppose a set of input patterns (input data vectors) are applied to the network, then the output response of the ANN is then compared with the desired response from the "teacher". The teacher would inform the network whether the output decision is correct or incorrect. Also, one could define an error criterion that would then be a basis to update the weights of the ANN so that the network will be trained with successive input patterns. This, in general is how a supervised network would be achieved.

Consider the example of pattern classification. Here, a sequence of input patterns is applied to the network. The desired output response is assigned as the “teacher”. If the network gives a correct response to a particular input then no change is made. But, if the response is incorrect, then the teacher provides an error difference between the actual response and the desired response, which is used to adjust the weights.

Unsupervised ANN: In any neural net, the most important factor is the memory stored as the values of the weights. During successive iterations of the ANN, it is this knowledge (past experiences or memory) that is being accumulated to update the weights and train the ANN. Based on the method to update the weights; ANNs are classified into supervised and unsupervised networks.

The unsupervised networks are also called self-organizing networks or rather, *competitive learning* networks. These competitive learning networks do not have a “teacher’s” guidance. The basis of unsupervised networks is clustering techniques. They help in grouping similar patterns, where each cluster has patterns closer together. Some basic unsupervised models are the Self-Organized Feature Mapping ANN (SOM), the Vector Quantization ANN (VQ), and the RBF ANN. The basic idea in all of these networks is that the hidden layer of neurons should capture the statistical features of the input data. The hidden neurons have the ability to extract the features of the data set. In most cases, the hidden neurons compete with each other to determine which one of them is closest to the input data, and the “winning neuron” is updated, which means the winning neuron is moved closer to the input data point which is a vector in multi-dimensional space. Hence, this type of network is also called “winner-takes-all” network.

In this chapter different types of neural net structures such as the Multi-layer feed-forward (FF) ANN, Radial Basis Function (RBF) ANN, Principal Component (PCNN) ANN, Memory (MNN) ANN will be considered. In the next chapter, different supervised and unsupervised learning algorithms to train these ANN will be presented.

2.8.1 Multi-layer Feed-Forward (FF) ANN

First, consider the single layer FF network. A layer is a set of neurons or computational nodes at the same level. A set of inputs can be applied to this single layer of neurons. This would then become a single layer FF network. This single layer could be called the "output layer" (OL). Each node in the OL is called an output neuron.

This structure can be extended to a multi-layer FF ANN by adding one or more layers to the existing network. These additional layers then become "hidden layers" (HL). Each node in the HL is called a hidden neuron. They appear as intermediate neurons between the input and the output layers.

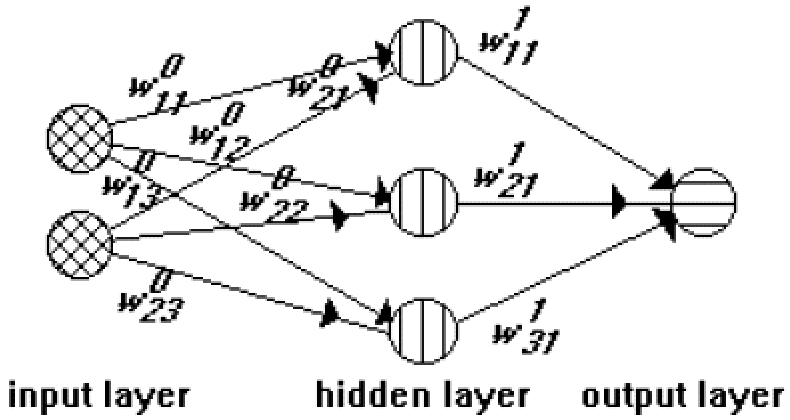


Figure 2.5 Feed-forward neural net structure

In Figure 2.5, the three layers of an ANN are shown. The first layer is the input layer where the input data vector is passed into the network. In Fig. 2.5, the input is a 2 dimensional vector. Following that is the hidden layer containing 3 hidden neurons. There are weights acting on the input data as it is passed into the hidden layer. These are the weights that are modified while training the network using training algorithms, which shall be discussed later in Chapter 3. Finally, following the hidden layer there is the output layer which has just one output neuron. Again, there are weights acting as connections between the hidden and output layers.

Each neuron acts as a local receptive field. But, when layers of neurons are added, then these layers gain the ability to extract statistical features from the input data vectors. Each node in the input layer is connected to every other neuron in the hidden layer (the next forward layer). Similarly each hidden neuron is, in turn, connected to every output neuron. Thus the architecture is such that the output response from a neuron in one layer is applied as input to neurons in the next forward layer. Suppose that there are 5 input neurons, 4 hidden neurons and 3 output neurons in an ANN, this ANN would be known as 5-4-3 network.

2.8.2 Radial Basis Function (RBF) ANN

In statistics, the concept of an RBF was introduced to solve multivariate interpolation problems. RBFs are powerful tools in “curve fitting” and “function approximation” problems. This idea was extended to an ANN and gave rise to an RBF ANN. Basically the RBF ANN looks similar to the FF ANN, but the only difference is that only one hidden layer is used in the case of the RBF ANN whereas in a FF ANN, more than one layer can be used. This hidden layer transforms the input vector (from the input layer) to a higher dimension, by applying a nonlinear transformation from the input space to the hidden layer vector space. Then, from the hidden layer, there is a linear transformation to the output layer. This ANN would be suitable for application to a pattern classification problem, because at a higher dimension, data can be classified well than at a lower dimension (geometrically speaking). The dimension of the hidden layer is defined by the number of hidden neurons. This is the reason why, in an RBF ANN, the dimension of the hidden layer is generally much higher than that of the input layer (dimension of the input vector).

Basically there are two types of *net inputs* to a neuron as mentioned earlier. These *net inputs* could be either Inner Product (Linear Basis type) or the Euclidean distance (Radial Basis type).

In the RBF case, the Euclidean norm is used. The hidden neurons have activation functions that are radial basis functions. The most simple is a Gaussian function defined as

$$G(\boldsymbol{x}, \boldsymbol{x}_i) = \exp\left(\frac{-\|\boldsymbol{x}-\boldsymbol{x}_i\|^2}{2 * \sigma_i^2}\right)$$

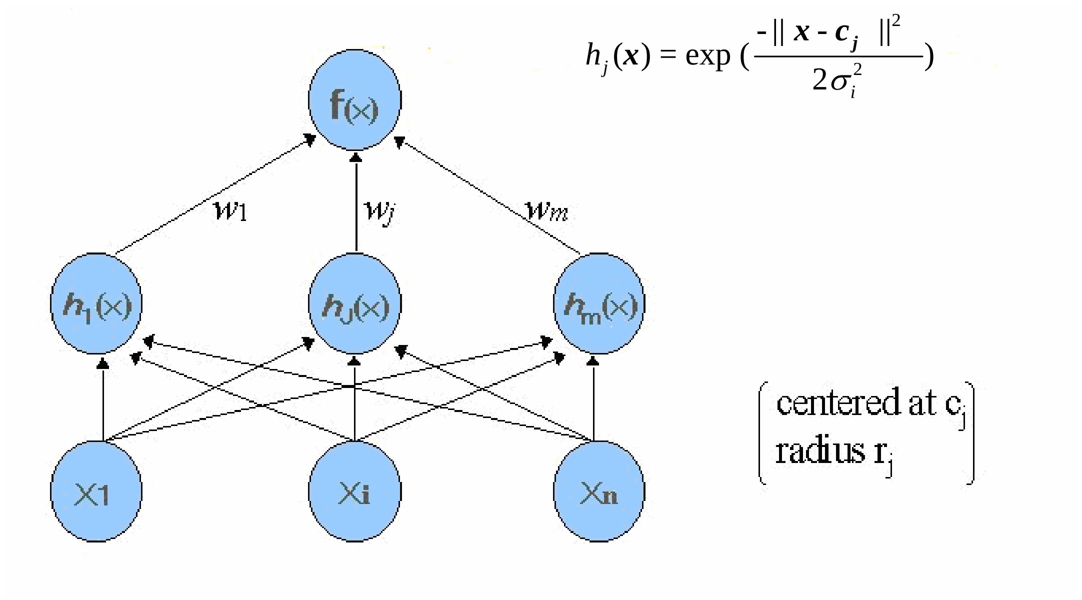
$\boldsymbol{x}$ - input feature vector

$\boldsymbol{x}_i$ – centers of radial basis activation functions of the hidden neurons

G – radial basis function

σ - radius of the radial basis function

$\|\boldsymbol{x}-\boldsymbol{x}_i\|$ - This is the Euclidean norm



$h_j(\boldsymbol{x}) = \exp\left(\frac{-\|\boldsymbol{x}-\boldsymbol{c}_j\|^2}{2\sigma_i^2}\right)$
f(x)
w_1
w_j
w_m
$h_1(x)$
$h_j(x)$
$h_m(x)$
centered at c_j
radius r_j
X1
Xi
Xn

Figure 2.6 Radial Basis Function neural net structure

As shown in figure 2.6, suppose that the hidden layer has m neurons, where each neuron is indexed by j = 1, 2…m; then, there are m radial basis functions. If the input is n-dimensional, then there would be n input nodes. Each of these nodes is indexed by i = 1, 2….n. The centers of the hidden neurons would be n-dimensional vectors. Each input feature vector would be compared to each of those m centers (which are again n-dimensional vectors) and the difference between them would be calculated. Then, the center that is nearest to the input vector is considered the "winner". Hence, these types of ANN are called competitive networks. Using 'competitive learning' the prototype hidden layer is obtained.

Once the prototype centers for the hidden neurons are obtained, then the responses of the hidden neurons to each input vector are then applied as multipliers to the linear weights into the output layer. The output layer responses are then used to classify the input patterns.

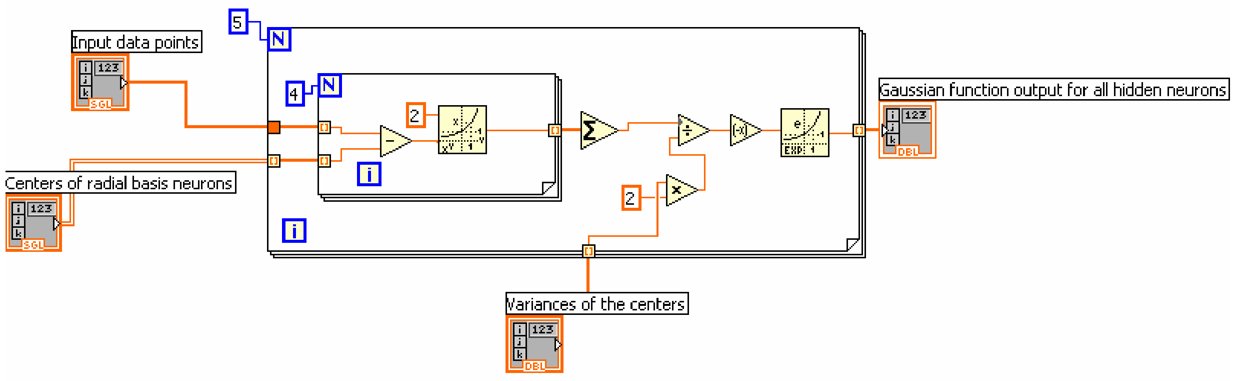

Input data points
Centers of radial basis neurons
Variances of the centers
Gaussian function output for all hidden neurons
5
N
4
N
2
2
i
i

Figure 2.7 Block diagram to evaluate the hidden layer responses of a RBF neural net

Figure 2.7 shows how the *net input* to the hidden layer neurons of an RBF ANN is calculated. There are 5 hidden neurons. The input data as well as the hidden neuron centers are 4 dimensional. It is seen that the input layer is a one-dimensional array (which is identified by single-lined wires) whereas the weight matrix is a 5×4 matrix (which is identified by the double-lined wires). There are 2 "for loops" involved here. The first one is to sequence the 5 RBF centers, one after the other. It is noted that the indexing is enabled for the "centers of radial basis neurons", whereas it is disabled for the "input data points". The second "for loop" is set-up for four iterations. '4' corresponds to the dimensions of the data points and the dimensions of the centers as mentioned earlier. Each time that this loop executes, a corresponding dimension of the data point and the center are subtracted from each other, and resulting value squared. In this way one can obtain the difference between two vectors by adding up all of the squared distances and then applying the square root function over the sum. This loop thus calculates the "Euclidean distance" between these two vectors.

2.8.3 Self-Organizing Feature Map (SOM)

The Self-Organizing Feature Map (SOM) is a special class of artificial neural nets. It is loosely based on the way the brain functions. Different sensory inputs to the brain, such as visual, auditory and the like are mapped into the brain in different zones or areas. SOM's are artificial mappings that learn through self-organization. The hidden neurons of a SOM sort-out or rather separate the input feature vectors into different domains based on the hidden neuron centers. Here, the Euclidean distances between an input pattern and the centers of the hidden neurons are calculated. The center with the minimum distance from the input vector is the "winner". Then, at the output layer, the input patterns are mapped onto a two dimensional topographic map. For example, one could assume a two dimensional 4×4 map whose coordinates range from (0, 0) to (3, 3). Based on some clustering techniques like 'competitive learning', and also using neighborhood functions, the SOM's are updated during each iteration, and thus finally, one can obtain classified data. In this method, underlying statistical features of the data are extracted. Various algorithms for updating a SOM are explained in the next Chapter.

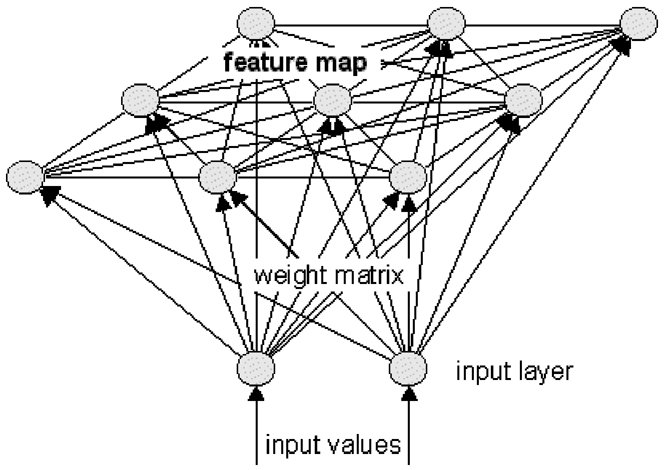


Figure 2.8 Self-Organizing map structure

2.8.4 Principal Component ANN (PCNN)

Principal Component analysis (PCA), also called principal component extraction is a technique adopted for dimensionality reduction. Using the PCA technique (explained in Chapter 3), basically a higher dimensional data space can be transformed onto a lower dimensional space. This transformation is also called the Karhunen-Louve transform or Hotelling transform. In this thesis the IRIS flower data set has been used for classification purpose. It consists of 150 4-dimensional vectors, 50 for each of 3 classes. While building the feed forward neural net for the classification of the IRIS data, initially all four dimensions of the data were used for training. After that was successfully implemented, the PCA technique was used to reduce the number of dimensions from 4 to 2. Then, these data vectors with just 2 features were classified using a neural net.

Essentially, the PCA uses the covariance matrix obtained from the input data vectors. By looking at the covariance matrix of the data vectors, one can determine which of the dimensions are highly variable across the data space. Along the diagonal of the matrix, those dimensions with the highest variance can be selected, and the data points plotted along those dimensions. One would find that most of the data points are retained (visible) in those dimensions. This is a simple approach.

The PCA technique transforms the existing data space into a totally new data space that has a better variance (geometrically speaking) across the data and also has a reduction in the number of dimension. That is, the number of components in the data vector is reduced. At the same time, most of the data points are retained. With K-L transform, new features obtained are a linear combination of the original ones.

PCNN [1] [13] is a neural network approach to the PCA. Suppose a neural net has a d-dimensional input and one wants to reduce the number of dimensions to say m ($m < d$); then one has to consider building a neural net with m hidden nodes in the hidden layer. In this thesis, a PCNN has been used to code digital images. Images are coded using PCNN by choosing the number of hidden nodes equal to the number of principal components (principal dimensions) that one decides to select from the input multi-dimensional data space. Thus the hidden layer becomes the principal component extractor. The output layer of this PCNN is going to be the same as the input layer in terms of the number of neurons, since one wants to recreate (decode at the output layer) the original image (auto-associative mapping) after it is coded in the hidden layer.

CHAPTER 3

LEARNING ALGORITHMS

3.1 Supervised Learning Algorithms

The basic idea in using a neural network is for it to learn from its environment and improve its performance over time. There are many ways to train an ANN. It is known that an ANN has synaptic weights (free parameters) that need to be updated. Improving the performance of an ANN means optimizing these weights. In the previous chapter some ANN architectures were presented. In this chapter a few learning algorithms to train these ANN's will be presented. Basic ideas for updating the free parameters of the network will now be considered. This section will discuss some of the important supervised clustering algorithms.

3.1.1 Linear Filter

Assume a simple linear neuron. It is similar to a linear filter. Linear means that the activation function used is linear. If a nonlinear function is used as the activation function, then the neuron becomes nonlinear. Because the weights attached to neurons have to be adjusted, the neural network becomes an *adaptive filter*. Consider a simple algorithm to adjust the weights.

$\mathbf{X}(\mathrm{n}) = [\boldsymbol{x}(1), \boldsymbol{x}(2), \boldsymbol{x}(3)....\boldsymbol{x}(\mathrm{n})]^{\mathrm{T}}$, which is the $\mathrm{n} \times \mathrm{m}$ input data matrix to the neuron. $\boldsymbol{x}(1), \boldsymbol{x}(2), \boldsymbol{x}(3)....\boldsymbol{x}(\mathrm{n})$ are n input data points which are m-dimensional vectors given by

$\boldsymbol{x}(\mathrm{i}) = [x_1(\mathrm{i}), x_2(\mathrm{i}), x_3(\mathrm{i})....x_m(\mathrm{i})]$ where i varies from 1,2…n.

d(i) is the desired neuron response where i varies from 1 to n.

d(1) corresponds to the response of the input vector $\boldsymbol{x}(1)$; d(n) corresponds to the response of the input vector $\boldsymbol{x}(n)$

w(i) is a weight vector which is m-dimensional given by

$\mathbf{w}(\text{i}) = [\text{w}_1(\text{i}), \text{w}_2(\text{i}) \ldots \text{w}_\text{m}(\text{i})]$ where i varies from 1 to n.

e(i) - the scalar error difference between the actual output and the desired response at some time index i

Let the output of the neuron be **x**(i) * **w**(i), which is the dot product of **x**(i) and **w**(i). To design an adaptive filter, one needs to find the cost function $\in(\mathbf{w})$. In this case it is the square of the error between the actual output and the desired outputs.

$$e(\text{i}) = \text{d(i)} - \boldsymbol{x}(\text{i}) * \mathbf{w}(\text{i}) \qquad \text{... (3.1)}$$

$\boldsymbol{x}(\text{i}) * \mathbf{w}(\text{i})$ is the dot product of the two vectors and i varies from 1 to n. The idea here is to reduce this cost function gradually with time. This can be done by adjusting the weights each time index .Using the Widrow-Hoff rule, which is generally called the Least-Mean-Square algorithm (LMS), in which the weights can be adjusted in the direction opposite to the error cost function $\in(\mathbf{w}(\text{i}))$ .Cost function is defined as

$$\in(\mathbf{w}(\text{i})) = \frac{e^2(\text{i})}{2}$$

where e(i) is error at some time index i

$$\frac{\partial \in (\mathbf{w}(\text{i}))}{\partial \mathbf{w}} = e(\text{i}) \frac{\partial e(\text{i})}{\partial \mathbf{w}} \qquad \ldots (3.2)$$

$$\text{where } \frac{\partial e(\text{i})}{\partial \mathbf{w}(\text{i})} = -\boldsymbol{x}(\text{i}) \quad ..(3.3).$$

$\frac{\partial \in (\mathbf{w}(\mathrm{i}))}{\partial \mathbf{w}} = \mathbf{g}(\mathrm{i})$ is the gradient of the cost function which is a gradient vector. In the method of *steepest descent*, the weights are adjusted according to the gradient of the error function, and based on that,

$$\mathbf{w}(\mathrm{i}+1) = \mathbf{w}(\mathrm{i}) - \mu \times \mathbf{g}(\mathrm{i}) \quad \text{... (3.4)}$$

where μ is a factor called the “learning rate”.

From the equations 3.2 and 3.3,

$$\boldsymbol{g}(\mathrm{i}) = -\boldsymbol{x}(\mathrm{i}) \times e(\mathrm{i}) \; \dots \text{(3.5)}$$

Hence from Eq. 3.4 & 3.5, the weight update equations becomes

$$\mathbf{w}(\mathrm{i}+1) = \mathbf{w}(\mathrm{i}) + \mu \times \boldsymbol{x}(\mathrm{i}) \times e(\mathrm{i}) \; \text{... (3.6)}$$

where μ is the learning parameter. So the LMS algorithm acts as a low pass filter (feedback) which passes only low frequency components of the error.

3.1.2 Back Propagation Algorithm

The Back Propagation algorithm is the most widely used algorithm for the supervised learning of an ANN. It is just an extension of the previously discussed LMS algorithm. Assume a multilayer network with one input layer, one hidden layer and one output layer. Let them be denoted as the i, j and k layers respectively. The input data is applied to the network at the input nodes. Basically, in this algorithm there is a forward pass and a backward pass through the layers of the network. The product of the weights and the input data vector, followed by the sum and evaluation of the activation functions take place in the hidden layer. The dot product of the outputs of the hidden nodes and the output layer weights are applied as input to the output layer. The forward pass ends here.

The backward pass begins by calculating the error at the output layer, where h is the squared error difference between the outputs and the desired values. Let there be k output neurons.

O_k – output of neuron k

Y_k – desired value of output neuron k

W_{ij} – weight matrix between the i and j layers. The matrix dimensions depend on the number of features (dimensions) of the input vector and the number of hidden neurons. Assuming a 3 dimensional input vector and a hidden layer with 5 neurons, the weight matrix would be a 3-by-5 matrix

W_{jk} – weight matrix between j and k layers. Since there are 5 hidden neurons and one output neuron the weight matrix would be a 5-by-1 vector

$$\varepsilon_k = O_k (1-O_k) (Y_k - O_k) \quad \dots (3.7)$$

This error function is used to update weights between output layer and next-to-last layer. W_{jk} weight vector between j and k layers is updated by the equation

$$W_{jk}^{new} = W_{jk}^{old} + \eta O_j \varepsilon_k \quad \dots (3.8)$$

η is the learning parameter whose value lies between (0, 1). Its value is assigned based on trial and error methods. The next step in the backward pass would be to calculate the error function of the hidden layer(s).The weight of the connection between i and j layers are given by

$$W_{ij}^{new} = W_{ij}^{old} + \eta O_i \varepsilon_j \quad \dots (3.9)$$

Where ε_j is given by

$\varepsilon_j = O_j(1-O_j)\sum_k W_{jk}\varepsilon_k$ where ε_j is the error function for each hidden neuron and ε_k is the error function of the output neuron..

This back propagation continues until we reach the input layer.

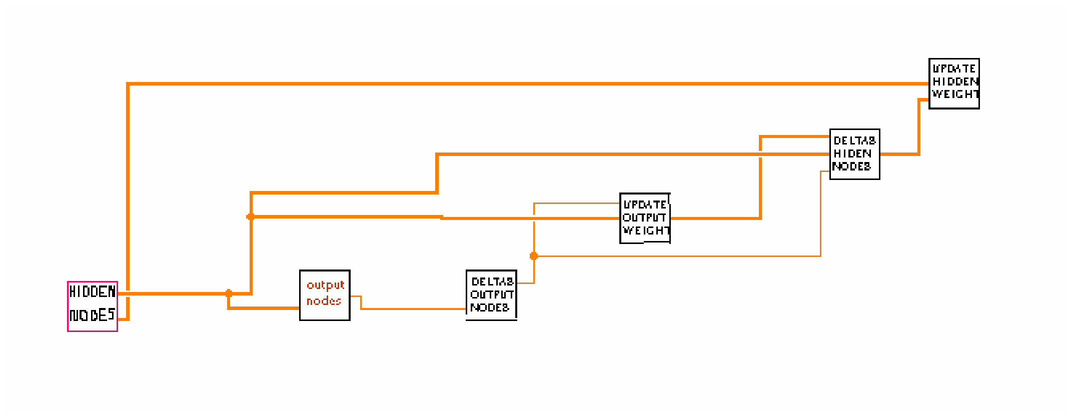


Figure 3.1 Block diagram showing the sub-VI's used

Figure 3.1 shows a part of the LabVIEW block diagram which utilizes the back propagation training algorithm to train a FF neural network to model (identify) a non-linear control system. The feed forward network has one input layer, one hidden layer and an output layer. The six sub-VI's shown in Figure 3.1 are similar to sub-routines (function calls) in a text based programming language. The six sub-VIs built are:

"Hidden_Nodes.VI" - This sub-VI evaluates the hidden neuron responses.

"Output_Nodes.VI" - Here the output neuronal responses are evaluated. Thus, so far there has been one forward pass through the network for each iteration. Now for the backward pass where the most important step of training is done, more sub-VIs are built.

“Deltas_Output_Nodes.VI” – Here the deltas (error gradients) of the output layer neurons are calculated.

“Update_Output_Weight.VI” - Once the output error gradients are known, then the output layer weights can be updated using this sub-VI.

“Deltas_Hidden_Layer.VI” – This sub-VI evaluates the error gradients of the hidden layer

“Update_Hidden_Weight.VI” - Here the hidden layer weights are updated.

Wires are connected between the various sub-VIs; and data is passed through these data connections. Also, to perform the above forward and backward pass repeatedly, one has to use a “for loop” or a “while loop” structure.

Figure 3.2 shows a part of the block diagram code designed for the neural network plant identification application. A “for loop” structure has been used, which helps in the repetition of the forward and the backward passes the required number of times.

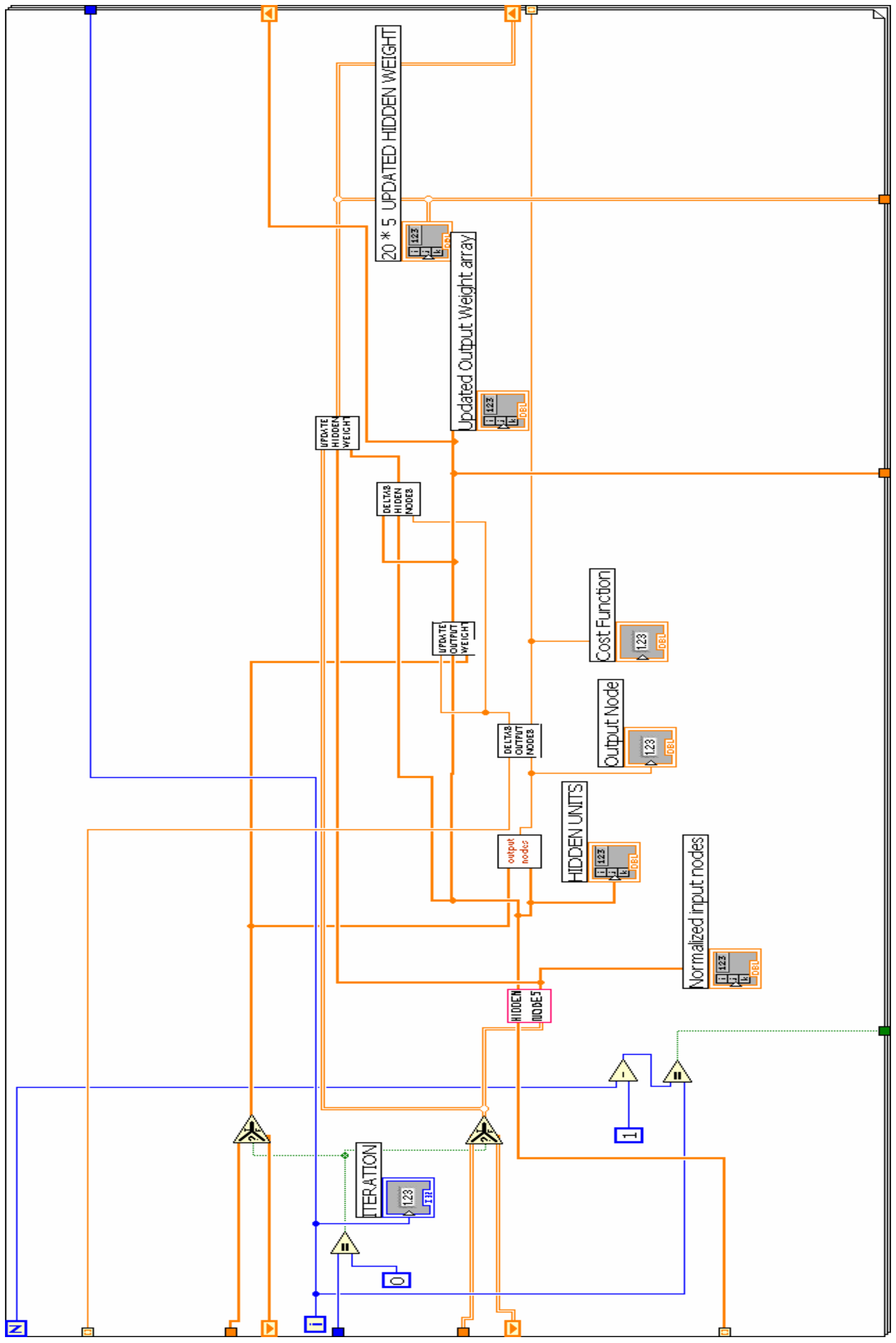


Figure 3.2 Block diagram of a feed-forward ANN built utilizing back propagation algorithm for control system modeling

3.2 Unsupervised clustering algorithms

Clustering techniques come under the concept of competitive learning. Clustering is one of the most important unsupervised training techniques. Basically it is a process by which objects that are closely related to each other are grouped. Similar objects form groups and they are dissimilar from other groups. For example, consider a radial basis network; it uses the Euclidean distance technique to find out the distance between the input m-dimensional vector and the 'prototype' radial basis centers. 'Prototype' center refers to the ideal center (mean value of a cluster) that needs to be found using learning algorithms. Each input point is an m-dimensional vector that lies in an m-dimensional feature space. The radial basis 'centers' are also m-dimensional vectors. Suppose initially centers are assumed for k Radial Basis Functions. Then, the center with the minimum distance from the input vectors is considered the 'winner' and is updated (moved closer to the input vector). The idea is to move the 'prototype' center nearer to the input vectors so that they form a group. During successive iterations, the winning vector is updated and moved closer to the input data. After a large number of iterations (epochs) and after comparing the input data points many times with the centers, one can find that the cluster centers have moved to their approximate means. Suppose that one has assumed 10 clusters; each cluster has a corresponding center (mean) point; then, after many iterations of the learning algorithm, one would be able to obtain 10 updated cluster center (mean) values. Thus 10 different clouds or clusters (either overlapping or non-overlapping) are formed. Now, new input feature vectors can be applied to the "learned network" and then one can find the closest cluster center to each input vector. Thus, one can classify future data points entering the network.

3.2.1 K-Means Clustering Algorithm

This is a well-known algorithm to determine the appropriate centers of the prototype hidden neurons in radial basis function networks. Suppose there are m centers, then

Step 1: Pick ‘m’ random points from the input data

Step 2: Select a vector randomly from the input space. Let it be **‘x’**

Step 3: Suppose k varies from 1 to m then let $t_k(n)$ be the center of the kth RBF at time step ‘n’. The minimum Euclidean distance between **x** and $t_k(n)$ is evaluated for all k varying from 1 to m. Let k(x) denote the index of the winning center.

$$k(x) = \arg\{\min(\| x(n) - t_k(n) \|)\}$$

where k = 1,2,…..,m and $\| x(n) - t_k(n) \|$ is the Euclidean norm

Step 4: The winning neuron center is updated as follows:

$$t_k(n+1) = t_k(n) + \eta[x(n) - t_k(n)] \quad \text{k is the index of the winning center}$$

$$t_k(n+1) = t_k \quad \text{, otherwise}$$

Step 5: The time step is incremented to the next iteration and then step 1 to step 4 are repeated again until the appropriate centers for the m clusters have been obtained.

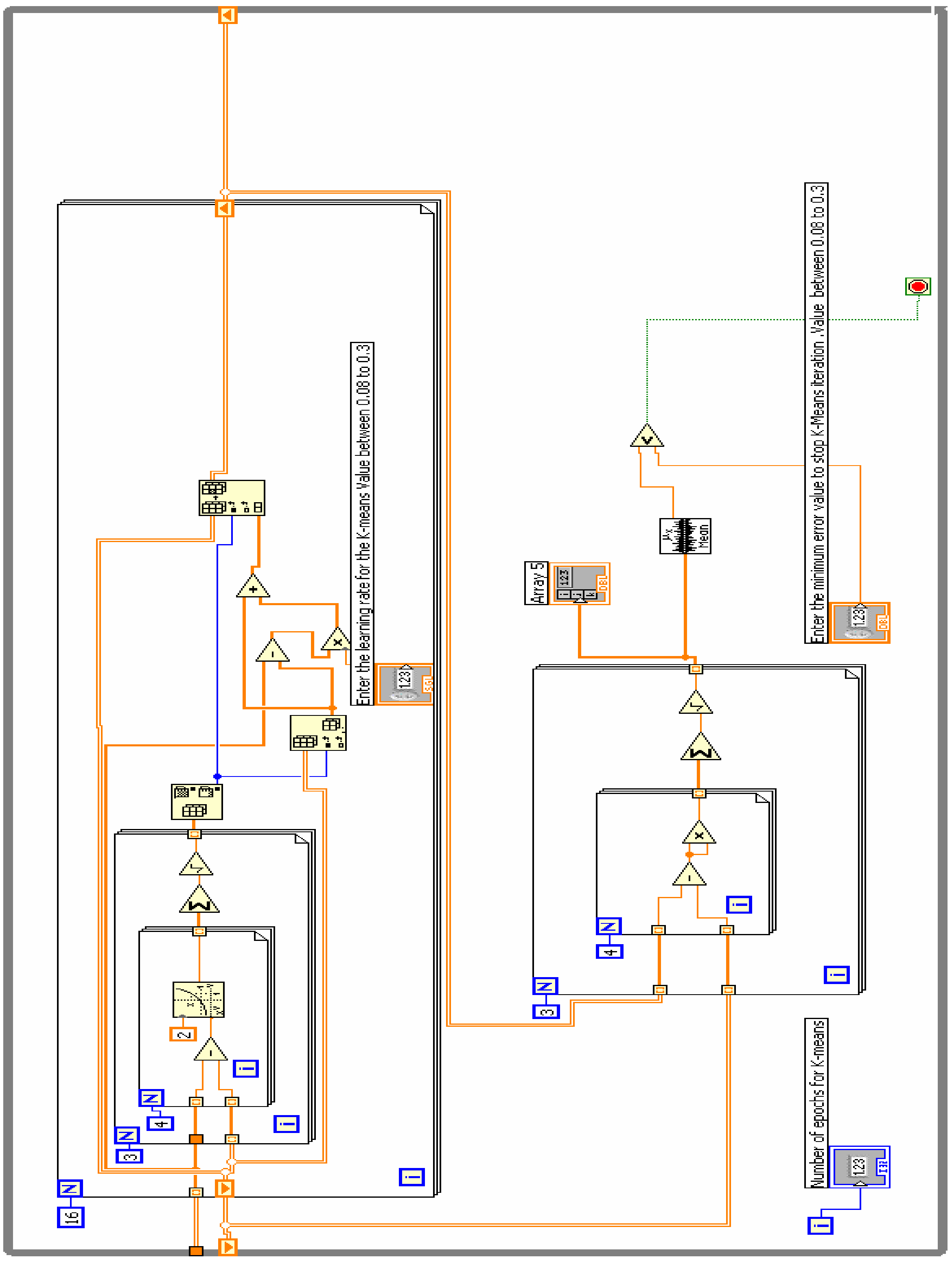

Enter the learning rate for the K-means Value between 0.08 to 0.3
Enter the minimum error value to stop K-Means iteration. Value between 0.08 to 0.3
Number of epochs for K-means
Array 5
Mean
16
3
4
2

Figure 3.3 K-Means Block Diagram

The block diagram shown in figure 3.3 executes the K-means clustering algorithm for IRIS data classification. In this thesis, IRIS data set [11] has been classified using a hybrid algorithm utilizing SOM/ K-Means algorithms. The program has two parts. The first part implements the SOM algorithm (explained in section 3.2.2) to obtain 16 ideal centers. Then, using the K-Means algorithm 3 ideal prototype centers (mean values of clusters) are obtained from these 16 possible centers obtained from the SOM algorithm. This block diagram implements the second part of the IRIS classification VI built using the hybrid algorithm, where K-Means clustering algorithm is performed. The first part (SOM algorithm) is explained in section 3.2.2. A “while loop” structure with a stopping criterion has been used here. The loop executes until the stopping criterion is achieved. The stopping criterion would generally be a pre-set level for the error. Once the error reaches the pre-set value, training stops. The ‘for loop’ with 16 iterations correspond to 16 centers obtained from the Self-organized mapping technique. The idea here is to pick 3 ideal prototype centers from those 16 possible cluster centers using the K-Means algorithm.

3.2.2 Self-Organized Feature Mapping Algorithm

The self-organizing feature mapping algorithm is an extension of the K-Means algorithm. The major difference between the two is that, in K-Means, as mentioned earlier, only the winning neuron is updated, and the center of the winning neuron is moved nearer to the input data point presented each time. But in the case of SOM, the winning neuron along with its neighbors are all updated and moved closer to the input data point during successive iterations. The main idea of SOM’s is to convert an n-dimensional input space into 1 or 2 dimensional output space with each output neuron having a distinct center from the other. The SOM algorithm [1] follows.

Step 1: Randomly choose m centers from the input vectors. Assume that are m neurons in the hidden layer.

Step 2: Select input vectors, one at each time step ‘n’ from the input space.

Step 3: Calculate the minimum Euclidean distance between each input vector and the center for each of the hidden centers and the winner is found as before using the k-means algorithm.

$$k(x) = \arg\{\min(\| x(n) - w_j \|)\}$$

where j = 1, 2…….m and k(x) refers to the index of the winning neuron

Step 4: Update the weights as follows:

$$w_j(n+1) = w_j(n) + \eta * h_{j,k(x)}(n) * [x(n) - w_j(n)]$$

Where k(x) refers to the index of the winning neuron; $h_{j,k(x)}$ is the neighborhood function in the output space defined by

$$h_{j,k(x)} = \exp\left(\frac{-\| r_k - r_j \|}{2 * \sigma^2(n)}\right)$$

Where r_k is the coordinates of the winning neuron in the 2 dimensional output feature space and r_j refer to the coordinates of all the neurons in the 2 dimensional feature output space where j = 1, 2…….m. Also $\sigma^2(n)$ refers to the variance of the Euclidean distances between the winning neuron and all the other neurons in the output space

Step 5: Repeat steps 1 to 4 until it is judged that the output space has appropriate weight vectors, which are statistically independent of each other and that the weight vectors have extracted exclusive statistical features of the input data (or input space).

Figure 3.4 shows the block diagram which implements the SOM algorithm. It is the first part of the IRIS data classification using the hybrid SOM/K-Means algorithm where SOM algorithm is implemented. Out of the 150 input data points, 16 points are randomly selected. Then, using the SOM algorithm these centers are updated during successive iterations. Finally 16 updated centers (cluster means), which are prototypes, are obtained. In order to map the 4 dimensional data into the 2 dimensional map, a “case structure” is used as shown in figure 3.4. Case structures require input conditions, based on which a case is selected. In this problem the condition for case structure is the value of the index of the winning neuron. Since there are 16 centers the index of the winning neuron may lie between (0, 15). So four cases are chosen based on the range in which the index of the winning neuron falls. If the index of the winning neuron is in the range between (0, 4) then first case is selected. If the range is between (5, 7) then the second case is selected. The range of values for the other two cases are (8, 11) and (12, 15).

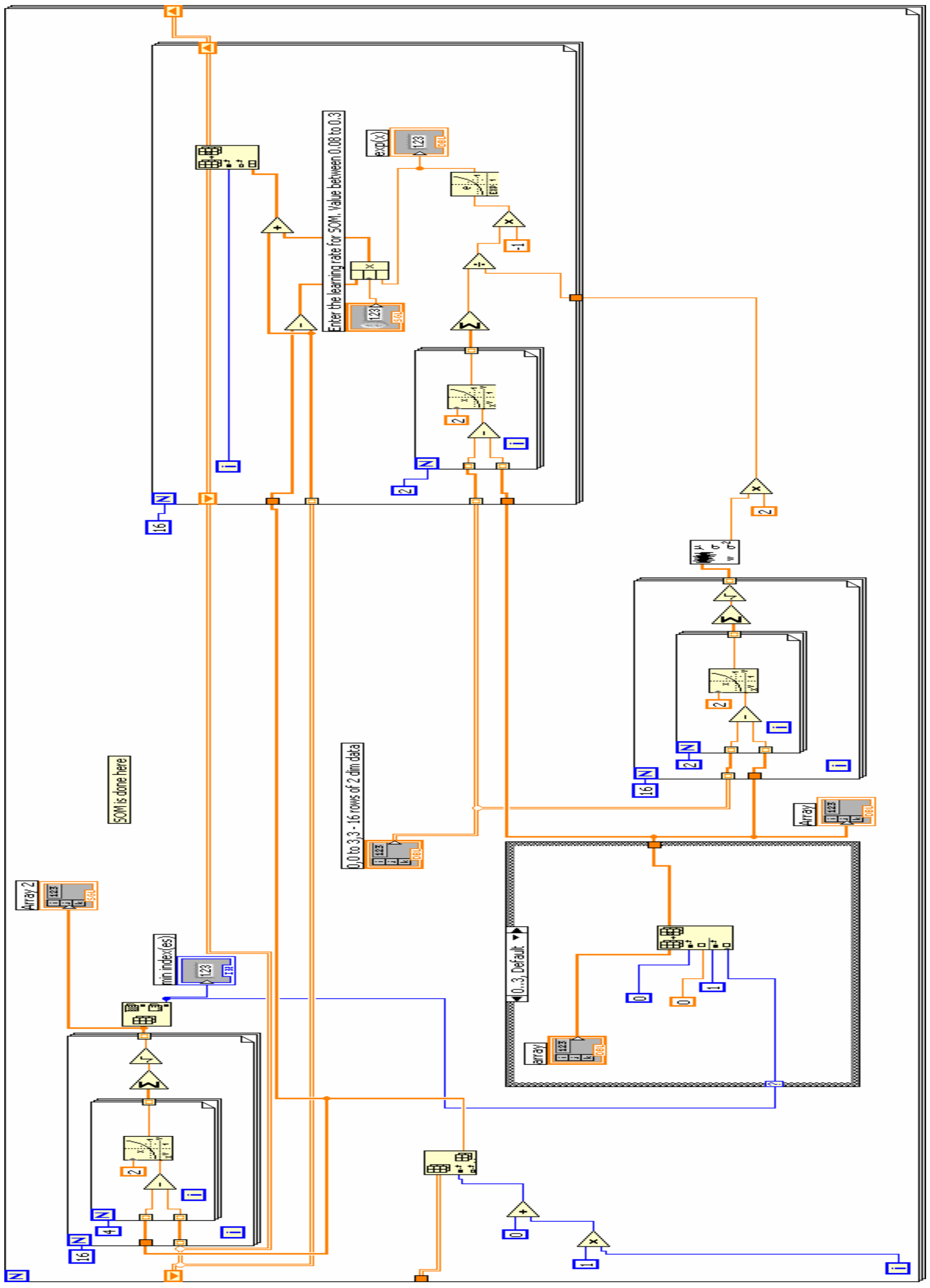

Enter the learning rate for SOM. Value between 0.08 to 0.3
exp(x)
SOM is done here
0,0 to 3,3 - 16 rows of 2 dim data
Array 2
min index(es)
Array
array
0..3, Default
16
4
2
-1
0
1

Figure 3.4 Block Diagram implementing the SOM algorithm

3.2.3 Fuzzy K-Means clustering algorithm

The Fuzzy K-Means algorithm has been very popular in pattern classification. Generally, in the 'classical' algorithms used for classification it is observed that the input data vector belongs to only one cluster. But in the fuzzy set approach one is not concerned about correct classification or misclassification, because, here it is initially assumed that the input vector simultaneously belongs to more than one cluster. Here class memberships are defined, where the vectors have a membership in all of the possible classes. This approach seems to be very successful, but later, it shall be seen that even this approach has some drawbacks. Since, in this thesis, most of the pattern classification results use the IRIS flower data [11], it is assumed that there is prior knowledge about the data, such as for example, the number of clusters (classes). The IRIS flower data consists of 4-dimensional vectors. Totally, there are 150 data points. This data set consists of feature extracted data so that no further feature extraction is required.

In fuzzy clustering, it is assumed that there are 'k' clusters that can be sorted or grouped from the input data vectors. Basically, one begins by building a membership matrix, also called the fuzzy matrix. Suppose that there are N data points and m clusters; then one can build a $\mathrm{N} \times \mathrm{m}$ fuzzy matrix. Each entry in the matrix can be denoted by u_{ij}, which would be the membership value of the i^th input data point in the j^th cluster. The values of the membership matrix lie between (0, 1). The total sum of membership values for each data point in all the clusters is 1. That is

$$\sum_{j=1}^{m} u_{ij} = 1, i = 1, ..., N$$

where $u_{ij} = \frac{1}{\sum_{k=1}^{m} (\frac{d(x_i,\theta_j)}{d(x_i,\theta_k)})^{\frac{1}{q-1}}}$ for various values of i=1 to N and j=1 to m; $d(x\ ,\theta\ )$ would be the distance between $\mathbf{x}_i$ and θ_j in the feature space; m is the number of clusters and N is the number of data points; θ_j would be the cluster center (parameter). The grade of membership of a point in one cluster also depends on its grade of membership in other clusters. All algorithms for clustering have an error function that needs to be minimized. Fuzzy clustering has an error function that is defined as follows:

$$J_q(\theta,U) = \sum_{i=1}^{N} \sum_{j=1}^{m} u_{ij}^q d(x_i,\theta_j)$$

where θ would be a parameter of each cluster which in general would be the mean or the cluster center. For a d-dimensional vector the parameter θ would also have d-dimensions. q is a new constant called the '*fuzzifier*' constant or degree of fuzzification. This constant directly affects the fuzziness by making it either fuzzier or less fuzzy (where it becomes hard clustering or 'crisp' clustering) similar to earlier algorithms like the K-Means approach where there is no fuzziness involved.

$d(x_i,\theta_j)$ is the distance between data point $\mathbf{x}_i$ and the cluster center θ_j. There are many different 'distances' that could be used. Here the distances refer to that between two vectors in the feature space. Some of the common distances are Euclidean, 'Minkowski' and the 'Mahalanobis' distances.

Mahalanobis distance is given by

$$d(x_i,\theta_j) = (x_i - \theta_j)^T A(x_i - \theta_j)$$

where ‘A’ is the covariance matrix of the data points for a particular cluster. For a d-dimensional feature space, each cluster would have a $\mathrm{d} \times \mathrm{d}$ covariance matrix associated with it.

Consider the actual algorithm [9] [10] which has been implemented using LabVIEW.

Step 1: Choose a random center θ_j for each cluster, j= 1, 2… m

Step 2: Select an input vector at each time step as before. Calculate

$$u_{ij} = \frac{1}{\sum_{k=1}^{m} \left(\frac{d(x_i,\theta_j)}{d(x_i,\theta_k)}\right)^{\frac{1}{q-1}}}$$

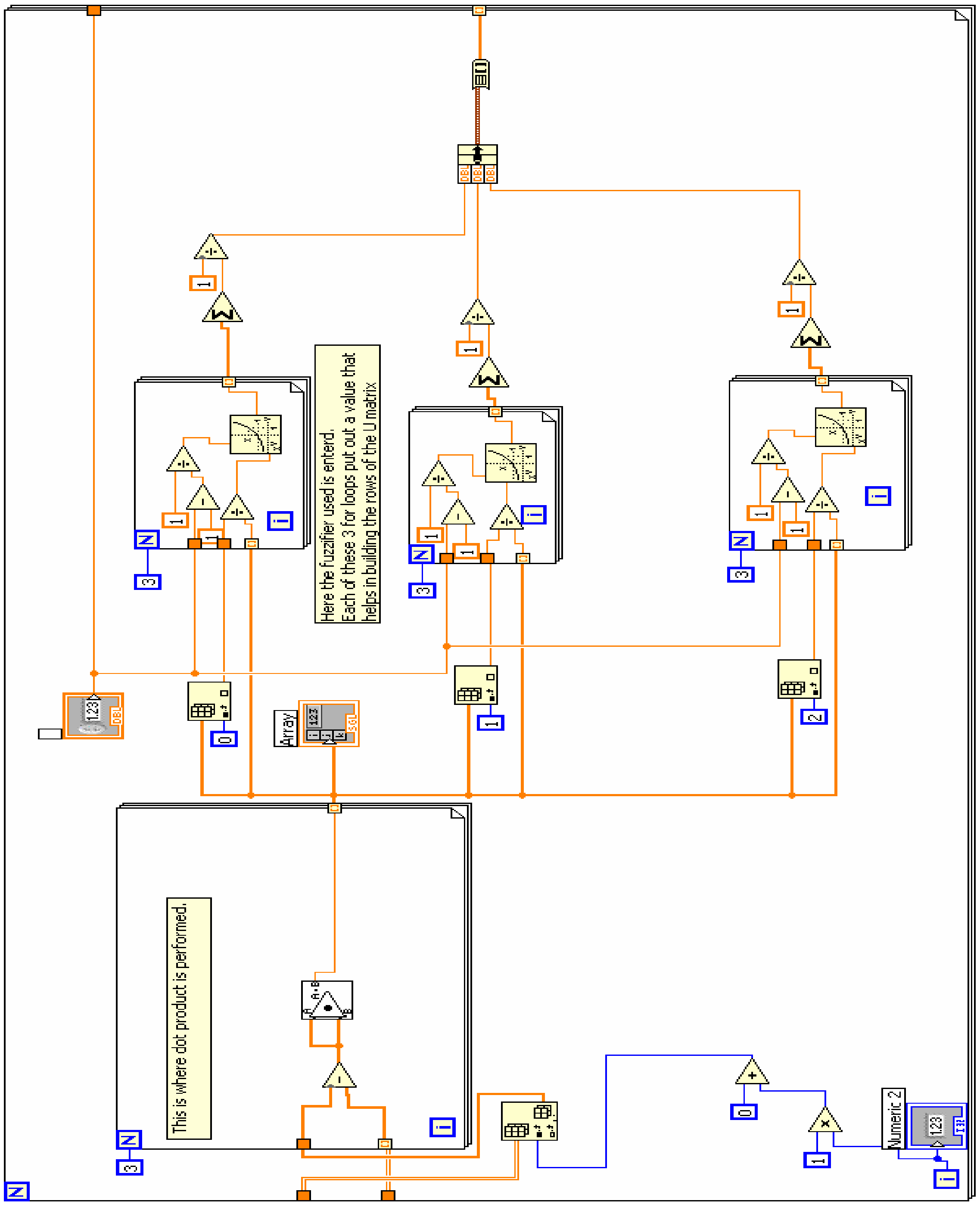
for various values of i=1 to N and j=1 to m; $d(x\ ,\theta\ )$ would be the distance between $\mathbf{x}_i$ and θ_j in the feature space. The denominator always lies in the range (0, 1).

Step 3: In the previous step the membership matrix has been initiated. In this step the goal is to update the center of the clusters and move it toward the center mean value.

$$\theta_j = \frac{\sum_{i=1}^{N} u_{ij}^{q} x_i}{\sum_{i=1}^{N} u_{ij}^{q}}$$

Here, there are a couple of methods by which one can update the centers, where the methods typically depend on various factors, such as, for example, using different distance measurements.

In Figure 3.5 the block diagram executes step 2 of the above algorithm for classification of the IRIS flower data. The outer “for loop” corresponds to the number of input data points whereas the inner “for loop” corresponds to the number of possible classes. So, the distance between each input data point and the three cluster centers is calculated and the sum of their ratios is stored in memory. This block diagram completes step 2 of the above algorithm in which the membership matrix is calculated.



Here the fuzzifier used is enterd.
Each of these 3 for loops put out a value that
helps in building the rows of the U matrix
This is where dot product is performed.
Array
Numeric 2

Figure 3.5 Block Diagram to evaluate the fuzzy membership matrix

Figure 3.6 shows the block diagram in LabVIEW that implements step 3 of the fuzzy K-Means algorithm. Here the cluster centers are updated at each iteration (each time an input data point is presented). The outer ‘for’ loop executes three times, once for each of the three clusters. The fuzzy matrix has three columns corresponding to the three clusters.

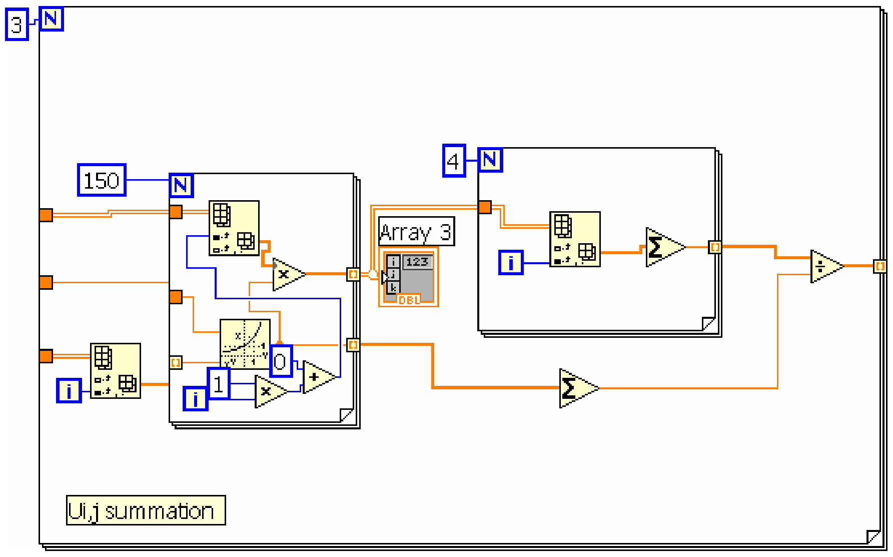


Figure 3.6 Block diagram to update the cluster centers

3.3 Principal Components Extraction Algorithm

Principal Components Analysis (PCA) is a technique used to reduce the number of dimensions of a multi-dimensional data set. In this thesis the 4 dimensional IRIS flower data set has been used for classification purposes. Out of the 4 dimensions, 2 of the principal dimensions (components) are extracted using this PCA technique. The algorithm to extract the principal components is discussed here in this section. The steps to obtain the principal features are as follows.

Step 1: The IRIS data has 4 features. Initially, the mean value for each of the 4 dimensions is calculated. Then the mean values are subtracted from each of the data dimensions.

Step 2: Then the covariance matrix corresponding to the 4 dimensions is calculated using this mean.

Step 3: The eigenvalues and the eigenvectors are calculated from the covariance matrix, since the covariance matrix is a square matrix. The 4 eigenvectors are along the 4 columns.

Step 4: Arrange the eigenvectors by their eigenvalue, principal to lowest. The eigenvector corresponding to the highest eigenvalue is the *principal component.* This *principal dimension* has the maximum variance across the data. Since, in this report, the idea is to reduce the dimensions from 4 to 2, the 2 eigenvectors with the smaller eigenvalues are dropped and only the two most significant components are chosen. Thus a matrix is formed with the two significant eigenvectors.

Step 5: Then, the transpose of the matrix with the two vectors is multiplied to the left of the original mean subtracted transposed data set.

$$FinalData = RowFeatureVector \times RowDataAdjust$$

where the *RowFeatureVector* is the transpose of the matrix with the two column vectors; *RowDataAdjust* is the original mean subtracted transposed data vectors; *FinalData* is the final data set with data items in columns and dimensions in rows. This gives the original data solely in terms of the vectors chosen (eigenvectors).

Thus the PCA algorithm is implemented. Also, one could get back the original data from this transformed data. In this report, the PCA is mainly used for pattern classification and not for data compression. So the reverse process of getting back the original data from the transformed data is not used here.

CHAPTER 4

AREAS OF APPLICATION

4.1 Pattern Classification

Pattern Classification is part of the much broader field called Pattern Recognition. Recognition basically has 2 parts

- Feature Extraction
- Pattern Classification

In this thesis, the LabVIEW VI's have been developed for pattern classification applications, assuming that the data is already defined over feature space (feature extracted form).

Classification is very important in numerous fields. A simple example is in a post office where mail to be delivered needs to be sorted. If one has an automatic system in which a computer can recognize the zip code and sort the mail based on zip codes then it would make the work much more efficient. What is needed is a scanner which reads the data into the system. Then, the system can use an effective algorithm to differentiate and sort the mail. There are many other applications where one could make use of such an automated data classifier.

There are many algorithms and techniques to classify data. In this thesis, LabVIEW VIs have been constructed for pattern classification applications that have a useful GUI and are very user-friendly. This chapter will discuss some statistical classification techniques followed by a practical neural net approach and other clustering algorithms to classify data.

Basically, a feature extracted data set could be of any dimension. In this thesis, a popular IRIS data set [11] has been used. This feature extracted data has been extensively used in the past and has been thoroughly analyzed. It is known that this data set consists of 4-dimensional vectors belonging to three classes. Out of the total of 150 vectors, each of the three classes consists of 50 vectors, each class defined by a cluster.

The idea of classification is to form clusters of data from a pool of data points. The statistical distribution of this IRIS data set has been studied in the past. Essentially, one could split the data analysis into two types: Parametric and non-parametric. In the parametric case, good prior knowledge is available about the data, such as the probability density function, *a priori* probabilities of occurrence for each class, and perhaps one or more moments for each class. For the non-parametric case, such knowledge is not assumed, but rather classification must be based on clustering. Next, Bayes' theorem will be addressed, and then the discussion will continue with building classifiers.

4.1.1 Statistics of Classification

Thomas Bayes was a mathematician who developed a method to classify data into clusters based on the probability theory known as maximum likelihood. A Bayes' classifier is a mathematical model or classification system. Since there are 3 classes and 50 points per class, then the *a priori* probability of occurrence for a sample value (point) to be from any one of the 3 classes can be taken to be 1/3, or $P(C_k) = 0.33$, k = 1, 2, 3.

These probabilities are assumed *a priori*. The probability of a feature vector $\boldsymbol{x}$ given that class C_k has occurred is denoted by $P(\boldsymbol{x}/C_k)$ which is called the class conditional probability. These conditional probabilities must be assumed or have to be calculated based on a set of rules. Then, let the probability of a particular pattern, $\boldsymbol{x}$, occurring, independent of any class be denoted by $P(\boldsymbol{x})$. Then, according to Bayes' theorem, the probability that pattern class C_k is the true class, given feature vector $\boldsymbol{x}$ is denoted by $P(C_k/\boldsymbol{x})$, the *posterior* probability. This probability is related to P($\boldsymbol{x}$) and $P(\boldsymbol{x}/C_k)$ as follows,

$$P(C_k/\boldsymbol{x}) = \frac{P(\boldsymbol{x}/C_k)P(C_k)}{P(\boldsymbol{x})}$$

The denominator is called the normalization factor that assures that,

$$P(C_1/\boldsymbol{x}) + P(C_2/\boldsymbol{x}) + P(C_3/\boldsymbol{x}) = 1$$

Essentially the theorem assigns a membership value between 0 and 1 to a particular feature corresponding to its membership in each of the 3 classes. The class C_p, corresponding to the largest value of $P(C_k/\boldsymbol{x})$, k = 1, 2, 3 for a given measured $\boldsymbol{x}$ is the winner, which means that $\boldsymbol{x}$ belongs to class C_p.

For classification [3], the basic idea is to estimate the density value of a particular feature vector in a particular region of feature space. Using parametric techniques as applied to the IRIS flower data set; one can determine the conditional probability densities and assume *a priori* probabilities. Using these and Bayes' theorem, one can determine the *posterior* probabilities for all of the three clusters. The highest value of the *posterior* probability $P(C_k/\boldsymbol{x})$ would be the winner, and $\boldsymbol{x}$, then belongs to the cluster with the highest value. This procedure is normally applied in statistical classification, and it is a very effective procedure.

Suppose that the data distribution is unknown. Then, one will need to employ non-parametric algorithms to classify the input data points. However, with the data set available, one can employ unsupervised algorithms that can automatically capture features from the data and cluster the data.

In classifying patterns in feature space based on geometric properties, the general procedure is to construct boundaries to separate clusters; the more well-placed that the boundaries are, the better the classification in terms of reducing the associated error of misclassification. If these boundaries are poorly chosen, the misclassification error will increase. These boundaries can be defined mathematically by 'discriminant functions'. In general, classification techniques are aimed at constructing these discriminant functions based on the input feature vectors.

4.1.2 Statistics and Neural Nets – an Analogy

Now, consider how a neural net functions in the process of pattern classification. Normally the standard feed forward neural net uses a non-linear transfer function to map the input feature space to the output decision space. For example, in this thesis the sigmoid function has been used for different nets. Suppose the input feature vector is **x** and the weight vectors are denoted by **w** as before, then the dot product and summation is given by

$$net = \sum x.w$$

Applying the sigmoid function to *net* gives the result

$$f(net) = \frac{1}{1+\exp(-net)}$$

as the output of the neuron. This $f(net)$ is nothing but the *posterior* probability function that was earlier obtained using the Bayes' theorem.

Consider a two-class problem with equal *a priori* probabilities of occurrence. Then, the *net* value that was calculated using the inputs and the weights is just the likelihood ratio of the class conditional probabilities expressed as,

$$net = \frac{p(\boldsymbol{x}/c_1)}{p(\boldsymbol{x}/c_2)}$$

If this likelihood ratio is greater than 1 then the $\boldsymbol{x}$ belongs to class C_1. If the likelihood ratios is lesser than 1 then $\boldsymbol{x}$ belongs to class $C_{2.}$ Thus the *net* input to a neuron is the class conditional probability $p(\boldsymbol{x}/c_k)$ where k= 1, 2 and the output is the *posterior* probability $P(C_k/\boldsymbol{x})$. Thus, the neural net can be considered to be an implementation of Bayes' theorem by analogy.

The sigmoid function is called the 'S-shaped' function which compresses any input value onto a small range, (0, 1). The sigmoid activation function acts a discriminant function to classify data in feature space. Although the sigmoid function is non-linear, the decision boundary it creates is still linear.

Geometrically speaking, there are many kinds of data to be classified. Some data sets can be linearly separable. But some data distributions cannot be linearly separated into clusters or classes. Well-known examples of data sets that are not linearly separable are the binary XOR data and the IRIS flower data. In the case addressed here in regard to the IRIS flower data, it has been confirmed that for the three clusters or defined pattern classes, two of the classes or clusters overlap.

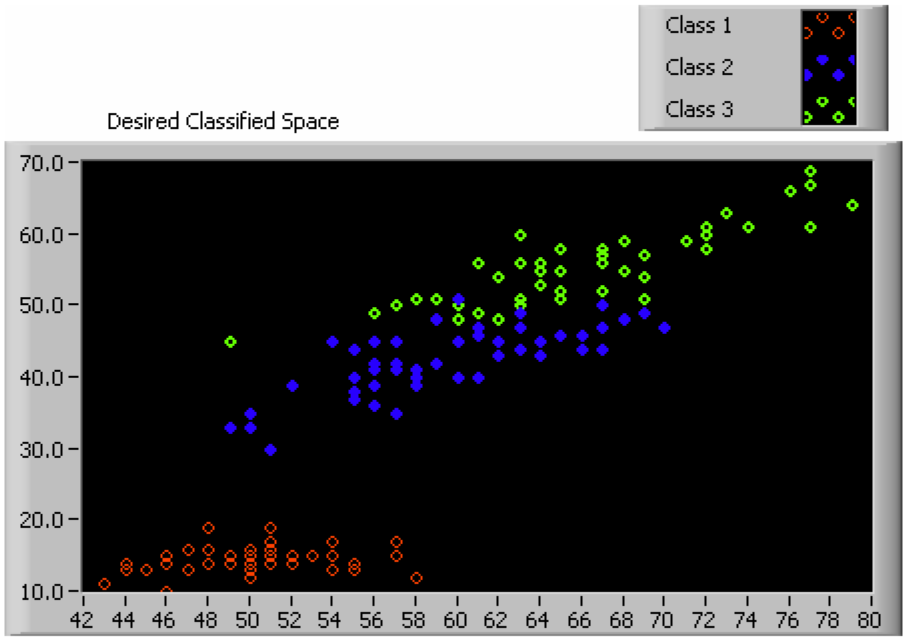


Fig 4.1 Based on the features 1 & 3, the class or cluster represented in red is linearly separable from the remaining two classes, represented in blue and green. The 4-dimensional vectors are reduced to 2 dimensional vectors using PCA. It was found that the feature 1 and feature 3 have high variance across the data. So, the 1st and 3rd dimensions were chosen to plot the IRIS flower data set. The x-axis in the above figure is the 1st dimension and y-axis is the 3rd dimension.

4.2 Image Compression

Digital images have become very popular in recent times. Every digital image is specified by the number of pixels associated with the image. Each pixel in a gray-level image is described by an intensity of the image at that point. An image that is 256×256, means that there are 65536 pixels (intensity points) in the image in a matrix form with 256 rows and 256 columns. Digital images are basically classified into two types: grayscale images and color images. Any color can be defined by the combination of the three primary colors – red, green and blue. A grayscale image has no color information. Therefore, every pixel in a grayscale image has different shade of gray which is commonly represented by 8 bits. So, there are 2^8 = 256 possible intensity values (shades of gray) for a grayscale image ranging from 0 to 255. The depth of the image is said to be 8 since 8 bits are used to represent each pixel.

Since 8 bits are used to represent each pixel, to represent an image which is of dimension 256×256, we need $256 \times 256 \times 8 = 524288$ bits are needed to represent the image. With limited memory space, it becomes useful to compress the digital image so that it occupies less memory and also becomes easier to share over a medium such as the internet. There are a lot of algorithms and techniques to compress images. This thesis applies a neural network based approach to compress digital images. The well-known '*Lena*' (bitmap format in this thesis) grayscale image (256×256) has been used to demonstrate the technique.

Each pixel in an image can be denoted as a coefficient, which represents the intensity of the image at that point. Then, the idea of compressing an image is to encode these coefficients with reduced bitswith lesser number of binary digits and at the same time, retain the quality of the image. Once compressed, these coded images which occupy less memory space can be transferred over the internet medium for sharing purposes. At the receiving end these compressed images need to be again decoded or decompressed so that one can recover the original image.

The quality of the received image can be tested by some standard error calculations. Each pixel of the original image can be subtracted from the corresponding pixel of the decompressed (received) image and the error between the two values can be squared. The mean of all the squared errors for all the pixels is called the MSE (Mean Square Error). The higher the value of this MSE, lower the quality of the decompressed image. Different compression algorithms are compared and the best of these is used for practical applications. A very well-known digital image compression algorithm is the JPEG (Joint Photographic Experts Group) algorithm. One part of this algorithm uses the Discrete Cosine Transform (DCT) technique to code the image coefficients (pixels).

Compression algorithms can be classified into two types – ‘lossy’ and ‘lossless’. If the recovered image (after decompression) does not have the same quality as the original image then there has been a loss of some image data during compression. This is called a ‘lossy compression algorithm’. But some algorithms have the ability to retain the quality of the image, even after the compression, and the decompression processes. Such algorithms come under the category of ‘lossless compression algorithms’.

4.2.1 Applying ANN to Compress Images

Recently, other than the classical techniques that have been traditionally used for image compression, some new ideas were developed using the ANN [14] [15] [16].
The Principal Component analysis technique (PCA) and the Principal Component ANN‘s (PCNN) were discussed in Chapter 2. In this thesis, LabVIEW Virtual Instruments (VI’s) which implement the PCNN’s have been built for digital image compression.

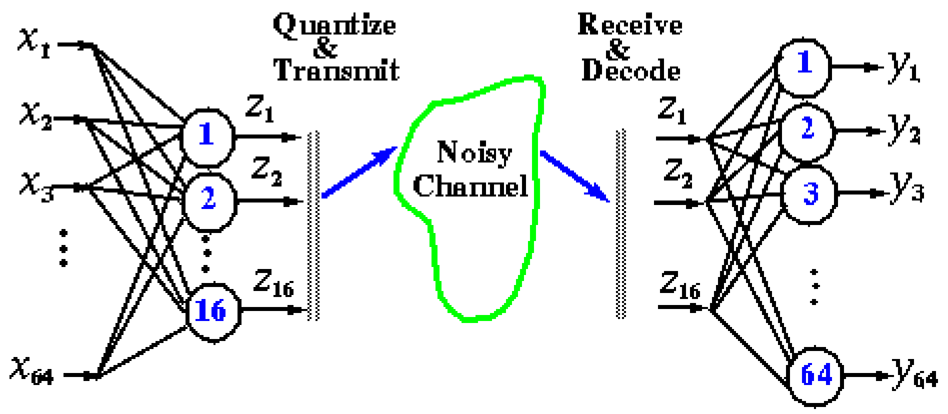


Figure 4.2 Obtained from [15]. Using a feed-forward neural network for compressing and transmitting images over a communication channel

Watta et.al [15] suggests using the multi-layer feed-forward neural net to compress images. The Lena image that has been used for compression purposes is a 256×256 image. This image can be broken into blocks of size 8×8. There will then be 64 pixels per block. Totally, there will be 32×32 = 1024 blocks. The 64 pixels in each block then becomes the input vector to the neural net. The main idea in using a neural network to compress images is to code these 64 coefficients using a reduced number of coefficients (bits per pixel). The neural network architecture is very helpful in this context. If one can reduce the number of dimensions in the hidden layer (number of hidden neurons) to be much less than the number of dimensions in the input layer, then there will be a reduction in the number of coefficients during the transition from the input to the hidden layer. An input layer with 64 dimensions and a hidden layer with 16 dimensions, for example, means that the 64 pixels of the image (8×8) block which is applied to the input layer has been transformed into 16 coefficients in the hidden layer.

Then, one could again use an output layer which has 64 dimensions to recover the original 64 pixels. The basic idea here is to learn the identity mapping or rather associative mapping which means the output of the neural net is the same as its input. Thus, with a 64 dimensional input layer, a 16 dimensional hidden layer, and a 64 dimensional output layer – a neural network can be used for image compression. One of the main purposes for compressing images this way is to make it easier to transfer the image over the internet.

The neural network can be split into two parts. The first part with a 64 dimensional input layer and a 16 dimensional hidden layer (16 hidden neurons) is used for encoding the image. Then, the responses of the 16 neurons in the hidden layer would be transmitted over the internet medium. The internet is denoted as a "noisy channel" in the Figure 4.2. At the receiving end, these 16 responses would be decoded and projected onto a higher dimensional layer. This higher dimensional layer would ideally be the output layer of the neural network.

In a feed-forward neural network, generally the log-sigmoid or the tan-sigmoid non-linear activation function is used. The tan-sigmoid function squashes all input values to the range (-1, 1). The 16 hidden neurons would respond with 16 values in the range (-1, 1). The number of bits to code these values would be infinity. So, one should quantize these values using a suitable quantization technique before transmitting them into the internet medium.

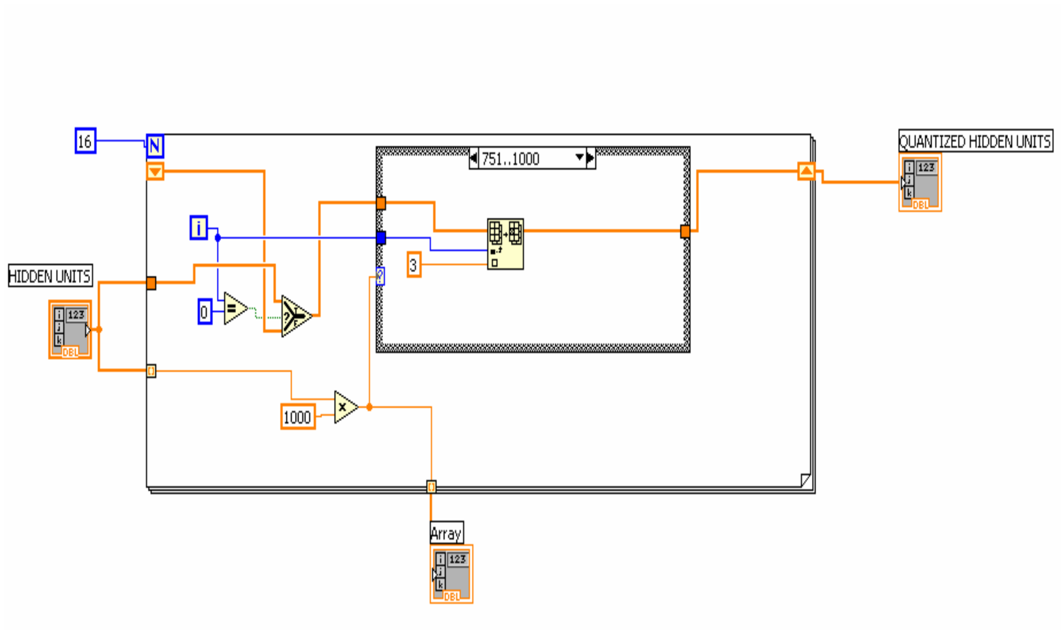
Table 4.1- Quantization table

Range of the hidden neuron response	Quantization Value
0, 0.25	00
0.26, 0.5	01
0.51, 0.75	10
0.76, 1	11

Using Table 4.1, the hidden neuron responses could be quantized and then transmitted. At the receiving end these quantized values are de-quantized (decoded) using Table 4.1 (inversely) and applied as the input to the hidden layer. Then, the responses of the hidden layer neurons are again projected to a higher dimensional output layer. Since there are 64 input features, and 8 bits to represent each pixel, one needs 64×8 =512 bits to represent the 64 features of an input vector (8×8 block). Assume that the 16 hidden neuron responses are quantized using a 2-bit quantizer; then the total number of bits to represent the 16 hidden neuron responses is $16 \times 2 = 32$ bits. As a result, there has been a reduction in the number of bits from 512 to 32 as the data passes from the input layer of the network to the hidden layer of the network. The compression ratio is given by $512/32 = 16$ which is 16:1. This is a very effective compression ratio. The compressed quantization values of the hidden neurons are transmitted over a medium like the internet and received at the other end of the communication channel. At the other end, the data is again de-quantized back and passed through the hidden layer. The hidden layer responses are again projected onto the higher dimensional output layer to recover the original image.

Before using a neural net with such a structure, one needs to train the neural net for an identity mapping where the neural net could produce an output which is the same as its input. A simple method to train this neural net would be to apply different 8×8 blocks (either randomly or sequentially across the image) of the image into the neural net and then train the neural net to produce the same pixel values of the output. Then, the target response vector for the neural net would be the same as the input vector.

Figure 4.3 shows the quantization and de-quantization part of the block diagram developed for the PCNN.



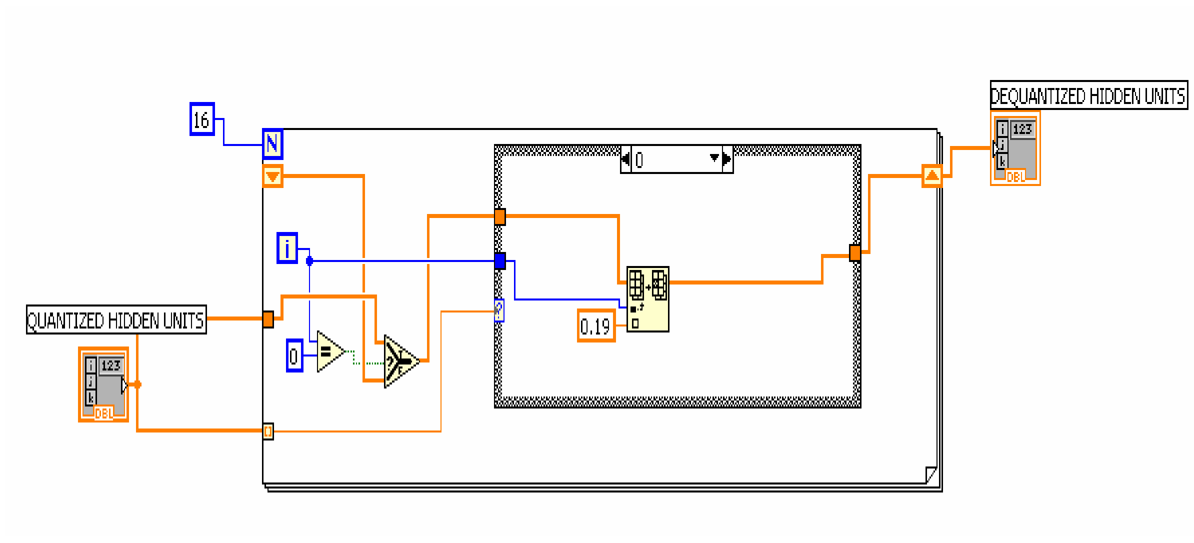


Fig 4.3 Block diagrams implementing the quantization and de-quantization (decoding) part

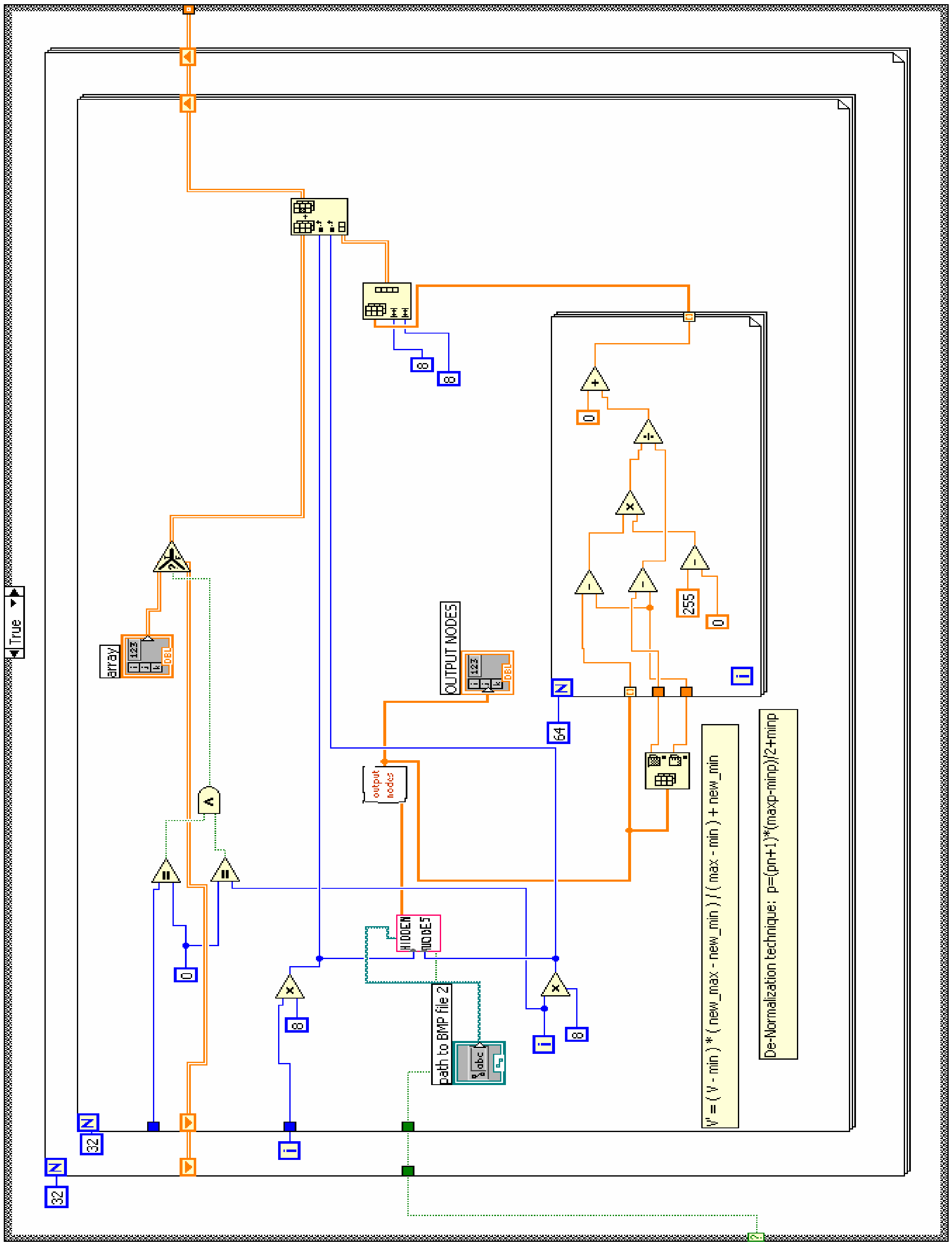

True
array
OUTPUT NODES
output nodes
HIDDEN NODES
path to BMP file 2
V' = (V - min) * (new_max - new_min) / (max - min) + new_min
De-Normalization technique: p=(pn+1)*(maxp-minp)/2+minp
32
32
64
255
8
8
8
8
0
0
0

Fig 4.4 This block diagram implements the ‘testing part’ of the image compression PCNN

Once the PCNN is trained then, for testing purposes, the image is sequentially scanned from left to right and from top to bottom for 8×8 tiny blocks. During each scan, each 8×8 block is applied as input to the PCNN and the 64 output responses are stored in memory. After entire image is scanned, the stored responses of the ANN could be used to reproduce the image.

4.3 Control System Modeling

In this thesis, LabVIEW VI's have been developed for control system identification. Any plant or control system can be modeled using mathematical analysis techniques. Extensive research has been done on plant modeling over the last few decades. Control systems can be classified into two types – Linear control systems and non-linear control systems. After a model is formulated for a control system (or a plant), then it becomes easy to analyze the nature of the system in terms of its outputs generated for a given set of inputs. Also, designing controllers for the plant then becomes easier. Neural networks can be used as tools for plant identification. References [16] and [17] discuss neural network techniques for plant modeling. Non-linear dynamic systems and techniques for identifying these non-linear systems using a novel recurrent neural network have been discussed in [18]. In this thesis a feed-forward neural network using the back propagation algorithm has been developed in LabVIEW for modeling a control system. The control system modeled is a dynamic nonlinear plant that was used in [19].

Control system modeling could be called as 'function approximation' or 'regression' in which a model of a function or a system is developed to learn that particular function or the system I/O. The inputs-output pairs of a system or plant are used to develop models. Since it is known that ANN's have the ability to learn a function by training, ANN's could be used for modeling plants. The input to the plant then become the input feature vector and the corresponding output then become the target vector. A control system could be either SISO (Single Input Single Output) or MIMO (Multiple Input Multiple Output) or some combination. To model a 'dynamic plant', one could also use the past outputs and the past inputs of the plant as the input feature vector for the ANN [19]. For example, the current output of the plant, the past two outputs, current input and the past two inputs could be used as the features of the input vector for the neural net, in which case the input feature vector would be 6 dimensional.

Statistically, the difference between classification and regression is the *posterior* probability. The neural net equivalent of the posterior probability is the output response of the output neurons. The difference between the statistics of classification and the statistics of regression is the *posterior* probability values that are obtained within the Bayesian framework.

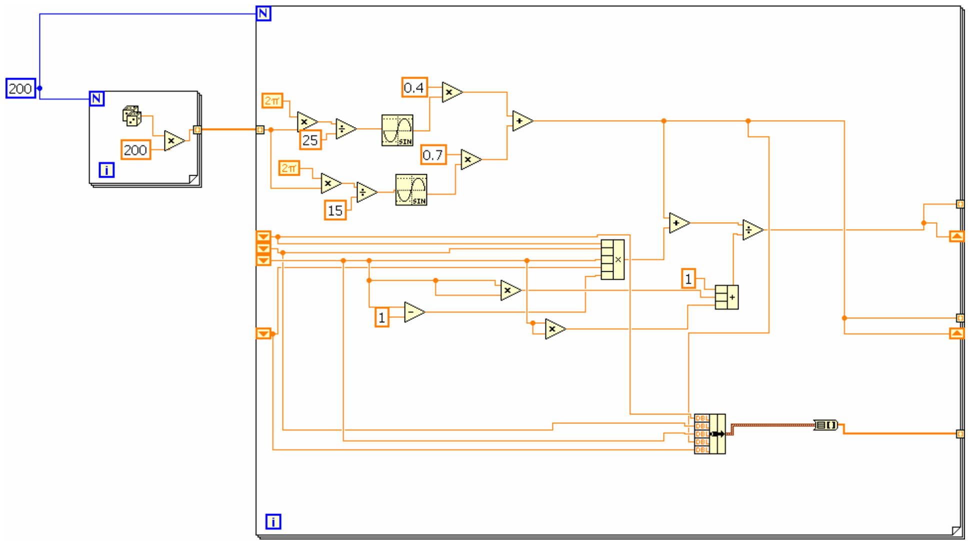


Fig 4.5 Block diagram for generating the outputs of the non-linear control system given in Eq. 5.1

CHAPTER 5

RESULTS AND DISCUSSIONS

5.1 Pattern Classification

5.1.1 Classifying IRIS Data Using a Feed-Forward ANN

The classification of the IRIS flower data set has been implemented with the standard feed-forward neural net that applies the back propagation algorithm, which had been constructed with LabVIEW. This neural net consists of a number of LabVIEW VI's. Since the IRIS data has 4 features and consists of 3 pattern classes, 4 input nodes and 3 output nodes are required. Initially, assume that there are 25 hidden nodes (arbitrarily chosen), followed by 3 output nodes, one for each class.

Essentially, the idea is to train the ANN with some data points and then test the ANN with the remaining points. This is a "cross-validation" technique to test the ANN. The back propagation algorithm has been explained earlier in Section 3.1.2.

While building the neural net in LabVIEW (main VI), six Sub-VI's were constructed, each of them performing a different function. These functions are listed as follows:

1. Generating hidden neurons
2. Generating output neurons
3. Generating deltas (error gradients) for output units
4. Updating weights for output units
5. Generating deltas (error gradients) for hidden units
6. Updating weights for hidden unit

Since, initially, 25 hidden neurons were assumed, there will be a 25×4 weight matrix from the input layer to the hidden layer. Also, there will be a 3×25 weight matrix from hidden layer to the output layer. Initially, randomly generated weights, with values in the range from 0 to 0.5 were used.

During successive iterations to modify the weights; the neural net is trained using the back propagation algorithm, these weights tend to converge towards values, that will reduce the mean square error (MSE) between the target (desired) response and the output layer response of the neural net. If the value of the MSE goes below a certain pre-set value, then the training is stopped. Then, the ANN can be tested using the cross-validation data points. When this stopping criterion is used, sometimes the ANN will stop its training after a relatively small number of iterations. In this case, the result of cross-validation is a large misclassification error. This is the result of finding a "local minimum". While applying the stopping criteria, the overall goal was to seek the global minimum of the error "curve", which would correspond to the best possible result (global optimum). To avoid local minima, different parameters such as the starting values of weights and the learning rates for updating the weights can be changed. Then, the ANN is trained in order to seek a lower error rate.

The initial weights are selected using a random number generator which generates random numbers in the range (0, 1); however, these weights can be scaled according to other requirements and the resulting performance. The results in Section 5.1.1 were obtained by scaling the weights by a value of 20. Also the weights were scaled with different values, and only the best results obtained are presented here.

The data set is assumed to be defined by a Gaussian mixture model; thus, the data points, as a set, belonging to difference classes are randomly mixed. Suppose that the first 100 data points are used for training; if, say, 400 sweeps are made with these 100 points, then 400 x 100 iterations will be performed. After 400 sweeps through these 100 points, one can test the network using the remaining 50 data points. Results will be presented here corresponding to two different cross-validation criteria.

Normalization of the input data is performed in a basic way. The data vectors (consisting of points belonging to all classes, each point consisting of 4 features) are first subtracted from the mean value and the result is then divided by the standard deviation to yield normalized values between 0 and 1. Neural networks function in an improved manner if input values are in this range.

The log sigmoid (non-linear) function is used as the activation function for all the neurons in the neural network used for pattern classification application. The sigmoid function has a very powerful compression effect as it compresses a wide range of values onto the range (0, 1).

Results in Section 5.1.1 consist of training error curves and observed and desired classification spaces. The error curves shown were obtained during the training period of the ANN. The observed classification spaces (for all three classes) and the confusion matrices were obtained while testing the ANN using *cross-validation* techniques.

While plotting the classification space as shown in Fig. 5.2, the dimensions of the data points had to be reduced from 4 to 2. There are 2 ways to do this. The first method would be to use the principal component analysis technique and obtain 2 *new* dimensions (as explained in Section 3.3) that have the maximum variance across the data. The second technique used, is to find the covariance matrix of the 4 input dimensions (features). The diagonal (looking from the top left to the bottom right) of the 4×4 covariance matrix for the IRIS data set gives the Eigen values (variances of the 4 dimensions). The 2 maximum Eigen values are chosen. The 2 dimensions corresponding to the two maximum Eigen-values have the maximum variance across the data and are chosen for plotting the data points in 2 dimensions. Using this technique, it was found that for the IRIS data set, out of the 4 possible dimensions, the first and the third dimensions have better variance. Also this technique *does not* transform the dimensions into *new* dimensions. Rather, the data points are plotted using the old dimensions, only the 2 most significant dimensions are chosen out of the four. The following *observed classification spaces* have been plotted across the first and the third dimensions. Though all the data points may not be visible across these 2 dimensions, most of the data points are visible as shown in Fig 5.2. The results follow.

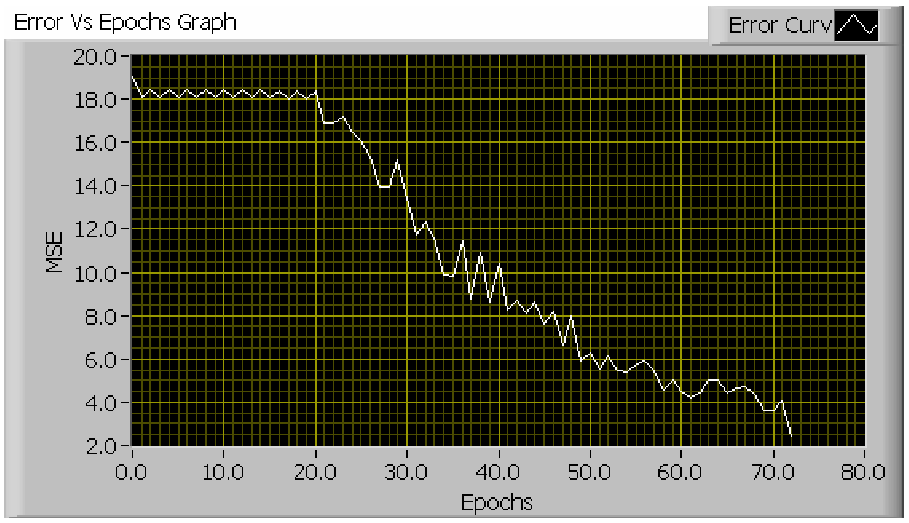


Fig. 5.1 Training Error versus Epochs curve

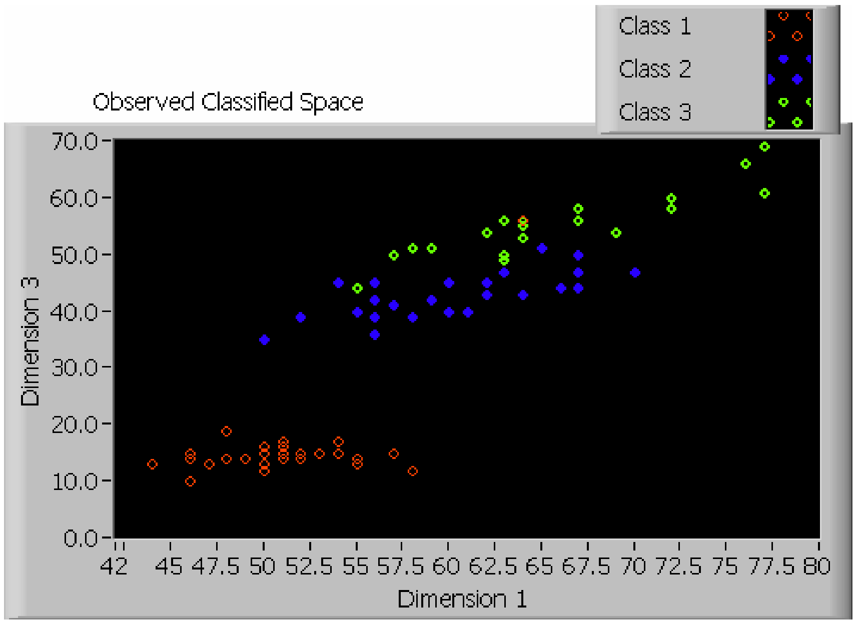


Fig. 5.2 Observed Classification Space

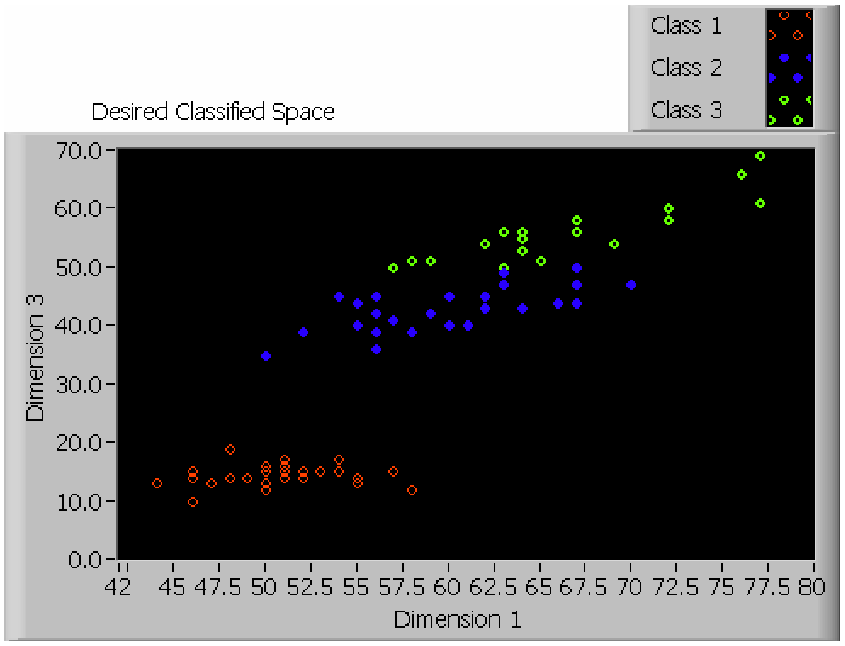


Fig. 5.3 Desired Classification Space

Figures 5.1 and 5.2 correspond to the training error curve and the observed classification space respectively after training the FF-NN using data points 0,2,4..148 and testing the FF-NN using data points 1,3,5…149. Training was stopped when the error level reaches a pre-set value of 3 (chosen by a trial and error method). The learning Rate $\eta = 0.15$.

Fig 5.3 is a plot of the desired classification space that one would observe, when known points corresponding to 1,3,5..149 are plotted in a 2-dimensional classification space. Refer to Table 5.1 for verifying the corresponding confusion matrix.

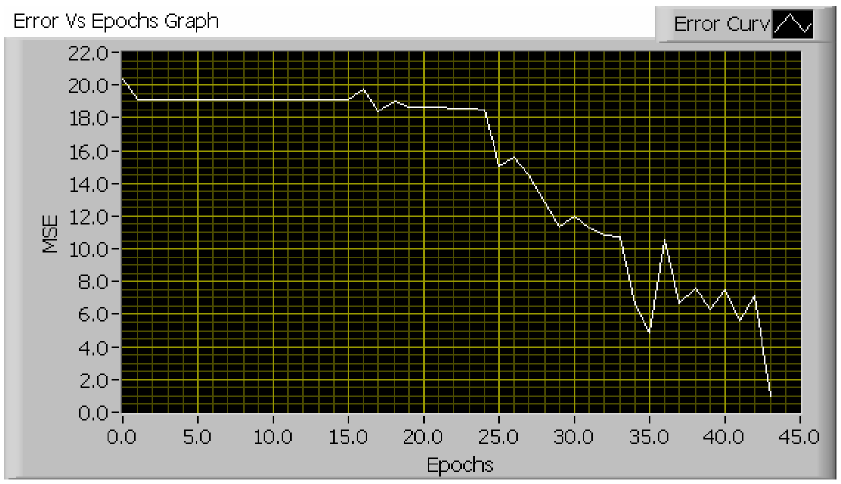


Fig. 5.4 Training Error versus Epochs curve

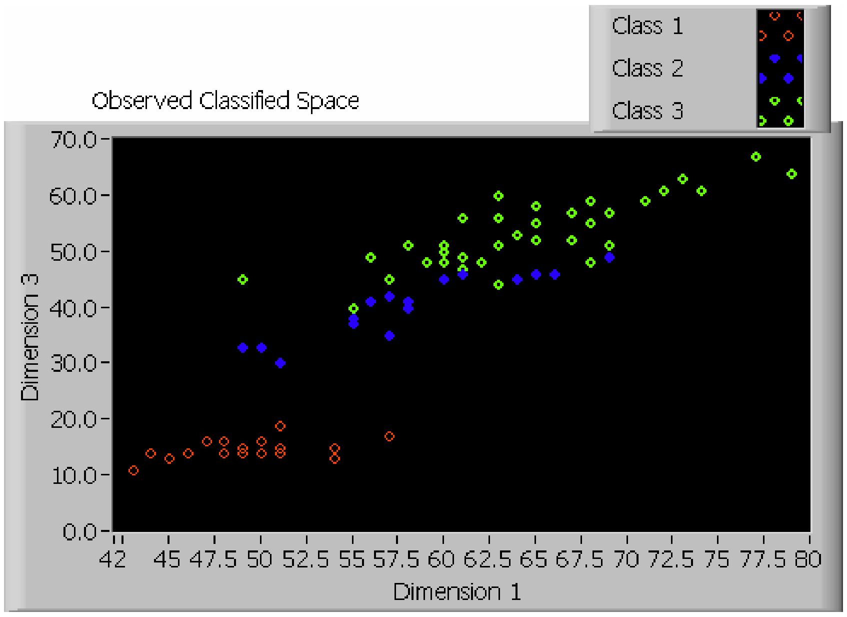


Fig. 5.5 Observed Classification Space

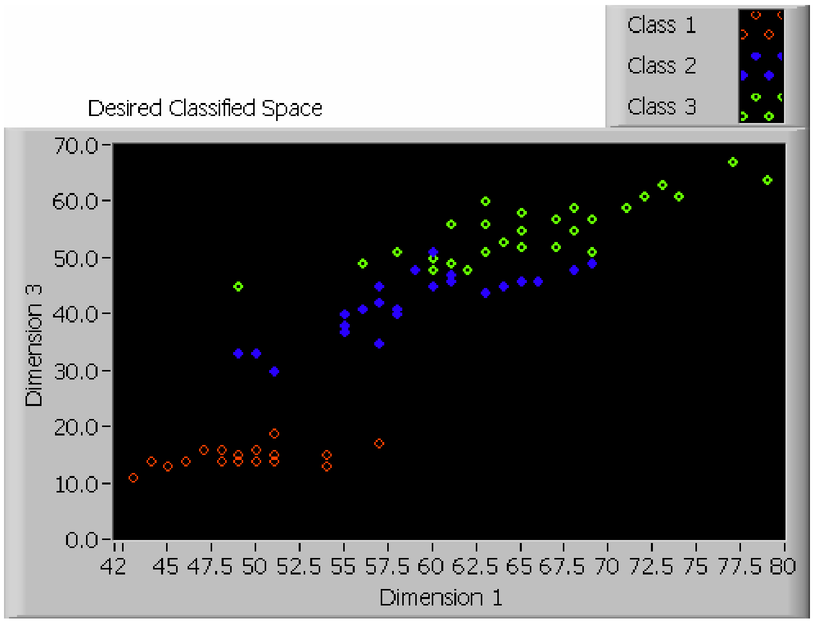


Fig. 5.6 Desired Classification Space

Figures 5.4 and 5.5 correspond to the training error curve and the observed classification space respectively after training the FF-NN using data points 1,3,5…149 and testing the FF-NN using data points 0,2,4..148. Training was stopped when the error level reaches a pre-set value = 3 (MSE value chosen by trial and error method). The learning rate used is 0.15.

Fig 5.6 is a plot of the desired classification space that one would observe when points corresponding to 0, 2, 4...148 are plotted in classification space. Refer to Table 5.1 for verifying the corresponding confusion matrix.

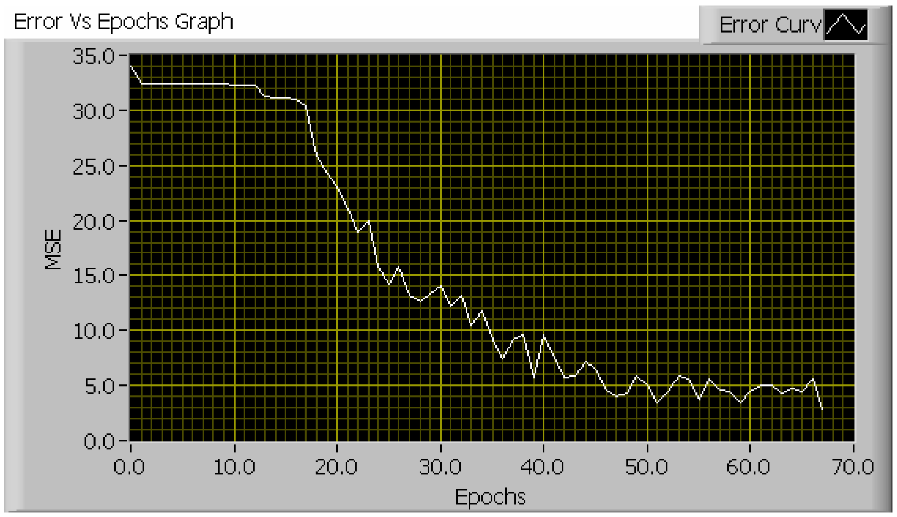


Fig. 5.7 Training Error versus Epochs curve

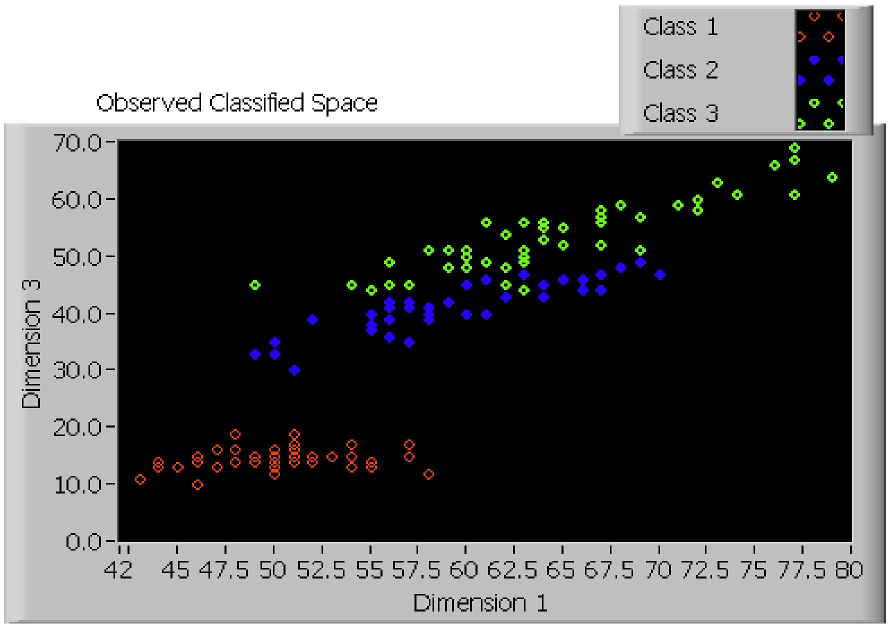


Fig. 5.8 Observed Classification Space

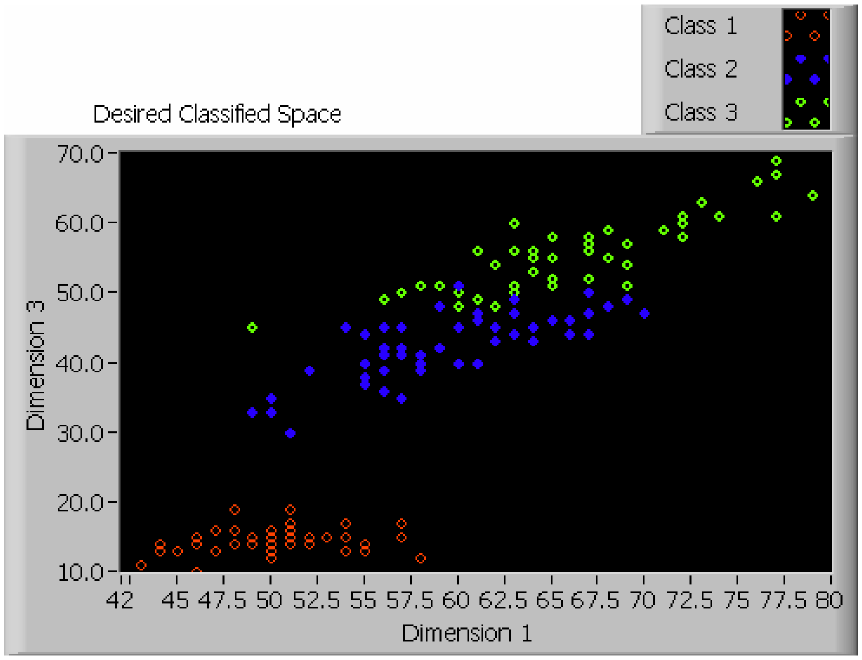


Fig. 5.9 Desired Classification Space

Figures 5.7 and 5.8 correspond to the training error curve and the observed classification space by training the FF-NN using all 150 data points and testing the FF-NN using all 150 data points. Training was stopped once the error level reaches a pre-set value = 3 (chosen by trial and error method). The learning rate used is 0.15.

Fig 5.9 is a plot of the desired classification space that one would observe when all the 150 data points are plotted in a 2-dimensional classification space. Refer to Table 5.1 for the corresponding confusion matrix. Learning rate of 0.15 was chosen after many values were tried. With a learning rate of 0.15 good confusion matrices were obtained. When a error pre-set level of 3 (MSE value) or less was used then better classification was achieved. The Epochs refers to the number of sweeps through the data used before the output layer Mean Square Error reaches the error pre-set level. The error was generated by comparing the target response and actual output layer response.

Table 5.1 Confusion Matrix table

LEARNING RATE	EPOCHS REACHED	STOPPING CRITERION	TRAINING DATA	TEST DATA	CONFUSION MATRIX		
0.15	72	3	0,2,4..148	1,3,5..149	30	0	0
					0	23	1
					0	2	19
0.15	43	3	1,3,5…149	0,2,4…148	21	0	0
					0	17	0
					0	8	29
0.15	67	3	0,1,2,3….149	0,1,2,3…149	50	0	0
					0	37	0
					0	13	50
0.15	63	3	0,1,2,…74	75,76,…149	13	0	0
					10	9	0
					0	21	22
0.15	65	3	75,76…149	0,1,2…74	27	0	0
					0	14	1
					0	6	27
0.15	51	3	0,1,2…99	100,101..149	15	0	0
					1	8	0
					0	12	14
0.15	82	3	0,1,2..49	50,51,..149	37	0	0
					0	31	0
					0	4	28

5.1.2 Classifying the IRIS Data Using the Feed-Forward ANN by Reducing the Input Feature Dimensions from 4 to 2

Section 5.1.1 discussed results obtained using all the 4 features of the IRIS input data vectors. In this section, out of the 4 features, only the two most significant features were selected using the *Principal Component Analysis Technique* (PCA). The Karhunen-Loeve Transform (KL Transform) has been applied to extract the first two principal components (which reduces the number of dimensions from 4 to 2). The feed-forward neural net is trained to classify IRIS data set using only those 2 significant features of the input data vectors. Results in this section were obtained by specifying the number of epochs the neural net has to train. That is, the stopping criterion is based on specifying the number of epochs for the neural net to train. At the end of training period, the net is tested using “cross-validation” techniques.

The feed-forward ANN used here has 2 input features, 4 hidden layer neurons and 3 output nodes. The weights were initialized using random number generators whose values lie in the range (0, 1). The weights were scaled suitably based on improved results. The observed classification spaces shown in Figures 5.11, 5.13 and 5.15 are plots using the two new dimensions obtained using the KL transform. The results follow.

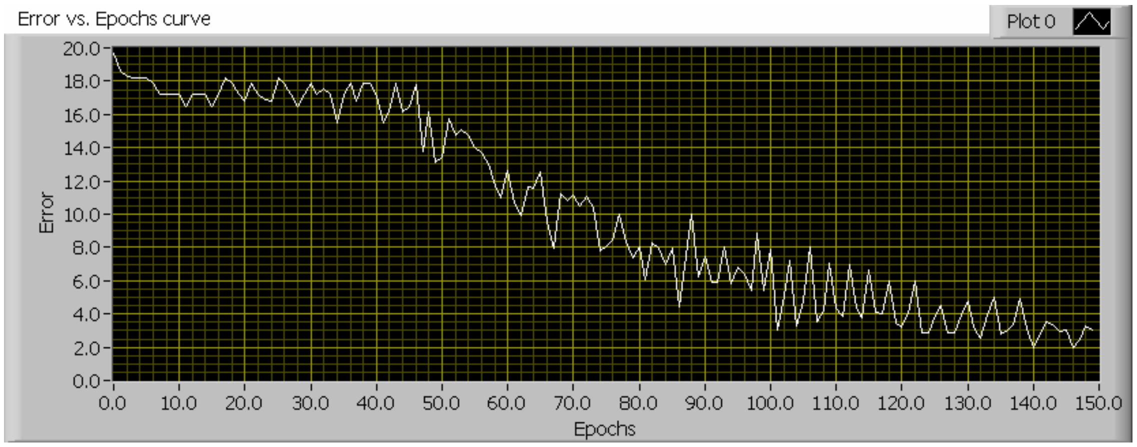


Figure 5.10

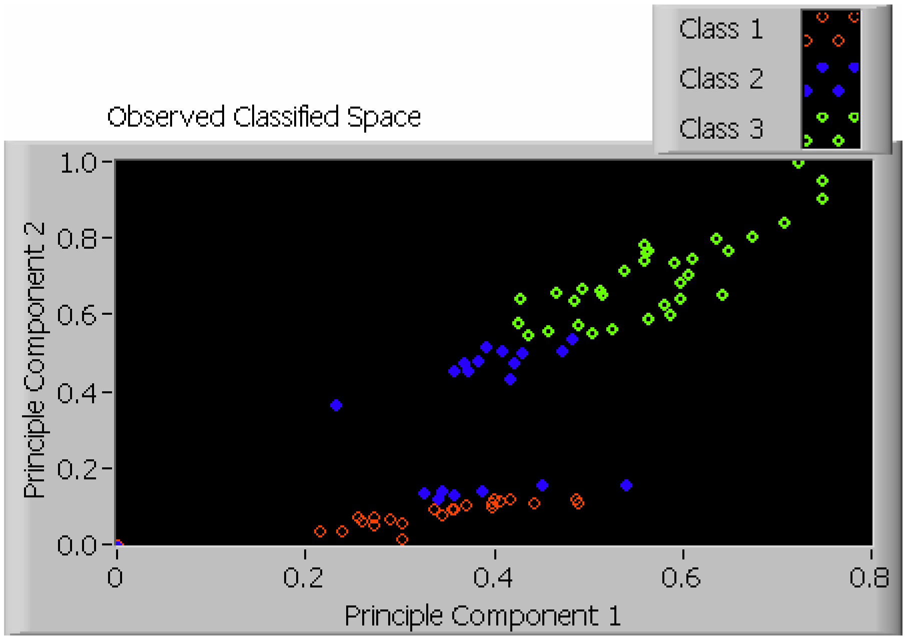


Figure 5.11 Observed Classification space

Figures 5.10 & 5.11 were obtained by training the ANN using data points 0,2,4…148 & testing the ANN using points 1,3,5…149. The ANN was trained for 150 epochs. Since the optimum number of epochs is known by trial and error method, different “epoch numbers” were tried. 100, 150, 200, 300 were the epoch values for which the ANN was trained for and it was found that the error stabilized after 140 epochs.

Table 5.2 shows the number of epochs versus the minimum MSE value reached

Table 5.2

Epochs	Min MSE reached
100	5
150	2.2
200	2
300	1.8

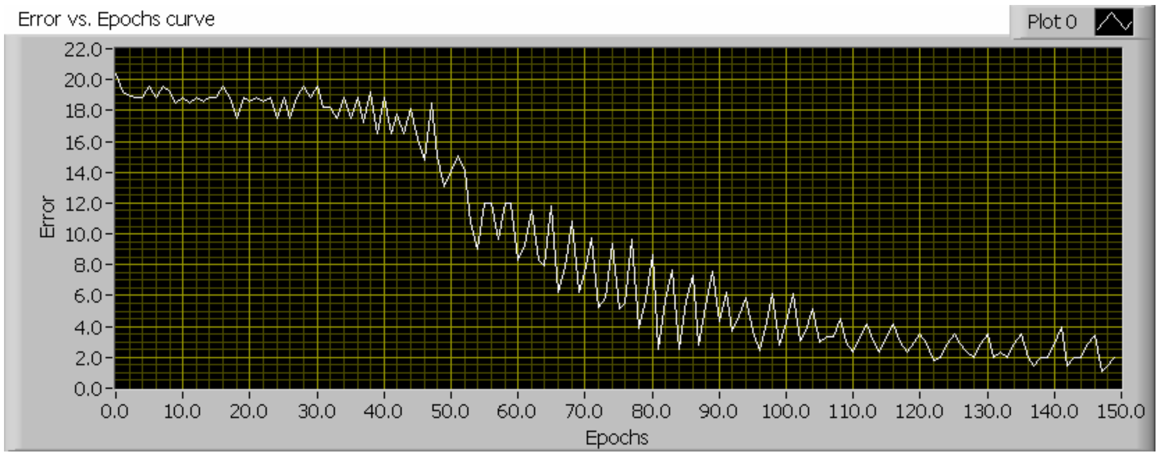


Figure 5.12 Training error versus epochs curve

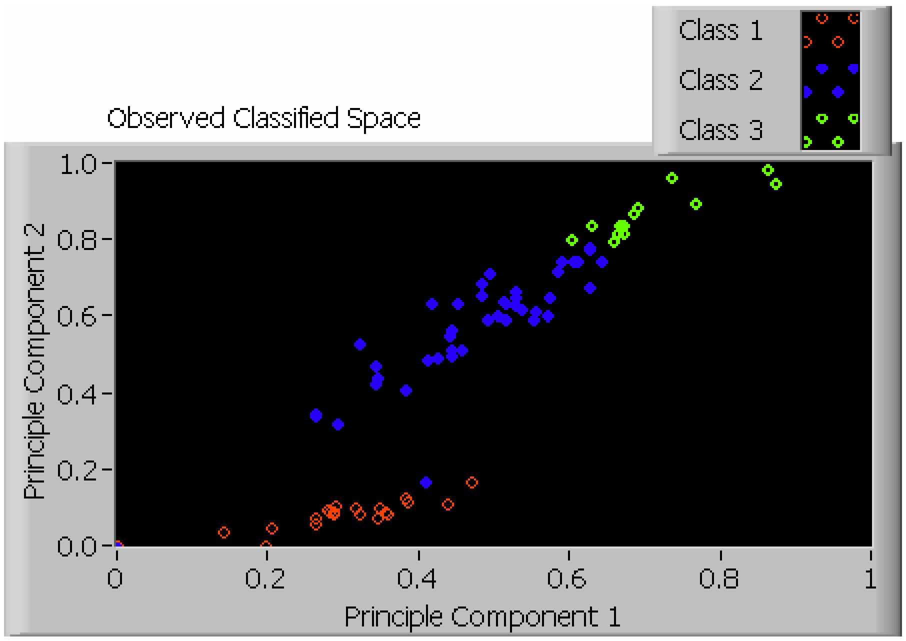


Figure 5.13 Observed Classification Space

Figures 5.12 & 5.13 were obtained by training the ANN using data points 1,3,5…149 & testing the ANN using points 0,2,4…148. The ANN was trained for 150 epochs. Since the optimum number of epochs is known by trial and error method, different "epoch numbers" were tried. 100, 150, 300, 400 were the epoch values for which the ANN was trained for and it was found that the error stabilized after 120 epochs.

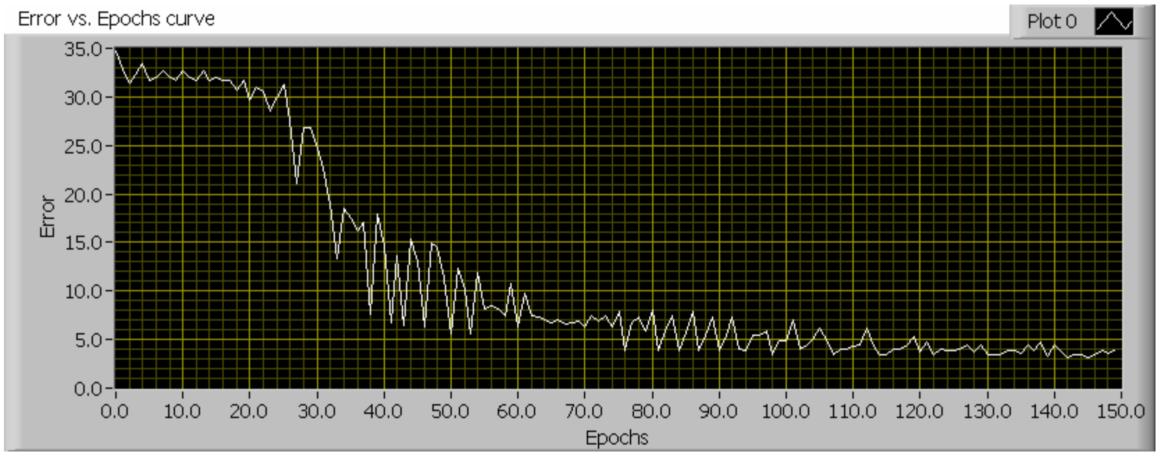


Figure 5.14 Training error versus epochs curve

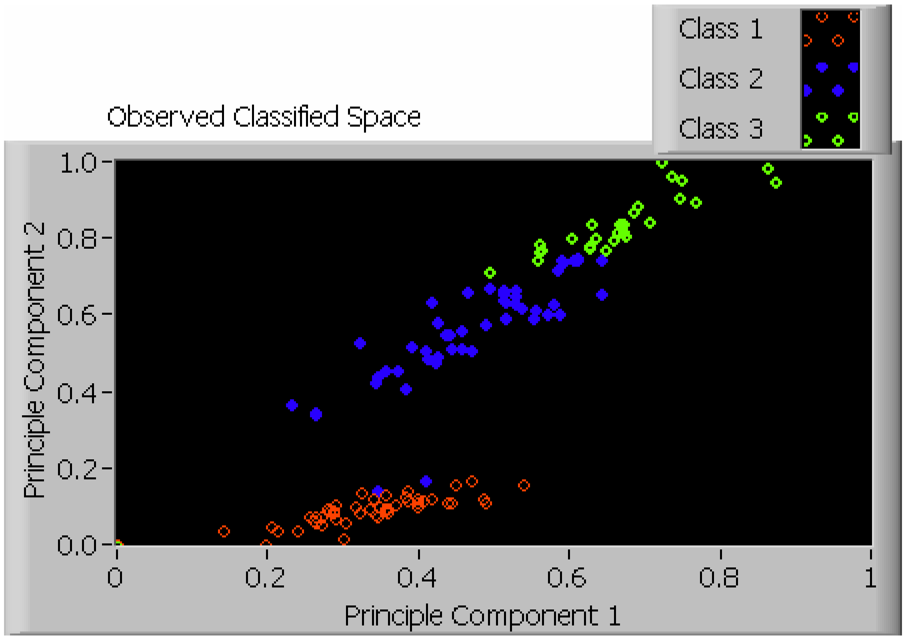


Figure 5.15 Observed Classification space

Figures 5.14 & 5.15 were obtained by training the ANN using all the 150 data points & testing the ANN using points all the 150 data points. The ANN was trained for 150 epochs. It was found that the error stabilized after 110 epochs.

Table 5.3 gives the confusion matrices for various "cross-validation" techniques. The learning rate used; the number of epochs to train the ANN; the data points used for training & testing; and the corresponding confusion matrices are shown in table 5.3.

Table 5.3 Confusion Matrices obtained using only 2 principal features

LEARNING RATE	EPOCHS	TRAINING DATA	TEST DATA	CONFUSION MATRIX		
0.2	150	0,2,4…148	1,3,5..149	22	0	0
				7	12	0
				0	13	21
0.2	150	1,3,5…149	0,2,4…148	20	0	0
				1	25	16
				0	0	13
0.2	150	0,1,2…149	0,1,2…149	48	0	0
				2	50	23
				0	0	27
0.2	200	0,1,2…74	75,76…149	21	0	0
				2	25	1
				0	5	21
0.2	200	75,76…149	0,1,2…74	24	0	0
				3	20	8
				0	0	20
0.2	200	0,1,2..99	100,101…149	16	0	0
				0	20	5
				0	0	9
0.2	150	0,1,2..49	50,51…149	37	0	0
				0	30	3
				0	5	25

5.1.3 Classifying IRIS Data Using the Radial Basis Function (RBF) ANN

In this section, the RBF ANN has been used to classify the IRIS data. Five hidden neurons have been assumed. The exponential Gaussian function has been used as the non-linear activation function in each of those hidden neurons. The K-Means clustering algorithm has been used to update the centers of the 5 hidden neurons. The output layer weights are updated using the LMS algorithm (explained in Section 3.1.1). The initial centers and weights are generated using random number generators whose values lie in the range (0, 1). In this section the observed classification spaces are presented for various cross-validation techniques. Also the confusion matrices obtained for various results have been presented in table 5.4. The weights and learning rates are obtained using trial and error methods. Various values of weights and learning rates were tried and only the best results are presented.

Figure 5.16 shows the observed classification space obtained after training the ANN using data points 0,2,4..148 and testing the ANN using data points 1, 3…149. Refer to Figure 5.3 for the desired classification matrix.

Figure 5.17 shows the observed classification spaces using data points 1, 3…149 for training and using data points 0, 2…148 for testing. Refer to Figure 5.6 for the desired classification matrix.

Figure 5.18 shows the observed classification space using all the 150 data points of the IRIS flower data. Refer to Figure 5.9 for the desired classification matrix.
For the corresponding confusion matrices, refer to Table 5.4.

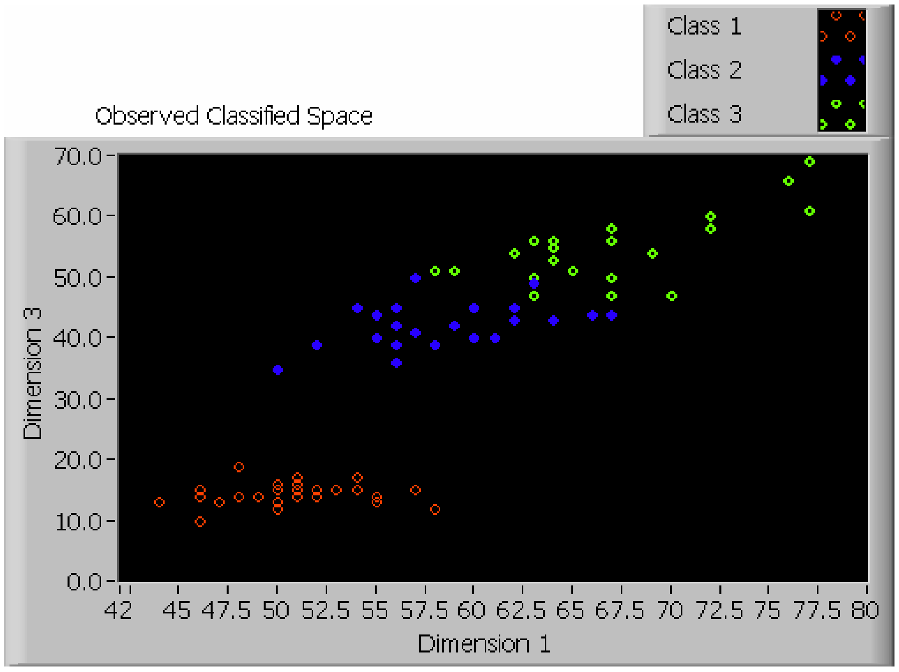


Figure 5.16 Observed Classification Space

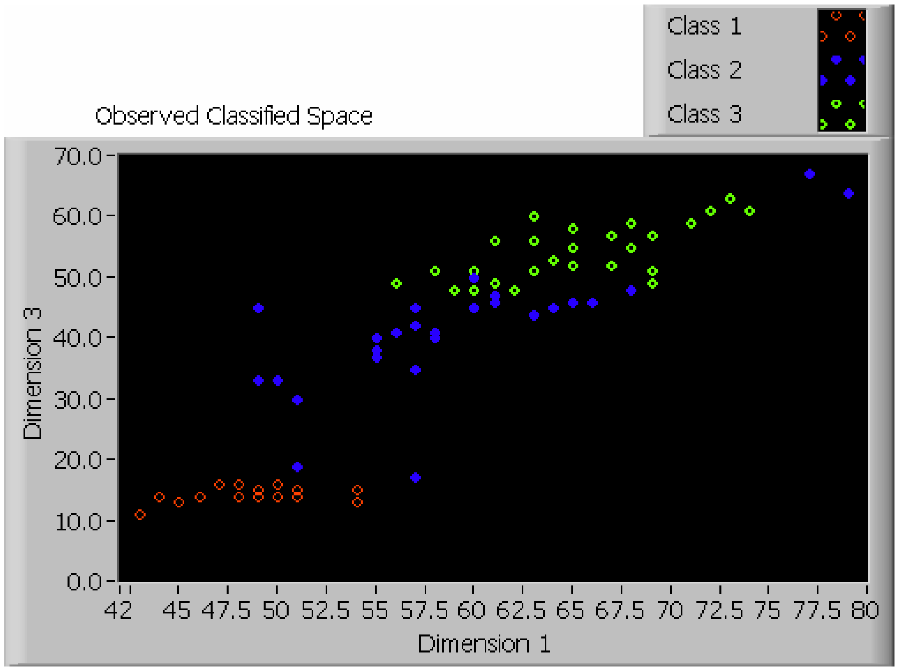


Figure 5.17 Observed Classification Space

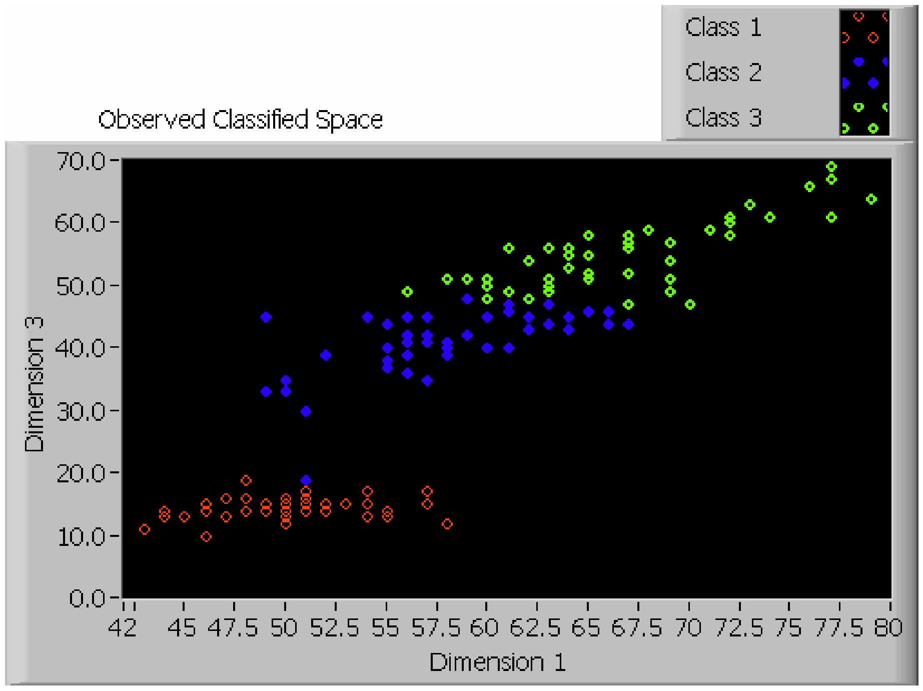


Figure 5.18 Observed Classification Space

Table 5.4 Confusion Matrices obtained using RBF ANN

Training data	Testing data	Epochs-K-Means	Epochs-LMS	Learning Rate	Confusion Matrix		
0,2,4..148	1,3,5…149	8000	8000	0.1	29	0	0
					0	21	4
					0	1	20
1,3,5..149	0,2,4..148	8000	8000	0.1	19	2	0
					0	22	3
					0	5	24
0,1,2…149	0,1,2…149	8000	8000	0.1	49	1	0
					0	43	7
					0	1	49
0,1,2..74	75,76…149	12000	12000	0.1	23	0	0
					0	24	6
					0	0	22
75,76…149	0,1,2…74	12000	12000	0.1	25	0	2
					0	19	1
					0	4	24
0,1,2..99	100….149	12000	12000	0.1	16	0	0
					0	15	5
					0	0	14
0,1,2..49	50,51…149	12000	12000	0.1	36	1	0
					0	26	9
					0	1	27

5.1.4 Classifying IRIS Data Using Fuzzy K-Means Clustering Algorithm

Results obtained by classifying the IRIS flower data using the Fuzzy K-Means algorithm are shown in the following pages. The algorithm has been explained in detail in the “Learning Algorithms” chapter. The advantage of a fuzzy set approach is that, initially, one would assign a membership value to each data point for each of the three possible clusters or classes. Thus, the first step would be to initialize the means for the three classes; two methods for accomplishing this have been addressed.

Method 1: One could randomly pick three values from the input data set and assume those three values as the initial centers (means) for the three classes.

Method 2: One could randomly generate three vectors, each 4-dimensional, and assume these vectors as the initial centers (means) for the three classes or clusters. Also, these three randomly generated centers could be scaled to obtain better results (which is done using a trial and error method).

In this work reported here, method 2 was followed to initialize the centers (means) for the classes. The stopping criterion based on reducing the error below a pre-set value has been applied. Error curves and the corresponding observed classification spaces have been presented here.

The results that are shown in this section used 0.25 as the pre-set error level. The training stops when the error reaches this level. Also, a degree of fuzziness (fuzzification factor) of 2 was assumed. For the desired classification spaces refer to Figures 5.3, 5.6 and 5.9.

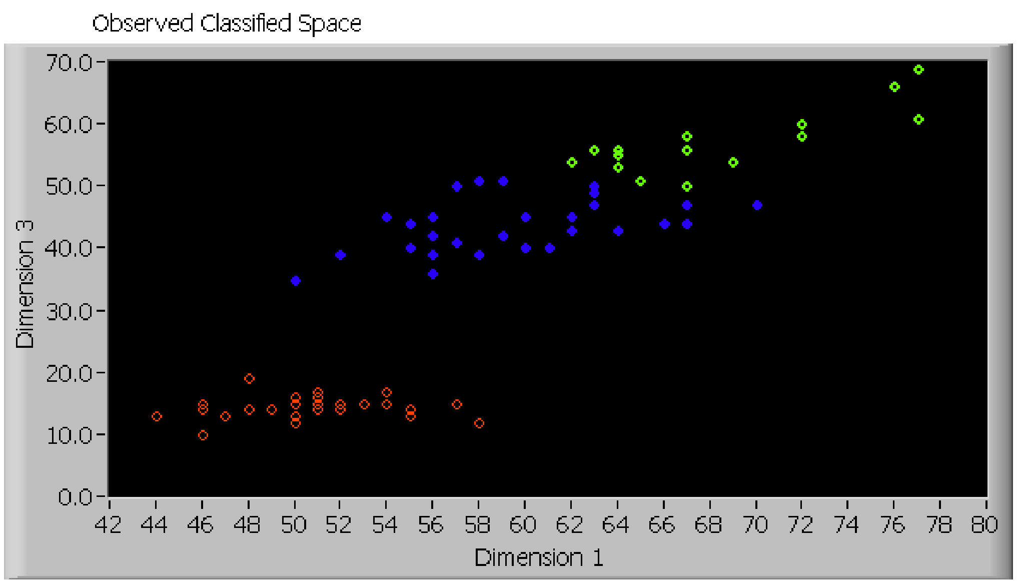


Fig. 5.19 Observed classification space obtained using data points 0, 2, 4…148 for training and data points 1,3,5…149 for testing. Refer to Figure 5.3 for the desired classification space.

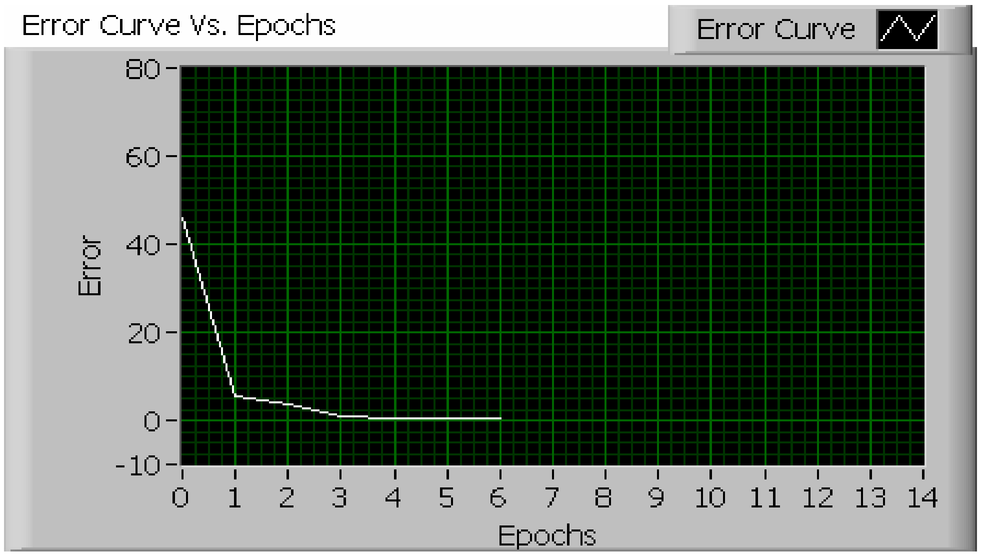


Fig. 5.20 Training error (MSE) curve obtained using data points 0, 2, 4…148 for training and data points 1,3,5...149 for testing.

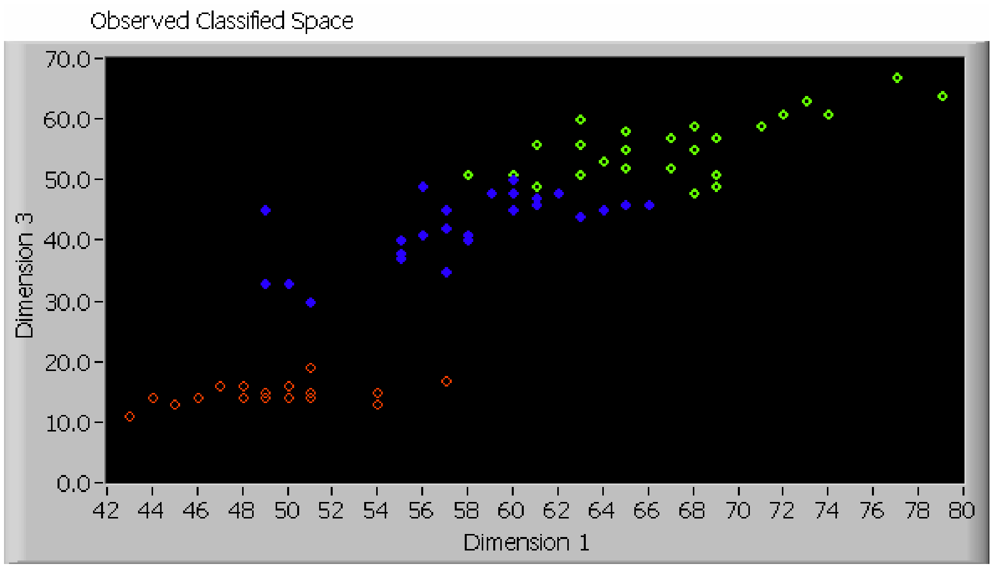


Fig. 5.21 Observed classification space obtained using data points corresponding to data points 1,3,5...149 for training and data points corresponding to 0, 2, 4…148 for testing. Refer to Figure 5.6 for the desired classification space.

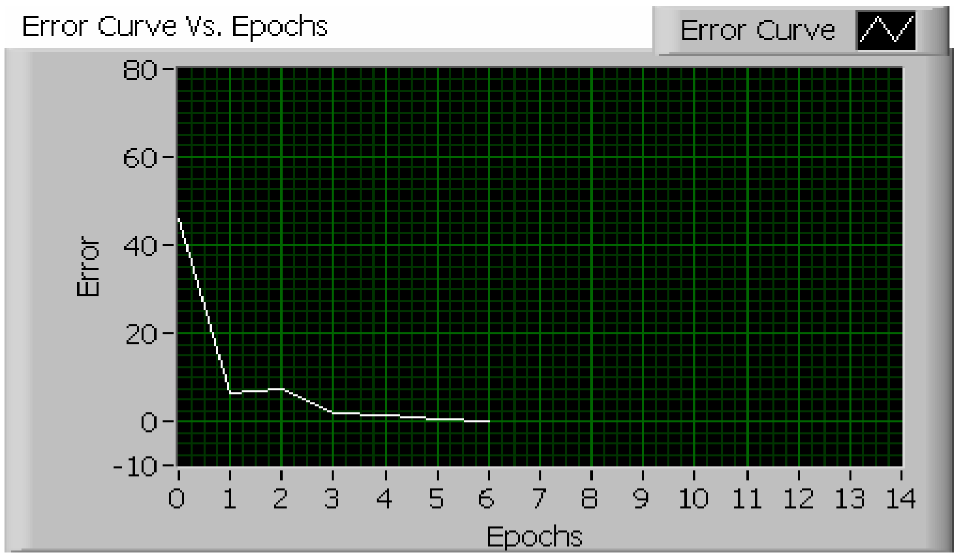


Fig. 5.22 Training error curve obtained using data points corresponding to 1,3,5…149 for training and data points corresponding to 0, 2, 4…148 for testing.

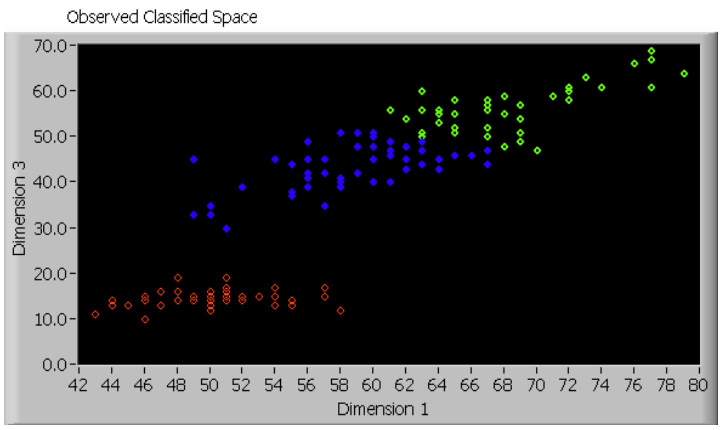


Fig.5.23 Observed classification space obtained using all the 150 data points for training and testing. Refer to Figure 5.9 for the desired classification space.

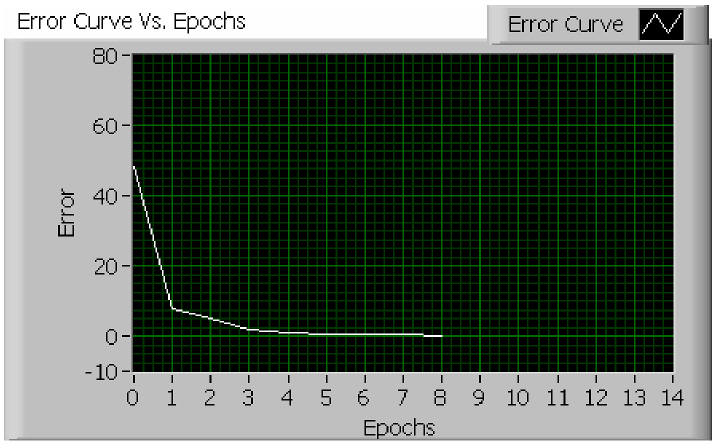


Fig. 5.24 Training error curve obtained using all the 150 data points for training & testing.

While obtaining the prototype centers for the 3 classes, it was found that indexes of the three centers keep changing during each run of the program. Since the index changes (between 0, 1 & 2) every time the program executes, the confusion matrix could not be constructed (as it was constructed in sections 5.1.1, 5.1.2 & 5.1.3). Hence the confusion matrix table has not been shown in this section. The 3×3 confusion matrix as shown in the previous sections was constructed by comparing the desired index & the observed index obtained using each input data point. If the desired & observed indexes match for a particular input, then the corresponding position in the diagonal is incremented by one. This procedure is utilized while testing the neural net.

5.1.5 Classifying IRIS Data Using Self-Organizing Feature Maps and K-Means Clustering Algorithms (Hybrid Algorithm)

Self Organizing feature maps (SOM) have been used to classify the IRIS flower data. The basic idea here is to implement a hybrid algorithm where initially SOM is used, followed by the K-Means clustering algorithm. While implementing, SOM algorithm is initially used to get 16 updated centers. Then from these 16 centers, 3 updated centers corresponding to each of the three classes are obtained using the K-Means clustering algorithm. Before mapping, the data points have to be normalized to values in the range (0, 1). The *Min-Max* normalization technique has been applied here which is given by,

$$V' = (V - min)\ \ (new\_max - new\_min) / (max - min) + new\_min$$

where *V'* is the new component of data vector corresponding to the old data vector component given by *V*. The *max* & *min* are the maximum and minimum values corresponding to the old data points. *new_max* & *new_min* correspond to the maximum and minimum values for the new data points. The range of values for the new data points depends on the *new_max* & *new_min* values. Normalization is usually done to scale large input values down to the range between (0, 1). Scaling the data points helps even in the case of unsupervised clustering algorithms.

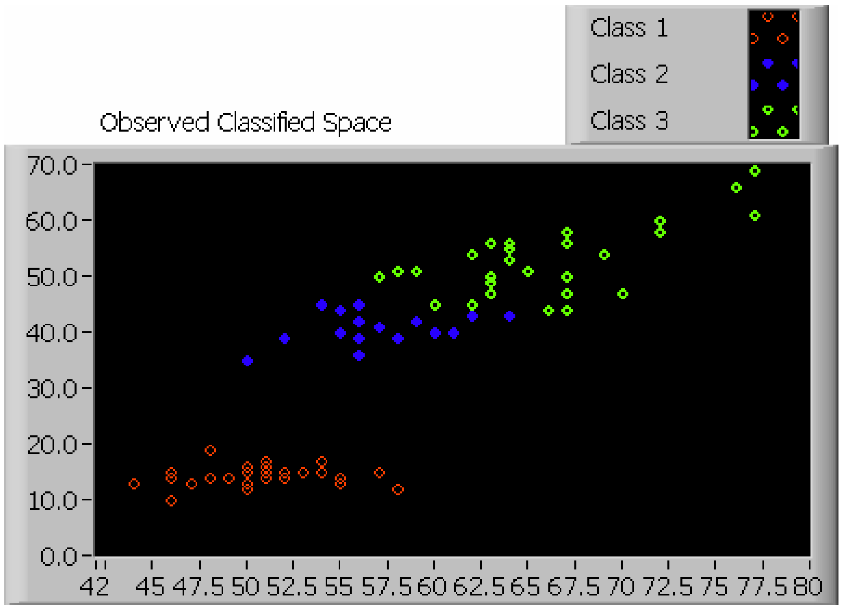


Fig. 5.25 Observed Classification space obtained using data points corresponding to 0,2,4..148 for training and data points corresponding to 1,3,5..149 for testing. Refer to Figure 5.3 for the desired classification space.

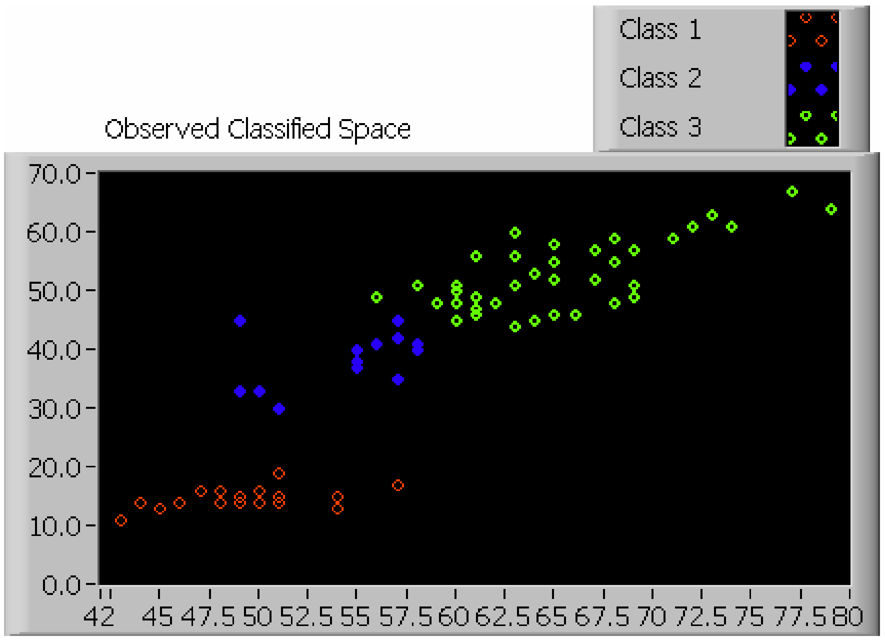


Fig. 5.26 Observed classification space obtained using data points corresponding to 1,3,5…149 for training and data points corresponding to 0,2,4..148 for testing. Refer to Figure 5.6 for the desired classification space.

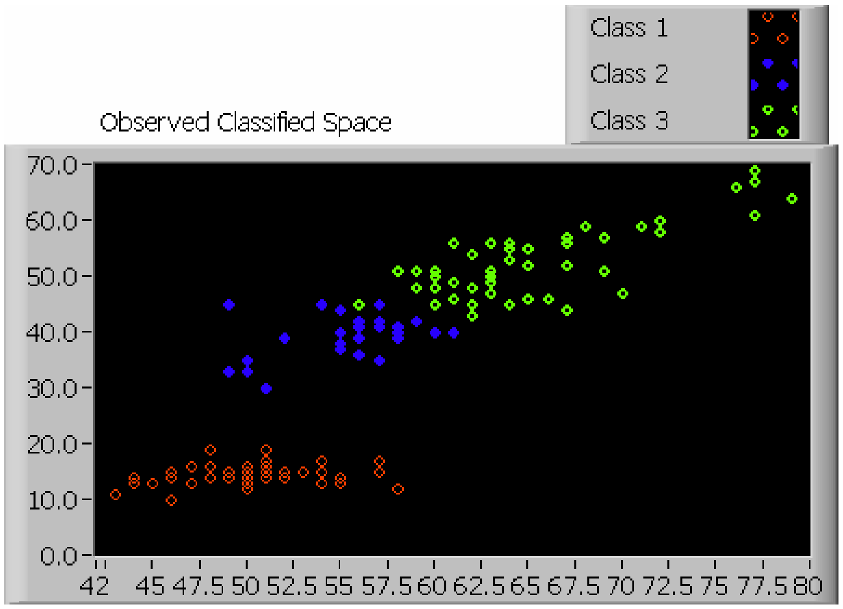


Fig. 5.27 Observed classification space obtained using all data points for training and testing. Refer to Figure 5.9 for desired classification space.

The confusion table has not been shown in this section because the indexes keep shifting during each run of the program (similar to Section 5.1.4).

5.2 Image Compression

This section addresses the application of a PCNN to image compression. The FF architecture of the PCNN is defined by:

Number of Input Neurons: 64

Number of Hidden Neurons: 16

Number of output Neurons: 64

Net Input to the neuron: Linear Basis input (dot product of input vector and the weight vector)

Activation Function: Non-linear Sigmoid function

Data Normalization: (0,255) gray scale pixel value is scaled to (0, 1) using *Min-Max* technique.

During the training phase, the 256×256 Lena image is scanned randomly to generate 8×8 blocks. Each randomly selected block is applied to the network (8×8 = 64) input units. The blocks are selected randomly a few thousand times, and each time the block is applied as input to the ANN and the neural net weights are updated each time using error back propagation algorithm, until the stopping error criteria reaches a particular pre-set value, which is chosen by trial and error method. Once the stopping criterion is reached, the training phase ends.

During the testing phase, the ANN is tested by sequentially scanning the image for 8*8 blocks of image from left to right and from top to bottom. Each scanned block is applied to the input neurons of the already trained neural net and the outputs are generated. These outputs are then stored in memory and the total image reproduced after scanning the entire image sequentially.

Fig. 5.28 Original Lena Image which has been chosen for compression using PCNN

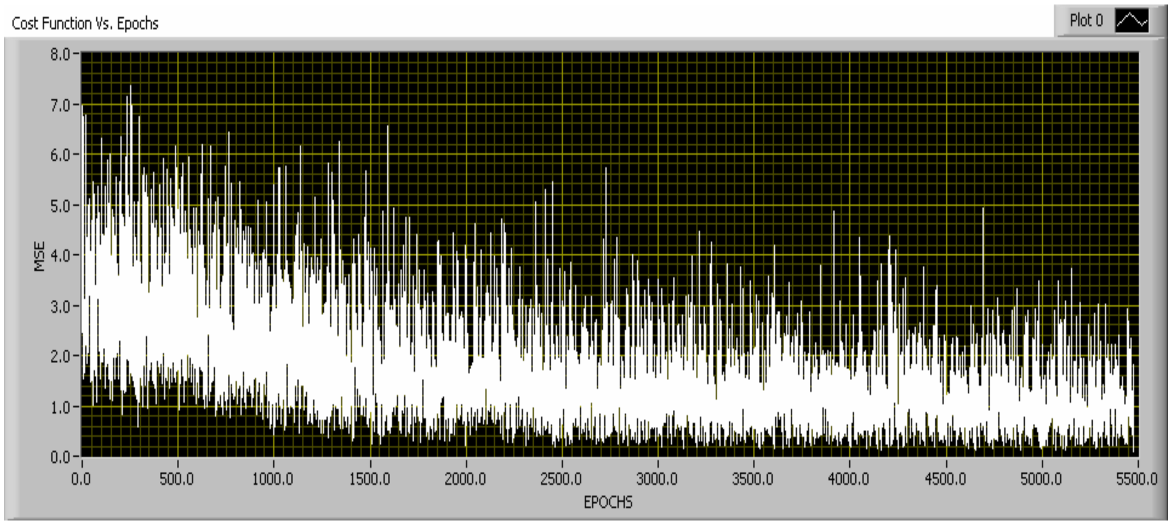


Fig. 5.29 Training error vs. Iterations graph after 5476 iterations for training using a stopping error pre-set error level < 0.1 and learning rate = 0.1 without quantization

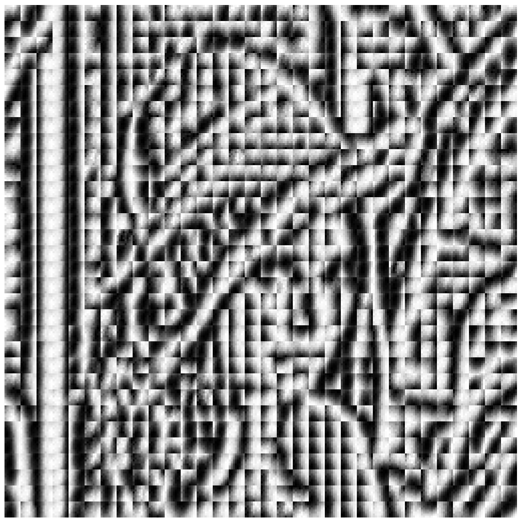


Fig 5.30 Reproduced Lena Image after 5476 iterations using a stopping error pre-set error level < 0.1 and a learning rate = 0.1 without quantization

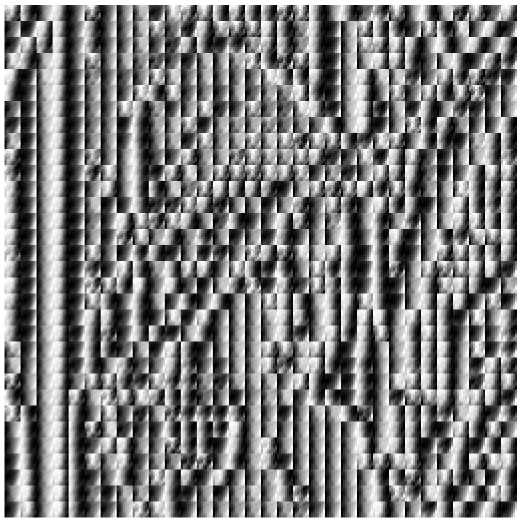


Fig. 5.31 Reproduced Lena Image after 2317 iterations using a stopping error pre-set level < 0.2 and a learning rate = 0.1 without quantization

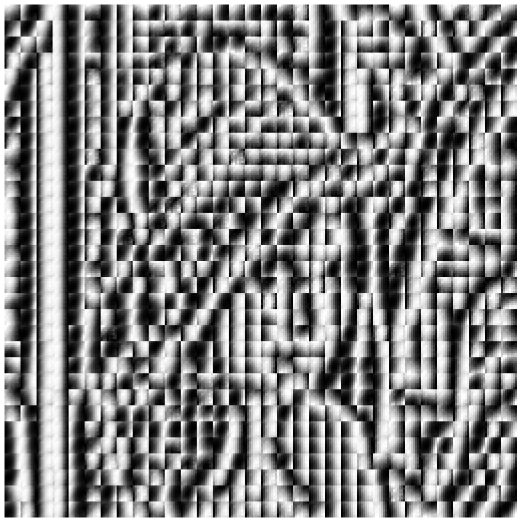


Fig. 5.32 Reproduced Lena Image after 3129 iterations using a stopping error pre-set level < 0.2 and a learning rate = 0.1 with 2-bit quantization

In the images obtained as shown in figures 5.30, 5.31 and 5.32 it is seen that the edges have been captured successfully. The edge transitions are visible clearly which means that the PCNN (with or without quantization) can extract the edges of an image. But the PCNN developed here does not have the ability to capture the texture of the image. If the texture & the edges of an image are captured, then the reproduced image would be as good as the original, at least, in terms of the human eye, if not in terms of the signal to noise ratio.

5.3 Control System Modeling

Control system to be modeled [19] is given by

$$y[k+1] = f(y[k],y[k-1],y[k-2],u[k],u[k-1])$$

$$= \frac{y[k]\ y[k-1]\ y[k-2]\ u[k-1]\ (y[k-2]-1)+u[k]}{1+y^2[k-2]+y^2[k-1]} \quad \ldots (5.1)$$

where $u[k] = 0.4 * \sin(\frac{2\pi k}{25}) + 0.7 * \sin(\frac{2\pi k}{15})$ is the input at discrete step k

A multi-layer feed-forward neural network with one hidden layer and one output layer has been considered as a model for the above non-linear control system. The output $y[k+1]$ depends on the three previous outputs and two previous inputs. Using "learning input" $u[k]$, it required a minimum of 300 iterations to train the ANN to generate the control system model.

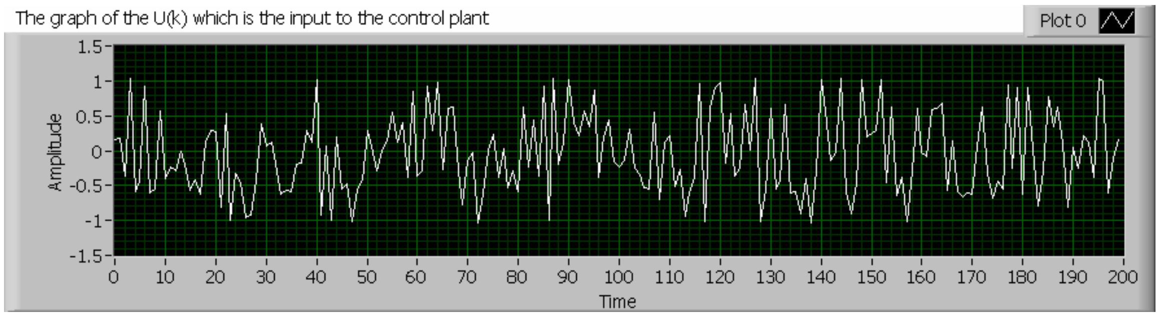


Fig 5.33 Graph for u(k), the “learning input”. 200 points are generated.

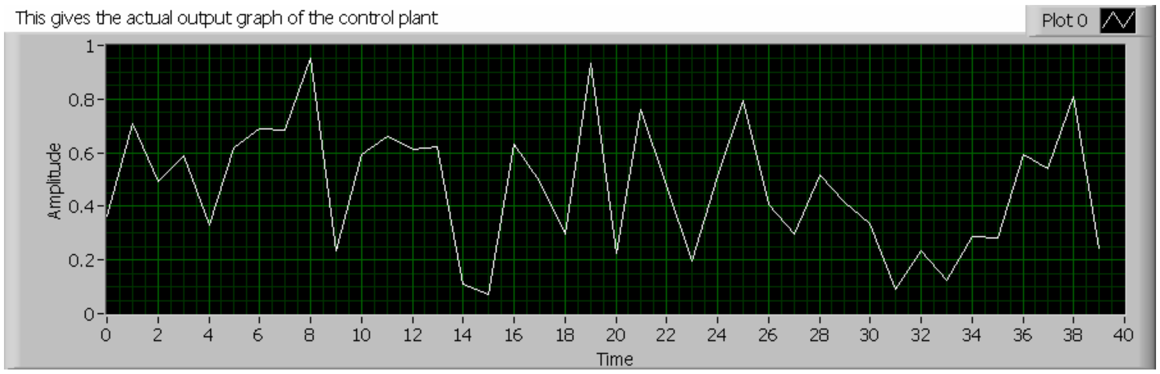


Fig 5.34 Actual output of the SISO control system given by Eq.5.1. The first 40 points are plotted here.

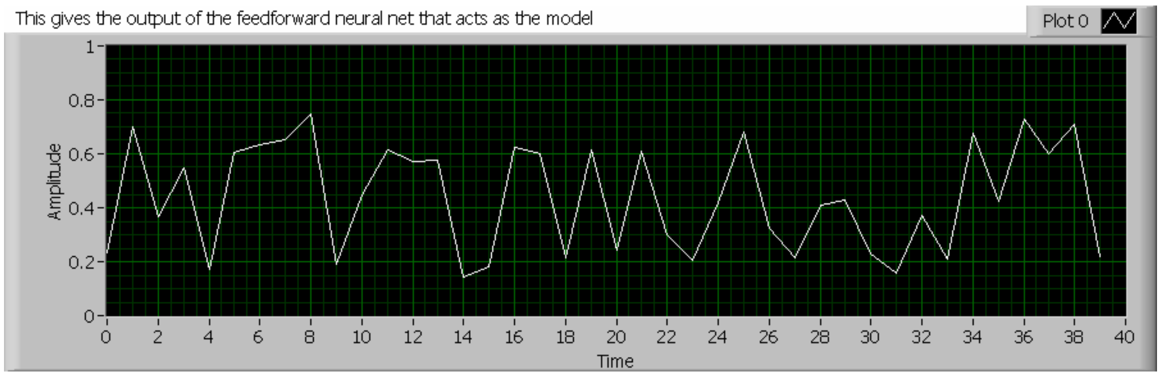


Fig 5.35 Output of the FF neural network with 5 input neurons, 20 hidden neurons and 1 output neuron which is trained for 500 epochs. The first 40 data points have been used for training and testing. Learning rate = 0.2

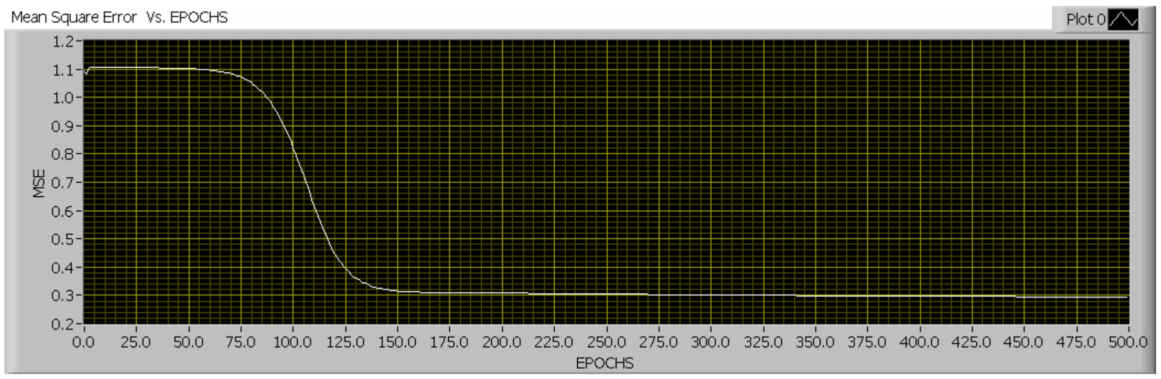


Fig 5.36 Error curve versus the number of epochs while training the neural net for 500 epochs

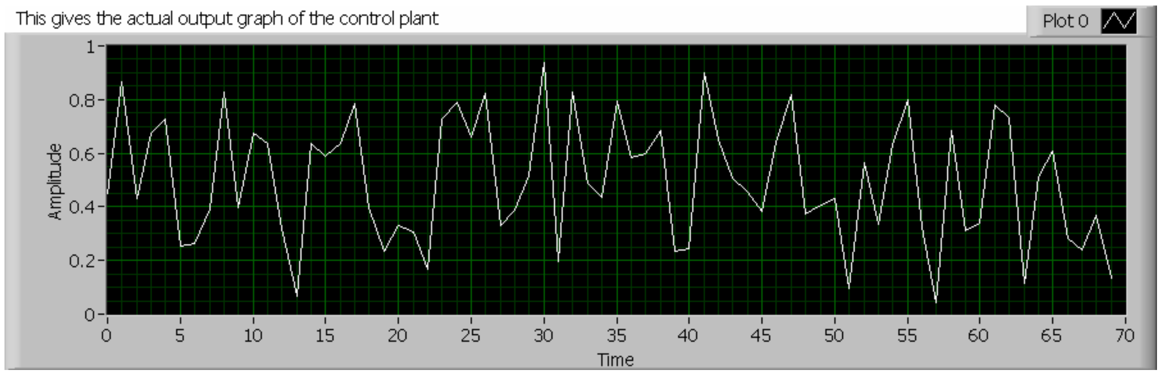


Fig. 5.37 Actual output of the SISO control system given by Eq. 5.1. The first 70 points are plotted here.

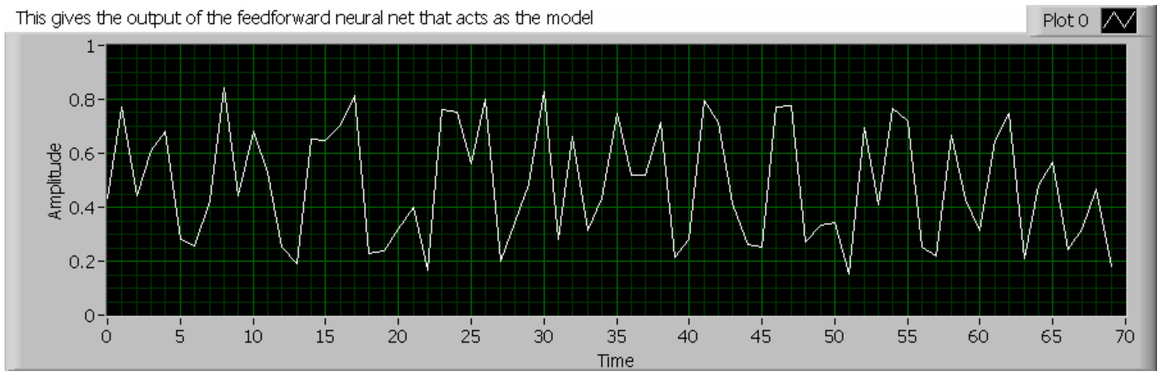


Fig. 5.38 Output of the FF neural network with 5 input neurons, 20 hidden neurons & 1 output neuron which is trained for 300 epochs. The first 70 data points have been used for training and testing. Learning Rate = 0.2

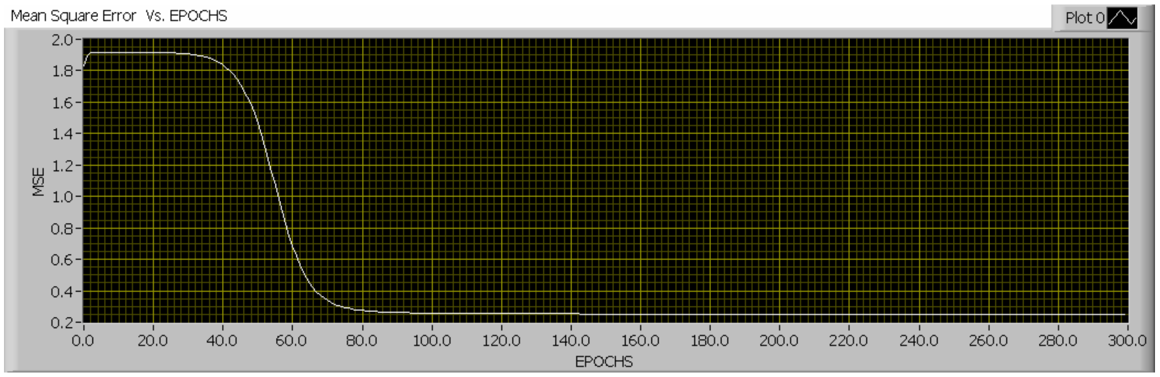


Fig 5.39 Error curve vs. Number of epochs obtained while training the neural net for 300 epochs. First 70 points were used for training.

In the above results, compare figures 5.34 & 5.35. Also compare figures 5.37 & 5.38. It is seen that the neural net, with proper training, has learned to trace the output of the control system. Thus successful results have been produced.

5.4 Conclusions and Future Work

In this thesis, a few neural network architectures have been successfully developed and trained accordingly to produce appropriate results using LabVIEW as the Application Development Environment (ADE). It has been proved that LabVIEW is a good platform for developing and testing computational intelligence algorithms. Neural Nets built in this thesis have a very good GUI and are very user friendly in terms of the presentation and the end result. The end-user can change the variable parameters of the neural net. For example, in a feed forward neural net the end-user can change the learning rate for updating the hidden & output weights; error pre-set value which is the stopping criterion. But the end-user who is not familiar with LabVIEW cannot change the number of neurons in the hidden layer or output layer. Because if one has to change the number of hidden neurons, then changes have to be made at various locations of the main program including the size of the hidden weight array, sub-VIs that calculate the hidden neurons & the gradients of the hidden neurons. Thus, an extension of this thesis would be to convert the currently developed neural nets into complete final products where the end-user can modify the entire architecture of the neural net, like for example, changing the number of hidden neurons, adding one or two hidden layers to the existing neural net, removing a hidden layer, or adding more output neurons. Such LabVIEW Virtual Instruments with excellent GUI could be developed as an extension of this work.

A memory neural net (MNN) [18], which is an extension of the recurrent neural net (RNN) is being currently built for the identification and control of a dynamic non-linear control system [19]. This MNN utilizes the dynamic back propagation algorithm as explained in [18]. An image coding application using the feed forward neural net (PCNN) & utilizing the Generalized Hebbian (GH) algorithm [1] is being developed using LabVIEW. This neural net codes images by extracting only those principal components that are significant across the data space. Also the structure of the neural net can be adjusted and varied so as to also capture the texture of the image. The neural net shown in section 5.2 used only one hidden layer to extract the principal components. Also back propagation algorithm was utilized for training the neural net. Though edges of the image could be detected, texture was not captured. So an extension of the results shown in section 5.2 would be to utilize GH algorithm to get better coded images.

These two projects could be developed and extended into user-friendly applications where the final user can vary all the parameters as well as the structure of the neural net and produce results as per requirement.

REFERENCES

[1] Haykin, S., 1999. *Neural networks* (2nd ed.), Prentice-Hall, Upper Saddle River, NJ.

[2] Kung.S.Y. Digital Neural Networks, Prentice Hall, Englewood Cliffs, NJ 07632

[3] Bishop, C.M. (1995) Neural Networks for Pattern Recognition, Oxford University Press

[4] Werbos, Paul.J (1990) Back propagation through time: what it does and how to do it. Proceedings of the IEEE, vol. 78, No. 10, October

[5] Back propagation algorithm for a generalized neural network structure, K.Krishnakumar

[6] "Classifying Facial Expression with Radial Basis Function Networks, using Gradient Descent and K-means "– Neil Alldrin, Andrew Smith, Doug Turnbull

[7] Chen, S., Cowan, C.F.N., and Grant, P.M. (1991), "Orthogonal least squares learning for radial basis function networks," IEEE Transactions on Neural Networks, 2, 302-309

[8] The Self-Organizing map, Teuvo Kohonen

[9] Bezdek J.C. Pattern recognition with fuzzy objective function algorithms. Plenum Press

[10] Pattern Recognition, Sergios Theodorodis , Second edition

[11] IRIS Flower data , University of California – Irvine database

[12] Machine Learning, Neural and Statistical Classification, Editors: D. Michie, D.J. Spiegelhalter, C.C. Taylor

[13] Neural network approaches to image compression , Proceedings of the IEEE , Vol. 83,No. 2,February 1995 Robert D.Dony , Simon Haykin

[14] Neural Network Technology for Image Compression, J.Jiang ,International Broadcasting Convention Conference, Publication No. 413

[15] http://neuron.eng.wayne.edu/bpImageCompression9PLUS/bp9PLUS.html

[16] K.S. Narendra, K. Parthasarathy: “Identification and control of dynamical systems using neural networks”, IEEE Transactions on Neural Networks, vol. 1, no. 1, 1990.

[17] A neural network based control scheme with an adaptive neural model reference structure , Marzuki Khalid , Sigeru Omatu , IEEE

[18] “Memory Neuron Networks for identification and control of dynamical systems”, P.S.Sastry, G.Santharam, K.P.Unnikrishnan, IEEE Transactions on neural networks, Vol. 5,No. 2,March 1994

[19] Modeling and Control of dynamic nonlinear systems using DRNN, Master’s thesis by Haihua Mao, University of Missouri

[20] A tutorial on Principal Components Analysis – Lindsay I Smith

[21] http://www.faqs.org/faqs/ai-faq/neural-nets

[22] www.ni.com