SIMPLIFIED NEURAL NETWORKS ALGORITHMS FOR FUNCTION APPROXIMATION AND REGRESSION BOOSTING ON DISCRETE INPUT SPACES

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER FOR THE DEGREE OF DOCTOR OF PHILOSOPHY IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2010

By

Syed Shabbir Haider

School of Computer Science

Table of Contents

Total Pages: 140

Total Word Count: 27710

List Of Figures

List Of Tables

Abstract

Function approximation capabilities of feedforward Neural Networks have been widely investigated over the past couple of decades. There has been quite a lot of work carried out in order to prove *‘Universal Approximation Property’* of these Networks. Most of the work in application of Neural Networks for function approximation has concentrated on problems where the input variables are continuous. However, there are many real world examples around us in which input variables constitute only discrete values, or a significant number of these input variables are discrete. Most of the learning algorithms proposed so far do not distinguish between different features of continuous and discrete input spaces and treat them in more or less the same way. Due to this reason, corresponding learning algorithms becomes unnecessarily complex and time consuming, especially when dealing with inputs mainly consisting of discrete variables.

More recently, it has been shown that by focusing on special features of discrete input spaces, more simplified and robust algorithms can be developed. The main objective of this work is to address the function approximation capabilities of Artificial Neural Networks. There is particular emphasis on development, implementation, testing and analysis of new learning algorithms for the Simplified Neural Network approximation scheme for functions defined on discrete input spaces. By developing the corresponding learning algorithms, and testing with different benchmarking data sets, it is shown that comparing conventional multilayer neural networks for approximating functions on discrete input spaces, the proposed simplified neural network architecture and algorithms can achieve similar or better approximation accuracy. This is particularly the case when dealing with *high dimensional-low sample cases*[1], but with a much simpler architecture and less parameters.

[1]. *High Dimensional-Low Sample Cases refers to real world applications where the number of explanatory or independent variables is relatively higher in comparison to the available training examples.*

In order to investigate wider implications of simplified Neural Networks, their application has been extended to the Regression Boosting frame work. By developing, implementing and testing with empirical data it has been shown that these simplified Neural Network based algorithms also performs well in other Neural Network based ensembles.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

First of all I would like to express my sincere gratitude and deep appreciation to my advisor, Dr. Xiao-Jun Zeng for his guidance, encouragement, and support throughout this research. I also want to thank the other committee members specially, Dr. Ludmil Mikhailov for their time in reviewing the initial research proposal and their invaluable suggestions.

I would also like to give my special gratitude to my employer (Mental Health Research Network) especially my Managers, Carly Cooper and Jane Ramsay for making this possible. Their continued help and support has played a very significant part in successful completion of this work.

My gratitude also goes to my parents specially my mother and my brother for their devotion, support and encouragement. Above all I would like to thank my wife for her endless patience and support throughout this research.

CHAPTER 1

INTRODUCTION

Designing machines that can behave like humans has been amongst one of the most extensively explored areas of research in the field of machine learning for many decades. Neural Networks are one of the major milestones in achieving that goal. Artificial Neural Networks are considered one of the hottest topics both at present and in the future of computing. They are indeed self learning mechanisms which don't require the traditional skills of a programmer. Extensive research in this field is underway at the moment, and it is claimed that these neuron-inspired processors can do almost anything, which is attracting more research and development in this field.

1.1 Neural Networks-A Brief Overview

There is no universally agreed upon definition of a Neural Network but there are certainly enough definitions to understand what a Neural Network is. According to [Hay96], "A Neural Network is a massively parallel distributed processor that has a natural propensity for storing experiential knowledge and making it available for use. It resembles the brain in two respects; knowledge is acquired by the network through a learning process and interneuron connection strengths known as synaptic weights are used to store the knowledge."

Some other popular definitions of Neural Networks can be found in [Kas96] and [Rip96]. At this point we can define a Neural Network (NN) or more precisely an Artificial Neural Network (ANN) as "a computational or mathematical model composed of a large number of simple, highly interconnected processing elements capable of learning, information processing and problem solving based upon the connectionist approach to computation" [Med98], we may also refer to [RS03] and

[Wal90] for a detailed history of connectionism. The analogy between a biological neuron and an artificial neuron is depicted in the figure 1.1.

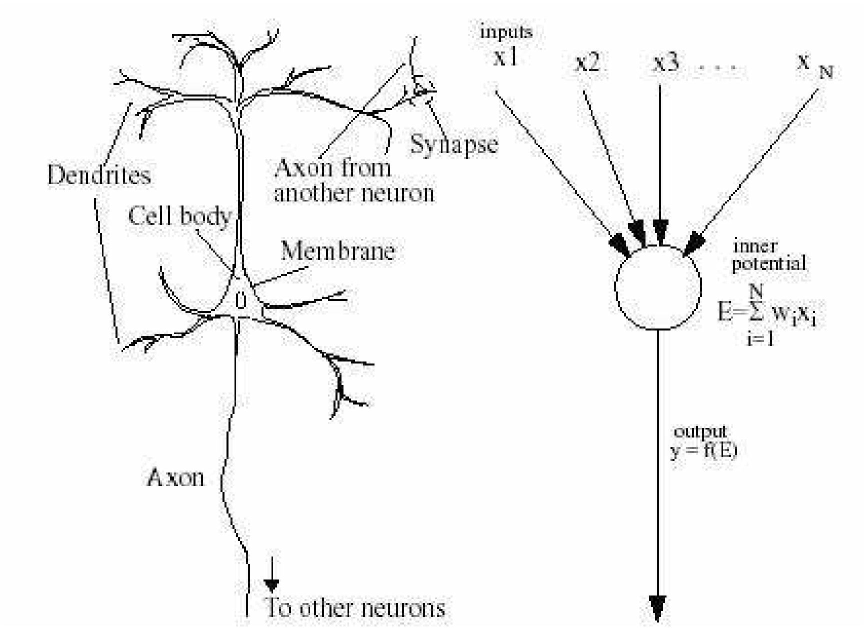

Figure 1.1 Biological Neuron Vs Artificial Neuron

Image Source: Negishi, M. 1998. Everything that Linguists have Always Wanted to Know about Connectionism. Department of Cognitive and Neural Systems, Boston University. URL: http://hemming.se/gslt/LingRes/NeuralNetworks.htm

1.2 Basic Terminology and Architectural Considerations

As defined earlier, an artificial Neural Network is a mathematical model composed of a large number of simple, highly interconnected, processing elements for studying learning and intelligence. According to [KS96], artificial Neural Networks are parallel computation models that have several distinguishing features:

1. A set of processing units.
2. An activation state for each unit, which is equivalent to the output of the unit.

3. Connections between the units. Generally each connection is defined by a weight W_{ij} that determines the effect that the signal of unit i has on unit j.
4. A propagation rule, which determines the effective input of the unit from its external inputs.
5. An activation function, which determines the new level of activation based on the effective input and the current activation.
6. An external input (bias, offset) for each unit.
7. A method for information gathering (learning rule).
8. An environment within which the system can operate, provide input signals and, if necessary, error signals.

As shown in figure 1.2, a processing unit receives a set of inputs X_i, $i = (1, 2, 3.....n)$; these inputs are then multiplied with corresponding connection weights W_{ij}, $i, j = (1, 2, 3.....n)$. The net input to a neuron is computed by summing all the individual products of network inputs, corresponding weight connections & bias i.e.

$$\sum_{i=1}^{n} w_{ij} x_i + b \tag{1.1}$$

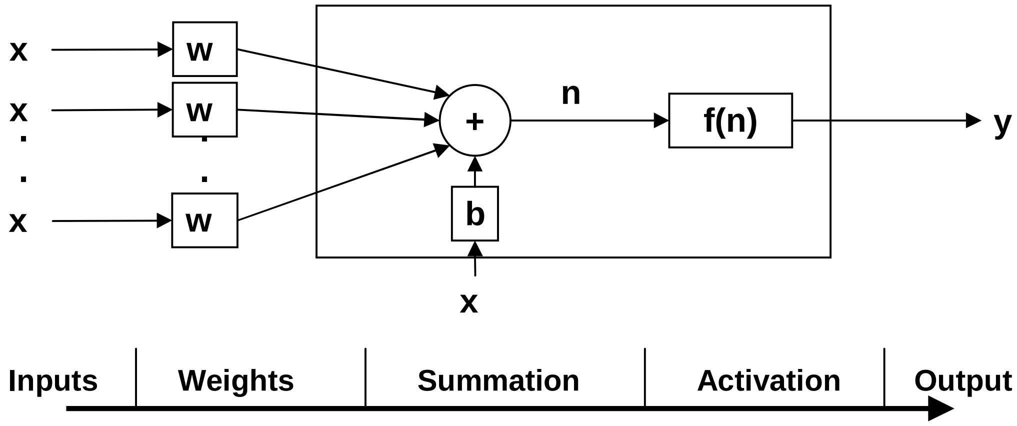

Figure 1.2 Block Diagram of An Artificial Neuron

Each non-input unit in a Neural Network combines values that are fed into it via synaptic connections from other units, producing a single value called net input. The function that combines values is called the *combination function*, which is defined by a certain propagation rule. In most Neural Networks we assume that each unit provides an additive contribution to the input of the unit with which it is connected. The total input to unit j is simply the weighted sum of the separate outputs from the connected units plus a threshold or bias term mentioned in many texts as θj:

$$y_j = \sum_{i=1}^{n} w_{ij} x_i + \theta_j \tag{1.2}$$

The contribution for positive W_{ij} is considered as an excitation and an inhibition for negative W_{ij}. The units having the propagation rule as shown in equation (1.2) are called Sigma Units. In some cases more complex rules for combining inputs are used. One of the propagation rules known as sigma-pi has the following format [KS96]:

$$y_j = \sum_{i=1}^{n} w_{ij} \prod_{k=1}^{m} x_{ik} + \theta_j \tag{1.3}$$

Lots of combination functions usually use a "bias" or "threshold" term in computing the net input to the unit. For a linear output unit, a bias term is equivalent to an intercept in a regression model. It is needed in much the same way as the constant polynomial ‘1’ is required for approximation by polynomials. The function $f(n)$ shown in the figure 1.2 is the unit's activation function. In the simplest case, f is the identity function, and the unit's output is just its net input. This is called a linear unit. There are many other popular choices for activation functions summarised in the table 1.1:

Activation Function	Transfer Characteristics	Network Type
Hard Limiting	$S(x) = 0$ if $x < 0$ $= 1$ if $x \geq 0$	Backpropagation
Symmetrical Hard Limiting	$S(x) = -1$ if $x<0$ $= 1$ if $x \geq 0$	Backpropagation
Linear	$S(x) = x$	ADALINE
Saturating Linear	$S(x) = 0$ if $x < 0$ $S(x) = x$ if $0 \leq x \leq 1$ $= 1$ if $x > 1$	ADALINE
Symmetrical saturating linear	$S(x) = -1$ if $x < 0$ $S(x) = x$ if $-1 \leq x \leq 1$ $= 1$ if $x > 1$	ADALINE
Log Sigmoid	$S(x) = 1/1+\exp^{-x}$	Backpropagation
Bipolar Sigmoid	$S(x) = \dfrac{1-e^{-x}}{1+e^{-x}}$	Backpropagation
Hyperbolic Tangent	$S(x) = \tanh(x) =$ $e^{x}-e^{-x}/e^{x}+e^{-x}$	Backpropagation
Sigmoid +ve Linear	$S(x) = 0$ if $x<0$ $= x$ if $x \geq 0$	Backpropagation
Radial Basis	$S(x,a,b) = k\left(\dfrac{x-a}{b}\right)$	RBF
Competitive	$S(x)=1$; for neuron with maximum ‘x’ $= 0$; for all others	LVQ

Table 1.1 Activation Functions And Their Transfer Characteristics

The architecture or topology of a network is defined by the number of layers, the number of units per layer, and the interconnection patterns between layers. They are generally divided into two categories based on the pattern of connections i.e. Feedforward Neural Networks and Recurrent Neural Networks as shown in figure 1.3.

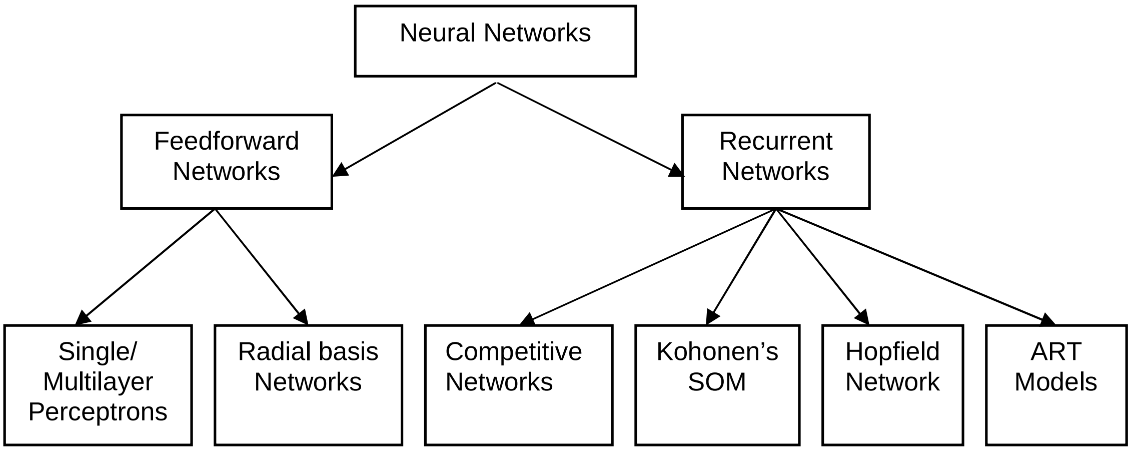

Figure 1.3 Neural Network Topologies

1) Feed-forward networks allows the data to flow from input units to output units in strictly one direction, this is the property that gives this architecture the name 'feed-forward'. The data processing can extend over multiple layers of units, but no feedback connections are present. That is, connections extending from outputs of units to inputs of units in the same layer or previous layers are not permitted as shown in the figure 1.4. Every unit only acts as an input to the immediate next layer. Obviously, this class of networks is easier to analyze theoretically than other general topologies because their outputs can be represented with explicit functions of the inputs and the weights.

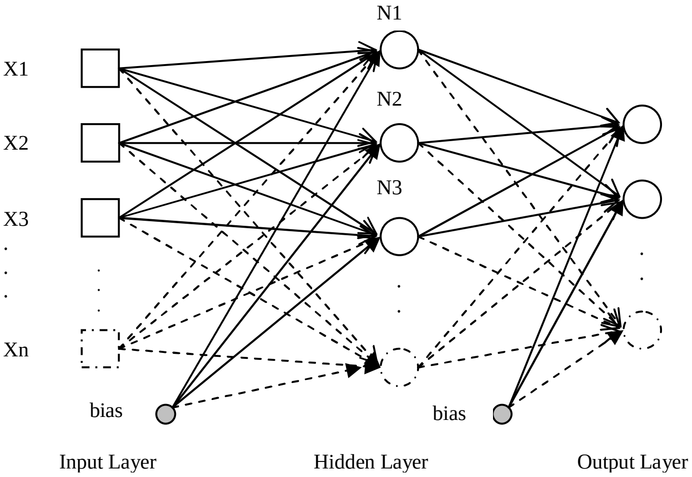

Figure 1.4 Feedforward Neural Network Architecture

Single Layer Perceptron, Multilayer Layer Perceptron (MLP's) and Radial Basis Networks are examples of feedforward network architecture. Feedforward networks trained with backpropagation algorithm are the main focus of this thesis. Details will be described in next chapter. The feed-forward networks provide a general framework for representing non-linear functional mapping between a set of input variables and a set of output variables. The representation capability of a network can be defined as the range of mappings it can implement when the weights are varied. The approximation and representation capabilities of feedforward networks are widely publicized and one may refer to [Sar97][RJ99][Bis95][Hor91] for a detailed review of the issue; at the moment it is sufficient to recognize the following facts about the representation capabilities of feedforward architecture:

- Single Layer Networks are capable of representing only linearly separable functions or linearly separable decision domains.
- One hidden layered network can approximate arbitrarily well any functional continuous mapping from one finite-dimensional space to another, provided that the number of hidden units is sufficiently large. To be more precise, feed-forward networks with a single hidden layer and trained by least-squares are statistically consistent estimators of arbitrary square-integral regression functions if assumptions about samples, target noises, number of hidden units, and other factors are all met. Feed-forward networks with a single hidden layer using threshold or sigmoid activation functions are universally consistent estimators of binary classifications under similar assumptions.
- Two hidden layered networks can represent an arbitrary decision boundary to arbitrary accuracy with threshold activation functions, and could approximate any smooth mapping to any accuracy with sigmoid activation functions.

2) *Recurrent Networks* allow feedback connections. This type of network has at least one feedback loop which can connect a unit to it self, see figure 1.5. In comparison to feed-forward networks, the dynamic properties of the network are important. In some

cases, the activation values of the units undergo a relaxation process such that the network will evolve to a stable state in which activation does not change further. In other applications in which the dynamic behaviour constitutes the output of the network, the changes of the activation values of the output units are significant. Common examples of Recurrent Neural Networks are Competitive Networks, Kohonen's Self Organizing Maps, Hopfield Network and ART Models [KS96].

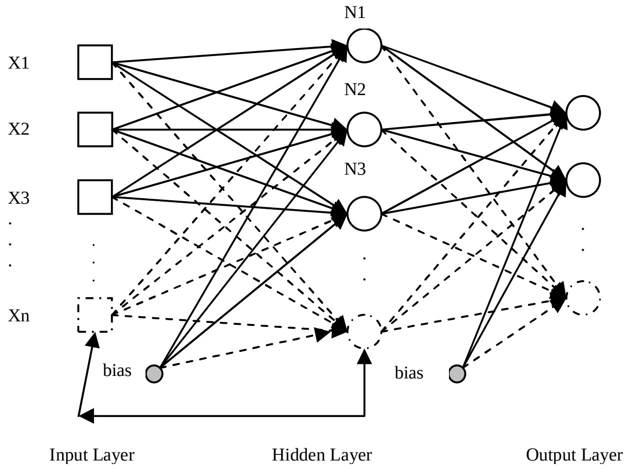

Figure 1.5 Recurrent Neural Networks Architecture

The issue of selecting architecture optimal for a specific problem is of prime importance. The representation capabilities of these networks allow us to choose the best architecture for a specific problem. In addition to a networks representation capabilities, a comprehensive problem specification also help define the network in many ways [HDB96]:

- Number of network inputs = number of problem inputs.
- Number of neurons in output layer = number of problem outputs.
- Output layer transfer function choice at least partly determined by problem specification of the outputs.

The last, but perhaps the most important consideration, is the learning process in Neural Networks. This is the most important feature of Neural Networks which allows them to learn from past experiences. The learning process is also very important with reference to this work and we will therefore discuss learning laws and corresponding algorithms in more detail in chapter 2.

1.3 Real World Applications Of Neural Networks

This evolutionary technology (ANNs) has been successfully applied to many real world applications, and performs very well on tasks involving Classification, Clustering, Pattern Recognition, Function Approximation and Time Series Prediction problems.

These capabilities of (NN) make them a very popular choice for many application areas such as Aerospace, Electronics, Banking, Forecasting, Manufacturing, Medicine, Entertainment, Defence and Bioinformatics. This technology has been successfully used in medical diagnosis (e.g. diagnosis of heart infection & epilepsy), system identification and control (e.g. vehicle control, process control), pattern recognition (e.g. face identification, radar systems, object recognition, etc.), sequence recognition (e.g. speech, handwritten text recognition, gesture,) game-playing and decision making (e.g. racing, backgammon, chess), financial applications, data mining, visualization and e-mail spam filtering. The list of Neural Network applications in real world is very long and the readers are referred to [HDB96][SS96][AB99] for more detailed review of these applications.

Most of the work in application of Neural Networks for function approximation has concentrated on problems where the input variables are continuous. However, there are many real world examples around us in which input variables constitute only discrete values, or a significant number of these input variables are discrete. For the purpose of this research we will focus on real-world function approximation problems, where the independent or input variables are mainly discrete. We will discuss special features of such applications in Chapter 3.

1.4 The Problem Statement

Approximation and representation capabilities of Artificial Neural Networks (ANN) are widely publicized, and to date it has been proved by many that Feedforward Neural Networks (FNN's) are capable of approximating any continuous function to reasonable accuracy; this property is known as 'Universal Approximation Property'. More recently, it has been shown that by focusing on the distinguished features of discrete input spaces, it is possible to have more simplified and possibly more accurate Neural Network architecture that can approximate functions defined on discrete input spaces with sufficient accuracy, and without any compromise on generalisation and approximation capabilities of existing NN schemes. Although standard NN approximation methods can be used for approximation of functions on discrete and mixed input spaces, when dealing with such problems these methods become unnecessarily complex, and less effective due to not taking into account special features of discrete input spaces. The main objective of this work is to address the function approximation capabilities of Artificial Neural Networks, with particular emphasis on development, implementation, testing and analysis of new learning algorithms for the simplified Neural Network approximation scheme for functions defined on discrete input spaces.

1.5 Motivations Behind Initiation Of This Research

The motivations that contributed towards initiation of this research are:

- *'Biological Analogy'*: The fact that Neural Networks resemble the human brain in their architecture and have the ability to learn from experience; just like humans.

- *'The Success of Feedforward Neural Network Architecture':* At present, only a few of Neural Network models, paradigms actually, are being used commercially. One particular model, the *feedforward back-propagation network*, is by far and away the most popular.
- *'Universal Approximation Property':* The ability of Feedforward Neural Networks to approximate any reasonable function to arbitrary accuracy is known as the universal approximation property.
- *'Nature of Input Variable Spaces':* Whilst proving the universal approximation property, almost all the approximation schemes have considered the independent variables (network inputs) to take on continuous values only. There are very few methodical results taking into account the true nature of input variable spaces, if there are any, they follow the same methodology as for continuous variables. A detailed review of the research and results obtained so far will be presented in Chapter 2, in connection with the review of existing techniques and methods.
- *'Discrete Nature of Variables':* In real world applications, many of the variables are discrete in nature i.e. they take on a countable number of values, as compared to continuous variables which can take on any number of values within a given interval. Categorical, nominal and binary variables are classical examples of discrete data. Many real world modelling problems have a large number of variables that just take on discrete values e.g. Location Market Condition Performance Modelling (LMCP) as described in [ZK08][ZGKL05].
- *'Separable Hierarchical Structure':* The property of functions defined on discrete input spaces to have a separable hierarchical structure as discussed in [ZK08].
- *'Limited Availability Of Training Data':* In order to achieve desired accuracy, it is necessary for any NN model to have sufficiently large amount of data available for training. In practice there are many cases when the availability of training data is limited as indicated in [ZK08][ZGKL05].

- *'Possibility Of A More Simplified Neural Network Architecture':* Keeping in mind the special properties of discrete variables (e.g. they take on a finite number of states), it is possible to have a more simplified feedforward Neural Network architecture; that exploits this nature of discrete variables.
- *'More Practical and Acceptable Architecture':* In practice, it is very hard to convince commercial organizations and other customers to employ NN technology to their specific problems because of the black-box nature of Neural Networks and complex computations associated with them. A more simplified architecture may be a better idea in filling that gap; besides the most apparent advantage of saving valuable resources such as processing time and memory while performing complex computations.

1.6 Research Objectives To Be Met

The main objectives of this research are to investigate the function approximation capabilities of Feedforward Neural Network Models, keeping in mind the limitations of standard Feedforward Neural Network model and special features of discrete input spaces. The main objectives of this research will be:

- To propose new simplified algorithms based on the simplified Neural Network approximation scheme proposed in [ZGKL05] for function approximation on discrete input spaces, to overcome the weakness of the existing NN algorithms.
- Development of the corresponding learning algorithms for these new proposed schemes.
- Implementation and analysis of the approximation capabilities of these newly proposed simplified Neural Network algorithms.

- Testing the performance of these algorithms based on empirical data such as in Quantitative Structural Activity Relationship Modeling (QSARs), and compare with the standard Neural Network model.
- Investigate the wider implications of simplified Neural Network approach to regression boosting.
- Propose new simplified regression boosting approach using simplified Neural Network model as base learner.
- Development and implementation of the new simplified regression boosting scheme along with corresponding algorithm.
- Analysis and performance comparison of simplified regression boosting algorithm, with standard regression boosting models employing Neural Networks as base/ weak learners.

1.7 Main Contributions

The main contributions of this research are listed below:

- A systematic review of function approximation capabilities of feedforward Neural Network model and universal approximation property.
- Detailed analysis and evaluation of simplified Neural Network approach.
- Simplified Neural Network based algorithms I and II for approximation of functions defined on discrete input spaces. By developing these learning algorithms, and comparing the performance of these algorithms with standard Neural Network model over benchmarking examples, it has been shown that these algorithms work in practice and achieve similar or better accuracy with employing relatively less parameters required for the model.
- Derivation of simplified backpropagation algorithm for simplified Neural Network algorithm I and II.
- Analysis of wider implications of simplified Neural Network approach in regression boosting frame work.

- Simplified regression boosting algorithm-III based on the simplified regression boosting approach. By implementing and comparing with a standard regression boosting model over benchmarking examples; it has been shown that this algorithm can be used for boosting regression estimates for selected domain.

Although all three algorithms are domain specific, and targets the approximation problems in high dimension-low sample cases for functions defined on discrete input spaces, they are simple enough to be easily extended to target mixed variable and high sample cases.

1.8 Structure Of Thesis

This thesis consists of six chapters, a brief outline is as follows:

Chapter one gives a brief overview and introduction of the chosen research area, with particular emphasis on Neural Network technology. Chapter one also contains a brief problem description, motivations behind this work, and a summary of research objectives.

Chapter two of this thesis focuses on the all important learning phase of Neural Network models. We presented different forms of learning, along with a discussion on learning in MLP models, with particular emphasis on feedforward Neural Network architecture, and the corresponding backpropagation learning algorithm.

Chapter three introduces the function approximation problem, with a detailed review of related work in this field, along with some recent advancement. Neural Network based ensemble methods have also been discussed with a particular focus on application of Neural Networks in regression boosting frame work. Chapter three also details the fundamentals of simplified Neural Network approach and special features of discrete input spaces.

Chapter four of the thesis details the proposed simplified algorithms based on simplified NN approach. A detailed analysis of approximation capabilities of simplified NN algorithms is also included in this chapter. This chapter also contains a discussion on the wider implications of the simplified Neural Network approach, and gives an overview of how simplified NN approach can be applied to regression boosting. We have given a brief introduction to regression boosting in this chapter, and discussed how a simplified regression booting scheme can be developed using simplified NN approach. We also propose a new algorithm for regression booting on functions defined on discrete input spaces in this chapter.

Chapter five of this thesis presents implementation and evaluation details. The obtained results are summarised in form of tables and graphs. A detailed analysis of the performance of the simplified Neural Network based algorithms I, II and simplified regression boosting algorithm-III is also given in chapter five.

Chapter six concludes this research with a detailed summary of the research carried out, results obtained, and contributions in literature. We also discussed important observations and future research directions in chapter six.

1.9 Summary

This chapter gives an introduction to the chosen area of research and gives a brief overview of the Neural Network technology and its applications. We have also included a summary of technological considerations and motivations behind initiation of this research. A summary of problem statement along with details of research objectives to be achieved are also presented in this chapter. This chapter concludes with a summary of all the six chapters of this thesis.

CHAPTER 2

LEARNING IN FEEDFORWARD NEURAL NETWORKS

Most of the Neural Networks used in practice do one or more of the tasks such as pattern classification, function approximation, noise reduction, optimization, data clustering etc. While performing any of these tasks an Artificial Neural Network maps a set of inputs to a set of outputs. This non-linear mapping is generally considered in a multidimensional surface. The objective of learning is to mould the decision surface according to a desired response, either with or without the training process RS03. Readers of this thesis are referred to [AB99] for a comprehensive understanding of theoretical foundation of learning in Neural Networks.

2.1 The Learning Process

Learning or training process is perhaps the back bone of Neural Network technology. As described earlier, functionality of a Neural Network is determined by the combination of the topology (number of layers, number of units per layer, and the interconnection paths between the layers) and the weights of the connections within the network. The topology is usually held fixed, and the weights are determined by a certain training algorithm. The process of adjusting the weights to make the network learn the relationship between the inputs and targets is called learning, or training.

Many learning algorithms have been invented to help find an optimum set of weights that result in a desired solution of the problems. The figure 2.1 presents taxonomy of learning process in a context ascribed by [Hay96]:

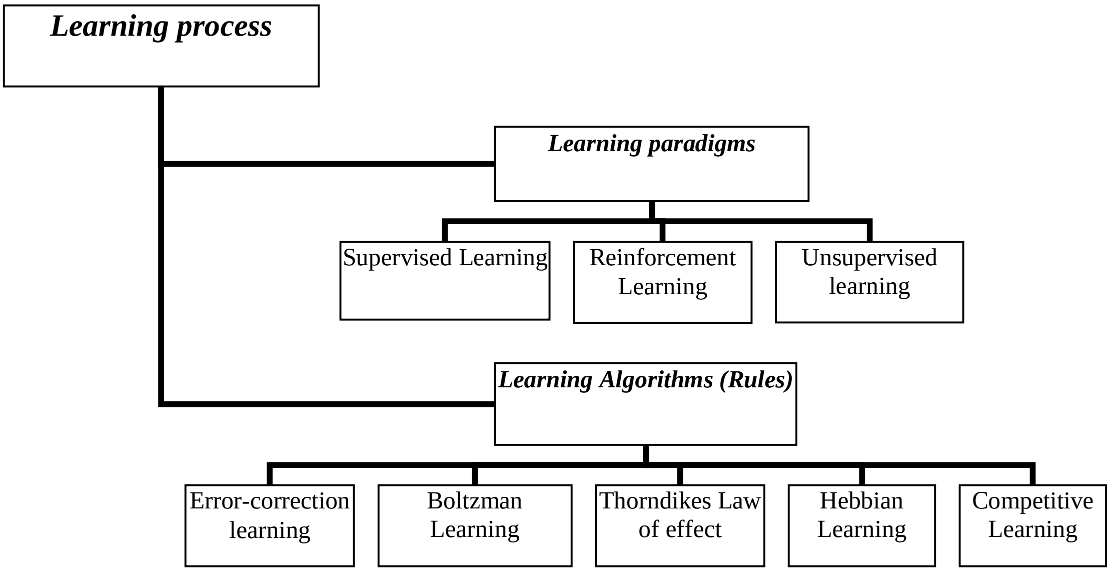

Figure 2.1 Neural Network Learning Process, Paradigms And Algorithms

2.1.1 Supervised Learning Laws

Neural Network Model that uses Supervised Learning are trained by presenting it with examples (also called training data) of inputs, and desired outputs (target values). These input-output pairs are provided by an external teacher, or by the system containing the network. The difference between the real outputs and the desired outputs is used by the algorithm to adapt the weights in the network. It is often posed as a function approximation problem - given training data consisting of pairs of input patterns ‘x’, and corresponding target ‘t’, the goal is to find a function f(x) that matches the desired response for each training input.

2.1.2 Unsupervised (Self Organizing) Learning

With unsupervised learning, there is no feedback from the environment to indicate if the outputs of the network are correct. The network must discover features, regulations, correlations, or categories in the input data automatically. In fact, for most varieties of unsupervised learning, the targets are the same as inputs. In other words, unsupervised learning usually performs the same task as an auto-associative network, compressing the information from the inputs.

2.1.3 Graded (Reinforcement) Learning

Graded or reinforcement learning is quite similar to supervised learning, except that instead of being presented by correct examples of network response on each individual trial, the network receives only a sequence of multiple training trials, i.e. at time intervals containing multiple input-output episodes; the network is given a numeric score or grade that represents the value of some network performance measurement function over this time interval. This type of networks are particularly used in control and process optimization problems where there is no way to know what the desired outputs should be [RS03].

Every learning algorithm follows a learning rule that dictates the whole learning process, in other words the conditions that have to be met by that learning algorithm. Hebb's rule and Delta rule (also called LMS i.e. least mean squared error rule) are two of the most basic and famous of the learning rules. The table 2.1 summarizes the different types of learning rules categorized under supervised and unsupervised learning methods.

Unsupervised Learning Laws	**Supervised Learning Laws**
Kohonen's self organizing maps	Delta rule
Hebb's rule/ signal Hebb law	Generalized delta rule
Competitive learning laws	Simulated Annealing
Differential Hebbian learning laws	Supervised Competitive Learning
Differential competitive learning laws	

Table 2.1 Supervised and Unsupervised Learning Laws

2.2 The Supervised Learning Laws For MLPs

This section details various supervised learning algorithms, with particular emphasis on multilayer feedforward networks trained with backpropagation algorithm, since this is the main focus of this research. Before we move on to a detailed analysis of these learning algorithms, the selection of an objective or cost function under which these algorithms operate, is very important. To train a network and measure how well it performs, an objective function (or cost function) must be defined to provide an unambiguous numerical rating of system performance. Selection of an objective function is very important because the function represents the design goals and decides what training algorithm can be taken. To develop an objective function that measures exactly what we want is not an easy task. A few basic functions are very commonly used. One of them is the sum of squares error function,

$$E = \frac{1}{NP}\sum_{p=1}^{P}\ \sum_{i=1}^{N}(t_{pi} - o_{pi})^2 \tag{2.0}$$

where 'P' indexes the patterns in the training set, N denotes the total number of patterns, 'i' indexes the output nodes, and 't_{pi}'and 'O_{pi}' are, respectively, the target and actual network output for the 'ith' output unit on the 'pth' pattern. In real world applications, it may be necessary to complicate the function with additional terms to control the complexity of the model.

2.2.1 The Perceptron Learning Rule

The McCulloch-Pitts (1943) neuron model has severe limitations e.g. the lack of learning capabilities mainly due to the presence of fixed set of weights and threshold. To overcome these severe shortcomings, several models were proposed that have the ability to some how adjust the synaptic weight connections [KS04]. The *perceptron learning* rule is perhaps the first of all supervised learning rules. It was introduced by Frank Rosenblatt in late 1950's. Although very basic in its computing capabilities, it nevertheless influenced extensive research taken in this field of computing. In perceptrons, training the weights are updated by altering the network parameters by an amount proportional to the difference between the target output and the actual output. One way to learn an acceptable weight vector is to begin with random weights, then iteratively apply the perceptron to each training example, modifying the perceptron weights whenever it misclassifies an example. This process is repeated, iterating through the training examples as many times as needed until the perceptron classifies all training examples correctly. Weights are modified at each step according to the *perceptron training rule*. Following is a description of basic steps in perceptron training rule.

Initialization: Set all the weights and node threshold to small random numbers. Note that the node threshold is the negative of the weight from the bias unit (whose activation level is set to one).

Computing activation level of units: The activation level of an input unit is determined by the instance presented to the network. However, the activation level of an output unit is determined as: $O_{\mathrm{j}} = f_h(a)$, where $a = \sum_{j=1}^{n} \left(w_{ji}\mathbf{x}_i - \theta_j \right)$, $f_h(a)$is a hard limiting function given by: $f_h(a) = 1$, if $a \geq 0$and, $f_h(a) = 0$if $a < 0$.

Weight Adjustment: Adjust weights by following the rule:

$$w_{ji}(new) = w_{ji}(old) + \Delta w_{ji} \tag{2.1}$$

where as change in w_{ji} can be computed as,

$$\Delta w_{ji} = \eta \left(t_i - o_i \right) x_i \tag{2.2}$$

where 'η' is a time dependent learning rate (0<η <1), t_i represents the target output where as o_i represents the actual output of the unit.

Iterations: Repeat the process until convergence is achieved.

Note that output value ‘o_i‘ is +1 or -1 (not a real); the perceptron rule is a learning rule for a threshold unit and to achieve convergence the training examples should be linearly separable and the learning rate should be sufficiently small.

2.2.2 The Widrow-Hoff Learning (DELTA) Rule

The very first extension of perceptron training rule was proposed in early 1960’s by Widrow called the delta rule. His model ADALINE has the ability to adjust the network synaptic weights according to *Widrow-Hoff learning rule* famously known as the *Least Mean Square (LMS) Algorithm*. The learning rule for ADALINE is formally derived using the gradient descent algorithm. The LMS rule adjusts the weights of the network by incrementing them every iteration step by an amount proportional to the gradient of the cumulative error of the network.

The basic differences in both the rules are summarized in table 2.2.

Perceptron rule	**Delta rule**
Thresholded output	Unthresholded output
Converges after a finite number of iterations to a hypothesis that perfectly classifies the training data, provided the training examples are linearly separable.	Converges only asymptotically toward the error minimum, possibly requiring unbounded time, but converges regardless of whether the training data are linearly separable.
Linearly separable data	Linearly non-separable data

Table 2.2 Perceptron Vs Delta rule

The delta training rule is best understood by considering the task of training an *unthresholded* perceptron; that is, a *linear unit* for which the output '*o*' is given by:

$$o(\vec{x}) = \vec{w}.\vec{x} \tag{2.3}$$

In order to derive a weight learning rule for linear units, let us begin by specifying a measure for the training error of a hypothesis (weight vector), relative to the training examples:

$$E(\vec{w}) \equiv \frac{1}{2}\sum_{d \in D}(t_d - o_d)^2 \tag{2.4}$$

Where the term t_d is the target and o_d refers to actual output of the linear units. The vector derivative of equation (2.4) is called the *gradient* of *E* with respect to (w) written as:

$$\nabla E(\vec{w}) \equiv \left[\frac{\partial E}{\partial w_o}\frac{\partial E}{\partial w_1}\frac{\partial E}{\partial w_2}\ldots\ldots\frac{\partial E}{\partial w_n}\right] \tag{2.5}$$

The gradient specifies the direction that produces the steepest increase in *E*. The negative of this vector therefore gives the direction of steepest decrease.

As we know that the training rule for gradient descent algorithm is:

$$\vec{w} \leftarrow \vec{w} + \Delta\vec{w} \tag{2.6}$$

where

$$\Delta w = -\eta \nabla E(w) \tag{2.7}$$

The negative sign is presented because we want to move the weight vector in the direction that decreases *E*. This training rule can also be written in its component form as shown in equation (2.8):

$$w_i \leftarrow w_i + \Delta w_i \tag{2.8}$$

Where

$$\Delta w = -\eta \frac{\partial E}{\partial w_i} \tag{2.9}$$

which makes it clear that steepest descent is achieved by altering each component w_i of w in proportion to $\frac{\partial E}{\partial w_i}$.

The vector of $\frac{\partial E}{\partial w_i}$ derivatives that form the gradient can be obtained by differentiating E

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \tag{2.10}$$

$$\frac{\partial E}{\partial w_i} = \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} (t_d - o_d)^2 \tag{2.11}$$

$$\frac{\partial E}{\partial w_i} = \frac{1}{2} \sum_{d \in D} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \tag{2.12}$$

$$\frac{\partial E}{\partial w_i} = \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w}.\vec{x}_d) \tag{2.13}$$

$$\frac{\partial E}{\partial w_i} = \sum_{d \in D} (t_d - o_d) (-x_{id}) \tag{2.14}$$

The weight update rule for standard gradient descent can be summarized as:

$\Delta w_i = -\eta \dfrac{\partial E}{\partial w_i}$ where,

$$\Delta w_i = \sum_{d \in D} (t_d - o_d)\ (-x_{id}) \qquad (2.15)$$

The major steps of this gradient descent learning algorithm are outlined in figure 2.2:

Recall that the training pairs are of the form $\langle x,t \rangle$, where x is the vector of input values and ‘ t ’ is the corresponding target values. ‘η ’ is a small value e.g 0.5, called the learning rate.

```
Step 1.  Initialization:  Initialize each $w_i$ to small random values.
Step 2.  Until termination condition is met, repeat:
         - Initialize each $\Delta w_i$ to zero.
         - For each $\langle \vec{x},t \rangle$ in training set, repeat
               Input each instance $\vec{x}$ to the unit and compute the output ‘O ‘.
               For each linear unit weight $w_i$ , Do
                   $\Delta \vec{w}_i \leftarrow \Delta \vec{w}_i + \eta\,(t - o)\,x_i$
         - For each linear unit weight $w_i$ , Do
               $w_i \leftarrow w_i + \Delta w_i$
```

Figure 2.2 The Widrow-Hoff Learning Algorithm

2.3 Backpropagation Algorithm For MLPs

The Backpropagation algorithm was first proposed by Paul Werbos in the 1970's. However, it was not until it was rediscovered in 1986 by Rumelhart and McClelland that BackPropagation became widely used.

As described earlier, linear approximation networks are too restrictive and nonlinear approximation networks offer much greater capacity. In order to enhance the approximation capabilities, it is critical to expand a single layer structure to a multilayer network. A typical multilayer Neural Network may consist of many layers of neurons that can be divided into three categories: Input, Output, and a Hidden layer. We have already seen how the Input and Output layers work, so now we will discuss the hidden layer. When it comes to using the gradient descent method for training a multilayer Neural Network, we run into some problems. Recall that the gradient descent technique basically measures the amount of error that our present output differs from the actual output we want. From the gradient descent technique described in simple Neural Networks, it was easy to calculate this change in proportional error because our weights are only found on input cells. Since our gradient descent really only calculate the change in weight proportion based on the input weights, how do we go about adjusting the hidden layer weights? One way of thinking is to re-calculate each hidden-layer units' weight based on their own individual inputs. While this would work, it would be quite time-consuming. One method that recursively does this, is the concept of backpropagation.

The idea behind backpropagation is to compute the individual error functions for each output node in our Neural Network and then sum them up. This summed up error represents the overall error function for our Neural Network. Now, since our error function is a summation of a group of output nodes' errors, we can determine the individual negative gradients for each output as the function is a continuous and differentiable function over the weights that contributed to that output nodes' error. We apply this same process recursively for each hidden layer of the Neural Network and update all of the weights. This recursive calculation of each layer's error and subsequent negative gradient calculation is known as backpropagation, as you are propagating the calculation back through the network layer by layer.

This algorithm is basically a generalization of the gradient descent method explained above. What we are in essence doing is treating each output as a single perceptron and updating the weights associated with it. We then recursively backpropagate this calculation through all the layers of the network until the Neural Network is trained. The combination of weights which minimizes the error function is considered to be a solution of the learning problem.

This algorithm will form the basis of our work and we will frequently refer to different steps in this algorithm throughout this thesis. Therefore, we have included a detailed derivation of the standard BP algorithm as appendix-A.

2.4 Special Issues in BP Learning and MLPs

The section below briefly describes some of the commonly addressed issues relating to backpropagation learning and Multilayer Perceptrons (MLPs).

2.4.1 Convergence, Stability And Plasticity

Convergence - We can say that the network has achieved *convergence* when the examples of the tasks are continuously presented, and the corresponding weight changes are carried out in such a way that the changes made during one iteration does not affect changes made in earlier alterations [RS03]. In other words, a situation when the network response for two consecutive cycles is the same and therefore no further iterations are required.

Stability - If weights are altered after each iteration, then convergence of weights should constitute towards the stability of the network. But in most situations it takes many more iterations than you desire to have output in two consecutive cycles to have the same response. Then a tolerance level on the convergence criterion can be used. With a tolerance level, an early and stabilized network state can be achieved.

Plasticity – Suppose a network is trained to learn some new examples, and in this process the weights are adjusted according to an algorithm. After learning those examples the network encounters a new example, the network then alters the model parameters again to learn that new example. But if the new weight structure is not responsive to the latest example; then the network does not possess plasticity. Thus the *Plasticity* is the ability to deal satisfactorily with new short -term memory (STM) while retaining the long-term memory (LTM) [RS03]. However, attempts to endow a network with plasticity may have some adverse effects on the stability of the network.

2.4.2 Selection of Hidden Layer Units (Activation function)

Since this method requires computation of the gradient of the error function at each iteration step, we must guarantee the continuity and differentiability of the error function. Obviously we have to use a kind of activation function other than the step function used in perceptrons, because the composite function produced by interconnected perceptrons is discontinuous, and therefore the error function too. One of the more popular activation functions for backpropagation networks is the sigmoidal activation function.

2.4.3 When To Stop Training?

Another important issue with backpropagation learning is when to stop the training. We know that in typical applications the weight update loop may be iterated thousands of times. The choice of termination condition is important because too few iterations can fail to reduce error sufficiently, on the other hand too much iterations can lead to over fitting the training data. Many researchers have suggested different solutions for termination criteria problem e.g. stopping the training session after a fixed number of iterations (epochs) have elapsed, stopping once the validation error meets some criterion, or once the error falls below some preset threshold value.

2.4.4 Local Minima

Since backpropagation uses a gradient-descent procedure, a Backpropagation network follows the contour of an error surface with weight updates moving it in the direction of steepest descent. For simple two-layer networks (without a hidden layer), the error surface is bowl shaped and using gradient-descent to minimize error is not a problem; the network will always find an errorless solution (at the bottom of the bowl). Such errorless solutions are called global minima. However, when an extra hidden layer is added to solve more difficult problems, the possibility arises for complex error surfaces which contain many minima. Since some minima are deeper than others, it is possible that gradient descent will not find global minima. Instead, the network may fall into local minima which represent suboptimal solutions.

2.4.5 Number Of Hidden Layers

We already know that networks with two hidden layers can represent functions with any kind of shapes. There is no theoretical reason to use networks with more than two hidden layers. It has also been proved that for the vast majority of practical problems, there is no reason to use more than one hidden layer. Problems that require two hidden layers are only rarely encountered in practice. Even for problems requiring more than one hidden layer theoretically, most of the time, using one hidden layer performs much better than using two hidden layers in practice [Mas93]. Training often slows dramatically when more hidden layers are used. Of course, it is possible that for a certain problem, using more hidden layers of just a few units is better than using fewer hidden layers requiring too many units, especially for networks that need to learn a function with discontinuities. In general, it is strongly recommended that one hidden layer be the first choice for any practical feed-forward network design. If using a single hidden layer with a large number of hidden units does not perform well, then it may be worth trying a second hidden layer with fewer processing units.

2.4.6 Number of Hidden Units

Another important issue in designing a network is how many units to place in each layer. Using too few units can fail to detect the signals fully in a complicated data set, leading to under-fitting. Using too many units will increase the training time, perhaps so much that it becomes impossible to train it adequately in a reasonable period of time. A large number of hidden units might cause over-fitting, in which case the network has so much information processing capacity, that the limited amount of information contained in the training set is not enough to train the network.

The best number of hidden units depends on many factors such as the numbers of input and output units, the number of training cases, the amount of noise in the targets, the complexity of the error function, the network architecture, and the training algorithm [Sar97]. There are lots of “rules of thumb” for selecting the number of units in the hidden layers as mentioned in [Mas93] [Sar97][Ara93] :

- Somewhere between the input layer size and output layer size.
- Two third of the input layer size plus the output layer size.
- Less than twice the input layer size.
- Squared input layer size multiplied by output layer size.

Those rules can only be taken as a rough reference when selecting a hidden layer size. They do not reflect the facts well because they only consider the factor of the input layer size and output layer size, but ignore other important factors that we have discussed earlier. In most situations, there is no easy way to determine the optimal number of hidden units without training, using different numbers of hidden units and estimating the generalization error of each. The best approach to find the optimal number of hidden units is trial and error. In practice, we can use either the forward selection (i.e. starting with a small number of hidden units and increasing gradually until convergence criteria is met) or backward selection (i.e. starting with a large number of hidden units and decreasing gradually until convergence criteria is met) to determine the hidden layer size.

2.4.7 Learning Rate and Momentum

The Backpropagation algorithm requires that the weight changes be proportional to the derivative of the error. The larger the learning rate, the larger the weight changes on each epoch, and the quicker the network learns. However, the size of the learning rate can also influence whether the network achieves a stable solution. If the learning rate gets too large, then the weight changes no longer approximate a gradient descent procedure. (True gradient descent requires infinitesimal steps). Oscillation of the weights is often the result. Ideally then, we would like to use the largest learning rate possible without triggering oscillation. This would offer the most rapid learning and the least amount of time spent waiting at the computer for the network to train. One method that has been proposed is a slight modification of the backpropagation algorithm so that it includes a momentum term. Applied to backpropagation, the concept of momentum is that previous changes in the weights should influence the current direction of movement in weight space. With momentum, once the weights start moving in a particular direction in weight space, they tend to continue moving in that direction which can help the network to "roll past" any local minima, as well as speed learning (especially along long flat error surfaces).

2.4.8 The Training Style

Updating the weights in a backpropagation network can be achieved by either of two ways:

1. *Online or Pattern By Pattern Learning,* in which the network parameters are updated after the presentation of each pattern. This type of learning is recommended

for application requiring high accuracy and can compromise on other factors such as time etc.

2. *Batch or Epoch Based Training*, where the network parameters are updated once or after all of the patterns in the training set have been presented. This method works out to be much faster then the online training methods.

2.4.9 Test, Training And Validation Sets

In NN methodology, the sample is often subdivided into "training", "validation", and "test" sets. The distinctions among these subsets are crucial; it is often argued that any performance comparison among two networks should be done on data that is not used for training or unseen examples. Neural network models are trained using the training data set examples, the performance is then compared using validation data set examples, this approach is known as 'hold-out' method [Bis95]. However, this approach can lead to some over-fitting in validation sets, therefore a third data set usually called test set is used to compare the performance of selected networks.

In [Spr97] author defines these three types of training data as:

- *Training set* - A set of examples used for learning that is to fit the parameters [i.e., weights] of the classifier.
- *Validation set* - A set of examples used to tune the parameters [i.e. architecture, not weights] of a classifier, for example to choose the number of hidden units in a Neural Network.
- *Test set* - A set of examples used only to assess the performance or generalization of a fully-specified classifier.

The crucial point is that a test set, by the standard definition in the NN literature, is never used to choose among two or more networks, so that the error on the test set provides an unbiased estimate of the generalization error (assuming that the test set is representative of the population, etc.). Any data set that is used to choose the best of two or more networks is, by definition, a validation set, and the error of the chosen network on the validation set is optimistically biased [Sar97].

To summarize the above discussion, we should remember that BP learns the weights for a multilayer network, given a network with a fixed set of units and interconnections. It employs gradient descent to attempt to minimize the squared error between the network output values and the target values for these outputs. The learning or network training is carried out in two phases. In forward stage, we calculate outputs given training examples of the form [**X**, t], and in backward stage, we update weights by calculating delta for all the hidden and input layers separately.

Many researchers and mathematicians have derived the BP algorithm in sufficient detail. The readers of this thesis are referred to [RS03][HDB96] and [Hay96] for an in-depth discussion and derivation of this algorithm. A detailed derivation of backpropagation algorithm for Multilayer Perceptrons is also presented in ‘Appendix-A’ for better understanding and further reference in this thesis.

2.5 Variants of the BP Learning

The gradient descent optimization method used in the standard back-propagation learning algorithm is widely used and proven very successful in many applications, but it does have some disadvantages i.e. the convergence tends to be extremely slow and convergence to the global minimum is not guaranteed. Many researchers [FM98][RJ99][Bis95][SH96][KP99] have suggested improvements to the standard gradient descent method, such as dynamically modifying learning parameters or adjusting the steepness of the sigmoid function. In appropriate circumstances, other optimization methods may be better than the gradient descent. Many converge much faster than gradient descent in certain situations, while others promise a higher probability of convergence to global minima [Wag02].

Conjugate gradient descent is one of the most often recommended optimization methods to replace the gradient descent [Mas93][RJ99][Bis95], this is a direction set minimization method. Minimization along a direction *'d'* brings the function 'E' to a place where its gradient is perpendicular to *'d'*. Instead of following the gradient at every step, a set of 'n' directions is constructed which are all conjugate to each other, such that minimization along one of these directions does not spoil the minimization along one of the earlier direction.

Gradient methods using second-derivatives (Hessian matrix), such as Newton's method, can be very efficient under certain conditions [Wag02]. Where first-order methods use a local linear approximation of the error surface, second-order methods use a quadratic approximation. Because such methods use all the first and second order derivative information in exact form, local convergence properties are excellent. Unfortunately, they are often impractical because explicit calculations of the full Hessian matrix can be very expensive in large problems. Some powerful, stochastic optimization methods such as simulated annealing [Mas93][RJ99] and genetic algorithms, which can overcome the local minima, have also been used successfully in a number of problems.

Methods discussed above are some of many improvements that have been suggested over a period of 10-15 years. For a detailed overview of these enhancements we may refer to the resources mentioned in section 2.5. In addition to that, there are many learning algorithms available in Matlab for experimentation and evaluation purposes e.g. Gradient Descent Learning with Momentum, Gradient

Descent Learning with Variable Learning Rate, Conjugate Gradient Learning, Levenberg-Marquardt Learning etc.

2.6 Summary

This chapter is a detailed overview of learning process in the Neural Networks. We have introduced different Learning paradigms and rules with particular emphasis on the Supervised Learning Laws for Multilayer Perceptions. We have also presented a detailed description of backpropagation algorithm used for training feedforward networks, and have discussed special issues relating to backpropagation learning process. Backpropagation algorithms remains the main focus of this work, therefore we have included a detailed derivation of all the steps in this algorithm as Appendix-A, which will be referred throughout this thesis for comparison with proposed simplified algorithms.

CHAPTER 3

APPROXIMATION CAPABILITIES OF FNNs AND RELATED WORK

Neural Networks have become very popular in many real life applications. As described earlier, the range of tasks and potential application areas for Neural Networks are ever increasing. Along with other recent advancements in the field of Neural Networks, there has been much research work being carried out in exploring the function approximation capabilities of NN’s i.e. the problem of estimating a function from a set of samples [HG92]. Historically, the two main areas of research in this field were classified as existence/constructive proofs for the ‘Universal Approximation Problem’ and ‘Tight Bounds on the Size needed by the Approximation Problem’. However, over the past decade, this focus has shifted more towards development of new and perhaps more efficient learning algorithms for Neural Networks to approximate functions.

3.1 Function Approximation-The problem

Function approximation is known to be a very common computational task in many science, engineering and real world applications. As a computational problem, Function approximation is very similar to non-liner regression, or learning a model depending on the disciplines and community involved. The problems may be dealt with differently in different communities, but the essence of the problem is the same. The aim of function approximation is to learn a mapping between an input and an output space from a set of input-output data i.e. the target function, call it f , may be unknown; instead of an explicit formula, only a set of points of the form $(x, f(x))$ is provided. Let,

$$x_i \in R^m, i = 1, 2, ..., N \text{ and } d_i \in R^1, i = 1, 2, ..., N \tag{3.1}$$

be the N input vectors with dimension m and N real number output respectively. We seek an unknown function $f(x): R^m \to R^1$ that satisfies the interpolation where

$$f(x_i) = d_i \text{ and } i = 1, 2, ..., N \tag{3.2}$$

The goodness of fit of d_i by the function f is given by an error function. A commonly used error function is defined by,

$$E(f) = \frac{1}{2}\sum_{i=1}^{N}(d_i - y_i)^2 = \frac{1}{2}\sum_{i=1}^{N}\left[\left(d_i - f(x_i)\right)\right]^2 \tag{3.3}$$

Where y_i is the actual response. In short, the main concern is to minimize the error function. In the other words, to enhance the accuracy of the estimation is the principal objective of function approximation.

3.2 FNN's As Universal Function Approximators

To date it has been proven by many researchers/ scientists that feedforward Neural Networks (FNN's) are capable of approximating any class of generic functions with sufficient accuracy [ST98] (i.e. NN as mathematical models are generally enough for most applications). This property is known as Universal Approximation. A detailed review of results on 'universal function approximation property' can be seen at [TKG03][Pin99][HSW89][AP97][Bau88][Bar93][LMB03].

The roots of universal approximation dates back to 1950s. Kolmogorov was perhaps the first of the researchers who proved that for any continuous mapping there must exist a three-layered feedforward Neural Network of continuous type neurons (having an input layer with n neurons, a hidden layer with (2n+1) neurons, and an output layer with *m* neurons) that implements *f* exactly, see [Bei98]. Cybenko [Cyb89] showed that any continuous function defined on a compact subset of R^n can be approximated to any desired degree of accuracy by a feedforward Neural Network with one hidden layer using sigmoidal nonlinearities. Many other papers have investigated the approximation capability of three-layered networks in various ways. Following the initial advancements in this area, Chen et al. [CCL95] pointed out that the boundedness of the sigmoidal function plays an essential role for its being an activation function in the hidden layer, i.e., instead of continuity or monotony, the boundedness of sigmoidal functions ensures the network's approximation capability of functions defined on compact sets in R .

In 1987, Hecht-Nielsen [HeN87] published a communication in which he turned attention to Kolmogorov's theorem. He pointed out a resemblance between the formal structure of Kolmogorov's expansion of continuous functions through other auxiliary functions with three layer feed-forward Neural Networks, condition of exactness of Kolmogorov formula, and there was only required that the formula only approximately represents continuous bounded functions.

Considerable breakthrough in this interesting field of theory of multilayer perceptrons was done by Hornik et al. [Hor91]. They demonstrated that an arbitrary continuous function can be uniformly approximated by three layer Neural Networks (with one layer of hidden neurons), where the hidden and output neurons are endowed by the so-called squashing transfer functions (sigmoid belongs between them).

Mhaskar & Hahm [MH97] presented generalized translation networks to uniformly approximate a class of nonlinear, continuous functionals defined on $L_p\left([-1,1]^s\right)$ for integer $s \geq 1, 1 \leq p < 1$ or $C\left([-1,1]^s\right)$. They obtained lower bounds on the possible order of approximation for such functionals in terms of any approximation process, depending continuously upon a given number of parameters. Their networks almost achieve this order of approximation in terms of the number of parameters (neurons) involved in the network. The training is simple and non-iterative. In particular, they avoided any optimization such as that involved in the usual back-propagation.

Stinchcombe [Sti99] proposed a characterization criteria for the set of activation functions, bounded or unbounded, that allow feedforward network approximation of the continuous functions on the classic two-point compactification of $R(1)$. The characterization fails when the set of targets are continuous functions on the classic compactifications of $R(n), n \geq 2$. Non-polynomial, analytic activation functions, with input-to-hidden weights in very limited sets, allow approximation of continuous function over compact sets in $R(n)$, while even sigmoidal activation functions with weights in limited sets cannot approximate continuous functions on compactifications. The abstract structure foregrounded by compactification leads directly to possibility results for multi-layer networks and possibility results for Neural Networks in infinite dimensional settings.

Selmic & Lewis [SL02] presented a new NN structure for approximating piecewise continuous functions. In their method a standard NN with continuous activation functions is augmented with an additional set of nodes with piecewise continuous activation functions. They proved that such a NN can approximate arbitrarily well any piecewise continuous function provided that the points of discontinuity are known. Since this is the case in many nonlinearities in industrial motion systems (friction, deadzone inverse, etc.) such a NN is a powerful tool for compensation of systems with such nonlinearities.

Hagan et al. [HDJ02] investigated the use of Neural Networks in control systems. They demonstrated the capabilities of this network for function approximation, and have described how it can be trained to approximate specific functions. They also presented three different control architectures that use Neural Network function approximators as basic building blocks. The control architectures were demonstrated on three simple physical systems.

Magoulas et al. [MVA99] presented three new gradient-based training methods. They claimed that these new methods ensure global convergence, that is, convergence to a local minimizer of the error function from any starting point. They compared their proposed algorithms with several training algorithms, and proved that their algorithms are numerically more efficient then its counterparts.

Park & Sandberg [PS93] proved that under certain mild conditions on the kernel function, radial-basis-function networks having one hidden layer and the same smoothing factor in each kernel, are broad enough for universal approximation. This provides an analytical basis for the design of Neural Networks using radial basis functions.

Poggio and Girossi [PG90] developed a theoretical framework for approximation based on regularization techniques that lead to a class of three-layer networks that called Generalized Radial Basis Functions (GRBF). They showed that GRBF networks are not only equivalent to generalized splines, but are also closely related to several pattern recognition methods and Neural Network algorithms. They introduced several extensions and applications of the technique and discussed intriguing analogies with neurobiological data.

Rossi and Conan-Guez [RCg05] showed that fundamental results for classical MLP can be extended to functional MLP. They obtained universal approximation results that showed the expressive power of functional MLP which is comparable to that of numerical MLP.

3.3 Approximation And Representation Capabilities of FNNs

Subsequent research in this field followed the pioneering works discussed above; many authors studied Universal Approximation by Feedforward Neural Networks. It is well known that a two-layered FNN, i.e. one that does not have any hidden layers, is not capable of approximating generic nonlinear continuous functions. On the other hand, four or more layer FNNs are rarely used in practice. Furthermore, the proof that they are universal approximators is simple. Hence almost all the work deal with the most challenging issue of the approximation capability of three-layered FNNs [ST98]. Under very mild assumptions on the activation functions in the hidden layer, it has been shown that a three-layered feedforward Neural Network is capable of approximating a large class of functions, including the continuous functions and integrable functions. The class of functions realized by a three-layered feedforward Neural Network can be represented as

$$\sum_{i=1}^{N} c_i g\left(x, \theta_i, b_i\right) \tag{3.4}$$

where N is the number of hidden nodes, $x \in R^n$ is a variable $c_i \in R$, $\theta_i \in R^n$, $b_i \in R$ are parameters, and $g\left(x, \theta_i, b_i\right)$ is the activation function used in the hidden layer.

Along with number of hidden layers another, very important consideration is the selection of activation function for the model. In order to explain the approximation capabilities of FNNs, many authors studied different types of activation functions. We can also classify the research in this field according to activation function used in the model. Radial and Ridge activation functions are two most commonly used activation functions in practice. We will briefly outline the research in both directions in the following section.

3.3.1 Ridge Activation Functions

As shown in [ST98], a ridge function has the following form: $g(x,a,b)=\sigma(a'x+b)$ where ‘‘ ’ ’’ is the transpose operator, a is a ‘ $d\times1$ ’ vector, usually referred to as the direction of the ridge function, and b is a scalar called the threshold. $\sigma(.)$ is a nonlinear function. The most common example is the logistic sigmoid function i.e.

$$lsig(x,a,b)=(1)\Big/\left(1+e^{-a'x-b}\right) \qquad (3.5)$$

Ridge activation functions are extensively studied by many authors mainly [Cyb89][Hor91][Hor93][LLPS93][Kur92][KKK97][CL92]. One of the earliest works was reported by Hecht-Nielson [HeN87] he used an improved version of Kolmogorov’s theorem which states that every continuous function $f:[0,1]^n \rightarrow R$ can be written as:

$$f(x)=\sum_{h=1}^{2d+1}\varnothing_h\left(\sum_{k=1}^{d}\lambda^h\psi(x_k+\varepsilon h)+h\right), \qquad (3.6)$$

where the real λ and the continuous monotonically increasing function ψ are independent of f, the constant ε is a positive number and the continuous function $\varnothing_h, 1\le h\le 2d+1$, depending on f.This formulation represented a three-layered network where the h^{th} hidden node computes the function $z(h)=\sum_{k=1}^{d}\lambda^h\psi(x_k+\varepsilon h)+h,$ and the output nodes compute $\sum_{h=1}^{2d+1}\varnothing_h(z_h),.$

The first non-constructive proof was given by Cybenko in 1988 [Cyb89] he showed that if the ridge activation function σ is a continuous sigmoid, then the set of $\sum_{i=1}^{N}c_i\sigma(\theta_i^T x+b_i)$ is dense in $C(K)$ where $C(K)$ represents the set of all continuous functions defined on K, with respect to the uniform norm. According to [Cyb89], if

σ be any continuous sigmoid-type function e.g. $\sigma(\xi)=1/\left(1+e^{-\xi}\right)$, then any continuous real-valued function f on $\left[0,1\right]^n$ (or any other compact subspace of R^n) and $\xi>0$, there exists vectors $a_1,a_2,.......a_n,b,c_i$ & c_0 and a parameterized function $Y(.,a,b,c):\left[0,1\right]^n \rightarrow R$ such that: $\left|Y\left(x,a,b,c\right)-f\left(x\right)\right|<\xi$, for all $x\in\left[0,1\right]^n$ where

$$Y\left(x,a,b,c\right)=NN(X)=\sum_{i=1}^{N}c_i\left(a'_i\;X+b\right)+c_0 \qquad (3.7)$$

And $a_i\in R^n$ & c_i,c_0 & $b\in R$ where $a=\left(a_1,a_2,.......a_n\right),c=\left(c_1,c_2,.......,c_n\right)$ and $b=\left(b_1,b_2,.......b_n\right)$". Also note that a_i is a $dx1$ vector usually referred to as the direction of the ridge function. More precisely, he proved that Neural Networks with one hidden layer of sigmoid-activation neurons and an output layer of linear neurons are universal function approximators i.e. they can approximate any reasonable function to arbitrary accuracy. Since then many enhancements have been proposed in order to facilitate convergence, or impose limits on the network size in the terms of number of layers and number of hidden units required for a particular set of problems.

Hornik [Hor91] and [Hor93] further extended these results. In particular, in [Hor93] some theorems are presented which encompass almost all recent results on FNNs with ridge functions. The theorems state that three-layered FNNs are universal approximators under very weak assumptions on the activation functions, and suggest that nonpolynomiality of the activation function is the key property. He proves also that the approximation can be performed by weights bounded as close to '0' as required and that for some activation functions, a single threshold for the hidden layer is sufficient.

Another approach was used by Chui and Li [CL92] to prove universal approximation. They showed that if the ridge activation function σ is a continuous sigmoid and the direction vector θ satisfies some interpolation conditions, then the set of $\sum_{i=1}^{N} c_i \sigma\left(\theta_i^T x + b_i\right)$ is dense in $C(K)$ with respect to uniform norm. They constructed their proof by showing that it is possible to realize polynomials as a sum of ridge activation functions. Since polynomials are dense in $C\left(R^n\right)$, it follows that the three-layered Neural Networks are dense in $C(K)$ with respect to uniform norm.

One of the most elegant results on ridge activation was presented by Leshno *et al.* [LLPS93]. They relaxed the condition for the activation function to 'locally bounded piecewise continuous' (*i.e.*, if and only if the activation function is not a polynomial), thus embedding as special cases almost all the activation functions that have been reported in the literature.

3.3.2 Radial Basis Functions

Radial basis function network was first introduced by Broomhead and Lowe in 1988 [BL88]. A Radial basis function (RBF) can be represented as:

$$g(x,a,b) = k\left(\frac{x-a}{b}\right) \tag{3.8}$$

where g depends on a centre a and a smoothing factor b. $k(.)$ is usually assumed to be integrable on R^d, and $\int_{R^d} k(x)\,dx \neq 0$. The radial basis functions adopted in applications usually depend only on the distance between its current value and the center, i.e. $g(x,a,b) = k\left(\|x-a\|/b\right)$, where $\|.\|$ denotes the usual Euclidean norm. The Gaussian radial basis function $gauss(x,a,b) = \mathbf{e}-^{\left(\|x-a\|^2/b\right)}$ is a common example of such functions [ST98].

Radial basis functions received relatively less attention compared to ridge activation functions. However, there has been quite a few very promising results found in literature. The most well-known result was presented by Park and Sandberg [PS93][PS91]. They showed that if the Radial basis activation function used in the hidden layer is continuous almost everywhere, bounded and integrable on R^n, and the integration is not zero, then a three-layered Neural Network can approximate any function in $L_p\left(R^n\right)$ with respect to the L_p norm with $1 \le p < \infty$. They further extended their initial results and showed that if $g\left(x,a,b\right) = k\left(\|x-a\|/b\right)$ is a RBF, k is integrable on R^d and that $\int_{R^d} k\left(x\right)dx \ne 0$; then Σ_g^3 is dense in $L^1(R^d)$. Similar results were also reported by [PG90][GP90] they also showed that RBFs posses the universal approximation property.

Another important result on radial basis functions was given by Chen and Chen [CC95]. They proved that if the radial-basis activation function $g \in C\left(R\right) \cap S'\left(R\right)$(i.e., all those continuous functions such that $\int_R g\left(x\right)s\left(x\right)dx$ makes sense for all $s \in S\left(R\right)$) then the set of functions $\Sigma_{i=1}^N c_i g\left(a_i \left\|x-\theta_i\right\|\right)$ is dense in $C\left(K\right)$ if and only if ‘g’ is not an even polynomial. Unlike Park and Sandbergs formulation this setting does not require radial-basis function to be integrable; however, it does require the activation function to be a continuous distribution function, which is a strong requirement. Furthermore, a norm was imposed on $\left(x-\theta_i\right)$, therefore, the network structure is not considered to be general enough.

Another simple, but effective technique for approximating a continuous function of variables with an RBF network was presented by Schilling *et al.* [SCAa05]. The method uses an -dimensional raised-cosine type of RBF that is smooth, yet has compact support. The coefficients of the RBF network are low-order polynomial functions of the input. More recently, [HSS05] coins the idea of a new sequential learning algorithm for radial basis function (RBF) networks referred to as generalized growing and pruning algorithm for RBF (GGAP-RBF). They first introduced the concept of *significance* for the hidden neurons and then uses it in the learning algorithm to realize parsimonious networks. The growing and pruning strategy of GGAP-RBF is based on linking the required learning accuracy, with the *significance* of the *nearest* or intentionally added new neuron. Significance of a neuron is a measure of the average information content of that neuron. The GGAP-RBF algorithm can be used for any arbitrary sampling density for training samples, and is derived from a rigorous statistical point of view. Simulation results for bench mark problems in the function approximation area show that the GGAP-RBF outperforms several other sequential learning algorithms in terms of learning speed, network size and generalization performance, regardless of the sampling density function of the training data.

3.3.3 Recent Advancements on Function Approximation by Feedforward NNs

As highlighted in the introduction of this chapter, the focus of research in the filed of Function Approximation by Feedforward Neural Networks (FNNs) has shifted more towards development of new and efficient algorithms for function approximation problems. A lot of research has been carried out in this direction in the past few years. We will summarize some of the recent advancements in this section.

In [HCS06] turned their attention to the fact that in most Neural Network implementations, tuning all the parameters of the networks may cause learning complicated and inefficient, and it may be difficult to train networks with non-differential activation functions such as threshold networks. Unlike conventional Neural Network theories, they proved, using an incremental constructive method, that in order to let Single Layer Feedforward Neural Network (SLFNN) as universal approximators, one may simply randomly choose hidden nodes, and then only need to adjust the output weights linking the hidden layer and the output layer. In such SLFNNs implementations, the activation functions for additive nodes can be any bounded non-constant piecewise continuous functions: and the activation functions for RBF nodes can be any integrable piecewise continuous functions $g : R \to R$ and $\int_R \ g(x)dx \neq 0$. The proposed incremental method is efficient not only for SLFNNs with continuous (including non-differentiable) activation functions but also for SLFNNs with piecewise continuous (such as threshold) activation functions.

In [ZP08] the authors investigated function approximation by using radial basis function network and Wavelet Neural Network (WNN). They used different types of basis functions as the activation function in the hidden nodes of the radial basis function network and the wavelet Neural Network. The performance is compared by using the normalized square root mean square error function as the indicator of the accuracy of these Neural Network models. They showed that WNN performs better in approximating a periodic function, whereas RBF Netwoks yields higher accuracy in estimating exponential function.

The authors of [GTMc08] presented a model with wavelet-like functions in the functional form of a Neural Network which is used for function approximation. They argued the fact that the scale parameters are mainly used, neglecting the usual translation parameters in the function expansion. They then investigated two training operations; first one consists of optimizing the output synaptic weights and the second one on optimizing the scale parameters hidden inside the elementary tasks. Building upon previously published results, it was found that if $(p+1)$ scale parameters merge during the learning process, derivatives of order p will emerge spontaneously in the functional basis. It is also found that for those tasks which induce such mergings, the function approximation can be improved and the training time reduced by directly implementing the elementary tasks and their derivatives in the functional basis.

One of the most significant achievements in the recent past is the idea of ‘Extreme Learning Machine (ELM)’ [HC07][HCS06][HZS06] which does not require any iterations in order to learn network parameters, and hence considerably reduces the network training time when compared to traditional BP algorithm. Although the testing performance of Standard NN models is better than that of the ELM but in terms of training time it is quite an efficient algorithm.

3.4 Neural Network Ensemble Methods

Along with other advancements in Neural Networks, ANN ensemble methods have also become very popular amongst Neural Network researchers in a variety of ANN application domains. We can think of a Neural Network ensemble as a learning paradigm where a collection of finite number of Neural Networks is trained for the same task. It is well-known that the generalization ability of Neural Networks, i.e., training many Neural Networks and then combining their predictions are better then a single NN model.

In general, a Neural Networks ensemble is constructed in two steps, i.e., training a number of component Neural Networks, then combining the component predictions. Using $f_1..........f_M$ to denote M individual NNs, a common example of ensemble for regression problem is, $f_{reg}\left(x\right)=\sum_{i=1}^{M} w_i f_i\left(x\right),$ where $w_i > 0$ is the weight of the estimator f_i in the ensemble.

Neural Network based ensemble methods was first proposed by Hansen and Salamon's (see [HS90]). In their work they showed that the generalization ability of a Neural Network can be significantly improved through ensembling a number of Neural Networks. Since then Neural Network Ensemble methods have been widely used to improve the generalization performance of the single learner.

Last decade has seen ever increasing interest in ensemble learning methods for NNs. There has been much literature published focusing on these methods, we can broadly classify these methods as either bagging and boosting or stacking. There are other popular ensemble learning techniques such as Mixtures of Experts [JJ94], Random Subspace [Hor98], Random Forests [Bre01] and Negative Correlation Learning [LY97][LY99]. However the application of Neural Networks as ensemble methods has been mainly studied in bagging and boosting framework. As the main objective of this work is to investigate approximation capabilities of Neural Networks therefore we will give a brief explanation of these two methods in the following section.

3.4.1 Bagging

Bagging is the common term used for a popular ensemble learning method called ‘‘Bootstrap Aggregation’’. This technique was proposed by Breiman [Bre96]. This approach is based on the bootstrap statistical resampling technique proposed by Efron et al. [ET93], to generate diverse training sets that are used to train the members composing an ensemble. Suppose the training set T consists of m instances. Each instance is assigned a probability of $1/m$, and the training set of a member network, is generated by sampling with replacement m times from the original training set T, using these probabilities. Thus many cases in T may be repeated several times in a member network, while others may be left out. This process is repeated, and each member network is generated with a different random sampling of the original training set. In [Bre96] the author concluded that bagging is effective on ‘‘unstable’’ learning algorithms. Predictors such as ANNs and regression trees are suitable for bagging. There has been other work in bagging [CC99][Zha99], which studied bagging in the context of ANNs, and concluded that model generalization ability can be significantly improved.

3.4.2 Boosting

Boosting has now become quite a familiar term in machine learning theory. We can define boosting, or leveraging, in simple terms, as a general way of improving the accuracy of any learning algorithm [FHT00]. Historically most of the work in the field of boosting or leveraging methods has concentrated on classification problems see [FS97], and related leveraging techniques [Bre98][Bre99][Fri01]. In comparison to regression/function approximation problems (i.e. the output variable *‘y’* is continuous), the application of boosting methods to classification problems have been well-studied, empirically tested and have good theoretical bounds and guarantees.

Boosting algorithms was first proposed by [Dru97]. They achieve improved performance by producing a series of predictors trained with a different distribution of the original training data. The algorithm trains the first predictor with the original training set, and the training set of a new predictor is assembled based on the performance of the previous predictors. The learning patterns whose predicted values obtained from the previous predictor differ significantly from their observed values are adjusted with higher probability of being sampled, so they will have a greater chance of appearing in the new training set than those correctly predicted. Thus different predictors are specialized in different parts of the observation space. A popular example is the AdaBoost algorithm [FHT00], which iteratively builds a classifier as a linear combination of the so-called weak classifiers. At each step, a new weak classifier is added optimizing the classification error rate with a new weighting on training samples.

3.4.3 Boosting for Regression Problems

Although less investigated, there have been quite a few very promising attempts to address the issue of boosting for regression/ approximation problems. Just like boosting for classification [FHT00] were the first ones to come up with boosting algorithms for regression problems. The much famous Adaboost.R was the first attempt to address this issue. The AdaBoost.R algorithm [FS97] attacks the regression problem by reducing it to a classification problem. To fit a set of (x, y) pairs with a regression function, where each $y \in [-1,1]$, AdaBoost.R converts each (x_i, y_i) regression example into an infinite set of $\left((x_i, z) \tilde{y} \right)$ pairs, where $z \in [-1,1]$ and $\tilde{y} = sign(y_i - z)$. The base regressor is given a distribution D over $(x_i - z)$ pairs and must return a function *f (x)* such that its weighted "error" $\sum_i \left| \int_{y_i}^{f(x_i)} D(x_i, z)\, dz \right|$ is less than 1/2.

Experimental results have shown that Adaboost.R and its variants, see [RMR99][BCP97][FS96][Sch90] are quite effective. However, performance of these models degrades due to the following two reasons. Firstly, the expansion of each instance in the regression sample into many classification instances. Although the integral above is piecewise linear, the number of different pieces can grow linearly in the number of boosting iterations. Secondly, the "error" function that the base regressor should be minimizing is not (except for the first iteration) a standard loss function. Furthermore, the loss function changes from iteration to iteration and even differs between examples on the same iteration. Therefore, it is difficult to determine if a particular base regressor is appropriate for AdaBoost.R.

3.4.4 Gradient-based Boosting

One of the most significant works in this area was presented in [FHT00]. They showed how adaptive boosting algorithms can be derived as gradient decent algorithms. This approach allows all model parameters to optimize one single common objective function, in comparison to traditional boosting methods that work by repeatedly calling weak (or base) learning method on modified samples to obtain different base rules. These are then combined into a master rule or hypothesis. The algorithm proposed in [Bre99] used the master algorithm to construct y_i values for each data-point x_i equal to the (negative) gradient of the loss of its current master hypothesis on x_i . The base learner then finds a function in a class f minimizing the squared error on this constructed sample.

As with traditional boosting methods, this view was well received in research community, and many authors' derived algorithms targeting classification problems. The work in [ZP01] was one of the first attempts to take advantage of this approach and extended it to tackle regression problems. They proposed a novel objective function for regression problems which lead to a simple boosting algorithm. They also proved that their method reduces training error when compared with other regression methods.

They used $J_T = \frac{1}{n}\sum_{i=1}^{n} \exp\left(\sum_{t=1}^{T} c_t \left[\left(f_t\left(\mathbf{x}^i\right) - y^i \right)^2 - \tau \right] \right)$ as objective function in [ZP01], where parameter C_t (combination co-efficients) and w_t (model weights), can now be derived using this objective function. They used the same objective function in the *WeakLearn* procedure, as the new hypothesis is the step in function space in the direction of steepest descent of this objective [ZP01]. This allows parameter C_t (combination co-efficients) and w_t , (model weights) to be derived using this objective function. The constant τ is used to distinguish between correct and incorrect responses and is chosen in problem-specific manner. As highlighted in [ZP01], this formulation allows each hypothesis to be trained to minimize the squared error of a weighted distribution. This also allows the objective function to be determined by simply re-weighting the training distribution. Another exciting fact is that the new weights of a training example only depend on its old weight and error produced in the last iteration.

This algorithm is presented with training set examples in the form $(\mathbf{x}_1, y_1).......(\mathbf{x}_n, y_n)$ where $y \in \mathbb{R}$, and the initial distribution of model parameters is chosen according to $p_1(\mathbf{x}^i) = p_1^i = w_1^i = \frac{1}{n}$. The next step in their algorithm is the call to *WeakLearn* procedure that produces a hypothesis $f_t(\mathbf{x})$ whose accuracy on the training set is judged according to the cost function *J* above. The algorithm then repeatedly calls the *WeakLearn* procedure on modified distributions in order to minimize J_t with distribution $p_{t.}$ On every call to the *WeakLearn* the algorithm checks the error 'ξ_t' and accepts iff $\xi_t = \sum_i p_t^i \exp\left[\left(f_t(\mathbf{x}_i) - y_i\right)^2 - \tau\right] < 1$. The combination coefficient C_t is then set to minimize J_t using simple line search. In order to generate next training distribution this algorithm modifies the model parameters according to $w_{t+1}^i = w_t^i * \exp\left(c_t\left[\left(f_t(\mathbf{x}^i) - y^i\right)^2 - \tau\right]\right)$, where $p_{t+1}^i = w_{t+1}^i / \sum_j w_{t+1}^j$ and finally estimate output 'y' on input **x** according to $\hat{y} = \sum_t c_t f_t(\mathbf{x}) / \sum_t c_t$.

Two important facts to be noted here is the way in which initial distribution is chosen i.e. $p_1(\mathbf{x}^i) = p_1^i = w_1^i = \frac{1}{n}$ and how the model parameters are updated by the *WeakLearn* procedure. In this work they used single hidden layer Neural Network (NN) as hypothesis and backpropagation as the learning procedure. In fact this setting has been a popular choice for regression boosting algorithms due to Neural Networks function approximation capabilities [FHT00][DH02].

3.5 Common Issues in FNNs & Problem Description

As shown in the above section, Feedforward Neural Network (FNN) architecture has been successfully applied to 'function approximation' problems in many real-world application domains. However this model has certain limitations. The most commonly faced situation is the problem of local minima i.e. the tendency of the model to get trapped in undesirable local minima in order to reach the global minimum of a very complex search space. Secondly, training of FNN is very time consuming task, due to the slow convergence of FNN training algorithms. Thirdly, FNN also fails to converge when high nonlinearities exist.

It is also important to understand that these "universal approximation" proofs are commonly used to justify the notion that Neural Networks can "do anything" (in the domain of function approximation). What is not considered by these proofs is that networks are simulated on computers with finite accuracy. And the fact that approximation theory results cannot be used blindly without consideration of numerical accuracy limits, and that these limitations constrain the approximation ability of Neural Networks, see [WGG95].

In addition to these limitations; the most important observation with reference to this work is the fact that almost all NN approximation schemes proposed so far are designed to approximation functions on continuous input spaces $U_i = [\alpha_i, \beta_i]$, i.e. the input-vector '$\mathbf{X}$' takes on continuous values [ZK08][PG90][ZGKL05][SM02]:

$$X = (x_1, x_1, ...x_n) \in R^n \qquad (3.9)$$

Another deficiency in these approximation schemes is that they are computationally very expensive, because one of the underlying assumptions is the availability of sufficiently large number of neurons in hidden layer(s). It is also seen in many practical applications that the size of the network increases very fast (some times exponentially) when it encounters new information in form of new examples or additional dimensions (inputs) or when some desired precision is to be achieved [Bei98].

Although these schemes can be used for approximation of functions on discrete input and mixed input spaces (i.e., some input variables are discrete values where other take continuous values), these schemes, when applying to approximate functions on discrete or mixed input spaces, are less effective and more complicated than necessary due to not taking into account special features of discrete input spaces [ZK08][ZGKL05].

3.6 Special Features Of Functions Defined On Discrete Input Spaces

When we say special features of discrete input spaces, what exactly do we mean by this? This is the issue of prime importance with regards to this research. The most apparent of these special features is the property of discrete variables to take on finite number of states, or in other words the points are isolated from each other in some sense.

3.6.1 Flexible Hierarchical Structure Property

Another very important feature of functions defined on discrete input spaces is their flexible (arbitrarily separable) hierarchical structure. As described in [ZK08], consider the following function:

$$G(x_1, x_2, x_3\ x_4, x_5) = 10\sin(\pi x_1 x_2) + 20(x_3 - 0.5) + 10x_4 + 5x_5 \qquad (3.10)$$

Let g_1 and g_{2j} $(j = 1,2)$ be

$$g_1(y_1, y_2, x_5) = y_1 + y_2 + 5x_5 \quad g_{2,1}(x_1, x_2) = \sin(\pi x_1 x_2) \quad g_{2,2}(x_3, x_4) = 20(x_3 - 0.5) + 10x_4 \quad (3.11)$$

Then

$$G(x_1, x_2, x_3\ x_4, x_5) = g_1[g_{2,1}(x_1, x_2), g_{2,2}(x_3, x_4), x_5] \qquad (3.12)$$

That is, $G(x_1, x_2, x_3, x_4, x_5)$ can be represented as a function with a hierarchical structure given in figure 3.1.

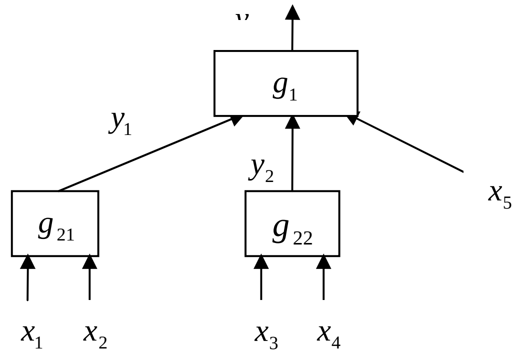

Figure 3.1 An Example Of A System With A Two Level Of Hierarchical Structure.

Figure 3.1 represents a two-level hierarchical structure. In the same paper [ZK08] authors also showed that, if $g_{2,j}$ is also a function with a hierarchical structure, then further levels of hierarchical structure are also possible. In other words, multi-level hierarchical structure for $G(x_1,..., x_n)$ is possible. Further in the paper they proved that, for a function with hierarchical structure, its hierarchical structure is not unique. This is illustrated as follows:

Consider the function $G(x_1, x_2, x_3, x_4, x_5)$ given above. If g_1 and $g_{2,j}$ $(j = 1,2)$ are chosen to be,

$$g_1(y_1, y_2) = y_1 + y_2, g_{2,1}(x_1, x_2) = \sin(\pi x_1 x_2), g_{2,2}(x_3, x_4, x_5) = 20(x_3 - 0.5) + 10x_4 + 5x_5 \quad (3.13)$$

Then,

$$G(x_1,\ x_2, x_3\ x_4,\ x_5) = g_1[g_{2,1}(x_1,\ x_2),\ g_{2,2}(x_3,\ x_4, x_5)] \tag{3.14}$$

That is, $G(x_1, x_2, x_3, x_4, x_5)$ can be represented as a function with a hierarchical structure which will be different from the hierarchical structure given in figure 3.1

The authors in [ZK08] also discussed a special case of the functions with hierarchical structure that is, when input variables in each sub-functions g_1 and $g_{2,j}\ (j = 1,2,...,m)$ are disjointed form each other. We can easily see that the input variable sets for g_1 and $g_{2,j}\ \ (j = 1,2)$ in equation (3.12) are disjointed, therefore the function $G(x_1,\ x_2,\ x_3\ x_4, x_5)$ given in equation (3.12) is one with separable hierarchical structure. On the other hand, the input variable sets for g_1 and g_{2j} $(j = 1,2)$ are also disjointed. That is, $G(x_1,\ x_2,\ x_3\ x_4, x_5)$ has another separable hierarchical structure. This shows that $G(x_1,\ x_2,\ x_3\ x_4, x_5)$ can be represented by different separable hierarchical structures. All the facts discussed above are formally summarized in the form of a theorem as follows, please refer to [ZK08] for a detailed proof of this theorem.

Theorem 1: Let $G(X)$ *be a MISO (multiple input single output) function given by* $y = G(X) = G(x_1, x_2,..., x_n)$, *where* $y \in V \subset R$ *is the output variable and* $X = (x_1, x_2,..., x_n) \in U = U_1 \times U_2 \times ... \times U_n \subset R^n$ *is the input variable vector in which* $x_i \in U_i$ *and* $U_i = \{u_{i,k} \mid u_{i,k} \in R,\ k = 1,2,..., N_i\}$, *in other words, input variable* x_i *takes discrete values. Then, for any disjoint grouping of the input variables* $\{x_1, x_2,..., x_n\}$ *into* $m+1$ *groups* G_1 *and* $G_{2,j}\ (j = 1,2,..., m)$ *satisfying the following conditions:*

$$G_1 = \left\{x_{i_1^{(1)}},..., x_{i_{n_1}^{(1)}}\right\} \qquad G_{2,j} = \left\{x_{i_1^{(2,j)}},..., x_{i_{n_{2,j}}^{(2,j)}}\right\} \quad j = 1,2,..., m \tag{3.15}$$

Where,

$G_1 \quad G_{2,j} = \varnothing, j = 1,2,...,m$ and $G_{2,j} \quad G_{2,j'} = \varnothing \;\; j \neq j', \quad j, j' = 1,2,...,m$ (3.16)

$G_1 \quad G_{2,1} \quad .... \quad G_{2,m} = \{x_1, x_2, ..., x_n\}$ (3.17)

then there exist functions g_1 and $g_{2,j}$ $(j = 1,2,...,m)$ [in particular, $g_{2,j}$ $(j = 1,2,...,m)$ can be as simple as linear functions or fuzzy systems] such that $G(X) = g_1[g_{2,1}(X_{2,1}),..., g_{2,m}(X_{2,m}), X_1]$. That is, any MISO function on discrete spaces has the arbitrary separable hierarchical structure.

This is a very interesting theorem and the results obtained have some significant implications on NN approximation schemes. The most significant of them are:

1. If $G(X)$ can be represented as a function with the given G_1 and $G_{2,j}$ $(j = 1,2,...,m)$ as its hierarchical structure is related to the existence of one-to-one mappings on discrete spaces. These one-to-one mappings not only exist but also can be realized by using some very simple functions.
2. For a discrete space U, there exist some simple functions which form one to one mappings from U to R. This is a property which holds only on discrete spaces but not on continuous spaces. This is because no one-to-one mapping from a multi-dimensional continuous space $U = \times_{i=1}^{n}[\alpha_i, \beta_i]$ $(n > 1)$ to R can be continuous (see [ZK08] for detailed discussion). As no continuous function can be found to form one-to-one mapping from a multi-dimensional continuous space to R, it is impossible to find a simple function which is a one to one mapping from multi-dimensional continuous space U to R.

3.7 Summary

This chapter presents a detailed literature review of our selected area of research. First of all it introduces the Function Approximation problem, followed by a detailed analysis of approximation and representation capabilities of Feedforward Neural Networks. A systematic review of related work on Universal Function Approximation Property has been presented. Recent advancement in this field has also been highlighted in this chapter, followed by a discussion on Neural Network based ensemble methods with a particular emphasis on application of Neural Networks in regression boosting frame work. We have also presented common issues and a formal problem description in this chapter. A comprehensive analysis of discrete nature of input spaces and 'Arbitrarily Separable Hierarchical Structure Property' of functions defined on discrete input spaces is also presented in this chapter.

CHAPTER 4

SIMPLIFIED NEURAL NETWORK (SNN) APPROACH AND ALGORITHMS

The special features of discrete input spaces discussed in the previous chapter, the capability of Feedforward Neural Networks to approximate any function arbitrarily well and the lack of systematic results focusing on discrete input spaces, are the main reasons behind the initiation of this research. The main objective of this research is to propose more simplified algorithms based on simplified NN approximation schemes that make use of these properties of discrete input spaces, without compromising on accuracy or generalization capabilities of the existing NN models and techniques.

4.1 The Simplified Neural Network (SNN) Approach

As we already know, the multilayer feedforward networks are usually arranged in many layers; input, output and one or more hidden layers. We also know that any mapping of the form $f : R^n \to R^m$ can be computed by m mappings $f_k : R^n \to R$ therefore it is sufficient to focus on networks with one output unit only [LLPS93]. This section gives a detailed analysis of simplified NN approach and shows how simplification is achieved with these schemes. In the following, it is always assumed that the input spaces are discrete ones i.e. $U_i = \{\alpha_{ij} \mid j = 1,2,..., N_i\}$.

We begin our discussion with a formal definition of standard Neural Network. In line with the famous Cybenko theorem [Cyb89] we can define a standard NN as:

$$y = NN(X) = \sum_{i=1}^{N} c_i \sigma(a_i^\tau X + b_i) + c_0 \tag{4.1}$$

where $X = (x_1, x_2, ..., x_n)$ are input variable, $X \in U \ = U_1 \times U_2 \times ... \times U_n \subset R^n$ which are input space, $y \in R$ is the output variable, τ is the vector transpose, $\sigma(.)$ is the activation function and the parameters $c_0 \in R$, $c_i \in R, a_i \in R^n$, and $b_i \in R$ $(i = 1,2,...,N)$. As described in [ZGKL05], the total number of parameters [i.e., $c_i \in R, a_i \in R^n$, $b_i \in R\,(i = 1,2,...,N)$ and $c_0 \in R$] is $(n+2)N+1$. For nonlinear complex function approximation, a large N is needed and very often N is subjected to exponential growth with the increase in dimension of n . As a result, a large number of parameters are needed in order to achieve good approximation accuracy.

To overcome these computational expanses new schemes are required which should be able to exploit the function approximation capabilities of Neural Networks for discrete input spaces.

4.1.1 Simplified Neural Networks (SNN)

We can define (see [ZGKL05] for a detailed discussion) a simplified Neural Network (SNN) as shown in equation (2.2):

$$y = SNN(X) = \sum_{i=1}^{N} c_i \sigma[\alpha_i (\alpha^\tau X + \beta) + \beta_i] + c_0 \tag{4.2}$$

where $c_0 \in R$, $c_i \in R$, $\alpha_i \in R$, $\beta_i \in R$ $(i = 1,2,...,N)$ and $\alpha \in R^n$, $\beta \in R$.

Let $z = L(X) = \alpha^\tau X + \beta$ (4.3)

and $y = NN_1(z) = \sum_{i=1}^{N} c_i \sigma(\alpha_i\, z + \beta_i) + c_0$ (4.4)

Then the proposed SNN given in (4.2) can be rewritten as follows:

$$SNN(X) = NN_1[L(X)] \tag{4.5}$$

In other words, the proposed SNN can be presented as a composition function of a linear function $L(X)$ given in (4.3), and one dimensional standard NN $NN_1(z)$ given in (4.4). The difference between the above simplified NN from the standard NN is that it uses a common linear function $a'X+b$ rather than $a_i^{'}X+b_i(i=1,2,...,N)$, which results in significant reduction of parameters required for the model. Such a simplified NN benefits through the following advantages:

1. A simplified NN requires approximately (3N + n + 2) parameters in most of the cases.
2. SNNs are more effective in overcoming the model over-fitting which is often the case with standard NN models. This is due to the fact that in the standard NNs, adding a new neuron [i.e., add an item $c_i\sigma(a_i^\tau X+b_i)$ in (4.1)] means adding $n+2$ parameters. As a result, it is an often faced situation in NN modeling, that adding one neuron causes overfitting but without adding results in underfitting, especially in the case where n is large but only a limited training data available. However, in SNNs, adding a new dimension or neuron in hidden layer means adding an item $c_i\sigma(\alpha_i z+\beta_i)$ which only adds three parameters. As a result, SNNs allow finer adding model parameters to overcome the model overfitting and underfitting, especially in the high dimension (i.e. large n) case.
3. More simplified learning algorithms can be developed. For example, in some cases, multi-dimension NN learning problem can be transformed to one dimensional NN learning problem and then the corresponding learning algorithms can be much simpler.

In the light of the above discussion, and advantages of simplified NN, we propose two algorithms which can be used with discrete input spaces for function approximation problems. As described in [ZGKL05], any algorithm developed under the assumptions discussed above have the universal approximation property and are general enough to approximate arbitrarily well any function defined on discrete input spaces. These facts are formally derived from the following two theorems.

Theorem 2: Let $G(X)$ *be a function defined on discrete space* $U = \prod_{i=1}^{n} U_i$ *. Then for any given* $\varepsilon > 0$ *, there exists a simplified NN* $y = NN(X) = \sum_{i=1}^{N} c_i \sigma(a' X + b) + c_o$ *such that,* $\| G - NN \| = \max_{X \in U} | G(X) - NN(X) | < \varepsilon$ *.*

Remark 1: The above theorem shows that SNNs can approximate any function on a discrete space to any degree of accuracy. In other words, SNNs, in spite of their simplified formula, reserve the universal approximation property of standard NNs and therefore are generally applicable for function approximation in discrete spaces. This theorem is very important with reference to this work, therefore a detailed proof of this theorem as appeared in [ZGKL05] is also included as appendix-C. Following the discussion in section 3.5 we can now introduce the following Lemma:

Lemma 1: Given a discrete input space $U = \prod_{i=1}^{n} U_i$ *, there exists a linear function* $a' X + b$ *which is one to one mapping on* $U = \prod_{i=1}^{n} U_i$ *.*

Theorem 3: Let $G(X)$ *be a function defined on discrete space* $U = \prod_{i=1}^{n} U_i$ *and* $L(X) = a' X + b$ *is any one to one mapping defined on* $U = \prod_{i=1}^{n} U_i$ *. Then for any given* $\varepsilon > 0$ *, there exists a simplified NN using* $L(X) = a' X + b$ *as the common linear function such that the simplified NN* $y = SNN(X) = \sum_{i=1}^{N} c_i \sigma\left[a' X + b) \right] + c_o$ *satisfies,* $\| G - NN \| = \max_{X \in U} | G(X) - NN(X) | < \varepsilon$ *.*

Remark 2: The above theorem shows that, for any given one to one linear function $L(X)$, simplified NN can be constructed based on $L(X)$ to form universal approximators.

What follows is a detailed description of these algorithms, whereas the derivation of backpropagation algorithm for these simplified algorithms is also presented in the following section.

4.1.2 Simplified NN Algorithm-I

1. *Initialisation*:
   - *a.* Identify a one to one linear mapping $z = L(X) = a'X + b$ on the input space that is both one to one and onto.
   - b. Training data transformation:
     Transform the training data $\{[y(t), X(t)]; t = 1,2,...,M\}$ to $\{[y(t), z(t)]; t = 1,2,...,M\}$ by using $z = L^*(X) = a'X + b$;
   - c. By using the optimisation algorithm such as gradient descent algorithm or other algorithms in order to identify,

     $$y = NN(z) = \sum_{i=1}^{N} c_i \sigma\left[\alpha_i z + \beta_i)\right] + c_o \,.$$

     Notice that this is a single variable function approximation;
   - d. Form the initial simplified NN as :

     $$y = NN(z) = \sum_{i=1}^{N} c_i \sigma\left[\alpha_i z + \beta_i)\right] + c_o$$

2. *Iterations:* Using the back-propagation algorithm to update the model.

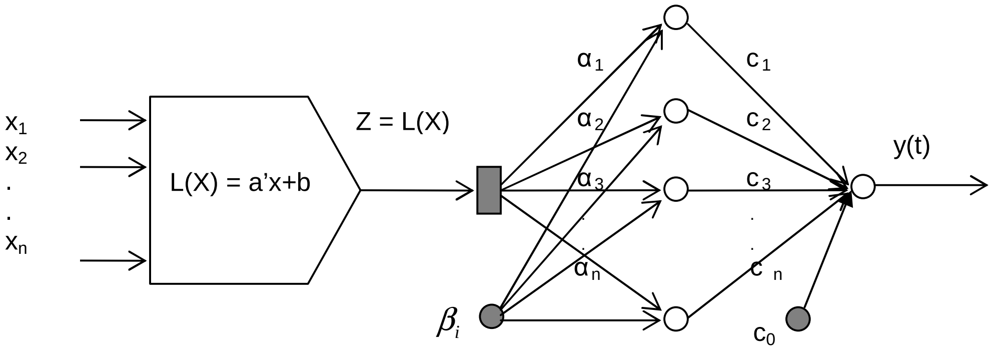

Figure 4.1 Architectural representation of Simplified NN Algorithm-I

4.1.3 Simplified NN Algorithm-II

1. *Input*: $S = (\mathbf{x}_1, y_1)\dots\dots(\mathbf{x}_n, y_n)$ where $y \in \mathbb{R}$, and training iterations T.
2. *Initialize:* The initial distribution of model parameters $p_1(\mathbf{x}^i)$ is chosen according to $p_1(\mathbf{x}^i) = p_1^i = w_1^i = \frac{1}{n}$.
    a. Compute the linear approximation . $z = L^*(X) = a'X + b$;
    b. Training data transformation:
    Transform the training data $\{[y(t), X(t)]; t = 1, 2, ..., M\}$ to $\{[y(t), z(t)]; t = 1,2,..., M\}$ by using $z = L^*(X) = a'X + b$;
    c. By using the optimisation algorithm such as gradient descent algorithm in order to identify,
    $y = NN(z) = \sum_{i=1}^{N} c_i\sigma[\alpha_i z + \beta_i)] + c_o$. Notice that this is a single variable function approximation;
    d. Form the initial simplified NN as :
    $$y = NN(z) = \sum_{i=1}^{N} c_i\sigma[\alpha_i(a'X + b) + \beta_i] + c_o$$
3. *Iterations:* Using the back-propagation algorithm to update the model.

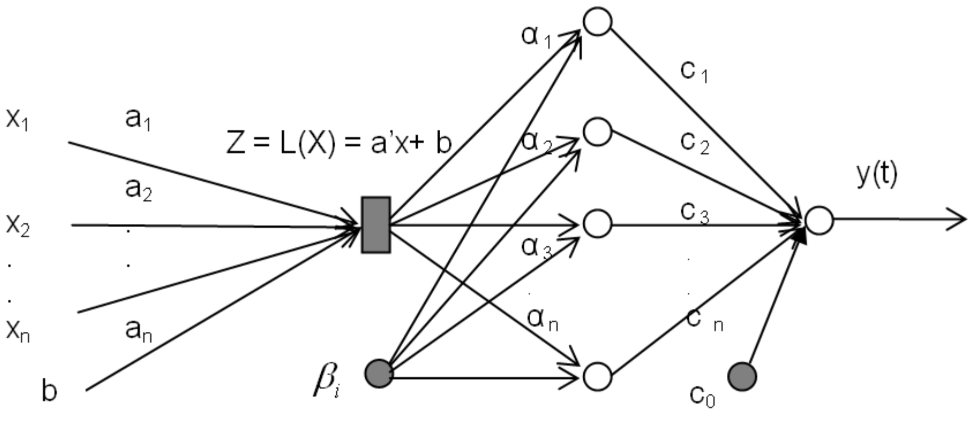

Figure 4.2 Architectural Representation of Simplified NN Algorithm-II

The basic difference in both the algorithms is their initialisation conditions. Algorithm-I uses a simple linear mapping to transform the input space in to a unidirectional one, this is a very simple method as by using the one-to-one linear mapping $L(X)$, the approximation problem is transformed to a simple learning problem of a single variable NN, figure 4.1 gives an architectural realisation of such a SNN. The first algorithm is based on the proof of Theorem 2 (See Appendix-C) which includes two steps; the first step is to find a one-to-one linear mapping $L(X)$ from U to R and then one dimensional function $g(z) = G[L^{-1}(z)]$ or $g[L(X)] = G(X)$ can be defined; the second step is using the available data $\{(X_t, y_t) | t = 1,2,..., N\}$ to get a set of training data for function $g(z)$ as $\{(z_t, y_t) | z_t = L(X_t), t = 1,2,..., N\}$ and then, for $g(z)$, apply the learning algorithms of the standard NN to find one dimensional NN approximator $NN_1(z)$ with the required approximation accuracy. Finally the SNN approximator can be obtained by $SNN(X) = NN_1[L(X)]$. Theoretically, this is a very simple method as by using the one-to-one linear mapping $L(X)$, the approximation problem is transformed to a simple learning problem of a single variable NN.

In the case where the number of input variables and the possible values of each input variables are small, then this is a good algorithm in practice due to its simplicity. However, this method is not suitable for high dimension (i.e., many input variables or n is large) with each input variable having many possible values (i.e., N_j is large). The is mainly due to the fact that; as the total number of all possible values of input vector $X = (x_1, x_2,..., x_n)$ are $\prod_{i=1}^{n} N_i$, it means that the total number of the possible function values of one-to-one mapping $z = L(X)$ is $\prod_{i=1}^{n} N_i$. When n and N_i $(i = 1,2,..., n)$ are large, this is impossible as all these possible values are beyond the representation accuracy of float numbers in today’s computers. Therefore, in the case when n and N_i $(i = 1,2,...,n)$ are large, the implementation of this algorithm, requires more specialised methods e.g. use of Extended Simplified Neural Networks (ESNN) as described in [ZGKL05]. The use of ESNN for such modeling problems is not discussed any further and remains a further research objective.

The second algorithm begins with initialising model parameters to $p_1(\mathbf{x}^i) = p_1^i = w_1^i = \frac{1}{n}$; the training data is then transformed $\{[y(t), X(t)]; t = 1, 2, ..., M\}$ to $\{[y(t), z(t)]; t = 1,2,..., M\}$ into single dimension by using a linear approximation $z = L^*(X) = a' X + b$. However, unlike algorithm-I, two additional parameters (see figure 4.2) are added to the one-dimensional Neural Network $SNN(X) = NN_1[L(X)]$. The second step in the algorithm is the application of the gradient descent optimisation algorithms to minimise, $E = \frac{1}{2}\sum_{t=1}^{T}\left[y_t - SNN(X_t)\right]^2$, where $SNN(X)$ is given in algorithm-II step 1.d, with the parameters $\{c_i, \alpha_i, \beta_i, \alpha, \beta, c_0 \mid i = 1,2,...,N\}$ to be identified. In this algorithm, it is not required that $z = L(X) = \alpha^\tau X + \beta$ is a one-to-one mapping (noticing that one-to-one mapping is a sufficient but not the necessary condition), rather parameters α and β are tuned by the learning algorithm to meet the approximation requirement. This algorithm is more complicated than the first one but likely it will handle high dimensional modeling situation [ZGKL05]. Architecture of such a SNN resembles the figure 4.2.

In the standard NNs we use to have weight connections i.e. a_{ij}, coming from each individual input to every hidden layer node. However, in the case of SNN of algorithm-II we transform the input vector $\mathbf{X}$ into one dimension using a linear function ($a' X + b$). The result of this setup is a scalar weight matrix representing the hidden layer weight connections rather than a vector representing all the hidden layer weights. The architecture shown in figure 4.2 also represents a one dimensional Neural Network because now we don’t have to update all the hidden layer weights associated with each neuron; instead only two parameter per neuron will be updated in the hidden layer i.e. the common weight connection and the bias attached to it.

An exciting fact to be noted here is the way the data is transformed into one dimension using the linear approximation $z = L^{*}(X) = a' X + b$. Such a linear approximation can also be found by applying multiple regression techniques. A multivariate or least squares fit model of the data is usually represented as $z = \alpha_0 x_0 + \alpha_1 x_1 + \alpha_2 x_2 .... \alpha_n x_n$. Therefore we have to solve for unknown coefficients $\alpha_0, \alpha\ , \alpha_2 .... \alpha_n$, by performing a least squares fit. We can then use these estimates to initialize network parameters to transform the training data before passing it on to our one dimensional NN. However, in algorithm-II, we have not adopted this approach since in standard NN models we do not perform any such data pre-processing and therefore the comparison of both the methods may be biased. Instead we will look at this approach i.e. use of multiple regression methods for data transformation in regression boosting frame work, see section 5.

The Algorithm-II presented above can be easily extended to be viewed as a regression boosting method for functions defined on discrete input spaces i.e. $X \in U = \prod_{i=1}^{n} U_i$. With similar error bounds and convergence guarantees as presented in [HZ09]. Based on these exciting facts we propose a new simplified approach to regression boosting for functions defined on discrete input spaces. We will refer to our approach as Simplified Regression Boosting (SRB) for discrete input spaces. Following is a step by step description of this approach for functions defined on discrete input spaces.

4.2 Backpropagation Algorithm for Simplified NNs

In-line with the definition and architectural representations of Simplified Neural Network algorithms, we can now define our simplified network parameters as: $X = (x_1,...,x_n)$ are input variable, $X \in U = \prod_{i=1}^{n} U_i$ which are input space, $y^m \in R$ is the output variable, *'m'* is the layer index and denotes output layer, the index of the layer just below output layer will be *'m-1'* and *'m-2'* and so on. α_i are the connection weights associated with input layer to hidden layer and in the Simplified NN case it will be represented as, $\alpha_i = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ . \\ . \\ . \\ \alpha_n \end{bmatrix}$ (4.6)

Notice the change in definition of this network parameter; in the case of standard NN this parameter was a $dx1$ vector where as in simplified case it is replaced by a scalar parameter.

β_i is the bias attached to hidden layer neurons, where as c_i & c_0 are the connection weight and bias from hidden layer to output layer respectively. 'σ' is the activation function and in the case of sigmoidal neurons it will be $\sigma(x) = \frac{1}{1+\exp^{-x}}$, and in the case of linear neurons it will be $\sigma(x) = x$. The output of hidden layer neuron j in the layer $m-1$ can therefore be computed as;

$$y_j^{m-1} = \sum_{i=1}^{N} \sigma\left[\alpha_i z + \beta_i)\right] \quad , \quad y_j^{m-1} = \sum_{i=1}^{N} \sigma\left[\alpha_i (a' X + b) + \beta_i)\right] \qquad (4.7)$$

The net input to our hidden layer neurons will be:

$$net_j^{m-1} = \sum_{i=1}^{N} \sigma[\alpha_i z + \beta_i)], \; net_j^{m-1} = \sum_{j=1}^{N} \alpha_i (a' X + b) + \beta_i \qquad (4.8)$$

The output of the last layer will be the same as its net input since the output layer uses the linear neurons. So the output of neuron 'i' in the layer 'm' (which is last layer) will be:

$$y_i^m = \sum_{i=1}^{n} c_i^m \; \sigma\left(y_j^{m-1}\right) + c_0^m \qquad (4.9)$$

where y_j^{m-1}can be computed as in equation (4.7)

4.2.1 Performance Index:

We know that our training set is of the form:

$\{X_1, t_1\}\{X_2, t_2\}$……………………..$\{X_k, t_k\}$, (4.10)

where X_k is the input vector and t_k is the corresponding target value and k = 1….p represents the *'kth'* iteration or pattern. Let *'W'* denote all the network parameters i.e. $W = [\alpha_i, \beta_i, a, b, c_i, c_0]$. Our objective is to minimize the cost function or the error measure i.e. sum of squared errors over whole the training set/ patterns which can be defined as:

$$\nabla E(W) = \frac{1}{2} \sum_{i=1}^{n} \; \sum_{k=1}^{p} \left(t_i(k) - y_i(k)\right)^2 \qquad (4.11)$$

And in the vector case we can define the above as:

$$E(W) = \Sigma\left[e^t e\right] = \Sigma\left[(t-y)^T (t-y)\right] \qquad (4.12)$$

Where '***e'*** is the sum of squared errors over all the training patterns. Therefore the approximate mean square error over a single sample (k) would be:

$$E^\wedge(X) = e^T(k) e(k) = (t(k) - y(k))^T (t(k) - y(k)) \qquad (4.13)$$

4.2.2 Updating Model Parameters:

We can define the approximate steepest descent or generalised delta rule for MLP's as follows:

$$W^{new} = W^{old} + \Delta W \tag{4.14}$$

where the parameters of our SNN are, $W = [\alpha_i, \beta_i, a, b, c_i, c_0]$ and,

$$\Delta W = -\eta \frac{\partial \hat{E}(W)}{\partial W} \tag{4.15}$$

where 'η' is the learning rate In the vector case we can write the equations (4.14) and (4.15) altogether as :

$$w\ (k+1) = w\ (k) - \eta \frac{\partial E^{\wedge}}{\partial w} \tag{4.16}$$

where *'k'* represents the *'kth'* iteration or pattern.

4.2.3 Gradient Calculation

Now we have to compute the gradients $\frac{\partial E^{\wedge}}{\partial W} = \left[\frac{\partial E^{\wedge}}{\partial c_i}, \frac{\partial E^{\wedge}}{\partial c_o}, \frac{\partial E^{\wedge}}{\partial \alpha_i}, \frac{\partial E^{\wedge}}{\partial \beta_i}, \frac{\partial E^{\wedge}}{\partial a}, \frac{\partial E^{\wedge}}{\partial b} \right]$, by using the chain rule of differentiation as follows:

$$\frac{\partial E^{\wedge}}{\partial C_i} = \frac{\partial E^{\wedge}}{\partial net_i^m} \wedge \frac{\partial net_i^m}{\partial C_i} \quad \text{and} \quad \frac{\partial E^{\wedge}}{\partial C_0} = \frac{\partial E^{\wedge}}{\partial net_i^m} \wedge \frac{\partial net_i^m}{\partial C_0} \tag{4.17}$$

$$\frac{\partial E^{\wedge}}{\partial \alpha_i} = \frac{\partial E^{\wedge}}{\partial net_j^{m-1}} \wedge \frac{\partial net_j^{m-1}}{\partial \alpha_i} \quad \text{and} \quad \frac{\partial E^{\wedge}}{\partial \beta_i} = \frac{\partial E^{\wedge}}{\partial net_j^{m-1}} \wedge \frac{\partial net_j^{m-1}}{\partial \beta_i} \tag{4.18}$$

$$\frac{\partial E^{\wedge}}{\partial a} = \frac{\partial E^{\wedge}}{\partial net_l^{m-1}} \wedge \frac{\partial net_l^{m-1}}{\partial a} \quad \text{and} \quad \frac{\partial E^{\wedge}}{\partial b} = \frac{\partial E^{\wedge}}{\partial net_l^{m-1}} \wedge \frac{\partial net_l^{m-1}}{\partial b} \tag{4.19}$$

Note that our initial simplified networks are of the form:

i. $y = NN(z) = \sum_{i=1}^{N} c_i \sigma[\alpha_i z + \beta_i)] + c_o$

ii. $$y = NN(z) = \sum_{i=1}^{N} c_i \sigma\left[\alpha_i (a'X + b) + \beta_i)\right] + c_o$$

This gives rise to two different scenarios as depicted above. We can proceed in two ways:

a. Following NN definition in (algorithm-I) compute the gradients $\frac{\partial E^{\wedge}}{\partial \alpha_i}, \frac{\partial E^{\wedge}}{\partial \beta_i}$

b. Following NN definition in (algorithm-II) also compute the gradients $\frac{\partial E^{\wedge}}{\partial a}, \frac{\partial E^{\wedge}}{\partial b}$.

The effect of computations in step ‘b’ will be the provision of two extra parameters for network tuning.

4.2.4 Computing Error Signals

Let $\frac{\partial E^{\wedge}}{\partial net_{i,j}^{m,m-1}} = s_{i,j}^{m,m-1}$, be the sensitivity or error signal for the output and hidden layers respectively. From the network definition above we can see that we have to compute the following gradients inline with the eqns. (4.17) (4.18) & (4.19)

$\frac{\partial net_i^m}{\partial C_i} = \frac{\partial}{\partial C_i}\left[\sum_{i=1}^{n} c_i \; y_j^{m-1} + c_0\right]$ and $\frac{\partial net_i^m}{\partial C_0} = \frac{\partial}{\partial C_0}\left[\sum_{i=1}^{n} c_i \; y_j^{m-1} + c_0\right]$ therefore,

$$\frac{\partial net_i^m}{\partial C_i} = y_j^{m-1} \text{ and } \frac{\partial net_i^m}{\partial C_0} = 1 \qquad (4.20)$$

Similarly,

$\frac{\partial net_j^{m-1}}{\partial \alpha_i} = \frac{\partial}{\partial \alpha_i}\left[\sum_{j=1}^{N} \alpha_i z + \beta_i\right]$ and $\frac{\partial net_j^{m-1}}{\partial \beta_i} = \frac{\partial}{\partial \beta_i}\left[\sum_{j=1}^{N} \alpha_i z + \beta_i\right]$

$$\frac{\partial net_j^{m-1}}{\partial \alpha_j} = Z \quad \text{and} \quad \frac{\partial net_j^{m-1}}{\partial \beta_i} = 1 \qquad (4.21)$$

And

$$\frac{\partial net_j^{m-1}}{\partial a} = \frac{\partial}{\partial a}\left[\sum_{j=1}^{N} \alpha_i(a'X+b)+\beta_i\right] \qquad \text{and} \tag{4.22}$$

$$\frac{\partial net_j^{m-1}}{\partial b} = \frac{\partial}{\partial \beta_i}\left[\sum_{l=1}^{N} \alpha_i(a'X+b)+\beta_i\right] \tag{4.23}$$

Therefore,

$$\frac{\partial net_j^{m-1}}{\partial a} = x_i \text{ and } \frac{\partial net_j^{m-1}}{\partial b} = 1 \tag{4.24}$$

Now we can re-write our steepest descent rule as follows:

1. For output layer weight and bias values:

$$c_i(k+1) = c_i(k) - \eta s_i^m y_j^{m-1} \quad , \quad c_0(k+1) = c_0(k) - \eta s_i^m \tag{4.25}$$

2. For hidden layer weight and bias values:

$$\alpha_i(k+1) = \alpha_i(k) - \eta s_j^{m-1} z \quad , \quad \beta_i(k+1) = \beta_i(k) - \eta s_j^{m-1} \tag{4.26}$$

and:

$$a\ (k+1) = a\ (k) - \eta s_j^{m-2} x_i \quad , \quad b\ (k+1) = b_i(k) - \eta s_j^{m-2} \tag{4.27}$$

4.2.5 Back-Propagating The Error Signal

The only thing left to be computed are the sensitivities i.e. $\frac{\partial E^{\wedge}}{\partial net_{i,j}^{m,m-1}} = s_{i,j}^{m,m-1}$.

This is the process which gives the name of back propagation to this algorithm. Note that the sensitivities are computed by starting at the last layer, and then propagating backwards through the network to the first layer. i.e. $S^m \to S^{m-1} \to S^{m-2} .....S^2 \to S^1$. For the last or output layer this sensitivity or error signal (i.e. how the error at the output is affected by the net input 'i') can be easily computed as follows:

$$s_i^m = \frac{\partial E^{\wedge}}{\partial net_i^m} = \frac{\partial}{\partial net_i^m}\left[\frac{1}{2}\sum_{i=1}^{n}\sum_{k=1}^{p} (t_i^m(k) - y_i^m(k))^2\right] \tag{4.28}$$

$= -(t_i(k) - y_i(k))\ \frac{\partial y_i(k)}{\partial net_i^m}$, where the term $\frac{\partial y_i(k)}{\partial net_i^m}$ is actually the derivative of our activation function i.e. $\frac{\partial y_i}{\partial net_i^m} = \frac{\partial \sigma(net_i^m)}{\partial net_i^m} = f\,'(net_i^m)$ (4.29)

Note that in the case of Sigmoidal neurons it will be:

$$\frac{\partial}{\partial(x)}\left[\frac{1}{1+\exp^{-x}}\right] = \frac{\exp^{-x}}{(1+\exp^{-x})^2} = \left[1 - \frac{1}{1+\exp^{-x}}\right]\left[\frac{1}{1+\exp^{-x}}\right] = (1 - x_i)\,x_i,$$

and in the case of linear neurons it will be $\frac{\partial}{\partial(x)}(x) = x$. (4.30)

Therefore we can see that the sensitivity/ error signal for output layer will be,

$$s_i^m = -(t_i^m - y_i^m)\ f\,'(net_i^m) \tag{4.31}$$

From here we can now compute the sensitivity of the hidden layer. Note that the error at hidden layer is not a direct function of its weight and bias; instead it is an accumulation of error from the layer just after this. So, we need another application of chain rule of differentiation to compute this error signal.

$$s_j^{m-1} = \frac{\partial E^\wedge}{\partial net_j^{m-1}} = \frac{\partial E^\wedge}{\partial net_i^m}\frac{\partial net_i^m}{\partial net_j^{m-1}} \tag{4.32}$$

Note that we have already computed the first term $\frac{\partial E^\wedge}{\partial net_i^m} = s_i^m$ in equation (4.28). Therefore, we are left with,

$$\frac{\partial net_i^m}{\partial net_j^{m-1}} = \frac{\partial}{\partial net_j^{m-1}}\left[\sum_{i=1}^{n} c_i\, y_j^{m-1} + c_0\right] = c_i\ \frac{\partial y_j^{m-1}}{\partial net_j^{m-1}} \tag{4.33}$$

$$\frac{\partial y_j^{m-1}}{\partial net_j^{m-1}} = \frac{\partial \sigma(net_j^{m-1})}{\partial net_j^{m-1}} = f\,'(net_j^{m-1}) \tag{4.34}$$

‘ $f\,'(net_j^{m-1})$ ’ is the derivative of activation function and can be computed following the derivation depicted in eqns. above (4.29) and (4.30).

By combining (4.32) and (4.33) we get,

$$s_j^{m-1} = s_i^m c_i\, f\,'(net_j^{m-1}) \tag{4.35}$$

We can now obtain the updated weight and bias values for our network by substituting the sensitivities/ error signal obtained in equation (4.31) and (4.35) into (4.17) (4.18) & (4.19) respectively.

4.3 SNN Extension To Regression Boosting

As discussed above we can see that Simplified NN approach has its distinct advantages over traditional NN approximation schemes. Especially when it comes to dealing with function defined on discrete input spaces. In order to investigate wider implications of SNN approach we will extend our approach to regression boosting, which will target the regression problems for our selected domain i.e. function approximation problems in high dimension-low sample cases where the model inputs constitutes of a significant number of discrete variables.

The algorithm-II presented above can be easily extended to be viewed as a regression boosting method for functions defined on discrete input spaces i.e. $X \in U = \prod_{i=1}^{n} U_i$. With similar error bounds and convergence guarantees as presented in [HZ09]. Based on these exciting facts we propose a new simplified approach to regression boosting for functions defined on discrete input spaces. We will refer to our approach as Simplified Regression Boosting (SRB) for discrete input spaces. Following is a step by step description of this approach for functions defined on discrete input spaces.

4.3.1 Simplified Regression Boosting (SRB):

Let G(x) be an objective function we wish to minimize this cost function, this could be any objective function such as one presented in [ZP01] or e.g. $G\left(x\right) = \frac{1}{2}\sum_{i=1}^{n}\sum_{k=1}^{p}\left(h_i\ (k) - y_i(k)\right)^2$. In traditional regression boosting settings $\mathbf{h}_i(k)$

is referred to as the hypothesis generated by the *WeakLearn* Procedure. The accuracy of this hypothesis on the training set is then measured according to cost function G(x). As highlighted before many regression boosting methods used Neural Networks as base regressor or *WeakLearn* procedure to generate a hypothesis $\mathbf{h}_i(k)$ at every iteration. In such situations the output or hypothesis generated by a standard Neural Network can be represented as $h_i(k) = NN(X) = \sum_{i=1}^{N} c_i \sigma(\mathbf{a}_i^{\tau} X + b_i) + c_0$, where $X = (x_1, x_2, \ldots, x_n)$ are input variable, $X \in U = U_1 \times U_2 \times \ldots \times U_n \subset R^n$ which are input space, $y \in R$ is the output variable, τ is the vector transpose, $\sigma(.)$ is the activation function and the parameters $c_0 \in R$, $c_i \in R, a_i \in R^n$, and $b_i \in R$ $(i = 1, 2, \ldots, N)$. In the following section we propose a new simplified version of the *WeakLearn* procedure to boost functions defined on discrete input spaces; we will refer to this simplified version as 'Simplified *WeakLearn*'.

Based on this approach we can derive algorithms for boosting regression problems for function defined on discrete input spaces. These will be a lot faster and simpler in architecture when compared to existing regression boosting models using Neural Networks as *WeakLearn* procedure. In fact, this approach can be used with any existing regression boosting algorithms using Neural Networks as *WeakLearn* procedure by simply replacing the Standard *WeakLearn* with the 'Simplified *WeakLearn*' discussed above. We can prove the convergence for this algorithm by following the approach used in [ZP01] and is included at the end of this thesis as appendix-D.

4.3.2 Simplified Regression Boosting Algorithm-III

1. *Input*:
    - Training set examples $S = (\mathbf{x}_1, y_1).......(\mathbf{x}_n, y_n)$ where $y \in \mathbb{R}$, training iterations T .
    - *Simplified WeakLearn*: A learning procedure that produces a hypothesis $\mathbf{h}_i^*(x)$
2. Identify a best linear approximation $z = L^*(X) = a'X + b$ which can be found by the least square algorithm;

(Note that we can represent a multivariate or least squares fit model of the data as: $z = \alpha_0 x_0 + \alpha_1 x_1 + \alpha_2 x_2 ....\alpha_n x_n$*. Therefore we have to solve for unknown coefficients* $\alpha_0, \alpha\ , \alpha_2 ....\alpha_n$ *, by performing a least squares fit i.e. multivariable regression)*

3. *Initialize:* Initialize the model parameters using $\alpha_0, \alpha\ , \alpha_2 ....\alpha_n$
4. Iterate:
    - Call Simplified WeakLearn-minimize cost function G(x) with initial model parameters. (accept iff $\xi_t = \sum_{k=1}^{p} \left(h_i^* \left(k^i \right) - y_i \right)^2 - \tau < 1$
    - Set combination co-efficient c_t to minimize G(x)
    - Modify model parameters using gradient descent algorithm in order to identify, $h_i^*(x) = \sum_{i=1}^{N} c_i \sigma\left[\alpha_i (a'X + b) + \beta_i\right] + c_o$
5. *Estimate Output:* $\tilde{y} = \sum_t c_t h_i^* \left(x \right) / \sum_t c_t$

In line with definition of a simplified NN presented in [HZ09] we define our ‘Simplified *WeakLearn*’ as $h_i^*(x) = \sum_{i=1}^{N} c_i \sigma\left[\alpha_i (a'X + b) + \beta_i\right] + c_o$, where $c_i \in R$, $a_i \in R^n$, $b_i \in R$ $((i = 1, 2, ..., N)$ and $c_0 \in R$. This Simplified *WeakLearn* approach will differ on two main aspects: the initialisation criteria and the total number of model parameters for the *WeakLearn* procedure. This algorithm will first identify a best linear approximation, then use these initial estimates to initialize the network weights, combining at a summing junction before the hidden layer neurons; then Call the Simplified *WeakLearn* procedure in order to minimize cost function *G(x)*. This approach has many distinct advantages. Firstly, a single common objective function is both used by the weak learning procedure to produce hypotheses and determines the other parameters in the algorithm. Secondly, the distribution of examples is used to control the generation of hypotheses and each hypothesis is trained to learn the same underlying function. Since the Simplified *WeakLearn* use simplified NN as the base learner, it also reduces the model parameters and enhances the performance. As highlighted section 4.1.1 the result of such a setup benefits in two ways; firstly in forward pass we have a good initial estimate as compared to individual inputs only and in backward pass we have two additional parameters associated with each input for further fine tuning of the initial estimates of the best linear approximation coefficients. This approach results in significant reduction of model parameters. As described in [HZ09], the total number of parameters required is 3N+n+2 as compared to a standard NN where the total number of parameters required for function approximation problems is (n+2)N+1, (n = number of network inputs, N= number of hidden layer neurons). Another distinct advantage of this approach is that when we add neurons in the hidden layer, we only add three parameters per neuron; this gradual increase in parameter helps in avoiding model over-fitting, a commonly faced situation in standard NN models.

4.4 Summary

The first part of this chapter is the introduction of simplified NN schemes and corresponding learning algorithms. A derivation of Backpropagation algorithm for these simplified NN algorithm is also outlined in detail. The simplified NN schemes and algorithms are mathematically analysed and an architectural representation of these algorithm is also presented in this chapter. A detailed analysis of approximation capabilities of simplified NN algorithms is also included in this chapter. This chapter also contains a discussion on the wider implications of the simplified Neural Network approach, and gives an overview of how simplified NN approach can be applied to regression boosting. We have given a brief introduction to regression boosting in this chapter, and discussed how a simplified regression booting scheme can be developed using simplified NN approach. We also propose a new algorithm for regression booting on functions defined on discrete input spaces in this chapter.

CHAPTER 5

IMPLEMENTATION AND EVALUATION OF THE SNN ALGORITHMS

There are many tools and applications available to simulate Neural Network based models for evaluating their performance. In order to assess the performance of our proposed simplified NN algorithms, and to compare the results with standard NN models, we have implemented these algorithms in Matlab 7.0. The reasons for choosing Matlab are: its familiarity in research community, success in recent years and availability of a range of learning and optimisation algorithms for NNs.

5.1 Data Collection

One of the most significant aspects in the success of any Neural Network application is the quality and availability of data. The availability of sufficient training data plays a very important role in success of a NN based model. As highlighted before unavailability of sufficient training data in certain application domains makes it difficult for standard NN models to achieve the desired results.

In order to analyse the performance of simplified Neural Networks (SNN) we first produced some dummy data sets and trained our SNN on these datasets. The dummy data sets are functions of varied complexity with two or three input variables as shown in table 5.1 and 5.2. For the dummy examples 25 (see table 5.1) and 40 (see table 5.2) cases of discrete values have been generated, independently each of which uniformly distributed over [0,1]. The values of the target variable Y was then computed using the equations shown in tables 5.1 and 5.2.

The obvious advantage of using dummy data sets is that we have prior knowledge of underlying function and we can easily monitor the performance of our proposed algorithms, as both the dependent and independent variables are under experimental control.

Once the performance of our network is verified on these dummy data sets, we identified some benchmarking examples to show that the proposed algorithms are general enough for any kind of approximation problems taking on discrete values. Selection of benchmarking data was a tedious task since our algorithms represent a special case of standard NN, therefore we need datasets that can meet the following criteria:

- All or at least a significant number of independent variables should be discrete. (Any continuous variables remaining in the data sets can be later rounded off to make it a discrete variable i.e. for experimentation purpose only).
- The number of independent variables should be large.
- The variables should be independent of each other.
- Availability of data is limited i.e. there are not enough examples for training a standard NN.

As argued earlier, most of the NN approximation schemes proposed so far consider the NNs to take on continuous inputs only. Therefore most of the benchmarking datasets have continuous values only. Alternatively, if there are any datasets available that has discrete values, they were used for classification problems instead.

There are many well known resources of experimental data available for use with NNs e.g. UCI Machine Learning Repository, Bilkent University function approximation repository, statlib data archives and Delve data sets etc. We have selected three different benchmarking examples from 'Bilkent University Function Approximation Repository'. For the Pyrimidines data set, the task consists of Learning Quantitative Structure Activity Relationships (QSARs) i.e. The Inhibition of Dihydrofolate Reductase by Pyrimidines. For the Triazines Dataset, the problem is to learn a regression equation, rule or tree to predict the activity from the descriptive structural attributes. A detailed description of the selected data sets and their past usage is given in Appendix-B.

5.2 Data Pre-Processing And Partitioning

Once the data is selected the next step is perform some data pre-processing. In practice, it is nearly always beneficial, sometimes critical, to apply pre-processing to the input data before they are fed to a network. There are many techniques and considerations relevant to data pre-processing e.g. simple filtering, principle component analysis and many others , please see [Sar97][Bis95][Mas93]. However, the aim of these pre-processing techniques is roughly the same i.e. transformation of the data into a form suited to the network inputs, selection of the most relevant data and reducing the number of inputs to the network.

In order to compare the performance of these simplified algorithms with standard NN models, we have used the method of three way splits, and partition the data into training sets, validation sets, and test sets. As defined earlier, validation sets are used to decide the architecture of the network, training sets are used to actually update the weights in a network and test sets are used to examine the final performance of the network. The crucial point is that a test set, by the standard definition in the NN literature, is never used to choose among two or more networks, so that the error on the test set provides an unbiased estimate of the generalization error (assuming that the test set is representative of the population, etc. Any data set that is used to choose the best of two or more networks is, by definition, a validation set, and the error of the chosen network on the validation set is optimistically biased [Sar97].

5.3 Simulation Results for SNN Algorithms I &II

Once the data pre-processing tasks are performed, the networks are ready for training. The selected data sets (i.e. dummy and real-world examples) are first used for training of a standard Neural Network. The objective is to set a standard for evaluation against our simplified algorithms. These standard NNs are actually feedforward Neural Networks of three layers i.e. input, hidden and output layer. According to the conventional setup, the hidden layer activation function is chosen to be sigmoid, whereas the output layer activation function is pure linear. With these initial parameters in place, we can now train the standard NN for approximation on the selected data sets. The same data sets are then used for training of proposed simplified NN models. The results obtained are summarized in (see tables 5.1-5.4), followed by comparative graphs (see figures 5.1-5.12) showing performance of these simplified NNs against standard NNs over testing data sets; where total number of training iterations or epochs are recorded on x-axis and mean squared error on y-axis.

Data Set	ANN			SNN-I			SNN-II		
	MSE	**No. of iterations**	**No. of parameters (n+2)N+1**	**MSE**	**No. of iterations**	**No. of parameters 3N+n+2**	**MSE**	**No. of iterations**	**No. of parameters 3N+n+2**
Dummy 1 $(2X_1+X_2^2)$	0.080659	100	(2-4-1) 17	0.494222	100	(1-4-1) 15	0.465661	100	(1-4-1) 15
Dummy 2 $(2X_1+2^{X2})$	0.360283	100	(2-4-1) 17	0.313595	100	(1-4-1) 15	0.4626	100	(1-4-1) 15
Dummy 3 $(\mathrm{Sin}(X_1+X_2))$	0.42679	100	(2-4-1) 17	0.0709674	100	(1-4-1) 15	0.0686217	100	(1-4-1) 15
Pyrimidines	0.0919559	300	(14-6-1) 97	0.0119983	300	(1-10-1) 33	0.0236143	300	(1-8-1) 27
Triazines	0.436261	300	(18-8-1) 161	0.0159458	300	(1-10-1) 33	0.0428223	300	(1-12-1) 39

Table: 5.1 Performance Comparison of Standard NNs Vs Simplified NNs

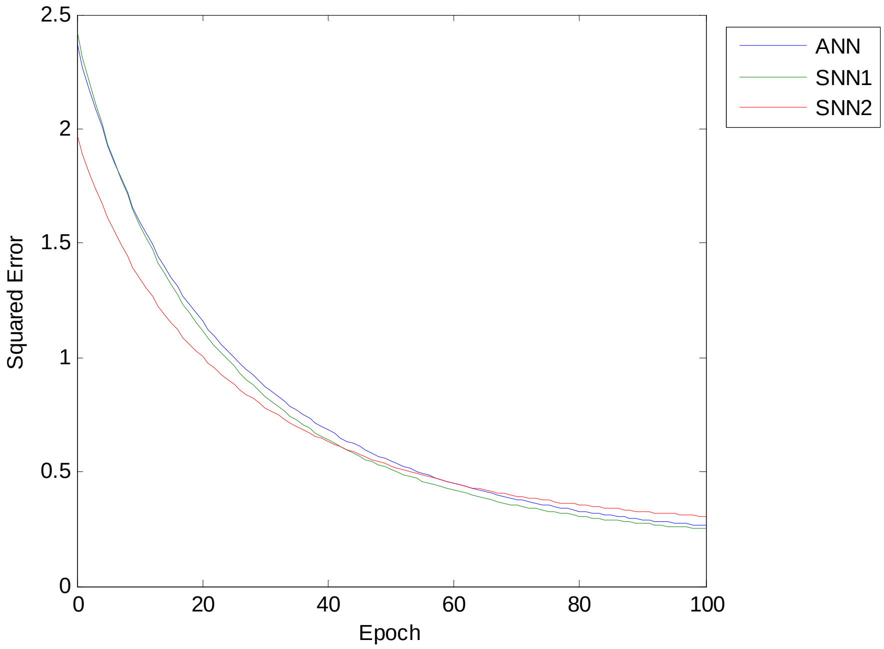

Figure 5.1 Performance of Standard NNs Vs Simplified NNs over Test Set (Dummy 1)

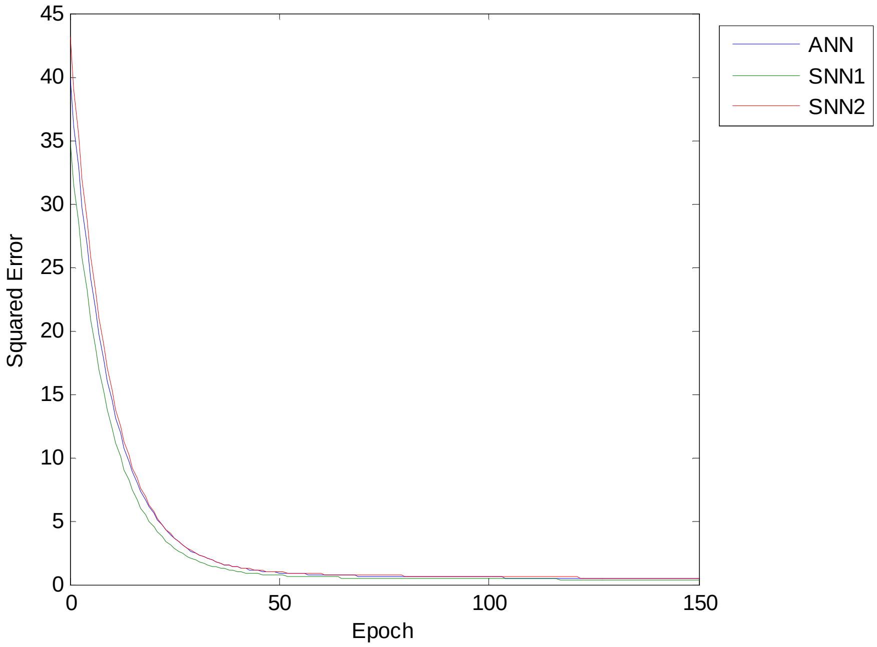

Figure 5.2 Performance Of Standard NNs Vs Simplified NNs over Test Set (Dummy 2)

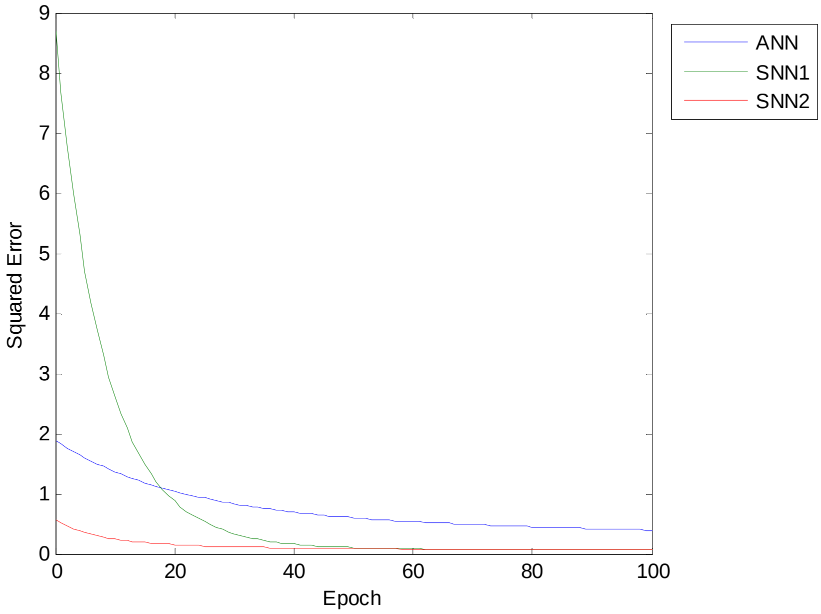

Figure 5.3 Performance of Standard NNs Vs Simplified NNs over Test Set (Dummy 3)

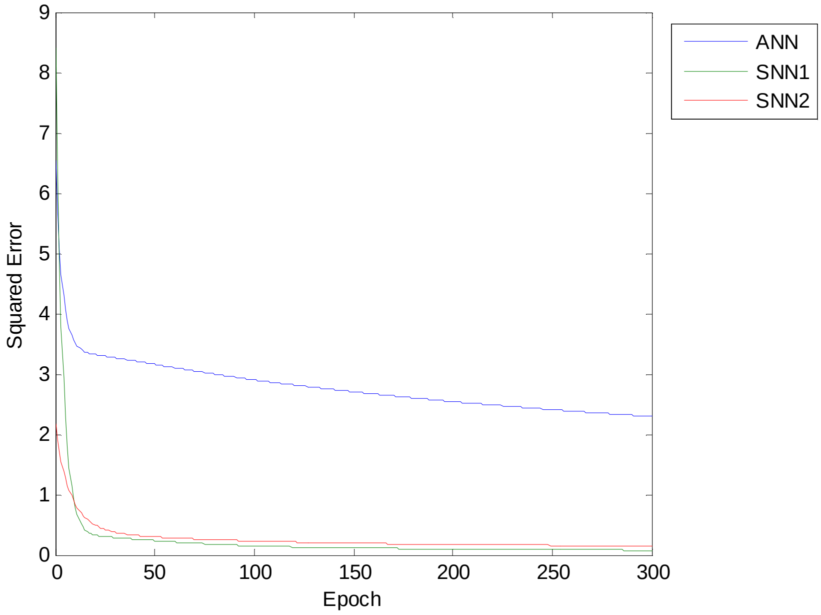

Figure 5.4 Performance of Standard NNs Vs Simplified NNs over Test Set (Pyrimidines)

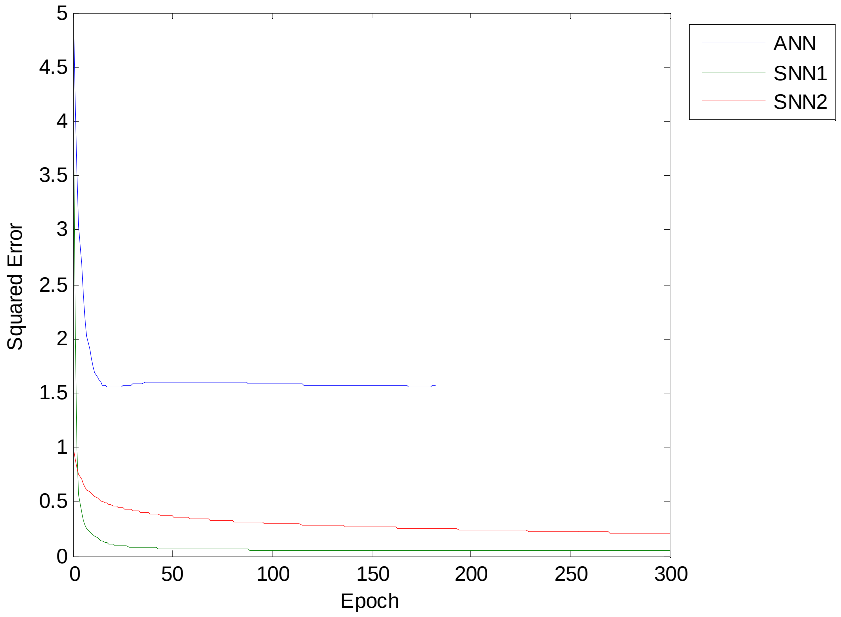

Figure 5.5 Performance of Standard NNs Vs Simplified NNs over Test Set (Triazines)

The initial experimental results obtained with dummy data sets do not reflect any significant improvement in terms of total number of parameters. The reason for that is we are only using two independent variables, and therefore the effect of simplification is not apparent. However, the results of benchmarking datasets show a significant reduction in the total number of parameters. These results supports our claim that simplified NNs are universal approximators for functions defined on discrete input spaces; since we have achieved approximately the same or in some cases even better accuracy, with significantly less parameters. Although the performance of the simplified algorithms was quite promising on the selected datasets, one may argue the simplicity of dummy datasets mainly consisting of two variables. We therefore extended our experiments to use more complicated dummy data sets with varying complexity and number of variables. We then used these datasets to experiment with simplified algorithm-II which yielded even better performance then before; please refer to table 5.2 and comparison graphs (figure 5.6-5.10). The experiments were initially performed with 100 training iterations for dummy datasets and 300 iterations for real world examples; in order to verify whether the performance of these models degrade upon increasing training iterations. Hence, increasing the number of iterations actually does not affect or add any value to the initial performance of our simplified NNs, we re-evaluated the performance of our algorithms against standard NN with no data pre-processing for both the models, we also reduced the number of training iterations significantly (i.e. 25) for Pyrimidines and Triazines datasets, see table 5.3 and 5.4 with corresponding comparison graphs as shown in figures 5.11 and 5.12.

Data Set	**Standard NN**			**Simplified NN-II**		
	MSE	**No. of iterations**	**No. of parameters (n+2)N+1**	**MSE**	**No. of iterations**	**No. of parameters 3N+n+2**
Dummy 4 SIN(2X_1+4X_2^2)	0.146185	100	17	0.074572	100	16
Dummy 5 2X_3^2+X_1^3+LOG(X_2)	0.126166	100	21	0.011201	100	16
Dummy 6 $Sin\left(6\frac{\pi}{4}x_1x_2\right)+0.5x_3^2$	0.210441	100	31	0.058809	100	28
Pyrimidines	0.079212	100	88	0.0035615	100	47
Triazines	0.035513	100	187	0.011004	100	74

Table: 5.2 Performance Comparison of Standard NNs Vs Simplified NNs for (SNN-II)

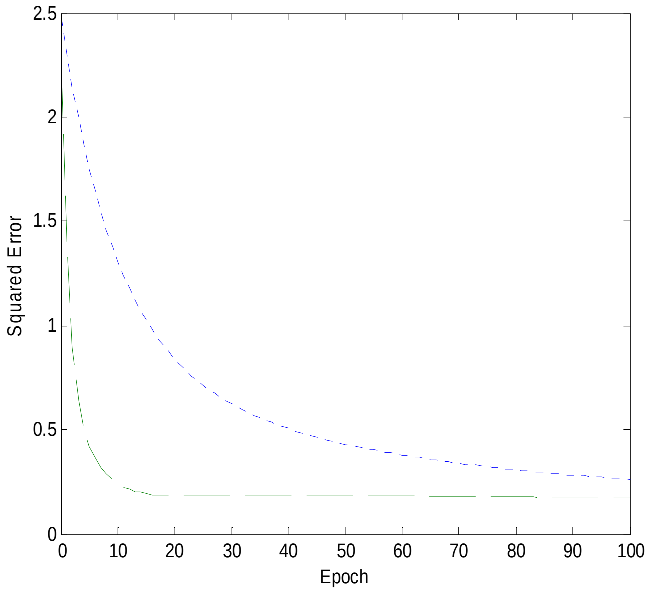

Figure 5.6 Comparison Graph, Standard NN Vs Simplified NN over Test Sets for SNN-II (Dummy 4)

Legend:

Standard NN: - - - - - - - - - - -----

Simplified NN: __ __ __ __ __

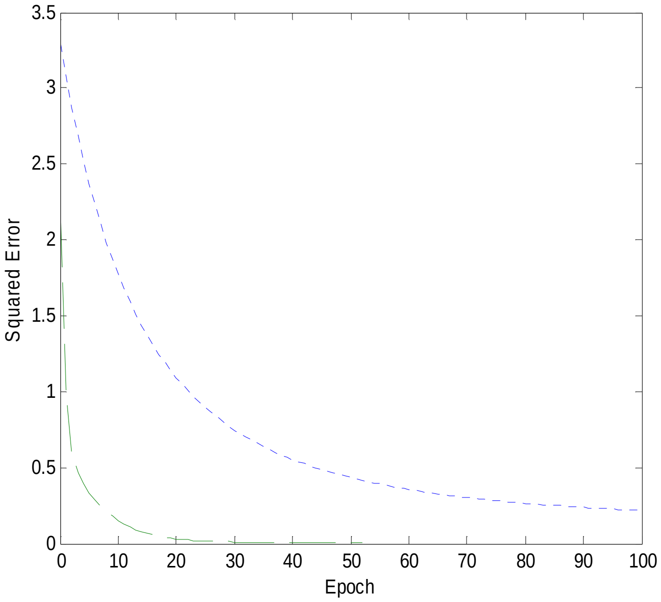

Figure 5.7 Comparison Graph, Standard NN Vs Simplified NN over Test Sets for SNN-II (Dummy5)

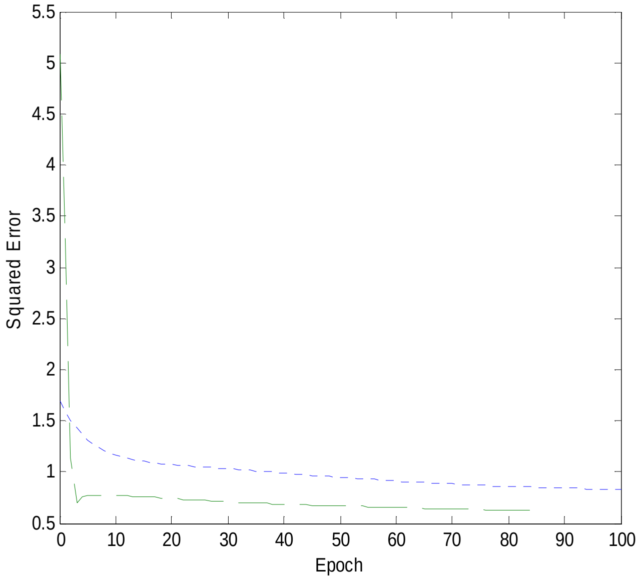

Figure 5.8 Comparison Graph, Standard NN Vs Simplified NN over Test Sets for SNN-II (Dummy 6)

Legend:

Standard NN: - - - - - - - - - - - -----

Simplified NN: ___ ___ ___ ___ ___

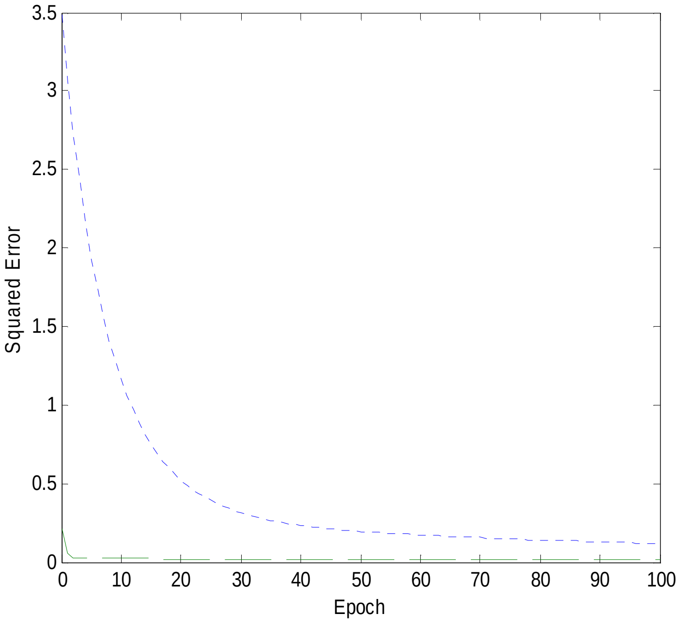

Figure 5.9 Comparison Graph, Standard NN Vs Simplified NN over Test Sets for SNN-II (Pyrimidines)

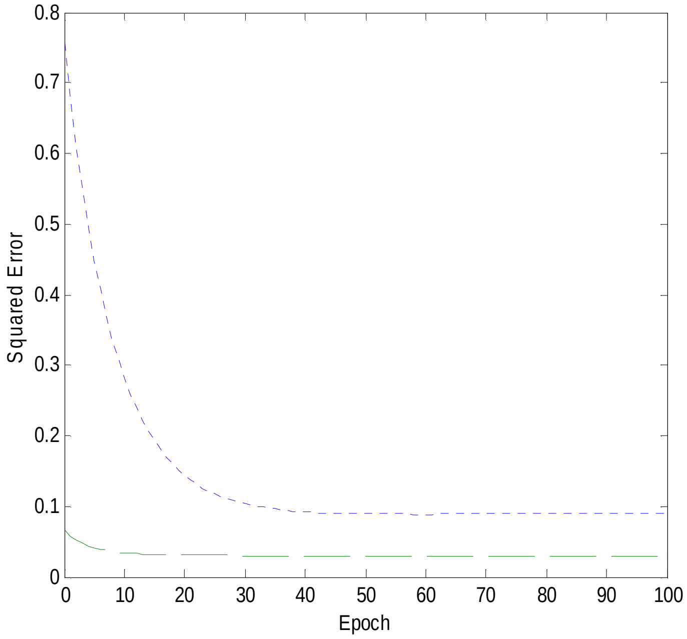

Figure 5.10 Comparison Graph, Standard NN Vs Simplified NN over Test Sets for SNN-II (Triazines)

Legend:

Standard NN: - - - - - - - - - - -----

Simplified NN: __ __ __ __ __

The experimental results show that these simplified networks have the ability to approximate functions defined on discrete input spaces to arbitrary accuracy by employing less number of parameters as compared to standard NN approximation schemes. The simplified algorithms have shown to be computationally inexpensive and simpler in architecture. Based on theses findings we decided to proceed with formal publication of our work.

One crucial point to be noted here is the fact that when comparing our results with standard NN models, we have not used any data pre-processing with standard NN models. For this particular reason, we either have to omit the data pre-processing stage from the simplified NNs and initialise the network parameters with random weights as in standard NN model, or do similar data pre-processing for standard NN model for a fair comparison. We have adopted the first approach and eliminated the data pre-processing stage from simplified NNs. With this setup in place, we re-evaluated the performance of our proposed simplified algorithms against standard NN model. Upon analysis of results presented above we can also see that the difference in performance of standard and simplified NNs is more apparent during initial training iterations. Hence, increasing the number of iterations actually does not add any value to the initial performance of our simplified NNs. For these reasons we re-evaluated the performance of our algorithms against standard NN with no data pre-processing for both the models, we also reduced the number of training iterations significantly (i.e. 25) for Pyrimidines and Triazines datasets.

For illustration, consider the example of Pyrimidines data set, which consists of 74 instances, 27 explanatory variables and 1 response variable. With five hidden layer neurons and over a set of 25 iterations, the performance of a standard NN in terms of mean squared error was recorded to be 0.2764 by employing a total of 146 parameters according to (n+2)N+1 (i.e. n = number of network inputs, N = number of hidden layer neurons). The same data are then used for training of our simplified NN. We obtained an accuracy of 0.0292 over 25 iterations by employing 47 parameters in total according to 3N+n+2. We have also achieved better accuracy in terms of means squared error. Also note that SNN has not only achieved similar accuracy but it has achieved that in relatively fewer training iterations or cycles, e.g. see the comparison graph for Triazines dataset, where similar accuracy is achieved in very fewer training cycles. These results support our claim that simplified NNs are universal approximators for functions defined on discrete input spaces, since we have achieved approximately the same or in some cases even better accuracy with significantly less parameters.

Pyrimidines			
	MSE	No. of iterations	No. of parameters
Standard NN	0.2764	25	146
Simplified NN	0.0292	25	47

Table 5.3 Pyrimidines Data set - Performance Comparison Over Testing Data for 25 Iterations

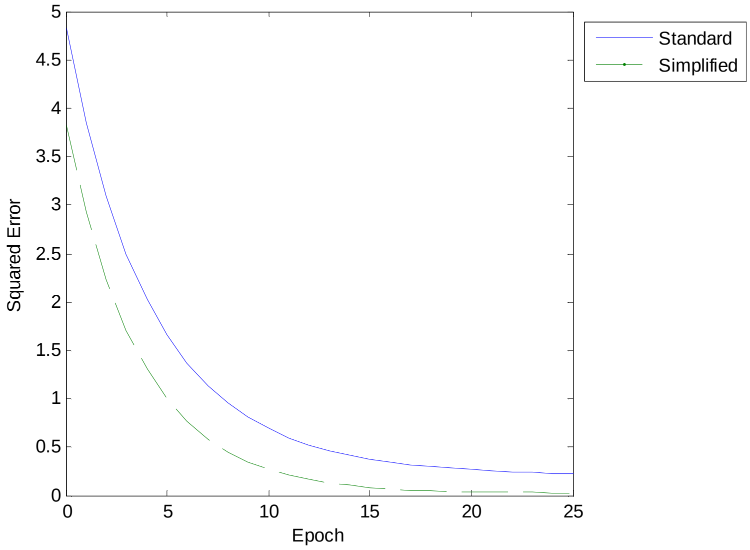

Figure 5.11 Pyrimidines Data set - Performance Comparison over Testing Data for 25 Iterations

Triazines			
	MSE	No. of iterations	No. of parameters
Standard NN	0.1032	25	311
Simplified NN	0.0225	25	77

Table 5.4: Triazines Data set - Performance Comparison over Testing Data for 25 Iterations

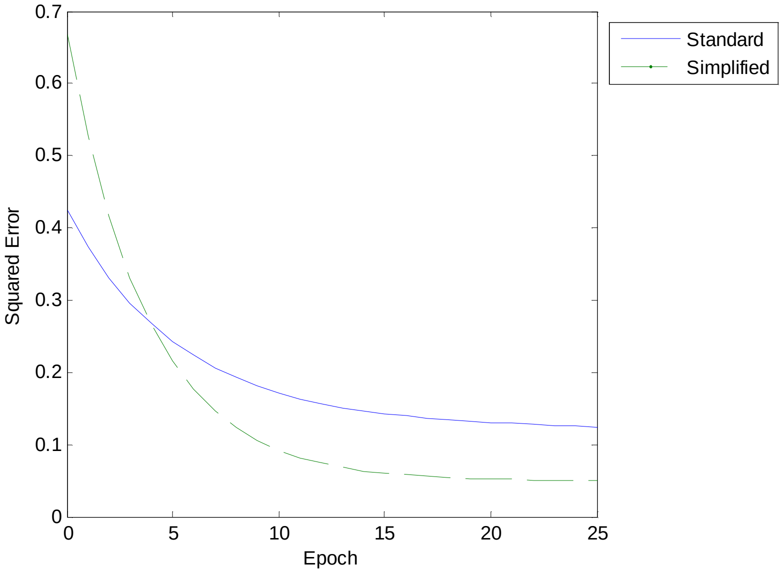

Figure 5.12 Triazines Data set - Performance Comparison over Testing Data for 25 Iterations

5.4 Simulation Results For Simplified Regression Boosting Algorithm-III

In order to evaluate the performance of the SRB Algorithm-III we have chosen three different benchmarking datasets: Pyrimidines and Triazines which are already used for evaluating simplified NN performance and a third example named F1 dataset, $y = 10\sin\left(\pi x_1 x_2\right) + 20\left(x_3 - .5\right)^2 + 10x_4 + 5x_5$. This first appeared in [Fri91] and then in [ZP01]. Since our focus is on function approximation problems for functions defined discrete input spaces, therefore we have not used standard data for this problem, this is because their input variables are continuous. Instead we have generated dummy samples for all the five explanatory variables which constitute discrete values. A total of 100 instances is produced and then partitioned into training, validation and test sets as per standard practice. For a fair comparison with [ZP01] we have used Neural Networks as the hypotheses and backpropagation as the learning procedure to train them. However our algorithm uses a simplified *WeakLearn* instead of a standard *WeakLearn* as used in [ZP01]. Each network had a layer of three 'tansig' activation functions between the input units and a single linear output. We used early stopping with a validation set in order to reduce over fitting in the hypotheses.

Performance of this algorithm is compared with a slightly modified version of the algorithm presented by Zimmel & Pittasi which appeared in [ZP01]. The first step in the simplified regression boosting algorithms is identifying a best linear approximation from the available data. The aim is to provide our Simplfied WeakLearn procedure. This can be achieved easily by applying multiple regression. In Matlab this can be done by using back-slash operator ("/"). We may refer to Matlab Neural Network toolbox help section for further details on specific implementation related issues.

The results were consistent for all the three examples and the training error was reduced steadily. Please refer to the comparison graphs (see figure 5.13-5.15) which show the performance of these examples over the test sets.

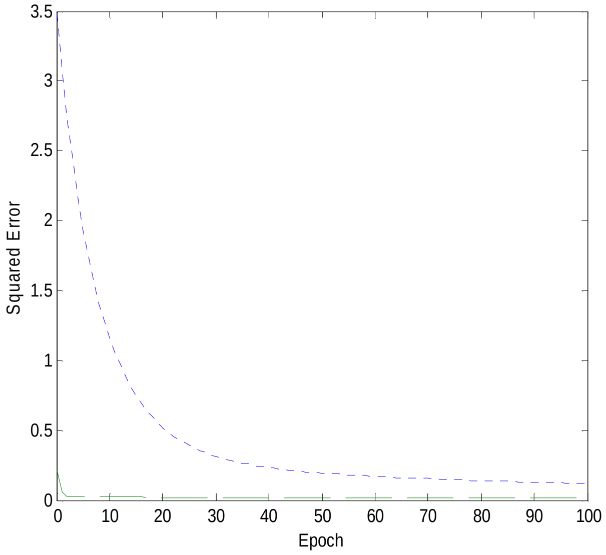

Figure 5.13 Performance Comparison of Simplified Regression Boosting Vs Standard Regression Boosting over Test Sets (Pyrimidines dataset)

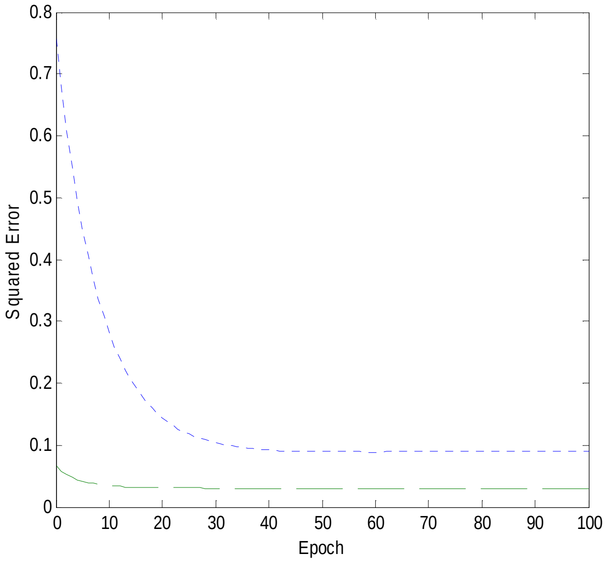

Figure 5.14   Performance comparison of Simplified Regression Boosting Vs Standard Regression Boosting over Test Sets (Triazines dataset)

Legend:

Standard Regression boosting: - - - - - - - - - -----

Simplified Regression boosting: __ __ __ __ __

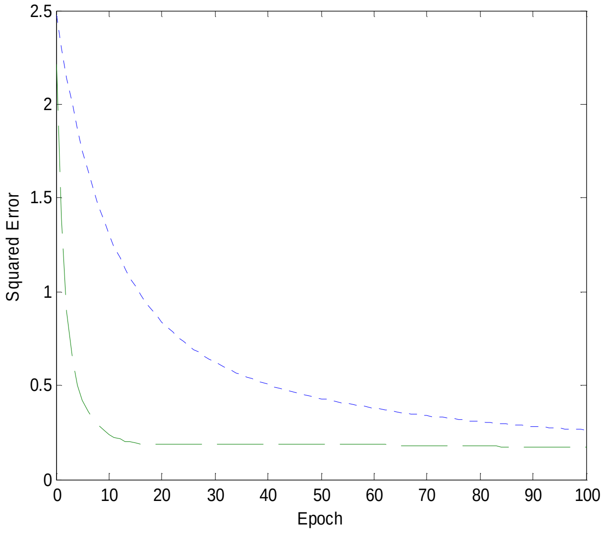

Figure: 5.15 Performance comparison of Simplified Regression Boosting Vs Standard Regression Boosting over Test Sets (F1 dataset)

Legend:

Standard Regression boosting: - - - - - - - - - -----

Simplified Regression boosting: __ __ __ __ __

Data Set	**Standard Regression Boosting using Z&P Algorithm**			**Simplified Regression Boosting**		
	MSE	**No. of iterations**	**No. of parameters (n+2)N+1**	**MSE**	**No. of iterations**	**No. of parameters 3N+n+2**
Pyrimidines	0.079212	100	88	0.0035615	100	38
Triazines	0.035513	100	187	0.011004	100	71
F1	0.310441	100	22	0.208809	100	16

Table 5.5: Performance comparison of Simplified Regression Boosting Vs Standard Regression Boosting over Test Sets

Performance comparison of simplified and standard regression boosting is summarized in table 5.5. On comparison of the obtained results we can see that the Simplified Regression boosting algorithm has achieved lower or approximately similar MSE on all the three examples. For instance, see the results obtained for F1 data set. We can see that we have achieved almost similar accuracy in terms of MSE. However, the number of parameters required for the model has been reduced to 16 from 22 in standard regression boosting algorithm. For F1, dataset reduction in parameters is not so significant due to the fact that F1 data set has only five inputs but if we compare the parameters required for both algorithms over Traizines and Pyrimidines datasets, we can see the effect of significant reduction in model parameters. For example in Traizines dataset we have achieved much better MSE by employing only 71 parameters as compared to 187 required for standard regression boosting model.

5.5 Summary

This chapter of the thesis discusses the implementation details of the simplified NNs. As illustrated earlier, these algorithms are implemented in Matlab 7.0 using Neural Network tool box functions. The algorithms are first implemented and then their performance is evaluated against standard NN approximation schemes. The data collection and pre-processing tasks are also discussed briefly. The proposed algorithms are initially tested on three dummy data sets, in order to understand the effects and these algorithms in detail, and then on two real world examples from Bilkent University Function Approximation repository. The experimental results are shown in the form of tables and graphs. A comparison of training, validation and test sets for all data sets are presented. Separate graphs showing the approximation and forecasting performance of these simplified NNs against standard NN scheme, on test sets, are also presented. Similarly, the implementation and evaluation details of simplified regression boosting algorithm are also given in this chapter. The performance evaluation and results for simplified regression boosting algorithm have been reported on three benchmarking datasets.

CHAPTER 6

CONCLUSION

Function approximation capabilities of feedforward Neural Networks have been widely investigated over the past couple of decades. However, use of these NN models is restricted due to complex computations attached with them. Over the years many improvements have been suggested but no particular attention has been paid to the nature of input spaces, the majority of the research undertaken ignores the fact the by focusing on distinguished features of discrete input spaces more simplified and robust algorithms can be developed. The main focus of this thesis is a special case of function approximation problems that take on discrete variables only.

6.1 Summary of Thesis

A survey of results on universal approximations properties followed, by a detailed analysis of simplified NN approach, along with a discussion on special features of discrete input spaces, provides us theoretical basis for further work. We then proposed simplified Neural Network algorithm I and II for function approximation in our selected domain i.e. functions defined on discrete input spaces with high dimensional-low sample case.

Experimental analysis, evaluation and comparison of these simplified Neural Network based algorithms have shown that these algorithms work well in the following situations:

- Limited availability of training data is the main reason for choosing SNN over standard NNs because any networks performance mainly depends on the number of training examples. Therefore, in the absence of adequate training data, it is hard for standard Neural Network to show high level of accuracy, which ultimately justifies the use of these simplified methods.

- When the input variables are independent of each other, it is easier to use aggregation methods, described in simplified algorithms. This will result in good initial starting solution which is the main objective behind using aggregation methods.

In order to investigate wider implications of the simplified Neural Network approach, we extended our approach to regression boosting problems. After a detailed analysis of existing regression boosting schemes, a simplified regression boosting approach was introduced. Based on the simplified regression boosting approach, we proposed algorithm-III, which is used for boosting regression problems in our selected domain.

6.1.2 Some Limitations

Like any other algorithms, these simplified algorithms have some limitations as well. Application of these algorithms to benchmarking data and examples have shown that it is hard to achieve desired results if the independent variables have too much variation, there are variables which take on continuous values, the number of values a discrete variable can take on is very large, and the input variables are not independent of each other.

The transformation phase of these algorithms may cause independent discrete variables to be continuous; thus requiring more parameters to achieve the desired approximation accuracy. Therefore special care is required while selecting a linear map that transforms multiple inputs to unidirectional data. Selection of an appropriate mapping, which can achieve desired accuracy, is a trivial task and hence proves the fact that functions defined on discrete input spaces have arbitrarily separable hierarchical structure which is not unique. Algorithm-II is not prone to this phenomenon, since each input is dealt separately.

The algorithms were implemented and their performance was compared with standard Neural Network models. Experimental results obtained so far, show that these schemes work in practice and have shown to achieve sufficient approximation accuracy. In most of the cases we have achieved approximately the same accuracy or even better by employing much less parameters as compared to standard NN models.

6.2 Future work

The results obtained in this research have many extensions which can be explored in order to carry out future research. One of the most obvious extensions is to extend our selected application domain to include mix input variables i.e. some inputs are discrete and some inputs are continuous. This extended simplified approach has already been discussed in [ZGKL05]. The idea is to use certain inputs as groups, and rather than having a single input Neural Network model, use more inputs, each representing separate groups. We can further extend this simplified approach to replace the lower level system with fuzzy systems or rule based system i.e. simplified neural fuzzy systems, see [ZK08][ZGKL05].

As highlighted in chapter 4, the simplified Neural Network approach uses ridge activation functions in the hidden layer. There are many other types of activation functions available for use in hidden layer, especially radial-basis activation functions, which have recently become very popular. The simplified Neural Network approach can therefore be investigated with other activation functions. Neural Network based ensemble methods have also become very popular mainly due to the fact many Neural Network models can generally produce better results than a single model. As shown in our simplified regression boosting scheme this approach can be applied to Neural Network based ensembles models. There are many other ensembles that can be investigated for application of these simplified methods.

Success of any Neural Network based model largely depends on the availability and reliability of training data. However, availability of data for certain application domains is always limited for different reasons e.g. LMCP modeling, QSAR modeling and many other. These schemes can be applied to many other application domains, where we are limited by the availability of data due to different reasons. One such example is the health care data, especially in United Kingdom, where access to patient related information is very restricted due to strict data protection rules. Health informatics itself, is a vast field and the opportunities for inter-disciplinary research employing these simplified methods for developing decision support systems, are immense.

6.3 Published Work

The outcomes of this research work have been formalized and have appeared in following paper:

- *Syed Shabbir Haider, Xiao-Jun Zeng, Simplified Neural Networks algorithm for function approximation on discrete input spaces in high dimension-limited sample applications, Neurocomputing, Volume 72, Issues 4-6,January 2009, Pages 1078-1083.*

6.4 Summary

This chapter is a brief summary of the research work undertaken. It includes a detailed discussion on advantages and limitations of these simplified Neural Networks. We have also highlighted future research directions in this field, followed by enlisting our published work.

Bibliography

[AB99] M. Anthony, P.L. Bartlett, Neural Networks Learning: Theoretical foundations, Cambridge University Press, 1999.

[AP97] J.G. Attali & G. Pagès. Approximations of functions by a multilayer perceptron: A new approach. Neural Networks, Vol 10, No. 6, pp. 1069–1081, 1997.

[Ara93] M. Arai. Bounds on the number of hidden units in binary valued three-layer neural networks. Neural Networks, Vol. 6, No. 6, pp 855–860, 1993.

[Bar93] A.R. Barron. Universal approximation bounds for superpositions of a sigmoidal function. IEEE Trans. Info. Theory, Vol. 39, No. 3, pp. 930–945, 1993.

[Bau88] E.B. Baum. On the capabilities of multilayer perceptrons. Journal of Complexity, Vol. 4, pp. 193–215, 1988.

[BCP97] A. Bertoni, P. Campadelli, M. Parodi, A boosting algorithm for regression. In W. Gerstner, A. Germond, M. Hasler, and J.-D. Nicoud (Eds.), Proceedings ICANN’97, Int. Conf. on Artificial Neural Networks Berlin: Springer. Vol. V of LNCS. pp. 343–348, 1997.

[Bei98] Valeriu Beiu, On Kolmogorov's Superpositions and Boolean Functions. 5th Brazilian Symposium on Neural Networks (SBRN '98), 9-11 December 1998, Belo Hoizonte, Brazil. pp. 55-60, 1998.

[Bis95] C.M. Bishop, Neural Networks for Pattern Recognition, Oxford University Press, 1995.

[BL88] D.S. Broomhead, D. Lowe, Multivariable function interpolation and adaptive networks. Complex Systems, Vol. 2, pp. 321-335, 1988.

[Bre01] L. Breiman. Random forests. Machine Learning, Vol. 45, No. 1, pp. 5-32, 2001.

[Bre96] L. Breiman, Bagging predictors, Mach. Learning, Vol. 24, pp. 123–

140, 1996.

[Bre98] L. Breiman, Arcing classifiers. The Annals of Statistics, Vol. 26, No.3, pp. 801–849,1998.

[Bre99] L. Breiman, Prediction games and arcing algorithms. Neural Computation, Vol. 11, pp. 1493–1517, 1999.

[CC95] T. Chen and H. Chen, Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems. IEEE Transactions on Neural Networks, Vol. 6, No. 4, pp. 911-917, 1995.

[CC99] J. G. Carney, P. Cunningham, The NeuralBAG algorithm: Optimizing generalization performance in bagged neural networks, in Proceedings of the 7th European Symposium on Artificial Neural Networks, pp. 35– 40, 1999.

[CCL95] T. Chen, H. Chen and R.W. Liu. Approximation capability in C(R^n) by multilayer feedforward networks and related problems. IEEE Trans Neural Networks Vol. 6, pp. 25-30, 1995.

[Cyb89] G. Cybenko, Approximation by superpositions of sigmoidal function, Mathematics of Control, Signals and System, Vol. 2, pp. 303-314. 1989.

[CL92] K. Charles, X. L. Chui, Approximation by ridge functions and neural networks with one hidden layer, Journal of Approximation Theory, Vol. 70, No. 2, pp. 131-141, 1992.

[DH02] Nigel Duffy, David Helmbold Boosting Methods for Regression, Machine Learning, Vol. 47, pp. 153–200, 2002.

[Dru97] H. Drucker, Improving regressors using boosting techniques. In Proceedings of the Fourteenth International Conference on Machine Learning, pp. 107–115), 1997.

[ET93] B. Efron, and T. J. Tibshirani, An Introduction to the Bootstrap, Chapman and Hall, New York, 1993.

[FHT00] Jerome H. Friedman, T. Hastie, and R. Tibshirani Additive logistic regression: a statistical view of boosting (With discussion and a rejoinder by the authors). Annals of Statistics Vol. 28, No. 2, pp. 337-407. 2000.

[FM98] Y. Fukuoka, H. Matsuki, A Modified Back-propagation Method to Avoid Local Minima, Neural Networks, , Vol. 11, pp. 1059-1072,1998.

[Fri01] J. H. Friedman, Greedy function approximation: A gradient boosting machine. Annals of Statistics, 2001.

[Fri91] J. H. Friedman, Multivariate Adaptive Regression Splines, Annals of Statistics Vol. 19, pp. 1-82. 1991.

[FS96] Y. Freund, and R. E. Schapire, Experiments with a new boosting algorithm, in Proceedings of the Thirteenth International Conference on Machine Learning, pp. 148– 156, 1996.

[FS97] Y. Freund, R. E. Schapire, A decision-theoretic generalization of on-line learning and an application to boosting. Journal of Computer and System Sciences, Vol. 55, No.1, pp. 119–139, 1997.

[GB00] H. Altay Guvenir and I. Uysal, Bilkent University Function Approximation Repository, See URL: http://funapp.cs.bilkent.edu.tr, 2000.

[GP90] F. Girosi and T. Poggio Networks and the best approximation property. Biological Cybernetics Vol. 63, pp. 169-176, 1990.

[GTMc08] Leila Ait Gougam, Mouloud Tribeche, Fawzia Mekideche-Chafa, A systematic investigation of a neural network for function approximation, Neural Networks, Vol. 21, No. 9, pp. 1311-1317, 2008.

[Hay96] S. Haykin Neural Networks-A Comprehensive Foundation, Macmillan College Pub., New York. 1996.

[HC07] G.B. Huang, L. Chen, Convex incremental extreme learning machine, Neurocomputing Vol. 70 (16–18), pp. 3056–3062. 2007.

[HCS06] Guang-Bin Huang, Lei Chen, Chee-Kheong Siew, Universal Approximation Using Incremental Constructive Feedforward Networks With Random Hidden Nodes, IEEE transactions on neural networks, Vol. 17, No. 4, 2006.

[HDB96] Martin Hagan, Howard Demuth and Mark Beale Neural Network Design, (Oklahoma State University), 1996.

[HDJ02] M. Hagan, H. Demuth, O. De Jesus, An Introduction to the Use of Neural Networks in Control Systems, International Journal of Robust and Nonlinear Control, Vol. 12, No. 11, pp. 959-985, 2002.

[HeN87] R. Hecht-Nielsen Kolmogorov's mapping neural network existence theorem. In: Proceedings Int Conf on Neural Networks, IEEE Press, New York, Vol. 3, pp. 11-13, 1987.

[HG92] C.M Higgins, R.M Goodman, Learning fuzzy rule-based neural networks for function Approximation International Joint Conference on Neural Networks, IJCNN 7-11, Vol. 1, pp. 251 – 256, 1992.

[Hor91] K. Hornik, Approximation capabilities of Multilayer Feedforward Networks, Neural Networks, Vol. 4, pp. 251-257, 1991.

[Hor93] K. Hornik, Some new results on neural network approximation, Neural Networks, Vol. 6, pp. 1069-1072, 1993.

[Hor98] K. Hornik, The random subspace method for constructing decision forests. IEEE Transaction on Pattern Analysis and Machine Intelligence, Vol. 20, No. 8, pp. 832-844, 1998.

[HS90] L. K. Hansen and P. Salamon. Neural network ensembles. IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 12, No. 10, pp.993-1001, 1990.

[HSS05] G. B. Huang, P. Saratchandran, Narasimhan Sundararajan,, A Generalized Growing and Pruning RBF (GGAP-RBF) Neural Network for Function Approximation, IEEE transactions on Neural Networks, Vol. 60, No.1, pp 57-67, 2005.

[HSW89] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators, Neural Networks, Vol. 2, pp. 359-366. 1989.

[HZ09] S. S. Haider, and X. Zeng, Simplified neural networks algorithm for function approximation on discrete input spaces in high dimension-limited sample applications. Neurocomputing Vol. 72 (4-6), pp. 1078-1083, 2009.

[HZS06] G.B. Huang, Q.-Y. Zhu, C.K. Siew, Extreme learning machine: theory and applications, Neurocomputing, Vol. 70, pp. 489–501, 2006.

[JJ94] M. I. Jordan and R. A. Jacobs. Hierarchical mixtures of experts and the em algorithm. Neural Computation, Vol. 6, No. 2, pp.181-214, 1994.

[Kas96] N. K. Kasabov, Fonundation of Neural Networks, fuzzy systems and knowledge engineering, Massachusetts Institute of technology, 1998.

[KKK97] V. Kurková, P.C. Kainen & V. Kreinovich. Estimates of the number of hidden units and variations with respect to half spaces. Neural Networks, Vol. 10, No. 6, pp.1061–1068, 1997.

[KP99] S. V. Kamarthi, S. Pittner, Accelerating Neural Network Training using Weight Extrapolations, Neural Networks, Vol. 12, pp. 1285-1299, 1999.

[KS04] F.O. Karray and C. De Silva, Soft Computing and intelligent systems design, pp. 236 Addison Wesley 2004.

[KS96] B. Kröse, Patrick V.D. Smagt, An Introduction to Neural Network. 8th Edition, The University of Amsterdam, 1996

[Kur92] V. Kurková. Kolmogorov’s theorem and multilayer neural networks, Neural Networks, Vol. 5, No. 4, pp. 501–506, 1992.

[LLPS93] M. Leshno, V. Y. Lin, A. Pinkus, S. Schocken, Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. Neural Networks, Vol. 6, No. 6, pp. 861-867, 1993.

[LMB03] Fei-Long, Li You-Mei XU and Zong-Ben, L^p simultaneous approximation by neural networks with one hidden layer. Journal of Software, Vol. 14, No. 11, pp. 1869-1874, 2003.

[LY97] Y. Liu and X. Yao. Negatively correlated neural networks can produce best ensembles. In Australian Journal of Intelligent Information Processing Systems Vol.4 (3/4), pp. 176-185, 1997.

[LY99] Y. Liu and X. Yao. Ensemble learning via negative correlation. Neural Networks, Vol. 12, No. 10, pp.1399-1404, 1999.

[Mas93] T. Masters, Practical Neural Network Recipes in C++. Academic Press, Inc., 1993.

[Med98] David A. Medler, A Brief History of Connectionism, Neural Computing Surveys Vol. 1, pp. 61-101, 1998.

[MH97] H.N. Mhaskar, Nahmwoo Hahm, Neural Networks for function approximation and System Identification, Neural Computation Vol. 9, pp. 143-159, 1997.

[MVA99] G. D. Magoulas, M. N. Vrahatis, and G. S. Androulakis Improving the Convergence of the Backpropagation Algorithm Using Learning Rate Adaptation Methods, Neural Computation Vol. 11, pp. 1769–1796, 1999.

[PG90] T. Poggio & F. Girosi. A theory of networks for approximation and learning, Networks for approximation and learning, Proc. IEEE (Special Issue on Neural Networks), Vol. 78, No. 9, pp. 1481–1497, 1990.

[Pin99] Allan Pinkus, Approximation theory of MLP model in Neural Networks, Acta Numerica, pp. 143-195. 1999.

[PS91] J. Park and I. W. Sandberg, Universal approximation using radial-basis function networks, Neural Computation, Vol. 3, pp. 246-257, 1991.

[PS93] J. Park & I.W. Sandberg. Approximation and radial-basis function networks. Neural Computation, Vol. 5, No. 3, pp. 305–316, 1993.

[RCg05] Fabrice Rossi and Breuc Conan-Guez Functional multi-layer perceptron: a non-linear tool for functional data analysis , Neural Network, Vol. 18, No.1, pp. 45-60, 2005.

[Rip96] B.D. Ripley, Pattern Recognition and Neural Networks. Cambridge University Press, 1996.

[RJ99] R.D. Reed and Robert J. Mark, Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks. The MIT Press, 1999.

[RMR99] G. Ridgeway, D. Madigan, & T. Richardson, Boosting methodology for regression problems. In D. Heckerman, & J. Whittaker (Eds.), Proc. Artificial Intelligence and Statistics, pp. 152–161, 1999.

[RS03] M Ananda Rao, J. Srivinas, Neural Networks, Algorithms and Applications, 2nd Edition, 2003.

[Sar97] W.S. Sarles, Neural Network FAQ, periodic posting to the Usenet newsgroup see comp.ai.neural-net, see URL: ftp://ftp.sas.com/pub/neural/FAQ.html, 1997.

[SCAa05] Robert J. Schilling, James J. Carroll, Ahmad F. Al-Ajlouni, Approximation of Nonlinear Systems with Radial Basis Function Neural Networks IEEE Transactions on Neural Networks, Vol. 12, No. 1, January 2001.

[Sch90] R. E. Schapire, The strength of weak learnability, Mach. Learning, 5, 197–227, 1990.

[SH96] R. Salomon, J. L. Hemmen, Accelerating Backpropagation through Dynamic Self-Adaptation, Neural Networks, Vol. 9, pp. 589-601, 1996.

[SL02] Ratsko R. Selmic and Frank L. Lewis, Neural Network Approximation of piecewise continuous functions: Application to Friction compensation, IEEE transactions on Neural Networks, Vol. 13, No. 3, pp. 745-751, 2002.

[SM02] Jeff Schneider, Andrew Moore, Active Learning in Discrete Input Spaces, Auton Paper, 2002. See URL: http://www.autonlab.org/autonweb/papers/y2002/14677.html.

[Spr97] D.A. Sprecher. A numerical implementation of Kolmogorov's superpositions, Neural Networks, Vol. 10, No. 3, pp. 447– 457, 1997.

[SS96] Christos Stergiou and Dimitrios Siganos Neural Networks-Online Technical Report. Department of computing, Imperial college of Science technology and Medicine, Surprise, Vol. 4, 1996.

[ST98] Franco Scarselli & AH Chung Tsoi, Universal Approximation using Feedforward Neural Networks: A survey of some existing methods, and some new results, Neural Networks, Vol. 11, No.1, pp. 15-37, 1998.

[Sti99] M.B Stinchcombe, Neural Network approximation of continuous functionals and continuous functionals on compactifications, Neural Networks, Vol. 12, pp. 467-477, 1999.

[TKG03] D. Tikk, L.T. Kóczy, T.D. Gedeon, A survey on the universal approximation and its limits in soft computing techniques International Journal of Approximate Reasoning, Vol. 33, No. 2, pp. 185-202, 2003.

[Wag02] W. P. Wagner, Daily Peak Load Electricity Forecasting using Artificial Neural Networks.2002. See URL: http://hsb.baylor.edu/ramsower/acis/papers/wagnerw.htm

[Wal90] S. F. Walker. A brief history of connectionism and its psychological implications. AI & Society, Vol. 4, pp. 17-38, 1990.

[Wat80] G. A. Watson, Approximation Theory and Numerical Methods, New York, John Wiley and Sons, 1980.

[WGG95] Jonathan Wray, G. Gary, R. Green, Neural networks, approximation theory, and finite precision computation, Neural Networks, Vol. 8, No. 1, Pages 31-37, 1995.

[Yao99] Xin Yao, Evolving Artificial Neural Networks, Proceeding of the IEEE, Vol. 87, No.9, pp.1423-1447, 1999.

[ZGKL05] Xiao-Jun Zeng, John Yannis Golermas, John A. Keane and Panos Liatsis, Approximation capabilities of Hierarchical Neural-fuzzy systems for function approximation on discrete spaces, International Journal of Computational Intelligence, Vol. 1, No.1, pp. 29-41, 2005.

[Zha99] J. Zhang, Developing robust non-linear models through bootstrap aggregated neural networks, Neurocomputing, Vol. 25(1–3), pp. 93–113. 1999.

[ZK05] Xiao-Jung Zeng, John A. Keane, Approximation capabilities of hierarchical fuzzy systems, IEEE Transactions on Fuzzy Systems Vol. 13, No. 5, pp. 659–672, 2005.

[ZK08] Xiao-Jun Zeng and John A. Keane, Hierarchical fuzzy systems for function approximation on discrete input spaces with Application, IEEE Transactions on Fuzzy Systems, Vol. 16, No. 5, pp. 1197-1215, 2008.

[ZP01] R. S. Zemel, T.A Pitassi, Gradient-Based Boosting Algorithm for Regression Problems Advances In Neural Information Processing Systems, No.13, pp. 696-702. 2001.

[ZP08] Zarita Zainuddin and Ong Pauline. Function approximation using artificial neural networks. WSEAS Trans. Math. Vol. 7, No. 6, pp. 333-338, 2008.

Appendix-A: Backpropagation Algorithms for Standard Neural Network Models.

We can define a standard Neural Network for function approximation problems as shown in equation (1.1). Note that it has been proved and widely accepted that Neural Networks with one hidden layer of sigmoid-activation neurons and an output layer of linear neurons are universal function Approximators i.e. they can approximate any reasonable function to arbitrary accuracy. More precisely, according to the definition of famous (Cybenko, 1989) theorem as:

"let σ be any continuous sigmoid-type function (e.g. $\sigma(\xi) = 1/ (1+e^{-\xi})$). Then any continuous real-valued function 'f' on $[0,1]^n$ (or any other compact subspace of R^n) and $\xi > 0$, there exists vectors $a_1, a_2 \ldots\ldots\ldots a_n$, b , c_i & c_o and a parameterized function $Y(\cdot, a, b, c) : [0,1]^n \rightarrow R$ such that:

$$|Y(x , a, b , c) – f (x)| < \xi \qquad \text{for all } x \in [0,1]^n$$

Where,

$$Y = (x,a,b,c) = NN(X) = \sum_{i=1}^{N} c_i \, \sigma(a'_i \, X + b) + c_0 \qquad (1.1)$$

And $a_i \in R^n$ & c_i , c_o & $b \in R$, where $a = (a_1,a_2\ldots\ldots, a_n)$, $c = (c_1,c_2\ldots\ldots, c_n)$ and $b = (b_1,b_2\ldots, b_n)$". Also note that 'a_i' is '*d* x *1*' vector usually referred to as the direction of the ridge function.

<u>Deriving The BP Algorithm For MLPs</u>

Let,

- $X = (x_1,...,x_n)$ are input variable.
- $y_m \in R$ is the output variable, 'm' is the layer index and denotes output layer, the index of the layer just below output layer will be 'm-1' and 'm-2' and so on.

- a_{ji} is the connection weight going from input **'i'** to hidden layer neuron **'j'**. And can be represented in matrix form as shown below:

$$a_{ji} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1i} \\ a_{21} & a_{22} & \cdots & a_{2i} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ a_{j1} & a_{j2} & \cdots & a_{ji} \end{bmatrix}$$

- b_j is the bias attached to hidden layer neuron 'j',
- Where as c_i & c_0 are the connection weight and bias from hidden layer to output layer respectively.
- σ is the activation function and in the case of sigmoidal neurons, $\sigma(x) = \frac{1}{1 + \exp^{-x}}$, and in the case of linear neurons it will be $\sigma(x) = x$.

  Since
- The output of hidden layer neuron ***'j'*** in the layer **'m-1'** will be;

$$y_j^{m-1} = \sum_{j=1}^{N} \sigma(a'_{ji} x_i + b_j) \qquad (1.2)$$

Where the subscript 'i' represents the ith input variable 'x'.
In vector/ matrix form this can be seen as:

- The net input to our hidden layer neurons will be:

$$net_j^{m-1} = \sum_{j=1}^{N} \sigma(a'_{ji} x_i + b_j) \qquad (1.3)$$

- The output of the last layer will be the same as its net input since the output layer uses the linear neurons. So the output of neuron **'i'** in the layer **'m'** (which is last layer) will be:

$$y_i^m = \sum_{i=1}^{n} c_i^m \; y_j^{m-1} + c_0^m \qquad (1.4)$$

where y_j^{m-1} can be computed as shown in equation (1.2).

Performance Index:

We know that our training set is of the form:

$\{\boldsymbol{X}_1, \mathbf{t}_1\}\{\boldsymbol{X}_2, \mathbf{t}_2\}\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\{\boldsymbol{X}_k, \mathbf{t}_k\}$ (1.5)

Where $\boldsymbol{X}_k$ is the input vector and $\mathbf{t}_k$ is the corresponding target value and k = 1….p represents the *‘kth’* iteration or pattern.

Let **‘W’** denote all the network parameters i.e. $W = \left[a_{ji}, b_i, c_i, c_0\right]$. Our objective is to minimize the cost function or the error measure i.e. sum of squared errors over whole the training set/ patterns which can be defined as:

$$\nabla E(W) = \frac{1}{2}\sum_{i=1}^{n}\sum_{k=1}^{p}\left(t_i(k) - y_i(k)\right)^2 \qquad (1.6)$$

And in the vector case we can define the above as:

$$\mathbf{E}(\mathbf{W}) = \sum\left[\mathbf{e}^{\mathbf{t}}\mathbf{e}\right] = \sum\left[(t-y)^T (t-y)\right] \qquad (1.7)$$

Where ‘***e***’ is the sum of squared errors over all the training patterns. Therefore the approximate mean square error over a single sample (k) would be:

$$E^{\wedge}(X) = \mathbf{e}^T(k)\mathbf{e}(k) = \left(t(k) - y(k)\right)^T\left(t(k) - y(k)\right) \qquad (1.8)$$

The Generalised Delta Rule/ Approximate Steepest Descent For Weight/ Bias Update:

We can define the approximate steepest descent or generalised delta rule for MLP’s as follows:

$$W^{new} = W^{old} + \Delta W \quad , \quad \text{where } W = \left[a_{ji}, b_i, c_i, c_0\right] \qquad (1.9)$$

And,

$$\Delta W = -\eta\frac{\partial \hat{E}(W)}{\partial W}, \quad \text{where ‘}\eta\text{’ is the learning rate} \qquad (2.0)$$

In the vector case we can write the equations (1.9) and (2.0) altogether as:

$$w(k+1) = w(k) - \eta \frac{\partial E^{\wedge}}{\partial w} \qquad (2.1)$$

where ‘k’ represents the ‘kth’ iteration or pattern.

Gradient Calculation

Now we have to compute the gradients $\frac{\partial E^{\wedge}}{\partial w} = \left[\frac{\partial E^{\wedge}}{\partial c_i}, \frac{\partial E^{\wedge}}{\partial c_i}, \frac{\partial E^{\wedge}}{\partial a_{ji}}, \frac{\partial E^{\wedge}}{\partial b_i} \right]$, by

using the chain rule of differentiation as follows:

$$\frac{\partial E^{\wedge}}{\partial C_i} = \frac{\partial E^{\wedge}}{\partial net_i^m} \wedge \frac{\partial net_i^m}{\partial C_i} \quad \text{and} \quad \frac{\partial E^{\wedge}}{\partial C_0} = \frac{\partial E^{\wedge}}{\partial net_i^m} \wedge \frac{\partial net_i^m}{\partial C_0} \qquad (2.2)$$

$$\frac{\partial E^{\wedge}}{\partial a_{ji}} = \frac{\partial E^{\wedge}}{\partial net_j^{m-1}} \wedge \frac{\partial net_j^{m-1}}{\partial a_{ji}} \quad \text{and} \quad \frac{\partial E^{\wedge}}{\partial b_i} = \frac{\partial E^{\wedge}}{\partial net_j^{m-1}} \wedge \frac{\partial net_j^{m-1}}{\partial b_i} \qquad (2.3)$$

Let $\frac{\partial E^{\wedge}}{\partial net_{i,j}^{m,m-1}} = s_{i,j}^{m,m-1}$, be the sensitivity/ error signal for the output and hidden layers respectively. From the network definition above we can see that we have to compute the following gradients inline with the eqns. (2.2) and (2.3) above :

$$\frac{\partial net_i^m}{\partial C_i} = \frac{\partial}{\partial C_i}\left[\sum_{i=1}^{n} c_i y_j^{m-1} + c_0 \right] \text{ and } \frac{\partial net_i^m}{\partial C_0} = \frac{\partial}{\partial C_0}\left[\sum_{i=1}^{n} c_i y_j^{m-1} + c_0 \right]$$

$$\frac{\partial net_i^m}{\partial C_i} = y_j^{m-1} \text{ and } \frac{\partial net_i^m}{\partial C_0} = 1 \qquad (2.4)$$

Similarly,

$$\frac{\partial net_j^{m-1}}{\partial a_{ji}} = \frac{\partial}{\partial a_{ji}}\left[\sum_{j=1}^{N} a'_{ji} x_i + b_i \right] \text{ and } \frac{\partial net_j^{m-1}}{\partial b_i} = \frac{\partial}{\partial b_i}\left[\sum_{j=1}^{N} a'_{ji} x_i + b_i \right]$$

$$\frac{\partial net_j^{m-1}}{\partial a_{ji}} = x_i \text{ and } \frac{\partial net_j^{m-1}}{\partial b_i} = 1 \qquad (2.5)$$

Now we can re-write our steepest descent rule in equation (2.3) as follows:

1. For output layer weight and bias values:

$$c_i(k+1) = c_i(k) - \eta s_i^m y_j^{m-1} \quad , \quad c_0(k+1) = c_0(k) - \eta s_i^m \tag{2.6}$$

2. For hidden layer weight and bias values:

$$a_{ji}(k+1) = a_{ji}(k) - \eta s_j^{m-1} x_i \quad , \quad b_i(k+1) = b_i(k) - \eta s_j^{m-1} \tag{2.7}$$

<u>Computing The Sensitivities (Back Propagation Of Error)</u>

The only thing to left to be computed is the sensitivities i.e. $\frac{\partial E^\wedge}{\partial net_{i,j}^{m,m-1}} = S_{i,j}^{m,m-1}$.

This is the process which gives the name of back propagation to this algorithm. Note that the sensitivities are computed by starting at the last layer, and then propagating backwards through the network to the first layer. i.e. S^M → S^{M-1} → ….. → S^2 → $S^{1.}$

For the last/ output layer this sensitivity or error signal (i.e. how the error at the output is affected by the net input ‘i’) can be easily computed as follows:

$$S_i^m = \frac{\partial E^\wedge}{\partial net_i^m} = \frac{\partial}{\partial net_i^m}\left[\frac{1}{2}\sum_{i=1}^{n}\sum_{k=1}^{p}\left(t_i^m(k) - y_i^m(k)\right)^2\right]$$

$$= -\left(t_i(k) - y_i(k)\right)\frac{\partial y_i(k)}{\partial net_i^m} \tag{2.8}$$

Where the term $\frac{\partial y_i(k)}{\partial net_i^m}$ is actually the derivative of our activation function

$$\text{i.e.} \frac{\partial y_i}{\partial net_i^m} = \frac{\partial \sigma(net_i^m)}{\partial net_i^m} = f'(net_i^m) \tag{2.9}$$

Note that in the case of sigmoidal neurons it will be

$$\frac{\partial}{\partial(x)}\left[\frac{1}{1+\exp^{-x}}\right] = \left[\frac{\exp^{-x}}{\left(1+\exp^{-x}\right)^2}\right] = \left[1 - \frac{1}{1+\exp^{-x}}\right]\left[\frac{1}{1+\exp^{-x}}\right]$$

$= (1 - x_i)\, x_i$ and in the case of linear neurons it will be:

$$\frac{\partial}{\partial(x)}(x) = x \tag{3.0}$$

Therefore, we can see that the sensitivity or error signal for output layer will be,

$$S_i^m = -\left(t_i^m - y_i^m\right) f^{'}\left(net_i^m\right) \qquad (3.1)$$

From here we can now compute the sensitivity of the hidden layer. Note that the error at hidden layer is not a direct function of its weight and bias. It is an accumulation of error from the layer just after this. So, we need another Application of chain rule of differentiation to compute this error signal.

$$S_j^{m-1} = \frac{\partial E^\wedge}{\partial net_j^{m-1}} = \frac{\partial E^\wedge}{\partial net_i^m} \frac{\partial net_i^m}{\partial net_j^{m-1}} \qquad (3.2)$$

Note that we have already computed the first term $\frac{\partial E^\wedge}{\partial net_i^m} = S_i^m$ in equation (3.1).

Therefore, we are left with,

$$\frac{\partial net_i^m}{\partial net_j^{m-1}} = \frac{\partial}{\partial net_j^{m-1}}\left[\sum_{i=1}^{n} c_i y_j^{m-1} + c_0\right] = c_i \frac{\partial y_j^{m-1}}{\partial net_j^{m-1}} \qquad (3.3)$$

$$\frac{\partial y_j^{m-1}}{\partial net_j^{m-1}} = \frac{\partial \sigma\left(net_j^{m-1}\right)}{\partial net_j^{m-1}} = f^{'}\left(net_j^{m-1}\right) \qquad (3.4)$$

$f^{'}\left(net_j^{m-1}\right)$’ is the derivative of activation function and can be computed following the derivation depicted in equations (2.9) and (3.0).

By combining (3.3) and (3.4) we get,

$$S_j^{m-1} = S_i^m c_i f^{'}\left(net_j^{m-1}\right) \qquad (3.5)$$

We can now obtain the updated weight and bias values for our network by substituting the sensitivities or error signal obtained in equation (3.1) and (3.5) into (2.6) and (2.7) respectively.

Jacobian Matrix

Note that the vector/ matrix representation of the term $\frac{\partial net_i^m}{\partial net_j^{m-1}}$ computed in equation (3.3) is of the form:

$$\frac{\partial net_i^m}{\partial net_j^{m-1}} = \begin{bmatrix} \frac{\partial net_1^m}{\partial net_1^{m-1}} & \frac{\partial net_1^m}{\partial net_2^{m-1}} & \ldots\ldots & \frac{\partial net_1^m}{\partial net_j^{m-1}} \\ \frac{\partial net_2^m}{\partial net_1^{m-1}} & \frac{\partial net_2^m}{\partial net_2^{m-1}} & \ldots\ldots & \frac{\partial net_2^m}{\partial net_j^{m-1}} \\ . & . & & . \\ . & . & \ldots\ldots & . \\ . & . & & . \\ \frac{\partial net_i^m}{\partial net_1^{m-1}} & \frac{\partial net_i^m}{\partial net_2^{m-1}} & \ldots\ldots & \frac{\partial net_i^m}{\partial net_j^{m-1}} \end{bmatrix} = c_i^m = f^{'}\left(net_j^{m-1}\right) \qquad (3.6)$$

Where as $$f^{'}\left(net_j^{m-1}\right) = \begin{bmatrix} \frac{\partial\sigma\left(net_1^{m-1}\right)}{\partial net_1^{m-1}} & 0 & \ldots\ldots & 0 \\ 0 & \frac{\partial\sigma\left(net_2^{m-1}\right)}{\partial net_2^{m-1}} & \ldots\ldots & 0 \\ . & & . & . \\ . & 0 & . & . \\ . & & . & . \\ 0 & 0 & \ldots\ldots & \frac{\partial\sigma\left(net_j^{m-1}\right)}{\partial net_j^{m-1}} \end{bmatrix} \qquad (3.7)$$

Therefore $\frac{\partial net_i^m}{\partial net_j^{m-1}} = c_i^m = f^{'}\left(net_j^{m-1}\right)$.

Appendix-B: Description of Data Sets

The Pyrimidines and Triazines data sets are taken from UCI Machine Learning Repository. A brief description of their past usage and original sources is given below.

A. Title of Database: Pyrimidines

1. Sources: Luis Torgo
   http://www.ncc.up.pt/~ltorgo/Regression/DataSets.html

2. Relevant Information: The task consists of Learning Quantitative Structure Activity Relationships (QSARs). The Inhibition of Dihydrofolate Reductase by Pyrimidines. The data and methodology are described in:

- R. D. King, S. Muggleton, R. A. Lewis, M. J. Sternberg, Drug Design by machine learning: the use of inductive logic programming to model the structure-activity relationships of trimethoprim analogues binding to dihydrofolate reductase. Proceedings of The National Academy of Sciences , Vol. 89, Issue 23, Pages 11322-11326, 1992.

5. Number of Instances: 74

6. Number of Attributes: 27 + 1 Response variable i.e. activity

7. Missing Attribute Values: None

B. Title of Database: Triazines

1. Sources: Luis Torgo
   http://www.ncc.up.pt/~ltorgo/Regression/DataSets.html

2. Relevant Information: The problem is to learn a regression equation, rule or tree to predict the activity from the descriptive structural attributes. The data and methodology is described in detail in:

- Ross D. King, Jonathan D. Hirst and Michael J.E. Sternberg, A comparison of artificial intelligence methods for modelling QSARs, Applied Artificial Intelligence, Vol. 9, Issue 2, Pages 213-233, 1995.

- Jonathan D. Hirst, Ross D. King and Michael J.E. Sternberg, Quantitative Structure-Activity Relationships by Neural Networks and inductive logic programming. I. The inhibition of dihydrofolate reductase by triazines. Journal of Computer Aided Molecular Design, Vol. 8, Issue 4, Pages 405-420, 1994.

3. Number of Instances: 186

4. Number of Attributes: 60 + 1 Response variable i.e. activity

5. Missing Attribute Values: None

C. Title of Database: F1

1. Sources:

(a) Original owners of database:

This is an artificial data set used by J.H. Friedman (1991) for MARS.

-BREIMAN, L. (1996): Bagging Predictors. Machine Learning, Vol. 24, Issue 3, Pages 123-140. Kluwer Academic Publishers.

-FRIEDMAN, J. (1991): Multivariate Adaptive Regression Splines. Annals of Statistics, Vol. 19, Issue 1, Pages 1-82.

2. Relevant Information: The cases are generated using the following method: Generate discrete values of 5 attributes, X_1, ..., X_5 independently each of which uniformly distributed over [0,1]. Obtain the value of the target variable Y using the equation below:

$$y = 10\sin(\pi x_1 x_2) + 20(x_3 - .5)^2 + 10x_4 + 5x_5$$

3. Number of Instances: 100

4. Number of Attributes: 5

5. Missing Attribute Values: None

Appendix-C: Proof of Theorem 2

For the given input space U , based on Theorem 1 in [ZGKL05], there exists a linear function: $z = L(X) = w_0 + \sum_{i=1}^{n} w_i x_i$ (C.1)

which is one to mapping from U to R . For every

$$X_{k_1 k_2 \ldots k_n} = \left(u_{1,k_1}, u_{2,k_2}, \ldots, u_{n,k_n}\right) \in U = \mathop{\times}_{i=1}^{n} U_i \text{ where } k_i = 1,2,\ldots,N_i \quad l = 1,2,\ldots,n \quad \text{(C.2)}$$

Define:

$$z_{k_1 k_2 \ldots k_n} = L\left(X_{k_1 k_2 \ldots k_n}\right) \quad \text{(C.3)}$$

That is, $z_{k_1 k_2 \ldots k_n}$ is the function value of $L(X)$ at $X_{k_1 k_2 \ldots k_n}$ and the set of all such values is denoted as :

$$V = \left\{ y_{k_1 k_2 \ldots k_n} \mid k_l = 1,2,\ldots,N_i, i = 1,2,\ldots,n \right\}, \quad \text{(C.4)}$$

which is the output variable space of function $L(X)$. As $L(X)$ is one-to-one mapping, then all elements of V are different to each other. Therefore, for every $z \in V$, there exists only one element X in U such that $z = L(X)$. Further, as U is a discrete space with finite elements, then V is a discrete space with finite elements.

Now define function $g(z)$ on V as follows: For every $z \in U$, let X be the unique element in U such that $z = L(X)$. Then define the value of g at z as follows:

$$g(z) = G(X) \quad \text{(C.5)}$$

For the function g defined in the above, it can be proved by the reverse process that for all $X \in U$.

$$G(X) = g[L(X)] \quad \text{(C.6)}$$

As $g(z)$ is a function on finite discrete space V which is bounded, based on [Wat80] it can extended to be a continuous function $\hat{g}(X)$ on $\hat{V} = [\underline{z}, \overline{z}]$ (where $\underline{z} = \min_{z \in V} z$, $\overline{z} = \max_{z \in V} z$) in the sense that:

$\hat{g}(X) = g(X)$ and $z \in V$ **.** (C.7)

As $\hat{g}(X)$ is a continuous function on $\hat{V}$, then it is implied immediately from the universal approximation property of standard NNs on continuous spaces that there exists a NN $NN_1(z)$ on $\hat{U}$ such that

$$\| \hat{g} - NN_1 \|_\infty = \max_{z \in \hat{V}} | \hat{g}(z) - NN_1(z) | < \varepsilon \tag{C.8}$$

Now define a SNN as $SNN(X) = NN_1[L(X)]$, then (C.6), (C.7) and (C.8) imply that, for any $X \in U$,

$$\begin{aligned} &| G(X) - SNN(X) | = | g[L(X)] - NN_1[L(X)] | \\ &\leq \max_{z \in V} \left| g(z) - NN_1(z) \right| \\ &\leq \max_{z \in \hat{V}} | \hat{g}(z) - NN_1(z) | < \varepsilon \end{aligned}$$

which leads to $\| G - NN \| = \max_{X \in U} | G(X) - NN(X) | < \varepsilon$ and hence complete the proof.

Appendix-D: Proof of Convergence Algorithm-III

By following the same approach as in [ZP01] we can prove the standard boosting property for our simplified regression boosting algorithm in the case where all combination coefficients $c_t = 1$. Let $p_t^i = w_t^i / \sum_{j=1}^{n} w_t^j$ and ξ_t denote the error that hypothesis t makes on its distribution, $\xi_t = \sum_{i=1}^{n} p_t^i \left[\left(h_t^*(\mathbf{x}_i) - y^i \right)^2 - \tau \right]$.

Theorem: Assume that for all $t \le T$ hypothesis t makes error ξ_t on its distribution. If the combined output $\tilde{y}$ is considered to be in error iff $\left(\tilde{y} - y \right)^2 > \tau$ then the output of the boosting algorithm (after T stages) will have error at most ξ where, $\xi = P\left[\left(\tilde{y} - y^i \right)^2 > \tau \right] \le \prod_{t=1}^{T} \xi_t$.

Proof: The proof presented below is based on the approach first appeared in [78] and then followed by [ZP01]. It is shown that the sum of weights at stage *T* is bounded above by the product of the ξ_t 's , while at the same time, for each input *i* that is incorrect, its corresponding weight w_T^i at stage *T* is significant.

$$\sum_{i=1}^{n} w_{T+1}^i = \sum_i w_T^i \left[\left(h_t^*(\mathbf{x}_i) - y^i \right)^2 - \tau \right] = \xi_t \sum_i w_T^i = \prod_{t=1}^{T} \xi_t$$

The second equality holds because, $\xi_t = \left(\sum_i w_t^i \left[\left(h_t^*(\mathbf{x}_i) - y^i \right)^2 - \tau \right] / \sum_i w_t^i \right)$.

Now

let $\bar{y}.^i = \Sigma_T h_t^* \left(\mathbf{x}^i \right) / T$, then the weight of example *i* at time *t* is:

$$w_t^i = \left(\sum_t \left[\left(h_t^*(\mathbf{x}_i) - y^i \right)^2 - \tau \right] \right) = \sum_t \left[\left(\left(h_t^*(\mathbf{x}_i) - \bar{y}.^i \right) + \left(\bar{y}.^i - y^i \right) \right)^2 - \tau \right]$$

$$= T \left[\mathbf{Var}\left(h^*\left(\mathbf{x}^i \right) \right) + \left(\bar{y}.^i - y^i \right)^2 - \tau \right] \ge T \left[\left(\bar{y}.^i - y^i \right)^2 - \tau \right]$$

The last in equality holds because $= \mathbf{Var}\left(h^{*}\left(\mathbf{x}^{i}\right)\right) = 1/T \sum_{t}\left(h_{t}^{*}\left(\mathbf{x}^{i}\right) - \bar{y}.^{i}\right)^{2}$. Now consider an example input *i* that produce an error, then we have $\left(\bar{y}.^{i} - y^{i}\right)^{2} > \tau \Rightarrow w_{t}^{i} \geq 1$, if ξ is the total error rate of the combination output, then $\sum_{i} w_{t}^{i} \geq \xi$. Thus we have, $\xi \leq \sum_{i} w_{T+1}^{i} = \prod_{t=1}^{T} \xi_{t}$.

One important fact to be noticed that there are no assumption about error rate ξ_t of individual hypothesis. Also if all $\xi_t < 1 = \Delta,$ where $\Delta < 1,$ then $\xi < \Delta^{T}$.

Appendix-E: Matlab Implementation for Simplified NN Algorithms

1. Importing data into Matlab:

The first step in experiments is to import the data sets in Matlab Work Area; Matlab does support many formats; we have got the data in Excel format with all the independent and dependent variables in one file with the last column having target values.

1.1 Initialisation:

Note: Matlab provides inbuilt functions to find the best linear approximation i.e. the task of finding a line or tangent plane that best fits the given data (Simple or Multiple Regression). Matlab represents a multivariate or least squares fit model of the data as:

$$\mathbf{y = a_0 + a_1\, x_1 + a_2\, x_2 + \ldots\ldots\ldots.+ a_n\, x_n}$$

We have to solves for unknown coefficients $a_{0,}$ $a_{1,}$ $a_{2,}$ and a_n , by performing a least squares fit. For this we have to construct and solve the set of simultaneous equations by forming the regression matrix, X, and solving for the coefficients using the backslash operator.

Step 1: Input Independent and dependent variables

a) Set x_i = [observations for all the independent variables x_i , i = 1….n]' // a transpose operator is used to later set the problem in matrix form //

b) Set y = [target values for each input pattern]’

// a transpose operator is used to later set the problem in matrix form //

Step 2: Solve for the least square fit model of the data (i.e. to Find the best Linear approximation z = L*(X) = a’X + b using Least squares algorithm)

c) Construct the regression matrix ‘X’ by using Matlab command

X = [ones(size(x_1)) x_1 x_2…….x_n];

// This will generate the matrix ‘X’ with all the independent variables appearing as columns with an extra column of ones in the beginning so that we can have the constant value ‘a_o’ in the equation above.//

d) Using backslash i.e. ‘A = X\y’ to solve for unknown coefficients;

// X = A\B Denotes the solution to the matrix equation AX = B //

Step 3: Training data transformation: Transform the training data $\{[y(t), X(t)]; t = 1,2,..., M\}$ to $\{[y(t), z(t)]; t = 1,2,..., M\}$ by using $z = L^*(X) = a'X + b$;

e.1) In the case of algorithm 1 Set P = [X] * [A]

// multiplying the Input variables matrix ‘X’ with the regression equation calculated in step 2(d) //

e.2) In the case of algorithm 2 Set net.iw {1,1} = A (i.e. $a_{1,}$ $a_{2,}$ $a_{3\ldots}$)&

net.b{1} = a_0

// this will set the weights & bias for the additional layer before the one dimensional NN to be the same as the coefficients of the regression equation computed in step 2.d.//

Step 4: Forming the initial simplified NN as $y = N(z) = \sum_{i=1}^{N} c_i \sigma[\alpha z + \beta)] + c_o$

f) Set P = P'

// Setting the resultant 'P' from step 3(e) as new independent variable //

g) Set T = [target values for each input P_i]

h) Now creating the feedforward network with one hidden layer of sigmoid activation units and linear activation neuron at the output (i.e to be consistent with the conventional FF NNs used for function approximation). $y = NN(z) = \sum_{i=1}^{N} c_i \sigma[\alpha(a' X + b) + \beta)] + c_o$.

The matlab command below will be used to create the architecture as above

```
net=newff(minmax(P),[Hid_N, Out_N],{'tansig','purelin'},'traingd');
```

// Hid_N = number of neurons in hidden layer & Out_N = number of neurons in output layer; always one in our case. The function minmax is used to determine the range of the inputs to be used in creating the network.//

1.2 Iterations:

Step 5: The network will be trained using traditional back-propagation (gradient descent) algorithm to identify and update the weight and bias values for our network as depicted in Matlab command in step 4(h),

a) Use the Matlab command as below to train the network,

```
[net,tr]=train(net,P,T)
```

Note: To allow for more flexibility with experimentation we may wish to change some of the default parameters associated with network training prior to training i.e. training progress record (net.trainParam.show), choice of number of training iterations (net.trainParam.lr), learning rate (net.trainParam.epochs) & training goal i.e. desired accuracy (net.trainParam.goal).

Step 6: The network can now be simulated to check its response for the input patterns.

a) By using following command

```
a = sim(net,p)
```

1.3 Forecasting:

Step 7: Once the network has been fully trained and performance goal for the training session has been achieved, we can predict the outputs for any new input pattern as below.

a) Set f_i = [input pattern to be forecasted, f_i , i = 1….n]'

b) F = [ones(size(f_1)) f_1 f_2…….f_n];

c) Set P = [F] * [A]

// multiplying the Input variables matrix 'f' with the regression equation calculated in step 2(d) //

d) Repeat step 6 (a) to obtain your forecast.