A Novel Hybrid Learning Algorithm for Artificial Neural Networks

Ranadhir Ghosh, BE (Comp. Sc.), MIT

School of Information Technology
Faculty of Engineering and Information Technology
Griffith University - Gold Coast Campus

Submitted in fulfillment of the requirements of the degree of Doctor of Philosophy
December 2002

Abstract

Last few decades have witnessed the use of artificial neural networks (ANN) in many real-world applications and have offered an attractive paradigm for a broad range of adaptive complex systems. In recent years ANN have enjoyed a great deal of success and have proven useful in wide variety pattern recognition or feature extraction tasks. Examples include optical character recognition, speech recognition and adaptive control to name a few. To keep the pace with its huge demand in diversified application areas, many different kinds of ANN architecture and learning types have been proposed by the researchers to meet varying needs.

A novel hybrid learning approach for the training of a feed-forward ANN has been proposed in this thesis. The approach combines evolutionary algorithms with matrix solution methods such as singular value decomposition, Gram-Schmidt etc., to achieve optimum weights for hidden and output layers. The proposed hybrid method is to apply evolutionary algorithm in the first layer and least square method (LS) in the second layer of the ANN. The methodology also finds optimum number of hidden neurons using a hierarchical combination methodology structure for weights and architecture. A learning algorithm has many facets that can make a learning algorithm good for a particular application area. Often there are trade offs between classification accuracy and time complexity, nevertheless, the problem of memory complexity remains. This research explores all the different facets of the proposed new algorithm in terms of classification accuracy, convergence property, generalization ability, time and memory complexity.

Statement of Originality

This work has not previously been submitted for a degree or diploma in any university. To the best of my knowledge and belief, this thesis contains no material previously published or written by another person except where due acknowledgement is made in the thesis itself.

Signature: …….

(RANADHIR GHOSH)

Date: ……

Acknowledgement

I would like to express my gratitude to all those who gave me the opportunity to complete this thesis. I want to thank the School of Information Technology, Griffith University, Gold Coast Campus for giving me the opportunity to commence this thesis in the first instance, to do the necessary research work and to provide all the resources that were required.

I am deeply indebted to my supervisor Dr Brijesh Verma whose help, stimulating suggestions and encouragement helped me in all the time of research for and writing this thesis. Also I would like to thank my associate supervisor Dr Vallipuram Muthukkumarasamy and faculty member of our school Dr Michael Blumenstein for their valuable suggestions many a times.

It will be incomplete without acknowledging the guidance of my dear brother Samir whose guidance helped me to overcome all the difficult phases during the period. Also, I would like to give my special thanks to my wife, my beloved Guddi whose patient love enabled me to complete this work.

Table of Contents

List of Figures

List of Tables

Abbreviation

AI :	Artificial intelligence
ANN :	Artificial neural network
BT-EALS-Tx :	Binary architecture search with EALS weight updating using Tx connection; x ∈ {1,2,3}
BT-GALS-Tx :	Binary architecture search with GALS weight updating using Tx connection; x ∈ {1,2,3}
CI :	Computational intelligence
EA :	Evolutionary algorithm
EALS :	Evolutionary algorithm with least square
EAWLS :	Evolutionary algorithm without least square
EBP :	Error back propagation
EP :	Evolutionary programming
GA :	Genetic algorithm
GALS :	Genetic algorithm with least square
GAWLS :	Genetic algorithm without least square
GV1 :	Is same as LI-EALS-T3
GV2 :	Is same as BT-EALS-T3
LI-EALS-Tx :	Linear architecture search with EALS weight updating using Tx connection; x ∈ {1,2,3}
LI-GALS-Tx :	Linear architecture search with GALS weight updating using Tx connection; x ∈ {1,2,3}
LS :	Least square
MLP :	Multi layered perceptron
NP :	Non polynomial

1 INTRODUCTION

Artificial neural network (ANN), fuzzy system and evolutionary computation are part of the soft computing or computational intelligence discipline [1] [2]. In last few decades several researchers have shown a lot of interests in these areas. Each of these fields or a combination of them has made possible to use for a large number of different application areas. One of the most important constituents of soft computing is artificial neural network (ANN). The basic architecture of an ANN consists of layers of interconnected processing units. These processing units, also known as nodes or processing elements, are the functional counterparts of biological neurons. The number of layers and the degree of interconnection between nodes vary among different ANN designs. The interconnections between the nodes represent the flow of information. Frequently the inner layers of an ANN are referred to as hidden layers since they neither receive input from nor output information outside of the network. The problem domain of the ANN is commonly formulated as a multivariate nonlinear optimization problem over a very high dimensional space of possible weight configurations. The problem of finding a set of parameters for a neural network, which allows it to solve the given problem, can be viewed as a parameter optimization problem. The range of the various parameters such as weights, thresholds and time constants can be bounded between a minimum and maximum value so that the size of the search space is finite. Though having conceded the fact that search space is finite, the optimization of such a problem is a very high order time complexity problem, often known as NP hard problem. The process of finding such optimal parameters is also known as learning algorithm of the ANN [3].

A learning algorithm is at the heart of the neural network based system. Over the past decade, a number of learning algorithms have been developed. However, in most cases learning or training of a neural network is based on a trial and error method. There are many fundamental problems such as a long and uncertain training process, selection of network topology and parameters that still remain unsolved. Learning can be considered as a weight-updating rule of the ANN.

Almost all the neural learning methods strictly depend on the architecture of the ANN. The connection rules determine the topological structure of an ANN. Two most important factors in the weight updating rules are the ordering of the nodes and the connection mechanism of the degree of inputs to the ANN. The number of nodes determines the dimensionality of the problem. The response of the ANN can be realized as a function of the weights of an ANN. These surface areas, which can be highly complex, are the areas of concern for a learning algorithm, so that a decision hyper-plane can be constructed by proper combinations of the weight vectors. There are many problems associated in the learning algorithm. For a classification, where the generalization ability of the ANN is important in terms of classification, because of the complexity of the error surface, it may be very difficult for the learning algorithm to achieve this goal, because many of the existing learning algorithms are very much depended upon the error surface information as a priori knowledge. Also, getting trapped in a sub-optimal solution is also a possibility. Another problem can be the step size of a learning process. A very high step size, which means a faster learning, can miss an optimal solution easily where as a very low step size can mean a very high time complexity for the learning process. There is a possibility that the step size can be adapted during the learning process, so that near optimal solution the value becomes less.

One of the learning techniques that attracted the researchers is to apply genetic algorithm based technique [4] [5]. The genetic algorithm is based on the "survival of the fittest" theory of the natural evolution. One of the major problems that can be solved using this method is that because of the stochastic nature of this algorithm the learning process can reach an optimal solution with much higher probability than many standard neural based techniques, which are based on so much on the gradient information of the error surface. One of the problems though, with this global search based technique is the time complexity of the algorithm. For a very large application size, a very powerful computation facility is required to solve the problem. There are other global search methods also such as simulated annealing etc.

GA based learning provides an alternative way to learn for the ANN. The task involves controlling the learning complexity by adjusting the number of weights of the ANN. The use of GA for ANN learning can be viewed as follows

- Search for the optimal set of weights
- Search over topology space
- Search for optimal learning parameters
- A combination to search for all the above [6]

The primary feature of the genetic algorithm, which distinguishes it from other evolutionary algorithms, is that it represents the specimen in the population as bit strings. Wright introduced the concept of an adaptive "landscape", which describes the fitness of organisms. Individual genotypes are mapped to respective phenotypes, which are in turn mapped onto the adaptive topography [7]. This is an analogue to the DNA strands used in nature to encode the traits of real organisms. The encoding allows the genetic algorithm to use a set of genetic operators to manipulate the bit strings when creating new specimen. These operators are similar to the types of operations that are naturally encountered by the DNA strands in real organisms during reproduction. The advantages of this approach lie in its generality. The ability of the genetic algorithm to produce progressively better specimen lies in the selective pressure it applies to the population. The selective pressure can be applied in two ways. One way is to create more child specimen then is maintained in the population and selects only the best ones for the next generation. Even if the parents were picked randomly this method would still continue to produce progressively better specimen due to the selective pressure being applied on the child specimen. The other way to apply selective pressure is to choose better parents when creating the offspring. With this method only as many child specimen as maintained in the population need to be created for the next generation. Because of its nature to find a global solution in the search space, normally GA based learning takes a very long time to train the ANN. So to have a better time complexity, it is a good idea to combine the global search GA with the standard neural learning methods, which perform some local search. Also, many proposals had been given by the researchers to apply such a hybrid algorithm, where the Evolutionary Algorithm

(EA) can find a good solution in terms of its weight and architecture of the ANN and then a normal local search procedure can be applied to find the final solution [8] [9] [10] [11].

Another topic of interests for a learning process is the matrix solution based techniques. There are many different matrix solution methods available to solve overly determined equation, based on the iterative nature of the method. One of the major problems in finding the solution is that finding the optimal values for the weights depends on the curve fitting technique. These are also known as least squares methods, because the goal is to reach the minimum error between the fitted curve and the actual target values. Most of the matrix-based solution depends on the inverse property of the matrix. Hence, there is a possibility to find an ill-conditioned matrix. The other challenging point is that the coefficient matrix of the resulting linear system may be full. The major advantage of these methods is the less time complex nature of these methods. Also, these methods can be considered as a local search method, to find optimal values for the weights. Many large-scale optimization problems display sparsity in both second derivative (Hessian) matrices and constraint matrices. Efficient optimization methods must exploit this fact. In recent years researchers have worked on this issue from a variety of directions resulting in several successful and original contributions. These include the development of techniques for the efficient estimation of sparse Jacobian and Hessian matrices (including complete theoretical analysis and high-quality software); an original analysis of the sparse null basis problem (given a sparse rectangular matrix with more columns than rows, determine a sparse (or compact) representation for the null space) and the development of algorithms for finding a sparse null basis, and new direct-factorization methods for solving large and sparse bound-constrained quadratic programming problems.

The computational demands of many large-scale optimization problems can be extremely high; therefore, it is important to explore the practical potential of parallelism in the solution of optimization problems. Researchers have been active in this arena in recent years, having considered several important computational problems related to optimization, including the fast parallel solution of triangular systems of equations; the parallel solution of general

systems of non-linear equations; and the parallel solution of non-linear least-squares problems.

In this research, the ideas of exploring these GA based method and the matrix solution methods are considered because of the respective advantages of the two techniques.

1.1 BACKGROUND HISTORY

The psychological background for ANNs came from William James. McCulloch and Pitts also studied formally simple networks. This happened at the beginning of the twentieth century. Ideas about elementary associations, synaptic changes and holistic learning were proposed. But because there was no computer available, these ideas weren't implemented. And the actual neuroscience was still at its very beginning. The field of Artificial Neural Networks is one aspect of research usually referred to under the broad heading of Connectionism [12]. Connectionism has its origins in the 1940's, and has undergone a colorful history of development [13]. The roots of the ANN research can be found in the work of McCulloch and Pitts. They had modeled the brain as a collection of nodes, which can either, be in “ON” or “OFF” state. The strength of the connection is known as weights. Asynchronously, each neuron readjusts its state according to the threshold rule. This postulate has become the basis for many multi layered perceptron (MLP) models today. Another fundamental insight was Hebb’s proposal. He mentioned that if one neuron repeatedly fires another neuron then the strength of the synaptic connection need to be increased the efficiency of such firing. This co relational synapse postulate has become the basis of many distributed associative model.

In computers of early 1950's, psychology and philosophy join together to form two distinct ideas of how to model human thought: One should either model human mind (cognitive) or human brain (connectionism). Computers offered a great quasi-empirical tool for both inquiries. It is at this point when connectionism and cognitivism were born and separated. Lashely had already presented biological evidence for distributed representations. Yet, throughout this

time, advocates of thinking machines continued to argue their case. In 1956 the Darmouth summer research project on AI provided a boost to both AI and neural network. In the years following the Dartmouth project, John von Neumann suggested imitating simple neuron functions by using telegraph relays or vacuum tubes. Also, Frank Rosenblatt, a neuro-biologist of Cornell, began work on perceptron. He was intrigued with the operation of the eye of a fly. Much of the processing which tells a fly to flee is done in its eye. The perceptron, which resulted from this research, was built in hardware and is the oldest neural network still in use today. Newell and Simon (1955) formulated their physical symbol system hypothesis while Hebb (1949) [14] suggested that a mass of neurons could learn if their strengths of connections could change according to what we today call the Hebbian rule.

However, it was not Hebb but Rosenblatt who made first artificial neural network, and this device was called the perceptron. It was based on Hebb's ideas of neural computing. This happened in 1956. The essential feature of the perceptron, and all other ANNs, is their ability to learn while the emphasis in Newell and Simon's work was on problem solving. Other significant models at the earlier time of ANNs were Selfridge's pandemonium and Widrow and Hoff's Adaline architecture.

The history of the ANN remained silent for almost two decades excepted at the back of mind of few dedicated researchers. The physical symbol system won the first race and became the main paradigm. Partly because of Minsky's and Papert's classical criticism on Rosenblatt's perceptrons: they showed that there are certain serious limits in Rosenblatt's neural network. More exactly, there are - in principle - some simple functions which Rosenblatt's perceptron is not able to learn. Nobody showed that the physical symbol system could have any such limits, so it seemed more promising.

However, advocates of neural networks continued their silent research while symbol manipulators run their own program elsewhere. Neural networks were to come again. To mention few ANN advocates who continued the research: Grossberg [15] [16], Anderson [17], Willshaw [18], Klimasauskas [19] and Kohonen [20]. There are also many interesting links to philosophy. While the western rationalist tradition was a certain background for physical symbol system

hypothesis, many disturbing voices were heard from Wittgenstein, Heidegger, Husser and Dreyfus. They all tribute for more holistic approach. And the same holism can perhaps be seen in neural networks.

Their criticism was right. Symbol systems didn't succeed as was predicted, and the problems were just those that 'continental' philosophers have proposed. For example, the task of writing out a complete theoretical account of everyday life turned to much harder than initially expected; something Heidegger and Wittgenstein have predicted. Minsky run into these problems and formulated the notorious frame problem and common sense problem. It was impossible to formulate our common sense knowledge. This happened around 1975 (Harnad 1993 for an wider introduction to the frame problem). The rationalist tradition had finally been but to an empirical test, and it had failed.

It was in the early 1980's when this disappointment gave space for neural networks. Rumelhart and McClelland published their classical PDP book and the new paradigm settled as a new basis for cognitive science. Many cognitive oriented researched turned to this new paradigm. At this point the explosion of connectionist research was so enormous, that it is pointless to view it historically. One of the main achievements was the analysis of multi-layer networks and their learning capabilities.

Extension of the MLP has made it possible for this type of network to solve more complex problems. Two different kinds of basic models can be found – static and dynamic network. Static networks can be considered as characterized by node equations without any memory. The dynamic network model output depends on the past/present value of the input/output. Perceptron is one of the most accepted static ANN model. It was first conceived by Rosenblatt in 1958. Given an n dimensional vector as input, perceptron performs a weighted sum of the n components of the input vector and add a bias value. Rosenblatt’s original model used a hard-limiting non-linearity. There are two ways to view the operation of a perceptron model. One is to view it as a discriminant function for two-class pattern recognition model. In that case perceptron can effectively partition the input space in two regions with a linear decision boundary. There are many problems that often require a non-linear partitioning. These problems were described by Minsky and Papert [21]. But, these problems could be solved with

the MLP, which cascades two or more layers of perceptron together, thus making it possible to form a hyper plane partitioning. One of the limitations of the Rosenblatt's original formulation of the MLP was the lack of adequate learning algorithm. Algorithms were eventually developed to overcome this limitation. The Backpropagation algorithm can be used in conjunction with Artificial Neural Networks. The invention of the Backpropagation algorithm has played a large part in the resurgence of the interest in artificial neural networks. Backpropagation is the systematic method for training multilayer artificial neural networks. It has dramatically expanded the range of problems to which artificial neural networks can be applied, and it has generated many successful demonstrations of its power.

1.2 BIOLOGICAL NEURON

The exact working of human brain is still a mystery. In particular the most basic element of human brain is a particular cell known as neuron. These cells cannot be regenerated unlike other body cells. Approximately there are 100 billions of neurons in human brain. These neurons can connect up to 200,000 other neurons, although 1,000 to 10,000 are typical. The power of human brain comes from these sheer numbers of such basic components and number of such a huge connectivity's among them. Signals are transmitted between neurons by electrical pulses (action-potentials or 'spike' trains) traveling along the axon. These pulses impinge on the afferent neuron at terminals called synapses. These are found principally on a set of branching processes emerging from the cell body (soma) known as dendrites. Each pulse occurring at a synapse initiates the release of a small amount of chemical substance or neurotransmitter, which travels across the synaptic cleft, and which is then received at post-synaptic receptor sites on the dendrite side of the synapse. The neurotransmitter becomes bound to molecular sites here, which, in turn, initiates a change in the dendrite membrane potential. This post-synaptic-potential (PSP) change may serve to increase (hyperpolarize) or decrease (depolarize) the polarization of the post-synaptic membrane. In the former case, the PSP tends to inhibit generation of pulses in the afferent neuron, while in the latter; it tends to excite the generation of pulses. The afferent neuron sums or integrates the effects of thousands of such

PSPs over its dendrite tree and over time. If the integrated potential at the axon-hillock exceeds a threshold, the cell `fires' and generates an action potential or spike which starts to travel along its axon. This then initiates the whole sequence of events again in neurons contained in the efferent pathway. The following figure (Figure 1-1) describes the structure and functionality of a biological neuron.

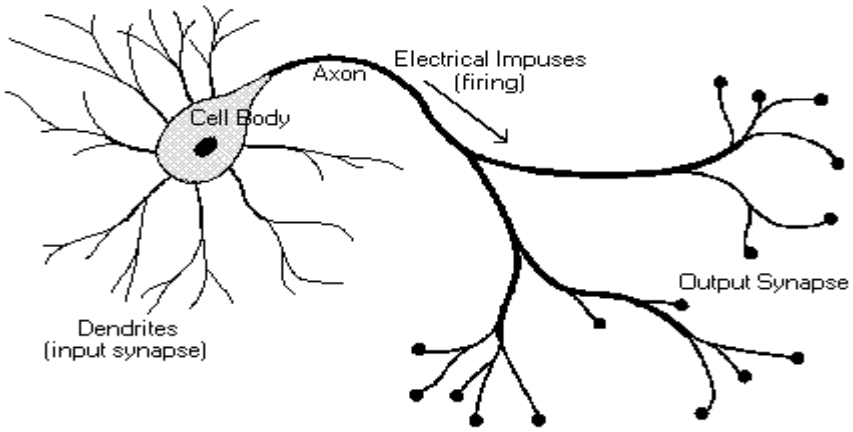

Figure 1-1 Biological neuron

1.3 ANN TAXONOMY

Neural networks, sometimes referred to as connectionist models, are parallel-distributed models that have several distinguishing features [3]:

- A set of processing units
- An activation state for each unit, which is equivalent to the output of the unit
- Connections between the units. Generally each connection is defined by a weight w_{jk} *that* determines the effect that the signal of unit j has on unit k
- A propagation rule, which determines the effective input of the unit from its external inputs

- An activation function, which determines the new level of activation based on the effective input and the current activation
- An external input (bias, offset) for each unit
- A method for information gathering (learning rule)
- An environment within which the system can operate, provide input signals and, if necessary, error signals.

1.3.1 Processing unit

A processing unit, also called a neuron or node, performs a relatively simple job; it receives inputs from neighbors or external sources and uses them to compute an output signal that is propagated to other units.

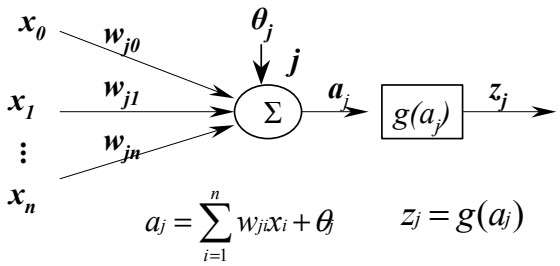

Figure 1-2 Artificial neuron

Within the neural systems there are three types of units:

- Input units, which receive data from outside of the network
- Output units, which send data out of the network
- Hidden units, whose input and output signals remain within the network.

Figure 1-2 describes an artificial neuron. Each unit *j* can have one or more inputs $x_0, x_1, x_2, \ldots x_n$, but only one output z_j. An input to a unit is either the data from outside of the network, or the output of another unit, or its own output.

1.3.2 Combination function

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 θ_j:

$$a_j = \sum_{i=1}^{n} w_{ji} x_i + \theta_j \tag{1-1}$$

The contribution for positive w_{ji} is considered as an excitation and an inhibition for negative w_{ji}. We call units with the above propagation rule *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 [3]

$$a_j = \sum_{i=1}^{n} w_{ji} \prod_{k=1}^{m} x_{ik} + \theta_j \tag{1-2}$$

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.

1.3.3 Activation function

Most units in neural network transform their net inputs by using a scalar-to-scalar function called an *activation function*, yielding a value called the unit's activation. Except possibly for output units, the activation value is fed to one or more other units. Activation functions with a bounded range are often called squashing functions. Some of the most commonly used activation functions are described in the following sub sections.

1.3.3.1 Identity function $g(x) = x$

It is obvious that the input units use the identity function (Figure 1-3). Sometimes a constant is multiplied by the net input to form a linear function.

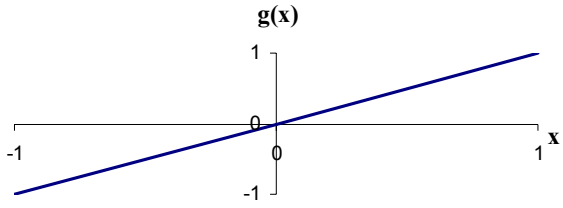

Figure 1-3 Identity function

1.3.3.2 Binary step function

Also known as *threshold function* or *Heaviside function*. The output of this function is limited to one of the two values:

$$g(x) = \begin{cases} 1 & \text{if}(x \geq \theta\ ) \\ 0 & \text{if}(x < \theta\ ) \end{cases} \tag{1-3}$$

This kind of function is often used in single layer networks. Figure 1-4 shows the plot of a binary step function.

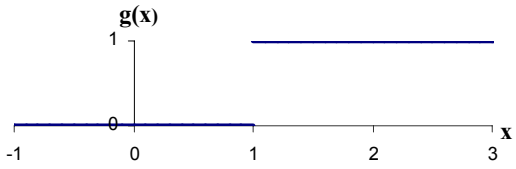

Figure 1-4 Binary step function

1.3.3.3 Sigmoid function

$$g(x) = \frac{1}{1+e^{-x}} \tag{1-4}$$

This function is especially advantageous for use in neural networks trained by back-propagation; because it is easy to differentiate, and thus can dramatically reduce the computation burden for training. It applies to applications whose desired output values are between 0 and 1. Figure 1-5 shows the plot of a sigmoid function.

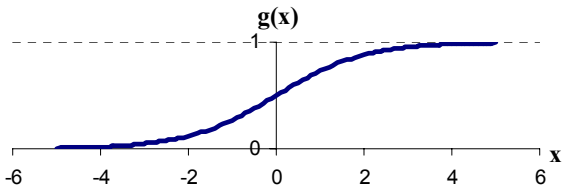

Figure 1-5 Sigmoid function

1.3.3.4 Bipolar sigmoid function

$$g(x) = \frac{1-e^{-x}}{1+e^{-x}} \tag{1-5}$$

This function has similar properties with the *sigmoid function.* It works well for applications that yield output values in the range of [-1, 1]. Figure 1-6 shows the plot of a bipolar sigmoid function.

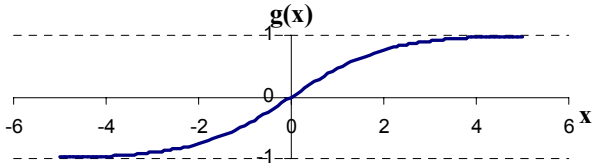

Figure 1-6 Bipolar sigmoid function

Activation functions for the hidden units are needed to introduce non-linearity into the networks. The reason is that a composition of linear functions is again a linear function. However, it is the non-linearity (i.e., the capability to represent nonlinear functions) that makes multi-layer networks so powerful. Almost any nonlinear function does the job, although for back-propagation learning it must be differentiable and it helps if the function is bounded. The sigmoid functions are the most common choices [5].

For the output units, activation functions should be chosen to be suited to the distribution of the target values. For binary [0, 1] outputs, the sigmoid function is an excellent choice. For continuous-valued targets with a bounded range, the sigmoid functions are again useful, provided that either the outputs or the targets to be scaled to the range of the output activation function. But if the target values have no known bounded range, it is better to use an unbounded activation function, most often the identity function (which amounts to no activation function). If the target values are positive but have no known upper bound, an exponential output activation function can be used [5].

Neural networks (NN) can be used for feature extraction, association, optimization, function fitting and modeling. It can be divided into two major categories – supervised net, where input patterns are associated with known output patterns and unsupervised where structures are found in the input patterns. In terms of its properties NN can be also divided according to its

- *Network properties* – Network topology, types of connections, the order of connection, weight range
- *Node properties* – Activation range, transfer function
- *System dynamics* – The weight initialization scheme, activation calculating formula, learning rule.

Neural networks can be explicitly programmed to perform a task by manually creating the topology and then setting the weights of each link and threshold. However, this bypasses one of the unique strengths of neural nets the ability to program themselves.

The most basic method of training a neural network is trial and error. If the accuracy of the network declines, undo the change and make a different one. It takes time, but the trial and error method does produce results. Unfortunately, the number of possible weightings rises exponentially as one adds new neurons, making large general-purpose neural nets impossible to construct using trial and error methods. In the early 1980s two researchers, David Rumelhart and David Parker independently rediscovered an old calculus-based learning algorithm. The back-propagation algorithm compares the result that was obtained with the result that was expected. It then uses this information to systematically modify the weights throughout the neural network. This training takes only a fraction of the time that trial and error method takes. It can also be reliably used to train networks on only a portion of the data, since it makes inferences. The resulting networks are often correctly configured to answer problems that they have never been specifically trained on.

As useful as back-propagation is, there are often easier ways to train a network. In the literature review section, some of the traditional ANN based learning and their hybrid counterpart such as GA based learning, least square based learning etc are discussed.

1.3.4 Genetic algorithm

Genetic algorithms are a stochastic search method introduced in the 1970s in the United States by John Holland (1976) and in Germany by Ingo Rechenberg (1973). Although genetic algorithms have been studied for over 25 years, implementing them is often as much as art as designing heuristics. Much of the genetic algorithm literature is devoted to relatively simple problems. Simplistic application of a genetic algorithm to a small problem often produces reasonable results, but naïve application of genetic algorithms to larger problems often results in poor performances. This is due to the nature of the genetic search and the relationship between the genetic representation and genetic operators. Direct representation of the problem, i.e. use of data types rather than bit string, promises further improvements in genetic algorithm's applicability, robustness

and performance. Continued reduction in computational cost along with increase in power and speed make genetic algorithms viable alternatives despite their significance computational overhead.

Most symbolic AI systems are very static. Most of them can usually only solve one given specific problem, since their architecture was designed for whatever that specific problem was in the first place. Thus, if the given problem were somehow to be changed, these systems could have a hard time adapting to them, since the algorithm that would originally arrive to the solution may be either incorrect or less efficient. Genetic algorithms (or GA) were created to combat these problems. They are basically algorithms based on natural biological evolution. The architecture of systems that implement genetic algorithms (or GA) is more able to adapt to a wide range of problems. A GA functions by generating a large set of possible solutions to a given problem. It then evaluates each of those solutions, and decides on a "fitness level" for each solution set. These solutions then breed new solutions. The parent solutions that were more "fit" are more likely to reproduce, while those that were less "fit" are more unlikely to do so. In essence, solutions are evolved over time. This way you evolve your search space scope to a point where you can find the solution. Genetic algorithms can be incredibly efficient if programmed correctly.

1.3.4.1 Outline of the basic genetic algorithm

Step 1: [Start] Generate random population of *n* chromosomes (suitable solutions for the problem)

Step 2: [Fitness] Evaluate the fitness *f(x)* of each chromosome *x* in the population

Step 3: [New population] Create a new population by repeating following steps until the new population is complete

Step 4: [Selection] Select two parent chromosomes from a population according to their fitness (the better fitness, the bigger chance to be selected)

Step 5: [Crossover] With a crossover probability cross over the parents to form a new offspring (children). If no crossover was performed, offspring is an exact copy of parents.

Step 6: [Mutation] With a mutation probability mutate new offspring at each locus (position in chromosome).

Step 7: [Accepting] Place new offspring in a new population.

Step 8: [Replace] Use new generated population for a further run of algorithm.

Step 10: [Test] If the end condition is satisfied, **stop**, and return the best solution in current population.

Step 11: [Loop] Go to step **2**

In Figure 1-7 some genetic operations on a sample example chromosome such as recombination and mutation are explained.

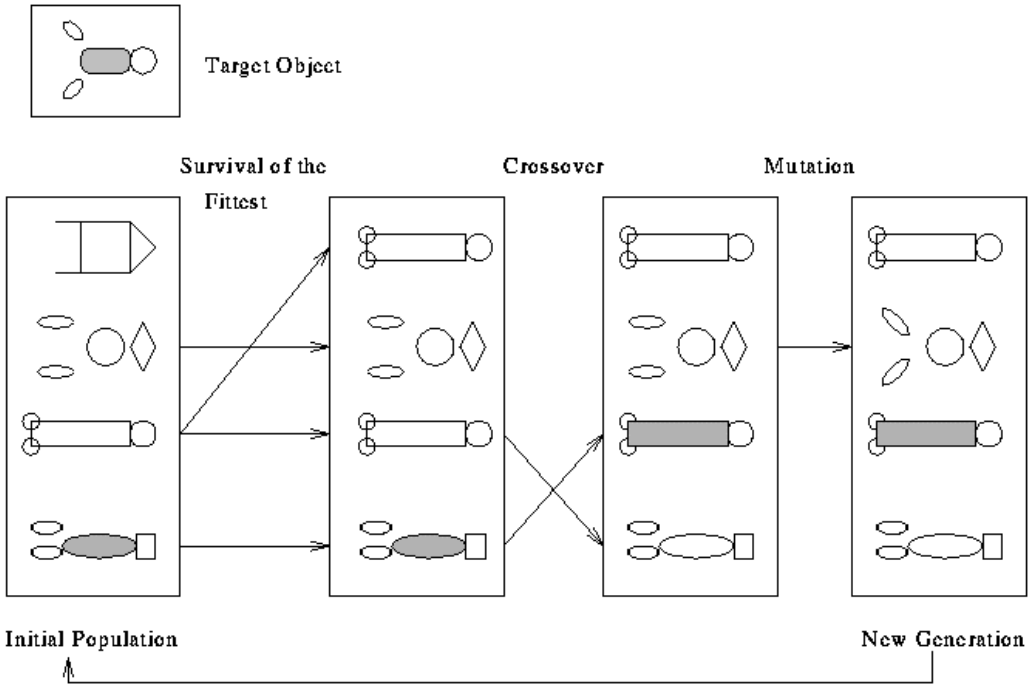

Figure 1-7 Genetic operations on genotypes

1.3.4.2 Application of genetic algorithm

The possible applications of genetic algorithms are immense. Any problem that has a large search domain could be suitable tackled by GAs. A popular growing field is *genetic programming* (GP).

1.3.4.2.1 Genetic programming

In programming languages such as LISP and Scheme, the mathematical notation is not written in standard notation, but in prefix notation. Some examples of this:

```
+ 2 1          :       2 + 1
* + 2 1 2      :       2 * (2+1)
* + - 2 1 4 9  :       9 * ((2 - 1) + 4)
```

Apart from the order being different between the left hand side to the right, there is no difference. The prefix method makes life a lot easier for programmers and compilers alike, because order precedence is not an issue. One can build expression trees out of these strings that then can be easily evaluated, for example, the following figure (Figure 1-8) shows the trees for the above three expressions.

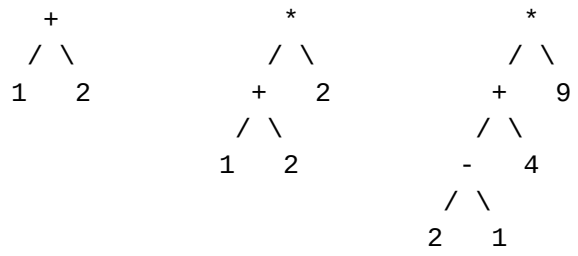

Figure 1-8 Genetic programming as a solver

Hence, expression evaluation becomes a lot easier. If for example we have numerical data and 'answers', but no expression to conjoin the data with the answers, a genetic algorithm can be used to 'evolve' an expression tree to create a very close fit to the data. By 'splicing' and 'grafting' the trees and evaluating the resulting expression with the data and testing it to the answers, the fitness function can return how close the expression is. The limitations of genetic programming lie in the huge search space the GAs have to search for - an infinite number of equations. Therefore, normally before running a GA to search for an equation, the user tells the program which operators and numerical ranges to search under. Uses of genetic programming can lie in stock market prediction, advanced mathematics and military applications.

1.3.4.2.2 Evolving neural networks

GAs have successfully been used to evolve various aspects of GAs - the connection weights, the architecture, or the learning function. GAs are perfect for evolving the weights of a neural network - there are immense number of possibilities that standard learning techniques such as back-propagation would take thousands upon thousands of iterations to converge to. GAs could (given the appropriate direction) evolve working weights within a hundred or so iterations.

Evolving the architecture of neural network is slightly more complicated, and there have been several ways of doing it. For small nets, a simple matrix represents which neuron connection which, and then this matrix is, in turn, converted into the necessary 'genes', and various combinations of these are evolved.

1.3.4.2.3 Other areas

Genetic Algorithms can be applied to virtually any problem that has a large search space. Al Biles uses genetic algorithms to filter out 'good' and 'bad' riffs for jazz improvisation, the military uses GAs to evolve equations to differentiate between different radar returns, stock companies use GA-powered programs to predict the stock market.

1.4 MOTIVATION AND AIM OF THE RESEARCH

The problems with the exiting neural learning methods can be summarized as follows

- Most of the calculus based algorithms depend on the gradient information of the error surface, which may not always be available or computationally expensive to find
- The algorithm may be trapped to a local minimum very easily
- The time complexity for algorithms with a global search capability is very high. Even algorithm that works similar as a local search has a very high time complexity for large training data set

- For iterative algorithms, there exists the potential problem of paralysis. In the paralysis problem, the weights become large and updating the weights becomes very difficult
- Many a times the given solution is not optimum, and the generalization ability of the algorithm is very poor
- Many a times the algorithms are very much sensitive to the initial condition, hence the results obtained are not very consistent. Many runs using the same architecture and same parameters are necessary to comment on the results
- For non-iterative based algorithms or direct solution based methods the memory complexity is very high

These are the problems that are the areas of concerns for this research. This research will address all the above-mentioned problems in a learning process for the ANN. The main areas of application for the learning strategy that was kept in mind were classification and function approximation problems.

The aim of this research is to develop a learning strategy for a fully connected feed forward MLP. This corresponds to finding a strategy of convergence for a feed-forward MLP considering its parameters such as weights and the architecture itself as variables. The main objectives of the research are as follows

- To propose a novel hybrid learning algorithm that combines evolutionary based search and matrix based method together for finding the weights of a feed forward ANN
- To propose a hierarchical combination dynamics for weights and architecture of a feed forward ANN, that combines the hybrid evolutionary search and the matrix based method with a naïve binary type search for the architecture search space
- To investigate the feasibility of using GA based learning algorithms and its variations for the ANN, and to conduct a comparative study of the results with the proposed hybrid learning algorithm that uses genetic search and least square method. The comparison is based on the correct classification rate, generalization ability, probability of

convergence and also the memory and time complexities of all the algorithms

- To investigate various variable parameters for GA based learning algorithms and to study their suitability for GA based learning algorithms for finding proper selection operator, genetic operators, the method of creating the genotypes, the method for generating the intermediate population, and to address issues such as elitism, competing convention problems, etc for such algorithms
- To investigate some of the popular least square based methods such as modified Gram-Schmidt, Singular value decomposition (SVD), etc for ANN learning
- To investigate the feasibility of finding a proper ANN architecture using the proposed hybrid learning algorithm as a base using a suitable stopping criteria.

1.5 ORGANIZATION OF THE THESIS

Chapter 1 contains the introduction of the GA based technique for ANN learning. In the beginning a background history of the ANN is given followed by the ideas of GA that can be applied for ANN learning. The introduction for the ANN includes the basic architecture and its working principle, different usage of activation function etc. Also, few different methodologies based on the basic GA are discussed such as genetic programming etc.

Chapter 2 contains the literature review section. Keeping in mind that mentioning all the work covering ANN and GA may not be possible for one volume of a thesis, only those learning algorithm are discussed which are based on either the basic or evolutionary based learning or matrix based methods. The chapter is mainly divided into two sections. In the first section, the different gradient based learning are discussed followed by the matrix based solution method. In the second section GA based learning for ANN are discussed. It also describes some of the popular evolutionary based learning models that have been implemented in

real application areas. At the end of this chapter some of the problems that are faced by some of the existing algorithms are discussed.

Chapter 3 contains the research methodology. The chapter is divided in such a way that a stepwise development of the final algorithm could be discussed. In the beginning of the chapter, the various GA / EA based module with variations are discussed. In the following section the least square method used in this thesis is discussed along with its mathematical basics. Then the various strategies for combining these two modules are discussed. The three possible combination strategies mainly T1, T2 and T3 are mentioned, along with their details flowchart. Finally in this chapter the hierarchical combination of the weight and architecture modules are discussed. A simple rule based system is given for various stopping criteria to show how the architecture module can be combined with the new algorithm to find optimum / near optimum architecture for the given ANN.

Chapter 4 presents all the experimental results. The algorithms that were implemented and tested for comparison purposes are – EBP, GAWLS (Genetic algorithm without least square), EAWLS (Evolutionary algorithm without least square), GALS/T (GALS-T1, GALS-T2, GALS-T3) Genetic algorithm with least square with different connection strategies, EALS/T (EALS-T1, EALS-T2, EALS-T3) Evolutionary algorithm with least square with different connection strategies, and finally the combination of weight and architecture modules GV1 (LI-EALS-T3) (Where the optimum number of hidden neurons is found using a simple linear search), GV2 (BT-EALS-T3) (where the optimum number of hidden neurons is found using a binary search type method). Before the experimental results are given a short description for all the algorithms along with various parameters are given. The experimental results are mainly divided into two sections. In the first section the results are given mainly for the comparison purposes with other algorithms, and in the second section experimental results are given to comment on specific properties for the new algorithm. The first section is further divided into two more sections. The first section, the results are given for changing various parameters such as number of populations, number of hidden neurons etc. Also for simplicity the results are given in different tables for different algorithm and data set. In the second

subsection the results are given after combining with the architecture modules. Hence only the best values from the combining modules are given.

Chapter 5 contains the analysis and discussion. It is broadly divided into two sections. In the first section the comparative study of the new proposed algorithm with the standard EBP and the Evolutionary method without least square based techniques are given. The comparison is based on test classification accuracy, RMS error, time and memory complexity, number of required number of hidden neurons and required number of generation / iteration. In the second section of the chapter some specific properties of the proposed algorithm are given. It includes the convergence property, sensitivity of the result to initial condition, fitness breaking effect etc. For analysis graphs are given followed by a short description for the discussion of the obtained result.

Chapter 6 contains the conclusion for the thesis and the possibility of extending the work for future research has been addressed.

2 LITERATURE REVIEW

Learning algorithm can be considered as a subset of machine learning algorithm in a broader sense. Machine learning terminology is often coined by the data mining community in general or more precisely by the predictive data mining community. Ever since the growing interest of AI community to build logical reasoning capability of human being in a machine [22], machine learning became an important aspect of the modeling. With the passage of time, the learning spectrum was clearly divided into the inductive (with no a priori knowledge) and deductive (with data driven), thus forming a spectrum with two extreme type of learning [23]. Not surprisingly, there were research interests also in hybrid kind of learning, which tried to merge the two extreme sides of data and knowledge driven learning. With the growth in the field of ANN for its universal approximation ability, a specific kind of learning became an area with special interest within AI research community. Thus, neural learning has been an area of much wider machine learning algorithm, which is mainly inductive type of learning, where the machine learns from the data without any a priori knowledge. One of the main factors that lead to the research interest in machine learning was probably to find a model that usually involve the fitting of very complex 'generic' models, that are not related to any reasoning or theoretical understanding of underlying causal processes; instead, the model can be shown to generate accurate predictions or classification in cross validation samples. Nevertheless there was research interest also to extract certain logical rules from the system after the learning process is over. To a larger extent the machine learning algorithms have different characteristic than their statistical data analysis counterpart [24] [25]. Unlike traditional statistical data analysis, which is usually concerned with the estimation of population parameters by statistical inference, the emphasis in data mining (and machine learning) is usually on the accuracy of prediction (predicted classification), regardless of whether or not the 'models' or techniques that are used to generate the prediction is interpretable or open to simple explanation.

Since the beginning, the learning algorithm for ANN has been a challenging task for the researchers. Till date there has been few hundreds of publication in this area, proposing various learning algorithms and architectures [2-25] [39-68]. It is not any more a possibility to report all the learning algorithms that have been published in this area in one thesis. In this thesis, a few popular learning algorithms have been considered, which is related to the new learning algorithm that has been proposed in this thesis. The criteria of selecting few among them in the literature review has been mainly those learning algorithms, which has been developed for fully connected feed forward network architecture, and also well cited in the research publications in ANN area. Also, mainly supervised learning algorithms have been considered [26] [27].

In summary, the aspects of learning algorithm for ANN can be viewed as follows

1. A training algorithm that can search for optimal parameters (for e.g. weights & biases)
2. A rule or algorithm that can determine the network complexity and suffice the condition of solving the given problem
3. A measurement metric to evaluate performance of reliability, generalization, time / memory complexity, etc.

All neural learning algorithms address the above issues – mostly point 1 and point 3 only and few addressing all the above. In mathematical terms, the goal of the ANN is to minimize the cost function as follows

$$g_T(f_{NS}(X,w),Y) \tag{2-1}$$

using the equation (2-2)

$$E\left[g_T\left(f_{NS}\left(X,w\right),Y\right)\right]=\int\int g_T\left(f_{NS}\left(x,w\right),y\right)f_{x,y} \tag{2-2}$$

Where the input output relationship can be realized as (2-3)

$$y = f_{NS}(x,w) \tag{2-3}$$

$F_{x,y}(x,y)$ denotes the joint probability distribution function (pdf) that depends on x and y. Given a network structure NS, a family of input output relationship (2-4)

parameterized by w, consisting of all network functions that may be formed with different choices of the weights can be assigned [28].

$$F_{NS} = \{ f_{NS}(x, w) \} \tag{2-4}$$

Most applications of neural network fall into one of the following categories:

1. Prediction [29] [30] [31] [32] [33] [34] [35]
2. Classification
3. Data association
4. Data conceptualization
5. Data filtering

A learning algorithm can have many characteristics, which may allow them to be fallen into different categories. In section 2.1, iterative learning algorithms are considered. In section 2.2, non iterative/non gradient types of learning algorithms are considered. Finally, in section 2.3, hybrid types of learning are considered.

2.1 ALGORITHM NOTATIONS

Below a listing of the commonly used notations used for all the algorithms is given. If some notation is only specific to a particular algorithm, it is described in the description of the algorithm.

x -	Input vector
t -	Target output vector.
$netOutputB_j$ -	Output of the jth neuron before activation.
Y_j -	Output of the jth neuron after activation.
W -	Weight matrix
w_{ij} -	Weight between input and hidden layer.
w_{jk} -	Weight between hidden and output layer.
N -	Number of inputs.
M -	Number of outputs.
hid -	Number of hidden units
L -	Number of layers.
Δ_{ij} -	Individual update in weight w_{ij}
E -	Error function
δ_j -	Local gradient of node j.
i -	Input layer index
j -	Hidden layer index
k -	Output layer index.

J -	$\left[\frac{\partial e_j}{w_{ij}}\right]_{M\times N}$
$\Delta\tilde{w}$ -	$P\Delta w$, where P is a pivoting matrix.
f -	Activation function.
f' -	First derivative of f.

2.2 ALGORITHM DESCRIPTION

In the following sections description of most popular algorithms are given. If the architecture for the algorithm requires any variation from the standard multi layered feed forward perceptron, then the figure for the architecture is also given.

2.2.1 Hebb rule

Hebb rule is the earliest and simplest among the ANN learning algorithms. The basic architecture is shown in Figure 2-1. Its postulate was based on the idea that if two interconnected neurons are on, then the synaptic strengths between them is increased. However a slight improvement of the original learning algorithm was done by McClelland and Rumelhart in1988. In this modified learning if the connected neurons are off then also their synaptic weights are increased.

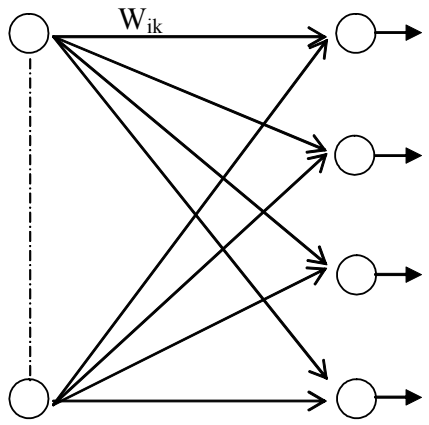

Figure 2-1 Hebb rule architecture

The Hebb rule algorithm can be described as follows:

Step 1:	Initialize all weights $w_{ik} = 0; i = 1:n; k = 1:m$	
Step 2:	For each training pair, **s** : t, do steps 3-5.	
Step 3:	Set activations for input units $x_i = s_i; i = 1:n$	
Step 4:	Calculate the net output $netOutB_k = b_k + \sum_{i=1}^{n} x_i w_{ik}$	(2-5)
Step 5:	Calculate the actual output $y_k = f(netOutB_k)$	(2-6)
Step 6:	Adjust the weights and bias for output layer $w_{ik}(t+1) = w_{ik}(t) + x_i y_k; i = 1:n; k = i:m$	(2-7)
	$b_k(t+1) = b_k(t) + y_k$	(2-8)

2.2.2 Hamming net algorithm

Hamming net is described by Lippmann [36]. It is a two layer feed forward network with the ability to classify noise corrupted patterns, which has the property of always converging toward one of the patterns that was stored during a training phase. The first layer (Figure 2-2) is known as the quantifier subnet. It computes the Hamming distance between the input pattern to be recognized and the patterns stored in the network as neuron's weight values. The second layer is the discriminator subnet that selects the first layer neuron with the highest output as the winner neuron, the one with smallest Hamming distance to the current input pattern

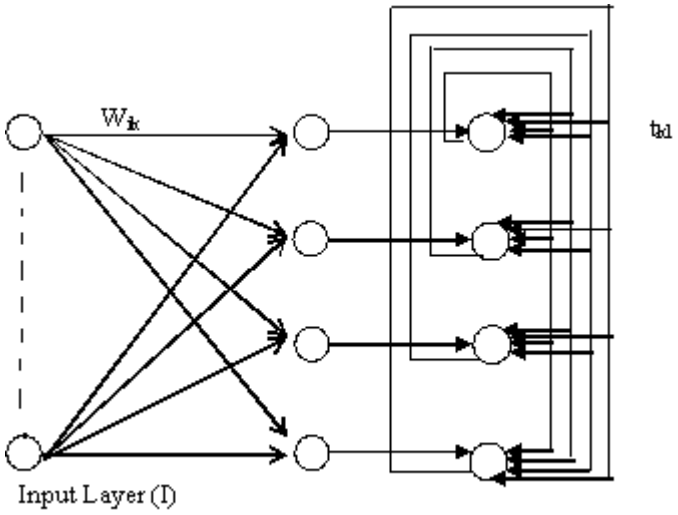

Figure 2-2 Hamming network

This algorithm can be described in the following steps:

Step1: Initialize the weights and offsets in the lower subnet

$$w_{ij} = \frac{x_i^f}{2}$$

$$\theta = \frac{N}{2} \tag{2-9}$$

$$0 < i < N-1, 0 < j < M-1$$

Step 2: Initialize the weights and offsets in the upper subnet

$$t_{kl} = \begin{cases} 1; k = l \\ -\varepsilon; k \neq l, \varepsilon < \dfrac{1}{M} \end{cases} \tag{2-10}$$

$$0 \leq k, l \leq M-1$$

Step 3: Initialize with unknown input pattern

$$\mu(0) = f\left(\sum_{i=0}^{N-1} w_{ij}(0) x_i(t) - \theta \right) \tag{2-11}$$

Step 4: Iterate until convergence as

$$\mu(t+1) = f\left(\mu(t) - \varepsilon \sum_{k \neq l} t_{kl} \mu(t) \right) \tag{2-12}$$

2.2.3 Kohonen self-organizing maps

Kohonen self-organizing network (Figure 2-3) was developed by Teuvo Kohno in 1982. It is an unsupervised learning clustering algorithm. It creates a map relationship among the pattern. During training, the output node with the minimum Euclidean distance is found. This will indicate the node (class) to which the input pattern belongs. Instead of determining the Euclidean distance, the "traditional" winner-take-all method may also be employed.

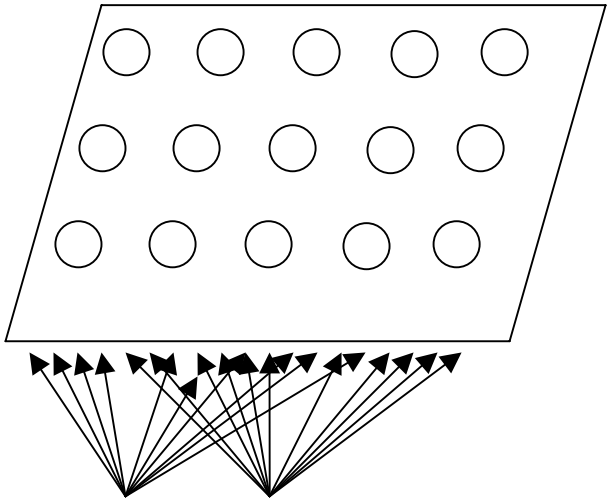

Figure 2-3 Kohonen self organizing network

This algorithm can be described in the following steps:

Step1:	Initialize the weights Set topological neighbourhood power Set learning rate $\eta(0 < \eta \leq 1)$
Step 2:	While stopping condition is false, do steps 2-9.
Step 3:	For each training pair, **s:t** , do steps 3-6.
Step 4:	Set activations of input units $x_i = s_i; i = 1:n$
Step 5:	For each j, compute the distance between input an each output

Contd..

$$D(j) = \sum_i \left(w_{ij} - x_i\right)^2 \quad (2\text{-}13)$$

Step 6: Find index J such that D(J) is minimum.

Step 7: For all units j within a specified neighbourhood of J, and all I, update the weight as:

$$w_{ij}(t+1) = w_{ij}(t) + \phi\eta\left[x_i - w_{ij}(t)\right] \quad (2\text{-}14)$$

where φ is the neighbourhood function.

Step 8: Update learning rate.

Step 9: Reduce radius of topological neighbourhood at specified times

Step 10: Test stopping condition:
If the number of iteration exceeds a specific value, stop; else continue.

.

2.3 GRADIENT BASED LEARNING

Gradient based or gradient descent learning is named because of the learning characteristic of the algorithm, which use gradient information of the error surface by the learning algorithm. These type of learning are mainly iterative in nature, where in each iteration, the current gradient information is used to update the weights of the ANN. Gradient based learning can further be divided into three types – first order, second order and reinforcement learning.

2.3.1 Frequency of updating weights in gradient descent algorithm

Depending on the frequency of updating the weights, gradient descent algorithm falls into two different types – data adaptive and block adaptive. The data-adaptive methods update the weights at every iteration, as opposed to the block methods, which execute the update upon the completion of each sweep over the whole training data set. The back propagation learning algorithm can be executed in either mode. The two approaches have exhibited significantly different numerical performances. Block adaptive methods are known to be more robust since the training step averaged over all the training patterns. In contrast, data

adaptive methods can be appealing for some on-line adaptation applications. They are more sensitive to the noise effect on individual patterns. Block methods are recommended for applications where real time learning is not necessary. The effectiveness of any learning algorithm lies in the selection of an optimal update direction. The direction of update depends on whether a first order or second order gradient method is adopted. In block adaptive methods, it could be useful to involve a second order momentum term in determining the direction of update. It often involves a direct or indirect computation of Hessian matrix and its subsequent inversion. Previous research suggests that the second order update direction seems to be more effective than its first order counterpart.

2.3.2 First Order gradient based learning

In first order gradient based method, the propagated error depends on the first order partial derivative information of the output function, with respect to the weights. Minimizing error with gradient descent is the least sophisticated but nevertheless in many cases a sufficient method. In the following some of the popular first order gradient based algorithms are discussed.

2.3.2.1 Perceptron rule

The perceptron learning rule is more powerful than Hebb rule. A number of different types of perceptron are described in Rosenblatt (1962) and in Minsky and Papert (1969, 1988). In general, the rule states that if weights exist to allow the net to respond correctly to all training patterns, then the algorithm will adjust the weights to find a value that the net does respond correctly to all training patterns in a finite number of steps.

This algorithm can be described in the following steps:

Step 1: Initialize weights and bias.
Set learning rate η ($0 < \eta \leq 1$)

Step 2: While stopping condition is false, do steps 3-8.

Step 3: For each training pair, **s:t** , do steps 4-6.

Step 4: Set activations of input units
$x_i = s_i; i = 1:n$

Step 5: Compute response of output unit

$$netOutB_k = b_k + \sum_{i=1}^{n} x_i w_{ik}$$

Step 6: Calculate the actual output
$y_k = f(netOutB_k)$

$$y_k = \begin{cases} 1 & if\ (netOutB_k > \theta) \\ 0 & if\ (-\theta \leq netOutB_k \leq \theta) \\ -1 & if\ (y\_in < -\theta) \end{cases} \quad (2\text{-}15)$$

Step 7: Update weights and bias if an error occurred for this pattern

If ($y_k \neq t_k$)

$$w_{ik}(t+1) = w_{ik}(t) + \eta x_i t_k; i = 1:n; k = i:m \quad (2\text{-}16)$$

$$b_k(t+1) = b_k(t) + \eta t_k \quad (2\text{-}17)$$

else

$$w_{ik}(t+1) = w_{ik}(t) \quad (2\text{-}18)$$

$$b_k(t+1) = b_k(t) \quad (2\text{-}19)$$

Step 8: Test stopping condition:
If no weights changed in Step 2, stop; else continue.

2.3.2.2 Pocket algorithm with ratchet

Perceptron learning is not well behaved for non-separable problems. While it will eventually visit an optimal set of weights, it will not converge at any set of weights. Even worse, the algorithm can go from an optimal set of weights to a

worst possible set in one iteration, regardless of how many iterations have been taken previously. The pocket algorithm [37] makes perceptron learning well behaved by adding positive feedback in order to stabilize the algorithm.

This algorithm can be described in the following steps:

Step 1: Set the vector of integral perceptron weights π as

$\pi =< 0,.....0 >$

Set

$$run_\pi = run_w = num\_ok_\pi = num\_ok_w = 0 \quad (2\text{-}20)$$

where

run_π = the number of consecutive correct classification using perceptron weight π

run_w = the number of consecutive correct classification using pocket weight W.

num_ok_π = total number of training examples that π correctly classifies.

num_ok_w = total number of training examples that W correctly classifies.

Step 2: Randomly pick a training example s_p with corresponding classification t_p

Step 3: If π correctly classifies s_p i.e.

$$\left.\begin{array}{l} \pi \bullet s_p > 0; t_p = +1 or \\ \pi \bullet s_p < 0; t_p = -1 \end{array}\right\} \quad (2\text{-}21)$$

then

$$run_\pi = run_\pi + 1 \quad (2\text{-}22)$$

Step 4: If $(run_\pi > run_w)$

Compute num_ok_π by checking every training pair.

If $(num\_ok_\pi > num\_ok_w)$

Set W=π

Set $run_w = run_\pi$ (2-23)

Set $num\_ok_w = num\_ok_\pi$ (2-24)

Contd..

If all training pairs are correctly classified then stop.
Else
From a new vector of perceptron weights

$$\pi = \pi + t_p s_p \quad (2\text{-}25)$$

Set $run_w = 0$ (2-26)

Step 5: If the specified number of iteration has not been taken then go to step 2.

2.3.2.3 Delta rule

This technique is usually ascribed to Widrow and Hoff, who trained Threshold Logic Units (TLUs) which had their outputs labeled -1, 1 instead of 0, 1. Details of this and related algorithms are provided in many texts (e.g., Widrow and Stearns, 1985; Bishop, 1995; Duda and Hart, 1973). These units they called Adaptive Linear Elements or ADALINEs [38]. The learning rule based on gradient descent with this type of node is, therefore, sometimes known as the Widrow Hoff rule, but more usually now, as the *delta rule* [39]. Widrow et al proposed two rules for ADALINE learning α-LMS and μ-LMS.

2.3.2.3.1 The μ-LMS algorithm

In μ-LMS algorithm [40], [41] can be described in the following steps:

Step 1: Initialize weights.
Set learning rate η ($0 < \eta \leq 1$)

Step 2: While stopping condition is false, do steps 3-7.

Step 3: For each training pair, **s:t** , do steps 4-6.

Step 4: Set activations of input units
$x_i = s_i; i = 1:n$

Step 5: Compute response of output unit

$$netOutB_k = b_k + \sum_{i=1}^{n} x_i w_{ik}$$

Contd..

Step 6:	Update weights and bias, for i = 1:n, $w_{ik}(t+1) = w_{ik}(t) + \eta(t_k - y_k)x_i$ $b_k(t+1) = b_k(t) + \eta(t_k - y_k)$ (2-27)
Step 7:	Test stopping condition: If the largest weight change that occurred in Step 2 is smaller than a specific tolerance, stop; else continue.

2.3.2.3.2 The α-LMS algorithm

In α-LMS algorithm [12] can be described in the following steps:

Step 1:	Initialize weights. Set learning rate η ($0 < \eta \le 1$)
Step 2:	While stopping condition is false, do steps 3-7.
Step 3:	For each training pair, **s:t** , do steps 4-6.
Step 4:	Set activations of input units $x_i = s_i; i = 1:n$
Step 5:	Compute response of output unit $netOutB_k = b_k + \sum_{i=1}^{n} x_i w_{ik}$
Step 6:	Update weights and bias, for i = 1:n, $w_{ik}(t+1) = w_{ik}(t) + \eta \frac{(t_k - y_k)x_i}{\|x\|^2}$ (2-28) $b_k(t+1) = b_k(t) + \eta(t_k - y_k)$
Step 7:	Test stopping condition: If the largest weight change that occurred in Step 2 is smaller than a specific tolerance, stop; else continue.

2.3.2.4 MADALINE rule I

Madaline Rule I (MRI) is used to train MADALINE (Many ADAptive LINEar Neuron), which was proposed by Widrow and Hoff [42]. In this algorithm only the weights between hidden ADALINES are adjusted, the weights for the output units are fixed.

MRI algorithm can be described in the following steps:

Step 1: Initialize weights.
Set learning rate η ($0 < \eta \leq 1$)

Step 2: While stopping condition is false, do steps 2-8.

Step 3: For each training pair, **s:t** , do steps 3-7.

Step 4: Set activations of input units
$x_i = s_i; i = 1 : n$

Step 5: Compute net input to each hidden ADALINE unit

$$netOutB_j = b_j + \sum_{i=0}^{N-1} w_{ij} x_i$$

Step 6: Determine output of each hidden ADALINE unit
$y_j = f(netOutB_j)$

Step 7: Determine output of the net

$$netOutB_k = b_k + \sum_{j=1}^{h} w_{jk} y_j$$

$$y_k = f(netOutB_k)$$

Step 8: Determine error and update weights
If (t = y)
No weight updates are performed.
Else
If (t = 1),
Update the weights and the bias on hidden unit Z_J, the units whose net input is closest to 0,

$$w_{iJ}(t+1) = w_{iJ}(t) + \eta(1 - netOutB_J)x_i \qquad (2\text{-}29)$$

Contd..

$$b_J(t+1) = b_J(t) + \eta(1 - netOutB_J) \quad (2\text{-}30)$$

If (t = -1)

Update the weights and the bias on all hidden units Z_J, that have positive net input

$$w_{iJ}(t+1) = w_{iJ}(t) + \eta(1 - netOutB_J)x_i \quad (2\text{-}31)$$

$$b_K(t+1) = b_K(t) + \eta(1 - netOutB_J) \quad (2\text{-}32)$$

Step 9: Test stopping condition:
If weight changes have stopped (or reached an acceptable level), or if a specific maximum number of weight update iterations (Step 2) have been performed, then stop; else continue.

2.3.2.5 MADALINE rule II

A more recent MADALINE training rule is MRII, proposed by Widrow, Winter, and Baxter, in 1988 [43]. It allows training for weights in all layers of the net.

This algorithm can be described in the following steps:

Step 1: Initialize weights.
Set learning rate η ($0 < \eta \leq 1$)

Step 2: While stopping condition is false, do steps 3-11.

Step 3: For each training pair, **s:t** , do steps 4-8.

Step 4: Set activations of input units
$x_i = s_i; i = 1:n$

Step 5: Compute net input to each hidden ADALINE unit

$$netOutB_j = b_j + \sum_{i=0}^{N-1} w_{ij}x_i$$

Step 6: Determine output of each hidden ADALINE unit

Contd...

$$y_j = f(netOutB_j)$$

Step 7: Determine output of the net

$$netOutB_k = b_k + \sum_{j=1}^{h} w_{jk} y_j$$

$$y_k = f(netOutB_k)$$

Step 8: Determine error and update weights

If (t ≠ y),
Do step 9-910 for each hidden unit whose net input is sufficiently close to 0. Start with the unit whose net input is closest to 0, then for the next closest and so on.

Step 9: Change the unit's output form +1 yo -1, or vice versa.

Step 10: Recompute the output of the net.
If the error is reduced
Adjust the weights on that unit
Use its newly assigned target value and apply the Delta rule.

Step 11: Test stopping condition:
If the weight changes have stopped or reached an acceptable level, or if a specific maximum number of weight update iterations (Step 2) have been performed then stop; otherwise continue

2.3.2.6 Back propagation algorithm

Probably, the most cited learning algorithm in ANN literature is the error back propagation (EBP) [44]. Werbos, Parker and Rumelhart developed the back propagation learning for the MLP (Figure 2-4) [45] [46] [47] [48]. It is based on the *gradient descent* minimization method. Back-propagation gets its name from the fact that the ANN is presented with an input pattern, for which an output pattern is calculated. Then, the error between desired and actual output can be determined, and passed backwards through the ANN. Based on these errors, weight adaptations are calculated, and errors are passed to a previous layer, continuing until the first layer is reached. The error is thus propagated back through the ANN [49].

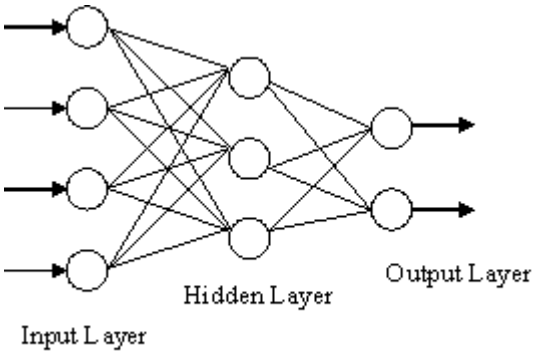

Figure 2-4 Multi layer perceptron

This algorithm can be described in the following steps:

Step 1: Initialize weights.
Set learning rate η ($0 < \eta \leq 1$)

Step 2: While stopping condition is false, do steps 3-8.

Step 3: For each training pair, **s:t** , do steps 4-7.

Step 4: Set activations of input units
$x_i = s_i; i = 1:n$

Step 5: Compute response of each unit

$$netOutB_j = b_j + \sum_{i=1}^{n} x_i w_{ij}$$

$$y_j = \frac{1}{1+\exp^{-netOutB_j}}$$

Step 6: Update weights and bias between input and hidden layer,
for i = 1:n,

$$w_{ij}(t+1) = w_i(t) + \eta \partial_j y_i \qquad (2\text{-}33)$$

$$b_j(t+1) = b_j(t) + \eta \partial_j \qquad (2\text{-}34)$$

the error is calculated as

Contd..

$$\partial_j = y_j * (1 - y_j) * \sum_k \partial_k w_{jk} \quad (2\text{-}35)$$

Step 7: Update weights and bias between hidden and output layer, for j = 1:h,

$$w_{jk}(t+1) = w_{jk}(t) + \eta \partial_k y_j \quad (2\text{-}36)$$

$$b_k(t+1) = b_k(t) + \eta \partial_k \quad (2\text{-}37)$$

the error is calculated as

$$\partial_k = y_k * (1 - y_k) * (d_k - y_k) \quad (2\text{-}38)$$

Step 8: Test stopping condition:
If the error is smaller than a specific tolerance or the number of iteration exceeds a specific value, stop; else continue.

2.3.2.7 Modification of EBP

There had been many modifications of the original EBP by many researchers. Following some of the modifications will be discussed.

2.3.2.7.1 Adding momentum factor

If a momentum term is added in the weight updating formula then the convergence becomes faster and also training is more stable. It involves adding a term to the weight adjustment that is proportional to the previous weight change. Once an adjustment is made the algorithm remembers its term and serves to modify all subsequent weight adjustments. Using the momentum method, the network tends to follow the bottom of narrow gullies in the error surface rather than crossing rapidly from side to side

This algorithm can be described in the following steps:

Step 1: Initialize weights.
Set learning rate η ($0 < \eta \le 1$)
Set momentum factor α ($0 < \alpha \le 1$)

Step 2: While stopping condition is false, do steps 3-8.

Step 3: For each training pair, **s:t** , do steps 4-7.

Step 4: Set activations of input units

$$x_i = s_i; i = 1:n$$

Step 5: Compute response of output unit

$$netOutB_j = b_j + \sum_{i=1}^{n} x_i w_{ij}$$

$$y_j = \frac{1}{1+\exp^{-netOutB_j}}$$

Step 6: Update weights and bias between input and hidden layer,
for i = 1:n,

$$w_{ij}(t+1) = w_{ij}(t) + \eta \partial_j y_i + \alpha\left(w_{ij}(t) - w_{ij}(t-1)\right) \quad (2\text{-}39)$$

$$b_j(t+1) = b_j(t) + \eta \partial_j + \alpha\left(b_j(t) - b_j(t-1)\right) \quad (2\text{-}40)$$

where
the error is calculated as

$$\partial_j = y_j * (1 - y_j) * \sum_k \partial_k w_{jk}$$

Step 7: Update weights and bias between hidden and output layer,
for j = 1:h,

$$w_{jk}(t+1) = w_{jk}(t) + \eta \partial_k y_j + \alpha\left(w_{jk}(t) - w_{jk}(t-1)\right) \quad (2\text{-}41)$$

$$b_k(t+1) = b_k(t) + \eta \partial_k + \alpha\left(b_k(t) - b_k(t-1)\right) \quad (2\text{-}42)$$

the error is calculated as

$$\partial_k = y_k * (1 - y_k) * (d_k - y_k)$$

Step 8: Test stopping condition:
If the error is smaller than a specific tolerance or the number of iteration exceeds a specific value, stop; else continue.

2.3.2.7.2 Exponential smoothing

Exponential smoothing was described by Sejnowski and Rosenberg in 1987. It is a method similar to adding a momentum term, but in this case a multiplication factor, which is also known as the smoothing term is added to the original weight updating formula. The smoothing factor is called as β, where β is in the range of 0 to 1. If β is 0, then smoothing is minimum; the entire weight adjustment comes from the newly calculated change. If β is 1, the new adjustment is ignored and the previous one is repeated. Between 0 and 1 is a region where the weight adjustment is smoothed by an amount proportional to β.

This algorithm can be described in the following steps:

Step 1: Initialize weights.
Set learning rate η ($0 < \eta \leq 1$)
Set Smoothing factor α ($0 < \alpha \leq 1$)

Step 2: While stopping condition is false, do steps 3-8.

Step 3: For each training pair, **s:t** , do steps 4-7.

Step 4: Set activations of input units
$x_i = s_i; i = 1:n$

Step 5: Compute response of output unit

$$netOutB_j = b_j + \sum_{i=1}^{n} x_i w_{ij}$$

$$y_j = \frac{1}{1+\exp^{-netOutB_j}}$$

Step 6: Update weights and bias between input and hidden layer,
for i = 1:n,

$$w_{ij}(t+1) = w_{ij}(t) + \eta\left(\alpha * \left(w_{ij}(t) - w_{ij}(t-1)\right) + \left((1-\alpha) * \partial_j y_i\right)\right) \quad (2\text{-}43)$$

Contd..

$$b_j(t+1) = b_j(t) + \eta\left(\alpha\left(b_j(t) - b_j(t-1)\right) + \left((1-\alpha) * \partial_j\right)\right) \quad (2\text{-}44)$$

the error is calculated as

$$\partial_j = y_j * (1 - y_j) * \sum_k \partial_k w_{jk}$$

Step 7: Update weights and bias between hidden and output layer, for j = 1:h,

$$w_{jk}(t+1) = w_{jk}(t) + \eta\left(\alpha * \left(w_{jk}(t) - w_{jk}(t-1)\right) + \left((1-\alpha) * \partial_k y_j\right)\right) \quad (2\text{-}45)$$

$$b_k(t+1) = b_j(t) + \eta\left(\alpha\left(b_j(t) - b_j(t-1)\right) + \left((1-\alpha) * \partial_k\right)\right) \quad (2\text{-}46)$$

the error is calculated as

$$\partial_k = y_k * (1 - y_k) * (d_k - y_k)$$

Step 8: Test stopping condition:
If the error is smaller than a specific tolerance or the number of iteration exceeds a specific value, stop; else continue.

2.3.2.7.3 Adaptive control

Researchers have shown that the primary learning parameters such as learning rate and momentum can serve a better purpose, if they can be changed dynamically [50], [51]. In the initial phase during the training, it can be assumed that the system is too far from the final goal; hence a comparatively large learning rate and momentum are required during this stage. As the learning progress, the value of the two parameters gets smaller so that the solution can not be missed.

This algorithm can be described in the following steps:

Step 1: Initialize weights.
Set learning rate η ($0 < \eta \leq 1$)

Step 2: While stopping condition is false, do steps 3-9.

Step 3: For each training pair, **s:t** , do steps 4-8.

Contd..

Step 4: Set activations of input units

$$x_i = s_i; i = 1:n$$

Step 5: Compute response of output unit

$$netOutB_j = b_j + \sum_{i=1}^{n} x_{ij} w_i$$

$$y_j = \frac{1}{1 + \exp^{-netOutB_j}}$$

Step 6: Update weights and bias between input and hidden layer, for i = 1:n,

$$w_{ij}(t+1) = w_{ij}(t) + \eta \partial_j x_i$$
$$b_j(t+1) = b_j(t) + \eta \partial_j$$

the error is calculated as

$$\partial_j = y_j * (1 - y_j) * \sum_k \partial_k w_{jk}$$

Step 7: Update weights and bias between hidden and output layer,

for j = 1:h,

$$w_{jk}(t+1) = w_{jk}(t) + \eta \partial_k y_j$$
$$b_k(t+1) = b_k(t) + \eta \partial_k$$

the error is calculated as

$$\partial_k = y_k * (1 - y_k) * (d_k - y_k)$$

Step 8: If the weight update decrease the cost function

$$\alpha = \alpha + 0.1$$

else

$$\alpha = \alpha - 0.1$$

Step 9: Test stopping condition:
If the error is smaller than a specific tolerance or the number of iteration exceeds a specific value, stop; else continue.

2.3.2.7.4 VM

Verma and Mulawka (1989) [52] described two similar techniques based on momentum method. In method 1, a new term vm1 has been added where vm1 is a constant number It has been demonstrated that with these easily implemented changes convergence time has been reduced by an average 60% for method 1. In second method with these changes the training time has been reduced by an average 70-80%. Experiments indicate good features of this approach.

Method 1 can be described in the following steps:

Step 1: Initialize weights.
Set learning rate η ($0 < \eta \leq 1$)
Set momentum factor α ($0 < \alpha \leq 1$)
Set vm1 factor vm1 ($0 < vm1 \leq 1$)

Step 2: While stopping condition is false, do steps 3-8.

Step 3: For each training pair, **s:t** , do steps 4-7.

Step 4: Set activations of input units
$x_i = s_i; i = 1:n$

Step 5: Compute response of output unit

$$netOutB_j = b_j + \sum_{i=1}^{n} x_{ij} w_i$$

$$y_j = \frac{1}{1 + \exp^{-netOutB_j}}$$

Step 6: Update weights and bias between input and hidden layer,
for i = 1:n,

$$w_{ij}(t+1) = w_{ij}(t) + \eta \partial_j y_i + \alpha\left(w_{ij}(t) - w_{ij}(t-1)\right) + \\ + vm1\left(w_{ij}(t-1) - w_{ij}(t-2)\right) \quad (2\text{-}47)$$

$$b_j(t+1) = b_j(t) + \eta \partial_j + \alpha\left(b_j(t) - b_j(t-1)\right) + \\ + vm1\left(b_j(t-1) - b_j(t-2)\right) \quad (2\text{-}48)$$

Contd..

the error is calculated as

$$\partial_j = y_j * (1 - y_j) * \sum_k \partial_k w_{jk}$$

Step 7: Update weights and bias between hidden and output layer, for j = 1:h,

$$w_{jk}(t+1) = w_{jk}(t) + \eta \partial_k y_j + \alpha \left(w_{jk}(t) - w_{jk}(t-1) \right) + \\ + vm1 \left(w_{jk}(t-1) - w_{jk}(t-2) \right) \quad (2\text{-}49)$$

$$b_k(t+1) = b_k(t) + \eta \partial_k + \alpha \left(b_k(t) - b_k(t-1) \right) + \\ + vm1 \left(b_k(t-1) - b_k(t-2) \right) \quad (2\text{-}50)$$

the error is calculated as

$$\partial_k = y_k * (1 - y_k) * (d_k - y_k)$$

Step 8: Test stopping condition:
If the error is smaller than a specific tolerance or the number of iteration exceeds a specific value, stop; else continue.

Method 2 can be described in the following steps:

Step 1: Initialize weights.
Set learning rate η ($0 < \eta \leq 1$)
Set momentum factor α ($0 < \alpha \leq 1$)
Set vm1 factor vm1 ($0 < vm1 \leq 1$)

Step 2: While stopping condition is false, do steps 2-8.

Step 3: For each training pair, **s:t** , do steps 3-7.

Step 4: Set activations of input units
$x_i = s_i; i = 1:n$

Step 5: Compute response of output unit

$$netOutB_j = b_j + \sum_{i=1}^{n} x_i w_{ij}$$

$$y_j = \frac{1}{1 + \exp^{-netOutB_j}}$$

Contd..

Step 6: Update weights and bias between input and hidden layer, for i = 1:n,

$$w_{ij}(t+1) = w_{ij}(t) + \\ +vm\left[vm*\alpha*\left(w_{ij}(t) - w_{ij}(t-1)\right) + \eta\partial_j y_i \right] \quad (2\text{-}51)$$

$$b_j(t+1) = b_j(t) + \\ +vm\left[vm*\alpha*\left(b_j(t) - b_j(t-1)\right) + \eta\partial_j \right] \quad (2\text{-}52)$$

the error is calculated as

$$\partial_j = y_j*(1-y_j)*\sum_k \partial_k w_{jk}$$

Step 7: Update weights and bias between hidden and output layer, for j = 1:h,

$$w_{jk}(t+1) = w_{jk}(t) + \\ +vm\left[vm*\alpha*\left(w_{jk}(t) - w_{jk}(t-1)\right) + \eta\partial_k y_j \right] \quad (2\text{-}53)$$

$$b_k(t+1) = b_k(t) + \\ +vm\left[vm*\alpha*\left(b_k(t) - b_k(t-1)\right) + \eta\partial_k \right] \quad (2\text{-}54)$$

the error is calculated as

$$\partial_k = y_k*(1-y_k)*(d_k - y_k)$$

Step 8: Test stopping condition:
If the error is smaller than a specific tolerance or the number of iteration exceeds a specific value, stop; else continue.

2.3.2.8 Nonlinear backpropagation (NLEBP)

Nonlinear backpropagation algorithm [53] was proposed by John Hertz, Anders Krogh, Benny Lautrup, and Torsten Lehmaan. The conventional linear backpropagation algorithm is replaced by the nonlinear version.

The algorithm can be described in the following steps:

Step 1: Initialize weights.
Set learning rate η ($0 < \eta \leq 1$)
Set learning rate α ($0 < \alpha \leq 1$)

Step 2: While stopping condition is false, do steps 3-9.

Step 3: For each training pair, **s:t** , do steps 4-8.

Step 4: Set activations of input units
$x_i = s_i; i = 1:n$

Step 5: Compute response of hidden unit

$$netOutB_j = b_j + \sum_{i=1}^{n} x_i w_{ij}$$

$$V_j = f(netOutB_j)$$

$$y_j = f\left(netOutB_j + \left[\sum_{k=1}^{M} (y_k - V_k) w_{jk} \right] \right) \quad (2\text{-}55)$$

Step 6: Compute response of output unit

$$netOutB_k = b_k + \sum_{j=1}^{h} V_j w_{jk}$$

$$V_k = f(netOutB_k)$$

$$y_k = f\left(netOutB_k + \frac{\eta}{\alpha}(T_k - V_k) \right) \quad (2\text{-}56)$$

Step 7: Update weights and bias between input and hidden layer,
for i = 1:n,

$$w_{ij}(t+1) = w_{ij}(t) + \eta\left(y_j - V_j\right) x_i \quad (2\text{-}57)$$

$$b_j(t+1) = b_j(t) + \eta\left(y_j - V_j\right) \quad (2\text{-}58)$$

Step 8: Update weights and bias between hidden and output layer,
for j = 1:h,

$$w_{jk}(t+1) = w_{jk}(t) + \eta\left(y_k - V_k\right) V_j \quad (2\text{-}59)$$

Contd..

$$b_k(t+1) = b_k(t) + \eta\left(y_k - V_k\right) \quad (2\text{-}60)$$

Step 9: Test stopping condition:
If the error is smaller than a specific tolerance or the number of iteration exceeds a specific value, stop; else continue.

2.3.2.9 Successive over relaxation EBP algorithm

The algorithm is based on the successive overrelaxation algorithm [54] for system of linear equations. This is a variation of classical backpropagation algorithm. The updating of weights uses always the most recently computed information about the weights on the other layer.

The algorithm can be described in the following steps:

Step 1: Initialize weights.
Set learning rate η ($0 < \eta \leq 1$)
Set learning rate α ($0 < \alpha \leq 1$)

Step 2: While stopping condition is false, do steps 3-9.

Step 3: For each training pair, **s:t** , do steps 4-8.

Step 4: Set activations of input units
$x_i = s_i; i = 1:n$

Step 5: Compute response of hidden unit

$$netOutB_j = b_j + \sum_{i=1}^{n} x_i w_{ij}$$

$$y_j = f\left(netOutB_j\right)$$

Step 6: Compute response of output unit

$$netOutB_k = b_k + \sum_{j=1}^{hid} y_j w_{jk}$$

$$y_k = f\left(netOutB_k\right)$$

Contd..

Step 7: Update weights between hidden and output layer.

$$w_{jk}(t+1) = w_{jk}(t) + \eta\left(t_k - y_k\right)\left(f'\left(netOutB_k\right)\right)y_j \quad (2\text{-}61)$$

$$y_k(t+1) = y_k(t) + \Delta w y_j \quad (2\text{-}62)$$

Step 8: Update weights between input and hidden layer.

$$w_{ij}(t+1) = w_{ij}(t) + \eta\sum_{k=1}^{M}\left(t_k - y_k\right)\left(f'\left(netOutB_k\right)\right)y_j\left(w_{jk}(t+1)\right)\left(f'\left(netOutB_j\right)\right)x_i \quad (2\text{-}63)$$

$$y_j(t+1) = y_k(t) + \Delta w x_i \quad (2\text{-}64)$$

Step 9: Test stopping condition:
If the error is smaller than a specific tolerance or the number of iteration exceeds a specific value, stop; else continue

2.3.2.10 RP propagation

Another quite well known heuristic method, suggested by several authors, is RP prop. In this algorithm the uses of individual learning rates for each weight are adjusted according to how `well' the actual weight is doing. Ideally, these individual learning rates are gradually adjusted to the curvature of the error surface and reflect the inverse of the Hessian. There are few other algorithms that use this methodology. But, probably the most promising among these schemes is Rprop. The algorithm combines the use of individual learning rates with the Manhattan updating rule, adjusting the learning step for each weight according to the formula [55] [56] [57]

The algorithm can be described in the following steps

Step 1: Initialize weights.

Step 2: While stopping condition is false, do steps 2-6.

Step 3: For each training pair, **s:t** , do steps 3-5.

Step 4: Set activations of input units
$x_i = s_i; i = 1:n$

Contd..

Step 5: Compute response of output unit

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

$$y_j = \frac{1}{1+\exp^{-netOutB_j}}$$

Step 6: Update weights , for i = 1:n, $0<\eta^-<1<\eta^+$

If $\left(\frac{\partial E}{\partial w_{ij}}(t-1) * \frac{\partial E}{\partial w_{ij}}(t) > 0\right)$ then

$$\Delta_{ij}(t) = \min\left(\Delta_{ij}(t) * \eta^+, \Delta_{\max}\right) \quad (2\text{-}65)$$

$$\Delta w_{ij} = -sign\left(\frac{\partial E}{\partial w_{ij}}(t)\right) * \Delta_{ij}(t) \quad (2\text{-}66)$$

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij} \quad (2\text{-}67)$$

else if $\left(\frac{\partial E}{\partial w_{ij}}(t-1) * \frac{\partial E}{\partial w_{ij}}(t) < 0\right)$ then

$$\Delta_{ij}(t) = \max\left(\Delta_{ij}(t) * \eta^-, \Delta_{\min}\right) \quad (2\text{-}68)$$

$$w_{ij}(t+1) = w_{ij}(t) - \Delta w_{ij} \quad (2\text{-}69)$$

$$\frac{\partial E}{\partial w_{ij}}(t) = 0 \quad (2\text{-}70)$$

else if $\left(\frac{\partial E}{\partial w_{ij}}(t-1) * \frac{\partial E}{\partial w_{ij}}(t) = 0\right)$ then

$$\Delta w_{ij} = -sign\left(\frac{\partial E}{\partial w_{ij}}(t)\right) * \Delta_{ij}(t) \quad (2\text{-}71)$$

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}$$

Step 6: Test stopping condition:
If the error is smaller than a specific tolerance or the number of iteration exceeds a specific value, stop; else continue.

2.3.2.11 Quick propagation

The basic theory of Quick prop algorithm is to estimate the weight changes by assuming a parabolic error surface to approximate the curvature information [58]. The weight changes are then modified by the use of heuristic rules to ensure downhill motion at all times. This updating corresponds to a 'switched' gradient descent with a parabolic estimate for the momentum term. To prevent the weights from growing too large, which indicates that QP is going wrong, a maximum scale is set on the weight update and it is recommended to use a weight decay term. The algorithm is also restarted if the weights grow too large.

The algorithm can be described in the following steps

Step 1:	Initialize weights. Set learning rate η ($0 < \eta \leq 1$)
Step 2:	While stopping condition is false, do steps 2-7.
Step 3:	For each training pair, **s:t** , do steps 4-6.
Step 4:	Set activations of input units $x_i = s_i; i = 1:n$
Step 5:	Compute response of output unit $netOutB_j = b + \sum_{i=1}^{n} x_i w_{ij}$ $y_j = \frac{1}{1+\exp^{-netOutB_j}}$

Step 6:	Update weights, for i = 1:n
	If $\left(\Delta w_{ij}(t) > 0\right)$ then
	If $\left(\frac{\partial E}{\partial w_{ij}}(t) > 0\right)$ then

Contd..

$$\Delta w_{ij}(t+1) = \eta * \frac{\partial E}{\partial w_{ij}}(t) \tag{2-72}$$

If $\left(\frac{\partial E}{\partial w_{ij}}(t) > \frac{\mu}{1-\mu} \frac{\partial E}{\partial w_{ij}}(t-1) \right)$ then

$$\Delta w_{ij}(t+1) = \Delta w_{ij}(t) + \mu * \Delta w_{ij}(t-1) \tag{2-73}$$

else

$$\Delta w_{ij}(t+1) = \Delta w_{ij}(t) + \frac{\frac{\partial E}{\partial w_{ij}}(t) * \Delta w_{ij}(t-1)}{\frac{\partial E}{\partial w_{ij}}(t-1) - \frac{\partial E}{\partial w_{ij}}(t)} \tag{2-74}$$

else if $\left(\Delta w_{ij}(t) < 0 \right)$ then

If $\left(\frac{\partial E}{\partial w_{ij}}(t) < 0 \right)$ then

$$\Delta w_{ij}(t+1) = \eta * \frac{\partial E}{\partial w_{ij}}(t)$$

If $\left(\frac{\partial E}{\partial w_{ij}}(t) < \frac{\mu}{1-\mu} \frac{\partial E}{\partial w_{ij}}(t-1) \right)$ then

$$\Delta w_{ij}(t+1) = \Delta w_{ij}(t) + \mu * \Delta w_{ij}(t-1)$$

else

$$\Delta w_{ij}(t+1) = \Delta w_{ij}(t) + \frac{\frac{\partial E}{\partial w_{ij}}(t) * \Delta w_{ij}(t-1)}{\frac{\partial E}{\partial w_{ij}}(t-1) - \frac{\partial E}{\partial w_{ij}}(t)}$$

else

$$\Delta w_{ij}(t+1) = \eta * \frac{\partial E}{\partial w_{ij}}(t)$$

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t) \tag{2-75}$$

Step 7: Test stopping condition:
If the error is smaller than a specific tolerance or the number of iteration exceeds a specific value, stop; else continue.

2.3.3 Second order gradient based learning

In second order gradient based learning, the propagated error is based on the second order partial derivative information with respect to the weights. Where as in the first order Gradient descent assumes a flat metric where the learning rate is identical in all directions in ω-space, in this case a more importance is given to the optimal value of learning, which can be changed dynamically. The reason behind the adaptive learning is that the first order learning does not provide us an optimal learning rate and it is wise to modify it according to the appropriate metric [59].

2.3.3.1 Conjugate gradient descent

Conjugate gradient descent works by constructing a series of line searches across the error surface. It first works out the direction of steepest descent, just as back propagation would do. However, instead of taking a step proportional to a learning rate, conjugate gradient descent projects a straight line in that direction and then locates a minimum along this line, a process that is quite fast as it only involves searching in one dimension. Subsequently, further line searches are conducted (one per epoch). The directions of the line searches (the conjugate directions) are chosen to try to ensure that the directions that have already been minimized stay minimized (contrary to intuition, this does not mean following the line of steepest descent each time).

The conjugate directions are actually calculated on the assumption that the error surface is quadratic, which is not generally the case. However, it is a fair working assumption, and if the algorithm discovers that the current line search direction isn't actually downhill, it simply calculates the line of steepest descent and restarts the search in that direction. Once a point close to a minimum is found, the quadratic assumption holds true and the minimum can be located very quickly [60] [61].

.

The algorithm can be described by the following steps

Step 1:	Initialize weights. Initialize the search direction $d(0)$
Step 2:	While stopping condition is false, do steps 3-6.
Step 3:	Calculate β(t) as follows

$$\beta(t) = \begin{Bmatrix} \nabla E(t+1) \bullet \left(\nabla E(t+1) - \nabla E(t)\right) / d(t) \bullet \left(\nabla E(t+1) - \nabla E(t)\right) \left(Hestens - Stiefel\right) \\ \nabla E(t+1) \bullet \left(\nabla E(t+1) - \nabla E(t)\right) / \nabla E(t) \bullet \nabla E(t) \left(Polak - Ribiere\right) \\ \nabla E(t+1) \bullet \nabla E(t+1) / \nabla E(t) \bullet \nabla E(t) \left(Fletcher - \mathrm{Re}eves\right) \end{Bmatrix} \quad (2\text{-}76)$$

Step 4: Find subsequent conjugate direction $d(t+1)$ as follows

$$d(t+1) = -\nabla E(t+1) + \beta(t) d(t) \quad (2\text{-}77)$$

Step 5: Estimates the minimization step η_t using the formula

$$\eta(t) = \frac{-d(t).\nabla E(t)}{d(t).(s(t) + \lambda(t) d(t))} \quad (2\text{-}78)$$

where λ is a fudge factor to make the denominator positive and **s** is a difference approximation of H **d**.

Step 6: Update the weights:

$$\Delta w_t = \eta_t d_t \quad (2\text{-}79)$$

$$w(t+1) = w(t) + \Delta w(t)$$

Step 7: Test stopping condition:
If the error is smaller than a specific tolerance or the number of iteration exceeds a specific value, stop; else continue.

2.3.3.2 Quasi newton

Another very popular second order gradient descent based algorithm is Quasi Newton method (Bishop, 1995; Shepherd, 1997. It has been cited in many ANN literature that in terms of time complexity this method is much better than the standard. Some earlier work suggested that with small number of weights (less

than couple of hundreds), the performance of this method is the best. If the network is a single output regression network and the given problem has low residuals, then Levenberg-Marquardt (LM) may perform better.

The frequency of updating weights in Quasi-Newton is a block update algorithm. For this reason, there is no shuffle option available with Quasi-Newton, since it would clearly serve no useful function. Also, selection of learning rate and momentum is not required in this method. Additive noise would destroy the assumptions made by Quasi-Newton about the shape of search space, and so is also not available.

As in the case of other second order algorithm, Quasi-Newton works by exploiting the observation that, on a quadratic (i.e. parabolic) error surface, one can step directly to the minimum using the Newton step - a calculation involving the Hessian matrix (the matrix of second partial derivatives of the error surface). The approximation calculation of the Hessian matrix is computed by some iterative method. The approximation at first follows the line of steepest descent, and later follows the estimated Hessian more closely.

The algorithm can be described by the following steps:

Step 1: Initialize weights.
Initialize the search direction $d(0)$
Initialize the Hessian matrix H as I

Step 2: While stopping condition is false, do steps 3-6.

Step 3: Find subsequent conjugate direction $d(t+1)$ as follows

$$d(t+1) = -H(t)g \qquad (2\text{-}80)$$

where g is the direction of steepest descent.

Step 4: Update the weight

$$\Delta w(t) = H(t)d(t) \qquad (2\text{-}81)$$

$$w(t+1) = w(t) + \Delta w(t)$$

Step 5: Update the Hessian matrix:

Contd..

$$H(t+1) = H(t) + \frac{\left(y(t+1)-y(t)\right)\otimes\left(y(t+1)-y(t)\right)}{\left(y(t+1)-y(t)\right)\bullet\left(\nabla w(t+1)-\nabla w(t)\right)} - \frac{\left[H(t)\bullet\left(\nabla w(t+1)-\nabla w(t)\right)\right]\otimes\left[H(t)\bullet\left(\nabla w(t+1)-\nabla w(t)\right)\right]}{\left(\nabla w(t+1)-\nabla w(t)\right)\bullet H(t)\bullet\left(\nabla w(t+1)-\nabla w(t)\right)} + \left[\left(\nabla w(t+1)-\nabla w(t)\right)\bullet H(t)\bullet\left(\nabla w(t+1)-\nabla w(t)\right)\right]u\otimes u \quad (2\text{-}82)$$

Step 6: Test stopping condition:
If the error is smaller than a specific tolerance or the number of iteration exceeds a specific value, stop; else continue.

2.3.3.3 Levenberg-Marquardt

Another well cited non linear optimization algorithm in ANN literature is Levenberg Marquardt (LM) algorithm (Levenberg, 1944; Marquardt, 1963; Bishop, 1995; Shepherd, 1997; Press et. al., 1992). It is reputably one of the fastest algorithms. However the use of LM algorithm is restricted as follows

2.3.3.3.1 Single output network

LM algorithm can only be used on networks with a single output unit.

2.3.3.3.2 Small networks

Levenberg-Marquardt has space requirements proportional to the square of the number of weights in the network. This effectively precludes its use in networks of any great size (more than a few hundred weights).

2.3.3.3.3 Sum-squared error function

Levenberg-Marquardt is only defined for the sum squared error function. If a different error function is selected for the network, it will be ignored during LM training. It is usually therefore only appropriate for regression networks.

Like other iterative algorithms, Levenberg-Marquardt does not train radial units. Therefore, LM algorithm can be used to optimize the non-radial layers of radial basis function networks even if there are a large number of weights in the radial layer, as those are ignored by LM. This is significant as it is typically the radial layer that is very large in such networks.

LM algorithm works by making the assumption that the underlying function being modeled by the neural network is linear. Based on this calculation, the minimum can be determined exactly in a single step. The calculated minimum is tested, and if the error there is lower, the algorithm moves the weights to the new point. This process is repeated iteratively on each generation. Since the linear assumption is ill-founded, it can easily lead Levenberg-Marquardt to test a point that is inferior (perhaps even wildly inferior) to the current one. The clever aspect of Levenberg-Marquardt is that the determination of the new point is actually a compromise between a step in the direction of steepest descent and the above-mentioned leap. Successful steps are accepted and lead to a strengthening of the linearity assumption (which is approximately true near to a minimum). Unsuccessful steps are rejected and lead to a more cautious downhill step. Thus, Levenberg-Marquardt continuously switches its approach and can make very rapid progress.

Levenberg-Marquardt algorithm is designed specifically to minimize the sum-of-squares error function, using a formula that partly assumes that the underlying function modeled by the network is linear. Close to a minimum this assumption is approximately true, and the algorithm can make very rapid progress. Further away it may be a very poor assumption. Levenberg-Marquardt therefore compromises between the linear model and a gradient-descent approach. A move is only accepted if it improves the error, and if necessary the gradient-descent model is used with a sufficiently small step to guarantee downhill movement.

The algorithm can be described by the following steps:

Step 1:	Initialize weights. Set learning rate η ($0 < \eta \leq 1$)
Step 2:	While stopping condition is false, do steps 3-7.
Step 3:	For each training pair, **s:t** , do steps 4-6.
Step 4:	Set activations of input units $x_i = s_i; i = 1:n$
Step 5:	Compute response of output unit $netOutB_j = b + \sum_{i=1}^{n} x_i w_{ij}$ $y_j = \frac{1}{1+\exp^{-netOutB_j}}$
Step 6:	Update weights, $\Delta w = -\left(Z^T Z + \alpha I\right)^{-1} Z^T \varepsilon$ (2-83) where E is the case error vector and Z is the matrix of partial derivatives of these errors with respect to the weights.
Step 7:	Test stopping condition: If the error is smaller than a specific tolerance or the number of iteration exceeds a specific value, stop; else continue.

2.3.3.4 Radial bias Function (RBF) network

Radial basis function networks are also feed forward, but have only *one* hidden layer. Like EBP, RBF nets can learn arbitrary mappings: the primary difference is in the hidden layer.

RBF (Figure 2-6) hidden layer units have a *receptive field* which has a *centre:* that is, a particular input value at which they have a maximal output. Their output tails off as the input moves away from this point.

Generally, the hidden unit function is a Gaussian function. The Gaussian function is shown in Figure 2-5.

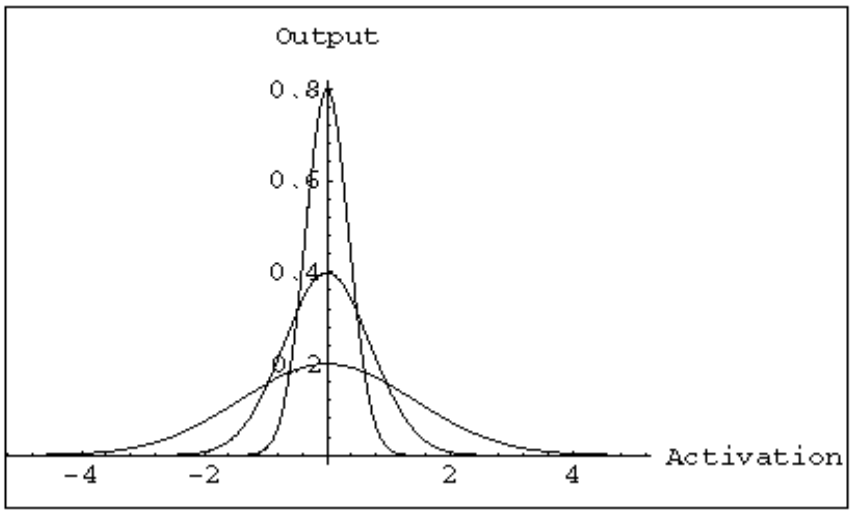

Figure 2-5 Gaussian function used for RBF

RBF networks are trained by deciding on how many hidden units there should be, deciding on their centres and the sharpness (standard deviation) of their Gaussians and training up the output layer.

Generally, the centres and standard deviation are decided on first by examining the vectors in the training data. The output layer weights are then trained using the Delta rule.

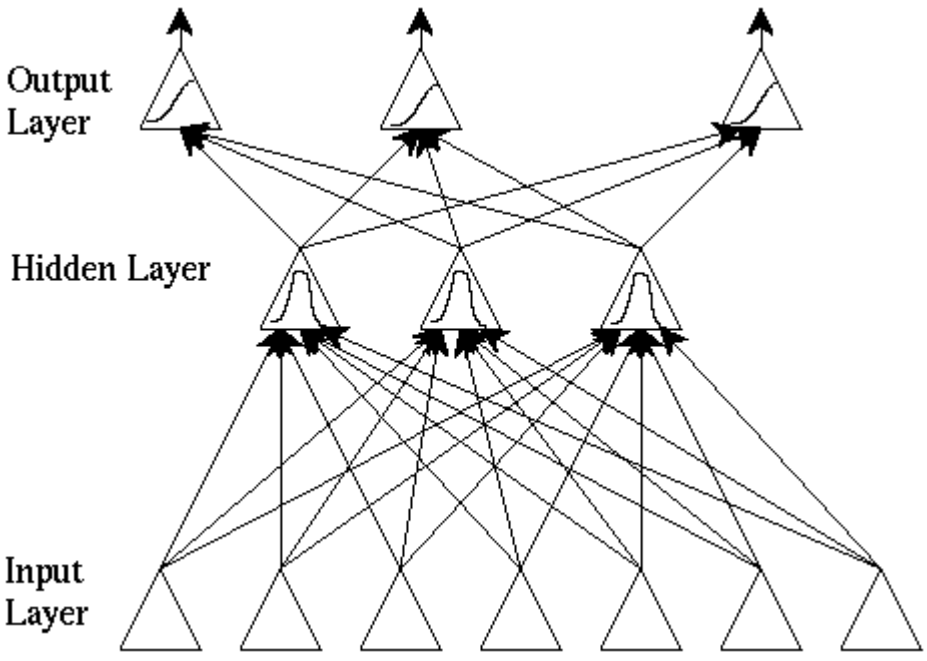

Figure 2-6 Radial bias network

The algorithm can be described in the following steps:

Step 1:	Initialize weights (c_j) between input and hidden layer, selecting randomly from the Input patterns.
Step 2:	Set smooth factor or the receptive field β ($0 < \beta \leq 1$)
Step 3:	For each training pair, **s:t** , do steps 4 -5.
Step 4:	Compute the matrix A where the element $a_{ij} = \Phi\left(\left\|x_i - c_j\right\|\right)$ (2-84)
	$\Phi(r) = e^{-(\frac{r^2}{\beta})}$ (2-85)
Step 5:	Calculate weight matrix between Hidden and output layer using R, modified Graham Schmidt or SVD method [62] [63] [64] [65] [66] [67] [68], where
	$U = SVD(A); GM(A); QR(A)$ (2-86)
	$w = U^{-1} y$ (2-87)

2.4 EVOLUTIONARY BASED LEARNING ALGORITHM

Since the publication of "The Origin of Species" by Darwin (1859), the idea about natural evolution has been fascinating. However because of the lack of support from proper computer hardware / software tool, conducting simulation experiments were not a reality until early 60 of twentieth century. Many different areas of research were evolved from this basic evolutionary theory, such as Evolutionary [69] strategy (ES), Evolutionary programming (EP), Simple genetic algorithm, Steady state genetic algorithm, Genetic Hillclimber, L-systems and genetic algorithm (GA) – some of them have already been discussed in the previous chapter. There have been few other global and local search strategies developed almost at the same time, which are also borrowed in ANN research area in parallel with the GA to some extent. Amongst them are – Exhaustive

search, Stochastic hillclimber, Simulated annealing etc. Evolutionary algorithm and their genetic counterpart will be discussed in the following subsequent section, before a short description of other comparatively less popular global search technique in the ANN pretext.

2.4.1 Exhaustive search

This is a total search through all possible solutions, obviously leading to a very high degree of time complexity. Even though theoretically exhaustive search cannot get stuck in local minima, it is not a very popular algorithm for any interesting problem, especially NP hard problem.

2.4.2 Genetic hill climber

Most often this is an alternative method for steady state genetic algorithm [70] [71], where the sexual recombination operator is ignored and the mutation operator is the driving force for creation of new genetic material. This search method is preferable when the recombination operator is considered to create unfit offspring with too high a probability. It is quite interesting to observe that like simulated annealing a genetic hillclimber may accept a new solution in the population even though the solution may have a reduced performance compared with the ancestor.

2.4.3 L- systems

In nature the precise form of a species is not coded in its genotype. Instead the genetic information of a species is more a kind of recipe. Since researchers had already used a kind of reverse engineering to come up with the idea to create ANN to obtain intelligent behavior with a computer program, Boers and Kuiper (Boers et al 1992) looked for a way to code neural network structures with recipe. They used a complex form of L system. L system was introduced by Lindenmayer to model the growth process of plants (Lindenmayer 1968). It is a kind of a grammar. The biggest difference with standard grammars is that all characters in a string are rewritten in parallel. This parallel rewriting property of L–system makes it possible to form a simple set of rules that are able to generate

strings that, given the right implementation are approximations of certain type of fractals [72] [73] [74].

2.4.4 Stochastic hill climber

A stochastic hill climber starts at a random point in the search space and iteratively makes random changes of the current solution. Each random change is only accepted if it increases the performance of the current solution. If the random change decreases the performance it is rejected. The algorithm has a high probability of getting stuck in the local minima.

2.4.5 Simulated annealing

This method also starts at a random point, but its performance is much better than the stochastic hillclimber, since it accepts a negative random change with a probability P. The probability P is decreased during the run by decreasing a temperature, which defines P. Accepting negative change, gives the simulated annealing approach the ability to escape local minima.

2.4.6 Genetic algorithm

The use of evolutionary based algorithms for training neural networks has recently begun to receive a considerable amount of attention. Although the origins of evolutionary computing can be traced back to the late 1950's, the field remains relatively unknown to the boarder scientific community for almost few decades. The fundamental work of Holand, Rechenberg, Schwefel and Fogel served to slowly change this picture during the 1970s [75]. Much of the research however has focused on the training of feed forward networks [Fogel, Fogel, and Porto, 1990; Whitley, Starkweather, and Bogart, 1990] [76] [77] [78]. Just as neurobiology is the inspiration for artificial neural networks, genetics and natural selection are the inspiration of the genetic algorithm (GA). It was developed by John Holland [79]. They are based on a Darwinian type survival of the fittest strategy. An advantage of using GAs for training neural networks is that they may be used for network with arbitrary topologies. Also, GA does not rely on calculating the gradient of the cost function. Cost functions need to be calculated

to determine their fitness [80] [81]. A learning method of this type basically has two parts: a merit function and a weight update method. The merit function quantitatively measures the performance of the network, and the update method determines the appropriate changes in the weight space.

There have been previous attempts to merge the concepts of GA and neural networks. One approach was to use a GA [82] [83] [84] [85] to design NN architectures followed by the application of the EBP algorithm to train each network. Another approach began with fixed network architecture, typically fully connected and then GA was applied to evolve the interconnected weights. Some researchers expanded these approaches by evolving the interconnected structure and weights for a NN but did not evolve the size if the NN. Few demonstrated that GA [86] [87] could be used to completely evolve the neural network size, interconnected structure, weights, I/O connections.

Genetic Algorithm (GA) [88] makes the search for near optimally possible in domains with rough and convoluted error surface. The error surface is a multidimensional plot of the error (E) versus the combination of the input independent variables or parameter. Gas as applied to PC networks use populations of networks, represented by the complete set of weights or parameters, to optimize a merit or fitness function. Each parameter is mapped onto the range of a integer or real values. The members of the sample population are randomly initialized chromosomes each representing a different network. The fitness of a chromosome is evaluated with the complement of the network pattern error. The fitness can be expressed in terms of RMS error (E) as follows

$$F(c) = \tan\left\{\Pi(1-E)/2\right\} + F0 \tag{2-88}$$

where F0 is the minimum fitness.

There are measures for fitness also exist, for example to maximize 1/MSE.

The goal toward which GA training proceeds is determined by the fitness function the user defines. This is probably the greatest advantage of GAs for ANN training.

2.4.6.1 Comparison between GA and gradient based training

The GA based learning method is more attractive than their gradient counterpart. The reason is that the GA can handle the global search problem better in a vast, complex, multi-modal, and non-differentiable surface. It is not based on the calculation of the derivative of the error surface, which can be unavailable or sometimes very costly to find. The GA also makes it easier for the ANN for decreasing the overall complexity and generalization for the fitness of its population. But, on the contrary, GA method is very slow, and it takes much longer time than some of the fast variants of EBP and conjugate gradient algorithms. But GA's are much less sensitive to the initial condition.

Kitano reported quite different result [89]. He found that a hybrid method of GA and EBP together "is, at best, equally efficient to faster variants of back propagation in very small scale of networks, but far less efficient in larger networks." The test problems he used to compare the two methods include the XOR problem, various size encoder decoder problems, and the two-spiral problem. However, there are other reports that says that hybrid learning of GA with EBP gives excellent result [90] [91] [92] [93]. A general GA algorithm can be described as follows

Step 1:	The initial population is filled with chromosomes that have randomly valued genes. The values can be either binary or real numbers. With binary value, the probability for a gene to have either 0 or 1, is equal. For real numbers the distribution can be normal, Cauchy and others. Two parameters are mainly considered for creating the initial population. First one is the size of the population and second one is the size of the chromosome.
Step 2:	For every member of the current population the fitness function f(x) is calculated.
Step 3:	If any member of the current population satisfied the stopping criterion, which can be based on the error function, the algorithm stops else proceed to step 4.

Contd…

Step 4:	Create an intermediate population by extracting members of the current population using reproduction scheme. This is also known as selection operator. Fitness normalization is used to force the fitness of the chromosome into certain range. There are various selection operators available. The most common are roulette wheel and tournament selection.
Step 5:	Create new population from the pool of the intermediate population by applying crossover and mutation operator with certain probabilities.

The flowchart of the algorithm is also given in Figure 2-7

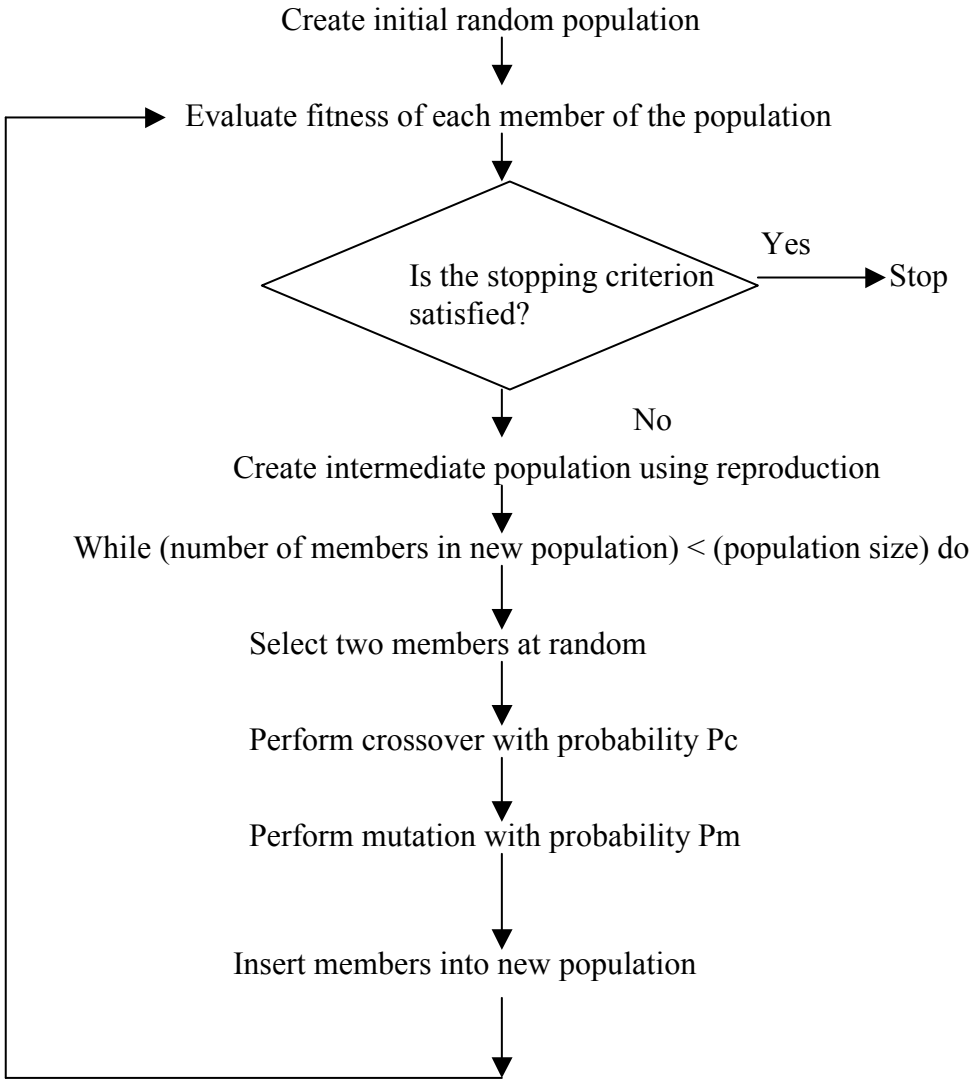

Figure 2-7 A flowchart for basic genetic algorithm

One of the problems in using GA for ANN is known as permutation problem or competing convention problem. It is caused by the many-to-one mapping from

the genotype to phenotype. The permutation problem makes crossover operator very inefficient and also ineffective in producing better offspring. A slightly modified algorithm proposed by Yao (1991) is given below [94] :

Step 1: Generate an initial population of μ individuals at random, and set k=1. Each individual is a pair real valued vectors, (W_i, η_i), $\forall i \in \{1,\ldots, \mu\}$, where W_i s are connection weight vectors and η_i s are the variance vectors for the Gaussian mutations (also known as strategy parameters in self-adaptive EA's). Each individual corresponds to an ANN.

Step 2: Each individual (W_i, η_i) , i = 1, 2, …, μ creates a single offspring (W_i', η_i')

for j = 1, 2, …, n

$$\eta_i'(j) = \eta_i(j)\exp\left(\tau' N(0,1) + \tau Nj(1,0)\right) \quad (2\text{-}89)$$

$$w_i'(j) = w_i(j) + \eta_i'(j) Nj(0,1) \quad (2\text{-}90)$$

where $W_i(j)$, $W_i'(j)$, $\eta_i(j)$, and $\eta_i'(j)$ denote the jth component of the vectors W_i, W_i', η_i, and η_i', respectively. N (0,1) denotes a normally distributed one-dimensional random number with mean and variance of 0 and 1 respectively. Nj(0,1) denotes that the random number is generated a new for each value of j. The parameter τ and τ' are commonly set to $\left(\sqrt{2\sqrt{n}}\right)^{-1}$ and $\left(\sqrt{2n}\right)^{-1}$. We can also use other mutation like Cauchy mutation.

Step 3: Determine the fitness of every individual, including all parents and offspring, based on the training error.

Step 4: Conduct a pair wise comparison over the union of parents (Wi, ηi) and offspring (W_i', η_i'), $\forall i \in \{1, \ldots, \mu\}$. For each individual, q opponents are chosen randomly with a uniform distribution from all the parents and offspring. For each comparison, if the individual's fitness is no smaller than the opponent's, it receives a win. Select μ individuals out of the parents and offspring that have most wins to form the next generation. This selection operator is also known as the tournament selection. There can be other selection operators like roulette wheel.

Step 5: Stop if stopping criterion is satisfied, otherwise, k=k+1, and go to step 2.

2.4.6.2 Evolution of architecture using GA

We can consider the architecture of an ANN to be its topological structure, i.e. connectivity and transfer functions for each node. Given a learning task, an ANN with only a few connections and linear nodes may not be able to perform all, due to its limited capability. A constructive algorithm for ANN structure design starts with a minimal network and adds more during training, while a pruning method works just in the opposite way. Angeline et al. indicated, "Such structure hill climbing methods are susceptible to becoming trapped at structural local optima." Design of the optimal architecture can be considered as a search space problem in the architecture space where each point represents a particular ANN architecture. The optimal architecture design is equivalent to find the global maxima. There are several characteristics of such a surface as indicated by Miller et al, which makes GA a better candidate [95] [96]. Those characteristics pointed out by miller are

- The architecture surface is infinitely large, hence unbounded for possible number of nodes and connections
- The surface is non-differentiable since changes in the number of nodes and connections are discrete
- The surface is complex and noisy since the mapping from the architecture to the performance is indirect, strongly epistasis, and dependent on the evaluation method used.
- The surface is deceptive since similar architectures may have quite different results.
- The surface is multi-modal since different architectures may have similar performance

A typical algorithm for the evolution of the ANN architecture is as follows:

Step 1:	Decode each individual in the current generation into architecture If the indirect coding scheme is used, some developmental rules or a training process specifies further detail of architecture.
Step 2:	Train each ANN with the decoded architecture by a predefined learning rule starting from different sets of random initial connection weights and, if any, learning rule parameters.
Step 3:	Compute the fitness of each individual according to the training result and other performance criteria such as complexity of the architecture.
Step 4:	Select parents from the population based on their fitness.
Step 5:	Apply search operators to the parents and generate offspring which can form the next generation.

2.4.6.3 Direct encoding Scheme

There are two different approaches in direct encoding scheme. In the first approach, each connection of the architecture is directly specified by its binary representation. An NXN matrix $C = (c_{ij})_{NXN}$ can represent an ANN architecture with N nodes, where c_{ij} indicates the presence or absence of the connection. Each matrix C has a direct one-to-one mapping for the corresponding ANN architecture. The binary string that represents an architecture is the concatenation of the rows (or columns) of the matrix. Such an encoding scheme can represent a recurrent network as well. This method is very suitable for the precise and fine tuned search of a compact ANN architecture and also, it facilitate rapid generation and optimization of tightly pruned interesting designs.

Another flexibility provided by this scheme stems from the fitness function. The training result pertaining to architecture can be used as the fitness function. Improvement on ANN's generalization ability can be expected if these criteria are adopted. Schaffer et al. have presented an experiment which shows that ANN designed by the GA approach has better generalization ability than EBP methods [97].

2.4.6.4 Indirect encoding scheme

In order to reduce the length of the genotype, indirect schemes can be used, where some of the characteristic of the architecture is encoded. The details can be predefined according to the prior knowledge or by a set of some deterministic development rules. Though it is more compact in genotype representation, it may not be that good in terms of its generalization ability. Few researchers argued that the indirect scheme is biologically more plausible because of its simplicity for genetic information encoded in chromosomes to specify independently the whole nervous system according to the discoveries of neuroscience [98].

2.4.6.5 Evolution of node transfer function

Stork et al [99]., were the first to apply GA to the evolution of both topological structure and node transfer functions. The structural gene contains the transfer function. It was much more complex than the sigmoidal function because they tried to model a biological neuron in a tail flip circuitry of crayfish. White and Ligomenides adopted a very similar approach. During the initial training 80% of the nodes used sigmoidal and 20% used the gaussian transfer function. Liu and Yao [100]. used GA with both sigmoidal and gaussian nodes. Their algorithm allowed growth and shrinking of the whole ANN by adding or deleting a node. Hwang et al evolved ANN topology, node transfer function, as well as connection weights for projection neural networks. Sebald and Chellapilla used node transfer function as an example to show the importance of evolving representations. Co-evolving solutions and their representations may be an effective way to tackle some difficult problems [101].

Some of the popular genetic algorithm methods that have been applied to the ANN learning method are given in the following section.

2.4.6.6 Prados's GenLearn method

Prados showed that training MLP by GA based method called, as GenLearn was significantly faster than methods that uses generalized delta rule. It has also advantages to escape the local minima in its search of weight space. In searching

for the fittest hidden neurons, GenLearn searches for a globally optimal internal representation of the input data.

In feed forward MLP, the search for an appropriate first layer weight matrix can be thought of as a search for powerful internal representation of the input data. The power of hidden neurons is due to their formation of these internal representations. The algorithm is described in the following

Step 1:	Initialize the weights and generate a set of random target hidden neuron patterns
Step 2:	Run several iterations of a chosen supervised learning rule to attempt to produce the target output patterns from the input patterns
Step 3:	Check termination conditions like RMS error and either stop if the condition is satisfied or continue to step 4
Step 4:	Measures the fitness of the hidden neuron
Step 5:	Replace the target hidden neuron patterns of the least fit hidden neurons

The first step in the GenLearn procedure is to generate a random value for each weight and to generate a random hidden-neuron pattern for each hidden neuron. These random hidden-neuron patterns give the initial target outputs for the hidden neurons. Next a learning rule is applied to attempt to produce the target hidden-neuron patterns from the input patterns [102].

2.4.6.7 GMU GANNET

This program developed by George Mason University implements GA that has evolved ANN, which exhibits properties of synchronous and asynchronous sequential machine. GANNET implements a simple neuron model. The neuron consists of a hard-limiting output function, integer weight, threshold values and allowable input output values (+1 / -1). The behavior of each neuron is specified by a 16 bits field, which serves to uniquely define one of 1882 neuron behaviors. Rather than map each weight and threshold to a section of genetic code, the GA was found to operate more effectively if behaviors were modified rather than

specific weights. There are multiple weight/threshold combinations, which map to a single behavior. Before the evolutionary process begins, an initial population of NN species must be created. The size of each NN is selected from a specified uniform distribution. The genetic code for each NN is generated with no a priori knowledge of the type of problem the experimenter is attempting to solve. Because of the unique genetic code used every binary representation generated can be successfully translated into a phenotype that is capable of being evaluated [103].

A set of fitness metrics was developed to guide the evolutionary process. The first and the most important fitness metric is the I/O performance. Before a NN is evaluated, it is set to a special dead state where the present state of every neuron is set to zero. The dead state provides a standard starting point from which a NN can start to process. To begin the process, a set of environmental inputs is presented to the NN. Each neuron computes the sum of its weighted inputs, factors in the threshold value, and generates a –/+1 signal as its next state. After all the neurons responses are computed, the next state of each neuron becomes its present state. The program will continue to find the next states until either the NN stabilizes or NN is considered to be unstable. This consideration is important because of the possibility of feedback, which allows for temporal information extraction [104].

2.4.6.8 EPNet

Yao and Liu developed an automatic system, EPNet, based on EP for simultaneous evolution of ANN architecture and connection weights [105]. It relies on a number of mutation operators to modify the architectures and weights. Functional evolution rather than genetic evolution is emphasized in EPNet. It uses rank based selection operator. And five mutation operators: hybrid training, node deletion, connection deletion and node addition. Hybrid training mutation modifies the ANN weights. It is based on a modified MP algorithm with an adaptive learning rate and simulated annealing. The other four mutations are used to grow and prune hidden nodes and connections. The number of epochs to train each ANN in a population is defined by two user defined parameters. This is known as partial training as there is no guarantee that the ANN will converge

even to a local optimum. It is used to bridge the behavioral gap between a parent and a offspring.

The five mutation operators are applied sequentially. If one mutation leads to a better offspring, then no further mutation will be applied. A validation set is used in EPNet to measure the fitness of an individual. EPNet has been tested extensively on a number of benchmark problems and achieved excellent results, including parity problem, two spiral problem, the diabetes problem, the heart disease problem, the thyroid problem, the Australian credit card problem etc. Very compact ANNs with good generalization ability have been evolved. The architecture of EPNet is shown in Figure 2-8.

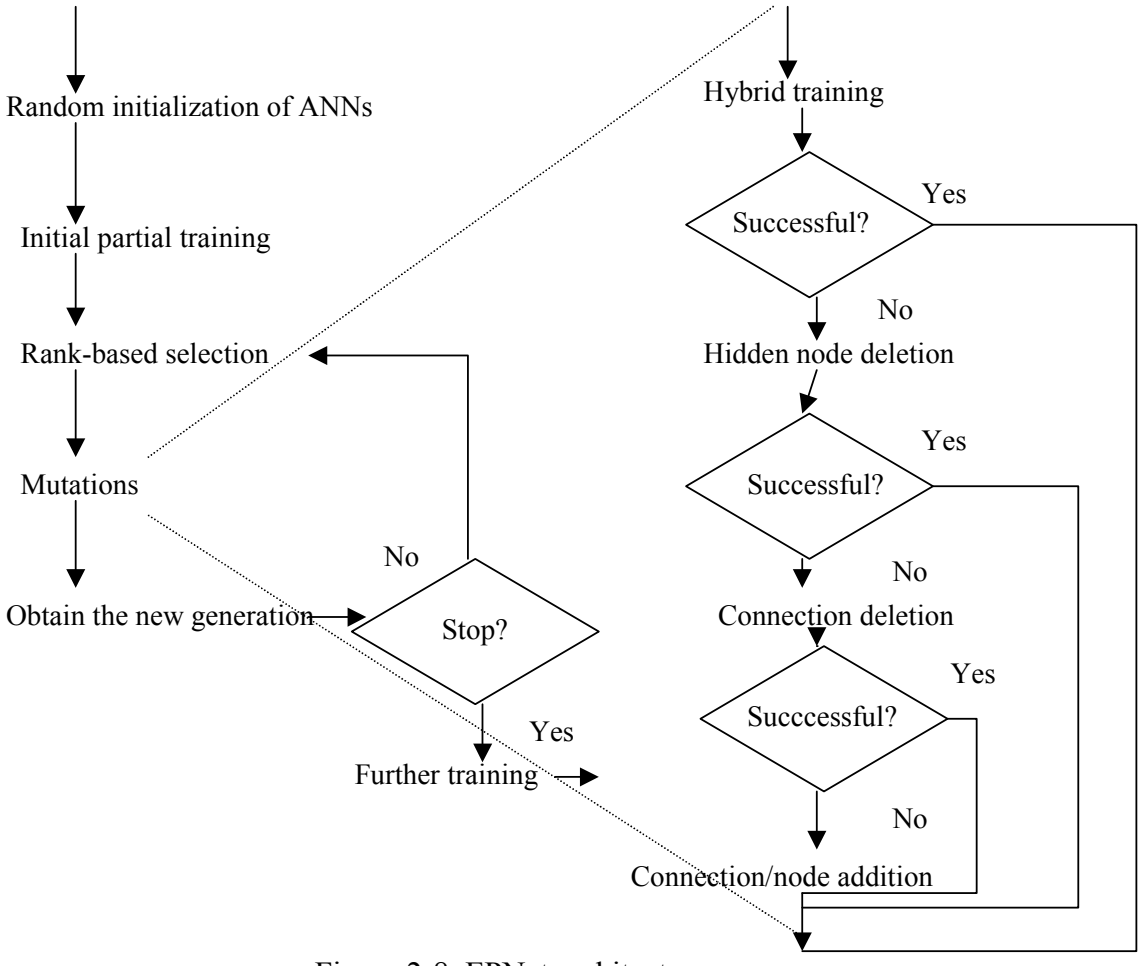

Figure 2-8 EPNet architecture

2.4.7 Hybrid Method

Whereas the evolutionary algorithm was proven useful in finding global optimum solution of a problem, there was a further need an improvement of this approach in terms of the time complexity and to some extent the quality of solution by conducting a fine tuning the search within the local neighborhood area of the global solution obtained by the evolutionary algorithm. Because of this nature of this algorithm, this is known as hybrid. Mainly most of the hybrid algorithm developed for ANN, used GA and some kind of a local search method. Amongst the local search EBP were used most extensively, because of its popular nature in neural learning. Earlier research in this area had shown that hybrid training was successful [10] [11] [106] [107]. There were a number of researchers who used GA and EBP and reported the improvement of the algorithm over traditional GA or EBP [108] [109]. Some recent work also suggested an improvement for hybrid algorithm by running several parallel combination of global and local search [110] [111] [112].

When an algorithm can be called as hybrid, is debatable. In this thesis the learning algorithm has been classified as hybrid if

- The algorithm uses a learning technique using matrix based methods or GA based methods.
- The algorithm uses different learning strategies for different layers
- The algorithm uses a combination of many different learning strategy as a convolution combination strategy or hierarchical combination strategy

2.4.7.1 Layer wise EBP based Kaczmarz algorithm

This is a learning algorithm for a Feed forward Neural Network (FNN) that combines the EBP strategy and optimization layer by layer [113] using Taylor series expansion of nonlinear operator. It constructs the objective function using the Taylor expansion of the nonlinear operator describing a FNN and proposes to

update weights of each layer by the EBP-based iterative Kaczmarz method (KM) [114]. The algorithm can be described by the following steps:

Step1: Initialize weights.
Initialize regularization factor μ.
Set the number of swaps Swp

Step 2: Choose the desired Mean squared error (MSE) value E_{des} and set current MSE value $E_{current}$ as $E_{current} >> E_{des}$.

Step3: Construct MSE Objective function as

$$E(W) = \frac{1}{M}\sum_{k=1}^{M} E_k(W) = \frac{1}{2MN}\sum_{k=1}^{M}\left(D_k(W) - y_k\right)^2 \qquad (2\text{-}91)$$

Step 4: Form the set of linear equations (SLE's) as $Aw_k = b$. Solve the matrix equation and compute the quasi optimal solution $w^{qopt}(k)$, performing Swp of the KM with the relaxation parameter α

Step 5: Form $W^{qopt}(k) = W^0(k) + w^{qopt}(k)$ (2-92)
and compute the MSE as

$$E(W) = \frac{1}{M}\sum_{k=1}^{M} E_k(W) = \frac{1}{2MN}\sum_{k=1}^{M}\left(D_k(W) - y_k\right)^2 \qquad (2\text{-}93)$$

Step 6: If $(E(W^{qopt}(k)) < E_{des})$

Continue with step 9.

Step 7: If $(E(W^{qopt}(k)) < E_{current})$

Set $E_{current} = E(W^{qopt}(k))$

Decrease regularization factor $\mu = \mu * 0.8$

Set $W^0(l-1) = W^{qopt}(l)\, l = L...1$ (2-94)

Go to step 8.

Else

Increase regularization factor $\mu = \mu * 1.2$

Go to step 3.

Step 8: Set $W^0(L) = W^{qopt}(0)$
Go to step 3.

Step 9: Set $W = W^{qopt}(l)$
End the learning process.

2.4.7.2 Global Extended Kalman Filter Algorithm (GEKF)

GEKF algorithm was described by Singhal et. al. in 1989 [115]. This algorithm is based upon the extended Kalman filter and training of a MLP by this algorithm can be viewed as a nonlinear system parametric identification problem. This algorithm is particularly successful as compared to the well known gradient descent method in terms of convergence and quality of the solution. But this algorithm achieves better performance at the expanse of much greater computational and storage requirements, which make the GEKF limited for training ANNs with a relatively large number of nodes.

The algorithm can be described by the following steps:

Step 1: Initialize the weights.

Step 2: While stopping condition is false, do steps 3-5.

Step 3: For input vector x(t) Compute the output of the network as'

$$y(t) = f\left(x(t), \hat{w}(t)\right) \quad (2\text{-}95)$$

where $\hat{w}(t)$ represents the matrix of estimated weights at time t

Step 4: Update the weight via first order EKF recursion

$$K(t+1) = P(t) J_{\hat{w}(t)} + \left(I + P(t) J^{T}_{\hat{w}(t)}\right)^{-1} \quad (2\text{-}96)$$

$$P(t+1) = P(t) - K(t+1) J_{\hat{w}(t)} P(t) \quad (2\text{-}97)$$

$$\hat{w}(t+1) = \hat{w}(t) K(t) E(t) \quad (2\text{-}98)$$

where

P(t) = error covariance matrix;
K(t) = Kalman gain;
$J_{\hat{w}(t-1)}$ = Jacobian of f, the neural network activation function, with respect to the current estimate of the weights.
E(t) = d(t)-y(t)

Step 5: Test stopping condition:
If the error is smaller than a specific tolerance or the number of iteration exceeds a specific value, stop; else continue.

2.4.7.3 Decoupled extended Kalman filter algorithm (DEKF)

The DEFF algorithm [116] partitioned the weight vector w into groups which are considered independent (decoupled), and the global error covariance matrix P is block-diagonalized, each block corresponding to a group of weights. The value of P-matrix outside the block is zero; the global recursion can be split into number of decoupled sub-recursions.

The algorithm can be described by the following steps:

Step 1:	Initialize the weights.
Step 2:	While stopping condition is false, do steps 3-5.
Step 3:	For input vector x(t) Compute the output of the network as' $y(t) = f\left(x(t), \hat{w}(t)\right)$ where $\hat{w}(t)$ represents the matrix of estimated weights at time t
Step 4:	Update the weight via first order EKF recursion $K^j(t+1) = P^j(t) J^j{}_{\hat{w}(t)} + \left(I + P^j(t) J^{jT}_{\hat{w}(t)}\right)^{-1}$ (2-99) $P^j(t+1) = P^j(t) - K^j(t+1) J^j{}_{\hat{w}(t)} P^j(t)$ (2-100) $\hat{w}(t+1) = \hat{w}(t) K^j(t+1) e(t+1)$ (2-101)
Step 5:	Test stopping condition: If the error is smaller than a specific tolerance or the number of iteration exceeds a specific value, stop; else continue.

2.4.7.4 Local linearized least squares algorithm (LLLS)

The local linearized least squares [117] based on viewing the local system identification sub-problems at the neuron level as recursive linearized least squares problems. The objective function of the least squares problem for each neuron is the sum of the squares of the linearized back-propagated error signals.

The algorithm can be described by the following steps:

Step 1:	Initialize the weights.
Step 2:	While stopping condition is false, do steps 3-5.
Step 3:	For input vector x(t) Compute the output of the network as' $y(t) = f\left(x(t), \hat{w}(t)\right)$ where $\hat{w}(t)$ represents the matrix of estimated weights at time t
Step 4:	Update the weight via first order EKF recursion $K_j(t+1) = P_j(t)\tilde{J}_j(t+1) + \left(1 + \tilde{J}_j(t+1) P_j(t) \tilde{J}_j^{\ T}(t+1)\right)^{-1}$ (2-102) $P_j(t+1) = P_j(t) - K_j(t+1)\tilde{J}_j(t+1) P_j(t)$ (2-103) $\hat{w}(t+1) = \hat{w}(t) K_j(t+1) e(t+1)$ (2-104)
Step 5:	Test stopping condition: If the error is smaller than a specific tolerance or the number of iteration exceeds a specific value, stop; else continue.

2.4.7.5 Precise linear sigmoidal network

D.R. Hush et. al. [118] proposed a computationally efficient algorithm for function approximation with precise linear sigmoidal network. A one hidden layer network is constructed one node at a time using well known method of filtering residual. The task of filtering an individual node is accomplished using a new algorithm that searches for the best fit by solving a sequence of quadratic programming problems. This approach offers significant advantage over derivative based approach algorithm (e.g. back propagation and its extensions) .RA MP The algorithm can be described by the following steps:

Step 1: Initialize weights and the parameters α, β

Step 2: For each training pair, **s:t** , do steps 4-6.

Step 3: Set activations of input units
$x_i = s_i; i = 1:n$

Step 4: Compute Z and partition the data into three regions S_-, S_1 and S_+,

$$z_i = W^T x_i \tag{2-105}$$

$$S_- = \{(x_i, t_i) : z_i < \alpha\} \tag{2-106}$$

$$S_1 = \{(x_i, t_i) : \alpha \le z_i \le \beta\} \tag{2-107}$$

$$S_+ = \{(x_i, t_i) : z_i \le \beta\} \tag{2-108}$$

Step 5: Compute R, r, β_- , β_+.

$$R = \frac{\left(\sum_{S_l} \tilde{x}_i \tilde{x}_i^T\right)}{N_l} \tag{2-109}$$

$$r = \frac{\left(\sum_{S_l} \tilde{x}_i t_i\right)}{N_l} \tag{2-110}$$

$$\beta_- = \frac{\left(\sum_{S_-} t_i\right)}{N_-} \tag{2-111}$$

$$\beta_+ = \frac{\left(\sum_{S_+} t_i\right)}{N_+} \tag{2-112}$$

where N_l , N_- and , N_+ represents number of sample in the region S_l , S_- and , S_+ respectively

Step 6: Update weight matrix $\tilde{W}$ as

$$\tilde{W} = R^{-1} r \tag{2-113}$$

Contd..

Step 7: Update α and β

$$\alpha = \frac{(\beta_{-} - w_0)}{\|W\|} \quad (2\text{-}114)$$

$$\beta = \frac{(\beta_{+} - w_0)}{\|W\|} \quad (2\text{-}115)$$

Step 8: Normalize the weight vector

$$W = \frac{W}{\|W\|} \quad (2\text{-}116)$$

Step 9: Repeat Step 2-8 until (W, α, β) converge.

2.4.7.6 Reduced complexity based on Jacobian deficiency

This is an advanced supervised learning method based on Jacobian rank deficiency and it is formulated, in some sense, in the spirit if Gauss-Newton algorithm. G. Zhou in 1998 developed this algorithm [119] improving convergence property while reducing the memory and computational complexity in supervised learning.

The algorithm can be described by the following steps:

Step 1: Initialize weights.

Step 2: For each training pair, **s:t** , do steps 4-7.

Step 3: Set activations of input units

$$x_i = s_i; i = 1:n$$

Step 4: Compute response of each unit

$$netOutB_j = b_j + \sum_{i=1}^{n} x_i w_{ij}$$

$$y_j = \frac{1}{1 + \exp^{-netOutB_j}}$$

Contd..

Step 5: Calculate Error vector E as

$$E(W)=\frac{1}{2}e(W)^T e(W) \tag{2-117}$$

where
$e=[e_1,.....,e_m]$ and $e_i=y_i-t_i$

Step 6: If $E\le\varepsilon$ then stop; else continue.

Step 7: Backpropagate the signal and calculate local gradients for each node

$$\partial_j = y_j*(1-y_j)*\sum_k \partial_k w_{jk}$$

$$\partial_k = y_k*(1-y_k)*(t_k-y_k)$$

Step 8: If $\left\|J^T e\right\|\le\varepsilon$ then stop,
else continue.

Step 9: Factorized $J^T J$ and $J^T e$ into D,L and b, respectively row by row until $d_p \le \rho_1 d_1$, where ρ_1 is a tolerance coefficient

Step 10: Compute Δw_1 as follows

$$\Delta\tilde{w}_1=-L_1^{-1}\left(D_1+\mu I_1\right)b_1 \tag{2-118}$$

$$\Delta\tilde{w}_2=0$$

where $\mu>0$

Step 11: Compute $E\left(W+P_1\left[\Delta\tilde{w}_1,0\right]^T\right)$ by the formula

$$E(W)=\frac{1}{2}e(W)^T e(W)$$

Step 12: If $E\left(W+P_1\left[\Delta\tilde{w}_1,0\right]^T\right) < E(W)-\partial$

then $\Delta\tilde{w}_1$ is accepted , go to step 13.
Else
$\mu=\mu\gamma$, where $\gamma>0$

Contd..

If $\mu \le \mu_{\max}$ repeat step 12.

If $\mu > \mu_{\max}$ restart the algorithm.

Step 13: If $\Delta \tilde{w}_2$ is to be used as necessary, execute step 15-16 else go to step 16.

Step 14: Decompose $D_1 L_2$ and $\mu\left(D_1 + \mu I_1\right) b_1 D_1$ into R and h respectively, column by column until $\left|r_{pp}\right| \le \rho_2 \left|r_{11}\right|$

Step 15: Compute $\Delta \tilde{w}_2$ as follows

$$\Delta \tilde{w}_{21} = -R_1^{-1}\left(D_2 + \nu I_2\right) D_2 h_1 \qquad (2\text{-}119)$$

$$\Delta \tilde{w}_{21} = 0 \qquad (2\text{-}120)$$

Step 16: If $E\left(W + P_1\left[\Delta \tilde{w}_1, P_2 \Delta \tilde{w}_2\right]^T\right) < E\left(W + P_1\left[\Delta \tilde{w}_1, 0\right]^T\right) - \partial$

then $\Delta \tilde{w}_2$ is accepted

$\nu = \dfrac{\nu}{\gamma}$

go to step 13.

Else

$\nu = \nu\gamma$

If $\nu \le \nu_{\max}$

repeat step 16.

If $\nu > \nu_{\max}$

$\Delta \tilde{w}_2 = 0$

Go to Step 17.

Step 17: Update weights

$$W\left(t+1\right) = W\left(t\right) + P_1 \Delta \tilde{w} \qquad (2\text{-}121)$$

$$\mu = \frac{\mu}{\gamma} \qquad (2\text{-}122)$$

Go to step 2.

2.4.7.7 Linearly constrained least square method (LCLS)

The LCLS method [120] uses only constrained minimum sample variance of fused information and thus the proposed fusion method can tackle the unknown covariance problem. Some neural network problems have been proposed for data fusion in [121], [122], where neural network approaches are difficult to obtain a good fusion solution. The propose method is easy to implement and converges quickly to the optimal solution due to its global exponential convergence and stability. The algorithm can be described by the following steps:

Step 1: Initialize weights.

Step 2: Set activations of input units

$$x_i = a_i s_i + GN; i = 1:n \qquad (2\text{-}123)$$

where a_i is a scaling coefficient and GN is the additive Gaussian noise. In vector notation

$\mathbf{X}(t)=\mathbf{a}\,\mathbf{s}(t) + \mathbf{GN}(t)$

Step 3: Compute response vector **Z** as follows

$$z(t) = w^T(t)x(t) = w^T(t)as(t) + w^T(t)GN(t) \qquad (2\text{-}124)$$

Step 4: Update the weight matrix as follows

$$\begin{pmatrix} w(t+1) \\ y(t+1) \end{pmatrix} = \begin{pmatrix} w(t) \\ y(t) \end{pmatrix} - h \begin{pmatrix} W_1 w(t) + W_2 y(t) - a \\ W_3 w(t) + a^T a y(t) \end{pmatrix} \qquad (2\text{-}125)$$

where $\begin{Bmatrix} W_1 = 4\alpha^2 R^T R + aa^T \\ W_2 = 2\alpha R a \\ W_3 = 2\alpha a^T R \end{Bmatrix}$,

h > 0 is a fixed step size and

α is a scale parameter such that $\|\alpha R\|_\infty \le 1$ and R is the covariance matrix.

Step 5: Compute error as $E\left[\left(z(t) - s(t)\right)^2\right]$

Step 6: Test stopping condition:
If the error is smaller than a specific tolerance or the number of iteration exceeds a specific value, stop; else continue.

2.4.7.8 Regularizer for recursive least squared algorithm

Recursive least square (RLS) [123] [124] based algorithms has an implicit weight decay term in its energy function but the weight decay term decrease linearly as the number of learning epochs increases, thus r rendering a diminishing weight decay effect as training progresses. C. Leung et. al. in 2001 [125] proposed two modified RLS algorithm to tackle this problem. In the first algorithm namely the true weight decay RLS (TWRLS) algorithm, a modified energy function is considered where by the weight decay effect remains constant, irrespective of number of learning epochs. The second version, the input perturbation RLS (IPRLS) algorithm, is derived by requiring robustness in its prediction performance to input perturbations. TWRLS algorithm can be described by the following steps:

Step 1:	Initialize the weights.
Step 2:	While stopping condition is false, do steps 3-5.
Step 3:	For input vector x(t) Compute the output of the network as $y(t) = f\left(x(t), \hat{w}(t)\right)$ where $\hat{w}(t)$ represents the matrix of estimated weights at time t
Step 4:	Update the weight via first order EKF recursion

$$K^*(t+1) = \alpha P(t)\left(I + \alpha P(t)\right)^{-1} \quad (2\text{-}126)$$

$$P^*(t+1) = P(t) - K^*(t+1)P(t) \quad (2\text{-}127)$$

$$K(t+1) = P^*(t+1)H(t+1)\left(I + H^T(t+1)P^*(t+1)H(t+1)\right)^{-1} \quad (2\text{-}128)$$

$$\hat{w}(t+1) = \hat{w}(t) - K^*(t+1)\hat{w}(t) + K(t+1)H^T(t+1)K^*(t+1)\hat{w}(t) + K(t+1)\left(T(t) - f\left(\hat{w}(t), x(t+1)\right)\right) \quad (2\text{-}129)$$

$$P(t+1) = P^*(t+1) - K(t+1)H^T(t+1)P^*(t+1) \quad (2\text{-}130)$$

where H is a N * hid matrix that is obtained by linearizing f(w,x) around the estimate of w i.e. $\hat{w}$

Step 5:	Test stopping condition: If the error is smaller than a specific tolerance or the number of iteration exceeds a specific value, stop; else continue.

IPRLS algorithm can be described by the following steps:

Step 1: Initialize the weights.

Step 2: While stopping condition is false, do steps 3-5.

Step 3: For input vector x(t) Compute the output of the network as
$y(t) = f\left(x(t), \hat{w}(t)\right)$

where $\hat{w}(t)$ represents the matrix of estimated weights at time t

Step 4: Update the weight via first order EKF recursion

$$K^*(t+1) = P(t)G(t+1)\left(I + G^T(t+1)P^*(t+1)G(t+1)\right)^{-1} \quad (2\text{-}131)$$

$$P^*(t+1) = P(t) - K^*(t+1)G^T(t+1)P(t) \quad (2\text{-}132)$$

$$K(t+1) = P^*(t+1)H(t+1)\left(I + H^T(t+1)P^*(t+1)H(t+1)\right)^{-1} \quad (2\text{-}133)$$

$$\begin{aligned}\hat{w}(t+1) = {} & \hat{w}(t) - K^*(t+1)G^T(t+1)\hat{w}(t) + \\ & K(t+1)H^T(t+1)K^*(t+1)G^T(t+1)\hat{w}(t) \\ & + K(t+1)\left(T(t) - f\left(\hat{w}(t), x(t+1)\right)\right)\end{aligned} \quad (2\text{-}134)$$

$$P(t+1) = P^*(t+1) - K(t+1)H^T(t+1)P^*(t+1) \quad (2\text{-}135)$$

where H is a N * hid matrix that is obtained by linearizing f(w,x) around the estimate of w i.e. $\hat{w}$ and

$$G(t+1) = \sigma\left[\Delta H_1(t+1) \mid \Delta H_2(t+1) \mid \ldots\ldots \mid \Delta H_N(t+1)\right] \quad (2\text{-}136)$$

where σ is variance

Step 5: Test stopping condition:
If the error is smaller than a specific tolerance or the number of iteration exceeds a specific value, stop; else continue.

2.4.8 Problems still unresolved

There are many problems associated with the current existing learning algorithms. Some of the aspects with existing learning methods for MLP can be summarized as

- The architecture selection is very costly. It needs expensive trail and error method to find the optimum number of hidden units.
- The convergence tends to be extremely slow.
- Learning parameters must be guessed heuristically.
- Convergence to the global minimum is not guaranteed.
- A network with more parameters than the number of data points available should not be used [126].

2.4.8.1 Local minima

The most well known difficulty that arises in general optimization problems is the issue of local minima. Univariate problem was the main concern of the mathematical programming and optimization researchers[127]. The local minima problem is described in Figure 2-9. In the one-dimensional case, the concept of local minima follows closely from the issue of convexity. If there are no local minima, then the optimization problem is trivial, and the error/cost function resembles a parabola. In case of ANN the issue of local minima is very important, as most of the classification problems are non-linear optimization with and without NP hard properties. Research indicates that empirical MLP error surfaces have an extreme ratio of saddle points to local minima [128] [129].

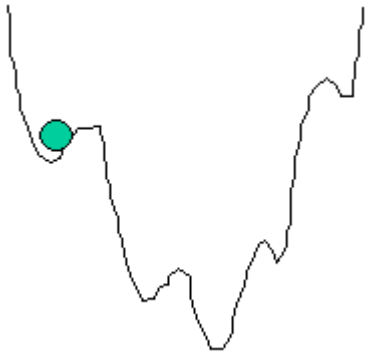

Figure 2-9 Local minima problem

2.4.8.2 Computation of gradients

The results and experience of research into the properties of the error surface have identified an important feature of MLP error surfaces, which has implications for successful training of the net. The presence of relatively steep and flat regions is a fundamental feature of the error surface. Properties of the second derivatives (Hessian matrix) of the error surface determine its relative steepness or flatness. Because of the complexity of the surface it is sometimes very hard and costly to compute the derivative for that. These are also known as the ill conditioning of the error surface. Algorithms that do not use gradient information directly will be affected implicitly through their reliance on the values of the error function. Algorithms such as Quasi-Newton (QN) and Levenberg-Marquardt, which use second order information, may not converge much faster than gradient methods in such a situation, and due to their increased computation effort may actually result in slower execution times. When this ratio is ``large", the Hessian/error surface is said to be ill conditioned [130]. In real applications, the error surface for which the gradient information is required is much more complex and many more dimensional than the following graph (Figure 2–10).

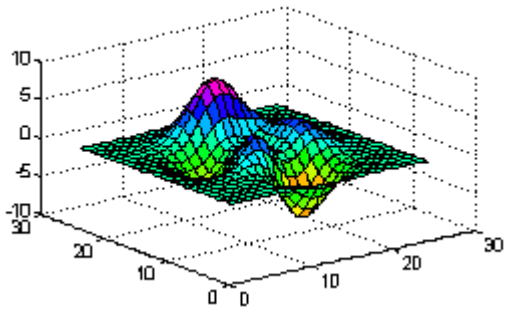

Figure 2-10 Error surface considering only one-weight and bias vectors

2.4.8.3 Paralysis problem

One of the problems with some of the existing learning algorithm for multi layered perceptron is known as paralysis. It occurs when the weights of the network become very high and hence, no change can occur during the learning process. So, the RMS error does not get change. So, one of the important aspect of a learning process is to check the network weights, so it does not become too high to be changed [1] [4] [5].

2.4.8.4 Generalization and convergence problem

ANN builds models of data by capturing the most important features during the training period. But sometimes because of the over complexity of the ANN model, it is possible that ANN can react on noise in the data. In statistics this is known as "statistical inference", In ANN term it is called "data fitting". Figures 2-11 and 2-12 show the training set results and corresponding validation check for a function approximator. The critical issue in developing a neural network is this generalization: how well will the network make classification of patterns that are not in the training set? Neural networks, like other flexible nonlinear estimation methods such as kernel regression and smoothing splines, can suffer from either under fitting or over fitting. Some of the techniques that have been proposed to solve this problem are jittering, weight decay, early stopping, Bayesian estimation etc.

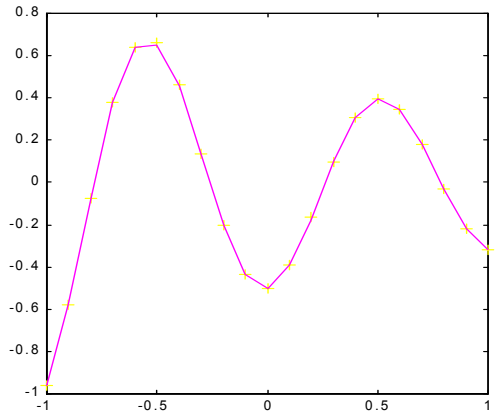

Figure 2-11 Training set result for a function approaximator

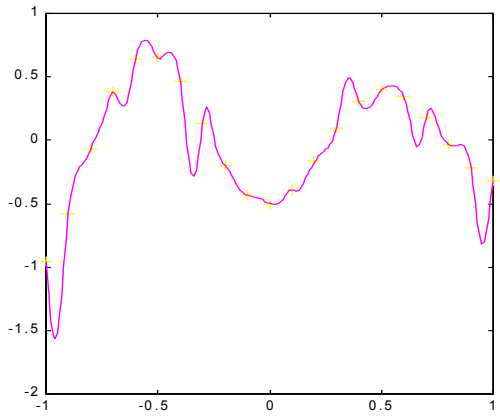

Figure 2-12 Validation set result by the same network

The earliest report by Martin and Pitmann for the generalization problem for MLP can be quoted as "We find only marginal and inconsistent indication that constraining net capacity improves generalization" [131]. There were also some views that with the excess capacity of the net can generalize well, when it uses back propagation learning and early stopping. [132].

Another way to view the generalization is the worst-case analysis to compute the bound on the generalization error. The difference between the estimate and the actual generalization can be bounded, and by increasing the number of training samples this bound can be very small. This was shown by Vapnik and Cervonenkis. They found that a useful bound can only be established, when the number of training samples exceeds a parameter called Vapnik-Chervonenkis dimension (Vcdim) [133].

2.4.8.5 Start state

Another problem with most of the learning algorithm is that the place at which one starts on the error surface, determines whether or not a good or the best solution is found. The initial state is determined by the initial weight values of the neurons, which are determined randomly most of the time. Thus a wrong starting point can no only take a very long time for convergence but also can give a very poor solution.

3 RESEARCH METHODOLOGY

The proposed novel algorithm consists of mainly two parts – finding the weights and finding the architecture (in terms of finding number of hidden neurons). The two modules are called findArchitecture (this module finds the number of hidden neurons) and findWeight (this modules finds the weight values using the hybrid method) respectively. The two modules are combined using a hierarchical structure, where a feedback mechanism exists for both the modules. In Figure 3-1, the flowchart for combining these two modules is given.

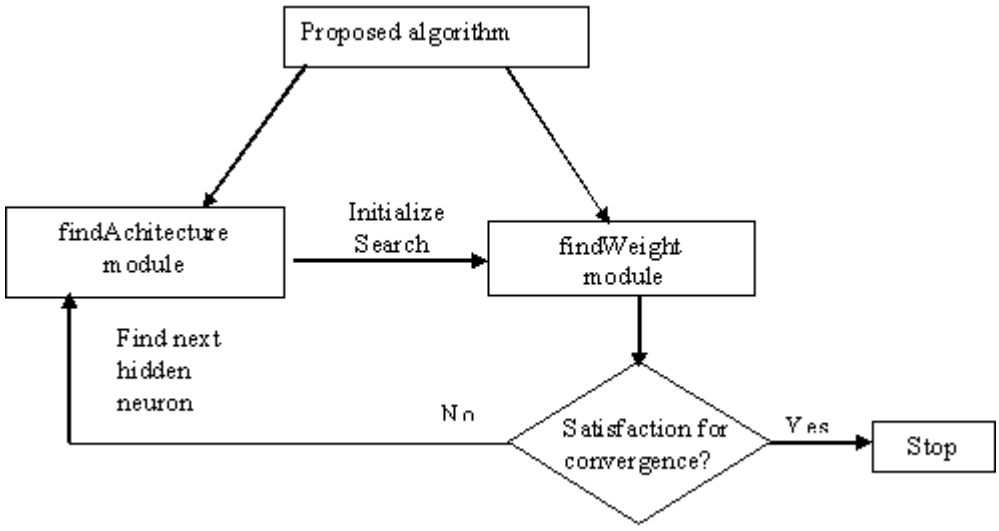

Figure 3-1 Combination dynamics for findArchitecture and findWeight modules

The findArchitecture module and detail of the flowchart (joining rules for the two modules) are discussed after the discussion of the findWeight module in details. In the following section detail of findWeight module is given.

3.1 MODULE DETAILS FOR FINDING WEIGHTS

The proposed findWeight module is mainly a hybrid approach, which uses genetic algorithm and the least square method. Hence there are two common

issues that need to be discussed, one is the ANN architecture where the proposed algorithm is applied, and because it is a hybrid method that uses two different techniques, the rules joining these two techniques are important. Because of this many possibilities due to the variations that exist in applying such a hybrid method, all possibilities of combinations for variations for the new proposed algorithm are studied. Finally based on the improvement of the results obtained from them, all the varied methods are ranked in a stepwise improvement before the final algorithm is given.

3.1.1 Common architecture details

A two-layer network architecture is considered. The input nodes for the ANN do the range compression for the input and transmit output to all the nodes in the hidden layer. The activation function used for all the nodes in hidden and output layer is sigmoidal. The first layer is composed of input nodes, which simply range compress the applied input (based on pre-specified range limits) so that it is in the open range interval (0, 1) and fan out the result to all nodes in the second layer. The hidden nodes perform a weighted sum on its input, and then pass that through the sigmoidal activation function before sending the result to the next layer. The output layers also perform the same weighted sum operation on its input and then pass that through the sigmoidal activation function to give the final result.

The weight variables for each layer are found using a hybrid method, which uses the genetic algorithm (GA) and a least square method. The architecture is shown in Figure 3–2. The genetic algorithm is applied for the first layer weight and the least square method is applied to find the weights for the output layer. We initialize the hidden layer weights with a uniform distribution with closed range interval [-1, +1].

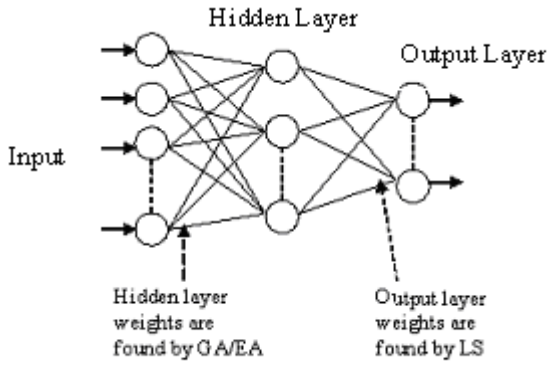

Figure 3-2 A two layer ANN architecture for the proposed hybrid learning method

3.1.2 Combination dynamics

The combination of the genetic algorithm and the least square method provides many possibilities of joining the two different methods [134] [135] [136] [137]. The rules of using the hybrid method can vary not only because there exists different variations of techniques that are suitable in general or may be specific to a particular type of problem, but also because the dynamics of combining the two can vary on the time of joining them. Depending on the different genetic algorithm strategies – two variations of genetic algorithm that can be applied for searching for weights for ANN are investigated. The first strategy is the primitive or straight GA which is applied to ANN phenotype using a direct encoding scheme. This GA methodology uses the standard two point crossover or interpolation as recombination operator and Gaussian noise addition as mutation operator. A slight variations of the standard GA that is used for testing called evolutionary algorithm, where the only genetic operation to be considered is mutation. Also, mutation is applied using a variance operator. The Table 3-1 shows the two different genetic algorithm strategies that are used in investigation of various combinations.

Table 3-1 Variations of strategies depending on the GA methods

Genetic Algorithm Strategy	Description
Genetic Algorithm (GA)	Uses the natural GA form using direct encoding schemes. Genetic operators are crossover and mutation with certain probabilities[1].
Evolutionary Algorithm (EA)	Uses the evolutionary form. Uses variance operator as a vector term for performing mutation. No crossover operation is used.

The stepwise operations for GA / EA are described next for a clear explanation of the two different genetic algorithm methodologies. The standard GA method can be described as follows:

Step 1: Initialize all the hidden layer weights using a uniform distribution of a closed interval range of [-1, +1]. A sample genotype for the lower half gene from the population pool for *n* input, *h* hidden units, *m* output, and *p* number of patterns can be written as

$$\left| w_{11} w_{12} \ldots w_{1n} w_{21} w_{22} \ldots w_{2n} \ldots w_{h1} w_{h2} \ldots w_{hn} \right| \quad (3\text{-}1)$$

Where, range(w) initially is set between the closed interval [-1 +1]. Also, the sample genotype will vary depending on the connection type as described later.

Step 2: The fitness for the population is calculated based on the phenotype and the target for the ANN.

$netOutput = f(hid * weight)$

Where **hid** is the output matrix from the hidden layer neurons, **weight** is the weight matrix output neurons and f is the sigmoid function

$$RMSError = \sqrt{\frac{\sum_{i=1}^{n} (netOutput - net)^2}{n * p}} \quad (3\text{-}2)$$

Contd..

[1] The mutation and crossover probability values will be analysed in the experimental results chapter.

$$popRMSError_i = norm(RMSError_i) \quad (3\text{-}3)$$

norm function normalized the fitness of the individual, so the fitness of each individual population is forced to be within certain range.

Step 3: Generate a random number x from a Gaussian distribution of mean 0 and standard deviation 1.

If (x < *crossOverRate*)
Select two parents from the intermediate population
ApplyCrossOver
End If

Generate another random number y from the same distribution

If (y < *mutationRate*)
ApplyMutation
End If

The crossover method that is used for this algorithm is known as linear interpolation with two points technique. Let's consider two genes $w_1 w_2 .. w_h$ and $w_1^{/} w_2^{/} ... w_h^{/}$

Two points are selected randomly, lets assume *point1* and *point2*, where *point1*<*point2*, and *point1*>1, *point2*<*h*

The two child after the crossover operation will be

$$\frac{2w_1 + w_1^{/}}{3} \frac{2w_2 + w_2^{/}}{3} \ldots \frac{2w_{point1} + w_{point1}^{/}}{3} \frac{w_{point1+1} + 2w_{point1+1}^{/}}{3} \ldots \frac{w_{point2\text{-}1} + 2w_{point2\text{-}1}^{/}}{3} \frac{2w_{point2} + w_{point2}^{/}}{3} \ldots \frac{2w_h + w_h^{/}}{3} \quad (3\text{-}4)$$

$$\frac{w_1 + 2w_1^{/}}{3} \frac{w_2 + 2w_2^{/}}{3} \ldots \frac{w_{point1} + 2w_{point1}^{/}}{3} \frac{2w_{point1+1} + w_{point1+1}^{/}}{3} \ldots \frac{2w_{point2-1} + w_{point2-1}^{/}}{3} \frac{w_{point2} + 2w_{point2}^{/}}{3} \ldots \frac{w_h + 2w_h^{/}}{3} \quad (3\text{-}5)$$

For mutation, a small random value between 0.1 and 0.2, is added to all the weights. Let us assume a parent string

$w_1 w_2 .. w_h$

After mutation the child string becomes

$$w_1 + \varepsilon \mid w_2 + \varepsilon \mid .. \mid w_h + \varepsilon \quad (3\text{-}6)$$

Where ε is a small random number [0.1 0.2], generated using a uniform distribution.

Step 4: If the convergence[2] for the GA is not satisfied then goto step 2.

[2] The convergence property for the GA will be described later

The evolutionary algorithm method (EA) can be described as follows:

Step 1: Initialize all the hidden layer weights using a uniform distribution of a closed interval range of [-1, +1]. A sample genotype for the lower half gene from the population pool for *n* input, *h* hidden units, *m* output, and *p* number of patterns can be written as

$$\left|w_{11}\mu_{11}w_{12}\mu_{12}..w_{1n}\mu_{1n}w_{21}\mu_{21}w_{22}\mu_{22}..w_{2n}\mu_{2n}..w_{h1}\mu_{h1}w_{h2}\mu_{h2}..w_{hn}\mu_{hn}\right| \quad (3\text{-}7)$$

Where, range(w) initially is set between the closed interval [-1 +1]. μ*'s* are the variance vectors, each values of μ is initialized by a Gaussian distribution of mean 0 and standard deviation 1. Also, the sample genotype will vary depending on the connection type as described later.

Step 2: The fitness for the population is calculated based on the phenotype and the target for the ANN.

$$netOutput = f(hid * weight)$$

where f is the sigmoid function

$$RMSError = \sqrt{\frac{\sum_{i=1}^{n}(netOutput - net)^2}{n * p}}$$

$$popRMSError_i = norm(RMSError_i)$$

norm function normalized the fitness of the individual, so the fitness of each individual population is forced to be within certain range.

Step 3: Each individual population vector $(\mathbf{w_i}, \boldsymbol{\eta_i})$, $i = 1, 2, \ldots, \mu$ creates a single offspring vector $(\mathbf{w_i'}, \boldsymbol{\eta_i'})$

for $j = 1, 2, \ldots, n$

$$\eta_i'(j) = \eta_i(j)\exp(\tau' N(0,1) + \tau Nj(0,1)) \quad (3\text{-}8)$$

$$w_i'(j) = w_i(j) + \eta_i'(j)\, Nj(0,1) \quad (3\text{-}9)$$

where $w_i(j)$, $w_i'(j)$, $\eta_i(j)$, and $\eta_i'(j)$ denote the *j*th component of the vectors $\mathbf{w_i}$, $\mathbf{w_i'}$, $\boldsymbol{\eta_i}$, and $\boldsymbol{\eta_i'}$, respectively. N(0,1) denotes a normally distributed one-dimensional random number with mean and variance of 0 and 1 respectively. Nj(0,1) denotes that the random number is generated a new for each value of *j*.

The parameter τ and τ' are set to $\left(\sqrt{2\sqrt{n}}\right)^{-1}$ and $\left(\sqrt{2n}\right)^{-1}$

Step 4: If the convergence[3] for the GA is not satisfied then goto step 2.

[3] The convergence property for the EA is described later

Depending on the connectivity for combining the GA/EA and LS method, three different connection topologies are devised. The variations of the methods are mainly due to the connection time for calling the least square method from the GA / EA. The three topologies are shown in Figure 3-3. On the basis of the time complexities for the three connections the three variations are known as T1, T2 and T3 on a descending order. The two independent modules of GA/EA and LS are connected together at some point for generating the solution for the weight matrix.

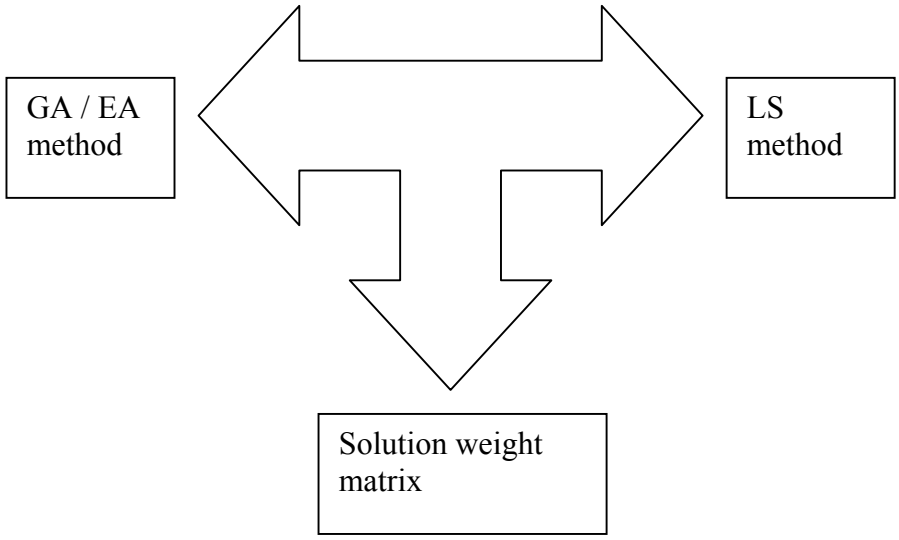

Figure 3-3 A general T (1/2/3) connection for GA and LA methods

The Table 3-2 shows a brief description of the three connection strategies before giving their individual details.

Table 3-2 Three different connection strategies for calling LS from GA/EA

Connection Strategy	Description
T1	The GA/EA and the LS method are called for every generation and every population to find the corresponding fitness value of the population.
T2	The LS method is called after one generation p.p for GA/EA method. The

	generation run for GA/EA method. The best fitness population is halved and the upper half is saved as the weights for output layers. Afterwards, the GA/EA is run for the remaining generation.
T3	The LS method is called after the convergence of the GA/EA is over. After certain number of generations for the GA/EA, the best fitness population is halved and the lower half is used as the weights for the hidden layer and those weights are used for the LS method.

The three connection methodologies are explained in simple flowcharts in the following section.

3.1.2.1 T1 connection

In T1 connection the LS method is called for calculating the fitness for every population and for every generation. The fitness is calculated using the weight represented by the population genome and the LS output of the output layer weights. The flowchart is given in Figure 3–4.

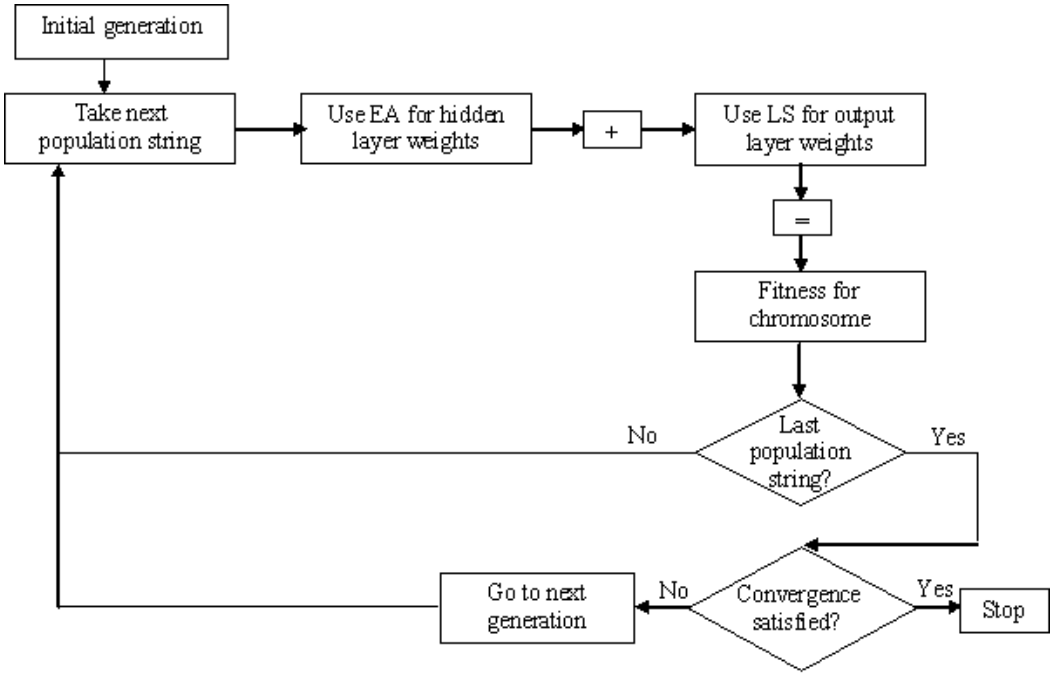

Figure 3-4 T1- connection architecture

3.1.2.2 T2 connection

In T2 connection the LS method is called for calculating the fitness for every population and only after the first generation. The fitness is calculated using the weight represented by the population genome and the LS output of the output layer weights. The flowchart is given in Figure 3-5.

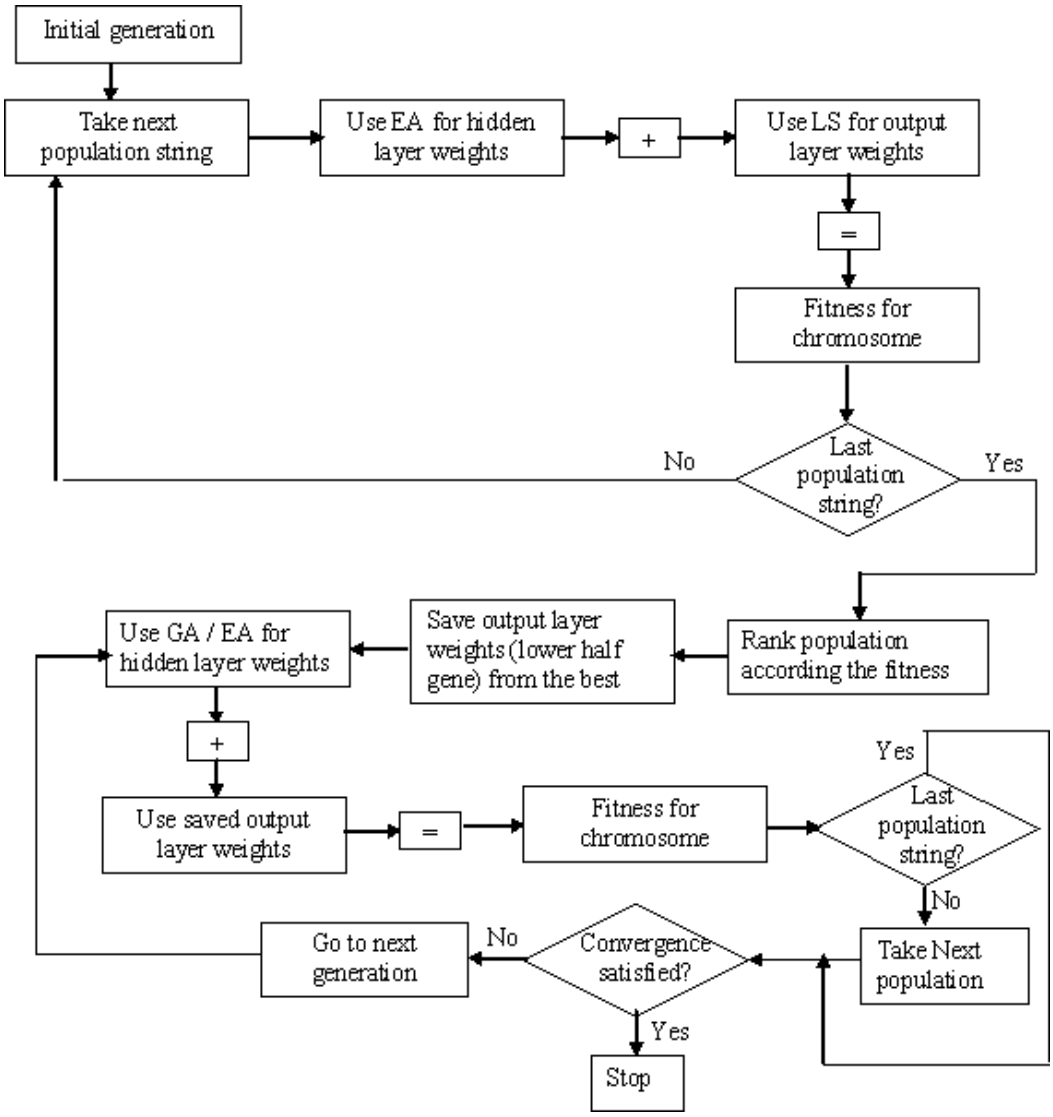

Figure 3-5 T2-connection architecture

3.1.2.3 T3 connection

In T3 connection the LS method is only after the stopping criteria of the GA / EA method is satisfied. The fitness is calculated using the weight represented by breaking the best population genome into half and combining the first half for the hidden layer and the LS output of the output layer weights. The flowchart is given in Figure 3-6.

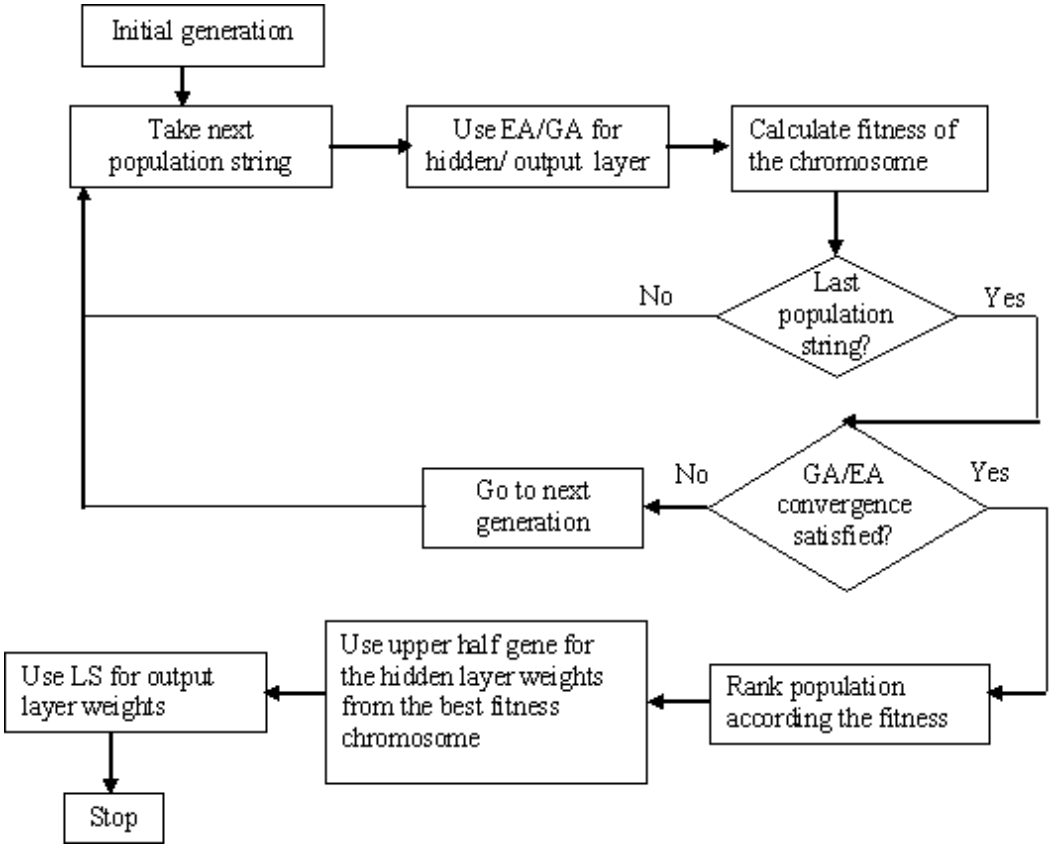

Figure 3-6 T3-connection architecture

3.1.3 Least square method

The weights for the output layer is computed using the linear least square method where the output of the hidden layer is computed as f(·) for the weighted sum of its input, where f is the sigmoid function. The output of each of the hidden neuron can be found using the equations:

$$h = f\left(\sum_{j=1}^{n} w_{ij} I_j\right), \tag{3-10}$$

where $i = 1,2,\ldots,h$, I is the mapped input or the normalized input and

$$f = \frac{1}{1+\exp^{-x}}, \tag{3-11}$$

where x is the output of the hidden neuron before the activation function.

After obtaining the corresponding weight gene from the genotype, as we use sigmoid activation function for the output also, there is a need to do the linearization [43], using the formula

$$netb_i = -\log(\frac{1-net_i}{net_i}), \tag{3-12}$$

where i=1,2,…,m, $netb_i$ is the output of the ith output neuron before the activation, and net_i is the output of the ith output neuron after the activation.

It is now required to solve the over determined equation

$$hid * weight = netb, \tag{3-13}$$

Least square method is used to solve this, which is based on the QR factorization technique to solve the equation for the weight matrix using the *qr* function

$$[Q, R] = qr(hid). \tag{3-14}$$

The function *qr* returns the orthogonal triangular decomposition of the **hid** matrix. It produces an upper triangular matrix **R** of the same dimension as **hid** and a unitary matrix **Q** so that **hid** = **Q***__R__. The mathematical foundation of the algorithm is as follows:

The set of equations **Ax**=**b** by calculating the QR factorization of **A** and solving first **Qy**=**b** (hence **y** = $\mathbf{Q}^{\mathrm{T}}\mathbf{b}$) and then **Rx**=**y** (a triangular system).

Let

$$\begin{aligned} &u \in R^m \setminus \{0\} \\ &H = I - 2\frac{uu^T}{\|u\|^2} \end{aligned} \tag{3-15}$$

is called a Householder transformation or a Householder reflection. Since **Hu** = -**u**, **Hv** = **v** if $\mathbf{u}^{\mathrm{T}}\mathbf{v}$ = 0,

This transformation reflects any vector **x** ∈ $\mathbf{R}^{\mathrm{n}}$ in the (n-1) dimensional hyper plane spanned by the vectors orthogonal to **u**. Each such matrix **H** is symmetric and orthogonal. The later follows because reflection leaves the Euclidian distance invariant, or by direct calculation

$$\left(I-2\frac{uu^T}{\|u\|^2}\right)^T\left(I-2\frac{uu^T}{\|u\|^2}\right)=\left(I-2\frac{uu^T}{\|u\|^2}\right)^2=I-4\frac{uu^T}{\|u\|^2}+4\frac{u(u^Tu)u^T}{\|u\|^4}=I. \tag{3-16}$$

Let $\mathbf{a} \in \mathbf{R}^m$ be the first nonzero column of **A**. We wish to choose $\mathbf{u} \in \mathbf{R}^m$

$$\left(I-2\frac{uu^T}{\|u\|^2}\right)a=a-2\frac{u^Ta}{\|u\|^2}u \tag{3-17}$$

and, we normalize **u** so that

$$2u^Ta=\|u\|^2 \tag{3-18}$$

(and $a \neq 0$)

Therefore $u_i = a_i, i = 2,\ldots,m$ and the normalization implies that

$$\begin{aligned} &2u_1a_1+2\sum_{i=2}^{m}a_i^2=u_i^2+\sum_{i=2}^{m}a_i^2 \Rightarrow \\ &u_i^2-2u_1a_1+a_i^2-\sum_{i=1}^{m}a_i^2=0 \Rightarrow . \\ &u_1=a_1\pm\|a\| \end{aligned} \tag{3-19}$$

It is usual to let the sign be the same as the sign of a_1, since $\|u\| << 1$ might lead to a division by a tiny number, hence to numerical difficulties. For large *m* there is no need to execute explicit matrix multiplication. Instead, it is required

$$\left(I-2\frac{uu^T}{\|u\|^2}\right)A=A-2\frac{u(u^TA)}{\|u\|^2} \tag{3-20}$$

to evaluate

$$\mathbf{w}^T=\mathbf{u}^T\mathbf{A} \tag{3-21}$$

subsequently forming

$$A - \frac{2}{\|u\|^2} uw^T . \tag{3-22}$$

Supposing that the Householder transformation has been applied *k*-1 times, so that the first *k*-1 columns of the resultant matrix **A** have an upper triangular form.

Columns for **A** are processed in a sequence, in each stage pre multiplying a current **A** by the requisite Householder transformation. The end result is an upper triangular matrix **R**. To determine **Q**, we set **Ω** = **I** initially, and for each successive reflection, we replace **Ω** by

$$\left(I - 2\frac{uu^T}{\|u\|^2} \right)\Omega = \Omega - \frac{2}{\|u\|^2} u\left(u^T \Omega\right) \tag{3-23}$$

As in the case of given rotation, by the end of the computation, $\mathbf{Q} = \mathbf{\Omega}^T$. However if we require just vector $\mathbf{c} = \mathbf{Q}^T\mathbf{b}$, rather than matrix **Q**, then we can set initially **c** = **b** and in each stage we replace **c** by

$$\left(I - 2\frac{uu^T}{\|u\|^2} \right)c = c - \frac{2u^T c}{\|u\|^2} u \tag{3-24}$$

If **A** is dense, it is in general more convenient to use Householder reflections. In extreme case, if an *n* X *n* matrix **A** consists of zeros underneath the first sub diagonal, they can be rotated away in just *n*-1 rotations at the cost of $0(n^2)$ operations.

The solution matrix can be found from the **R** matrix using one step iterative process as

$$x = \frac{R}{R^T / (hid^T * netb)} . \tag{3-25}$$

The error **e** can be calculated as

$$r = netb - hid * x \tag{3-26}$$

$$e = \frac{R}{R^T / (hid^T * r)}.$$

The final value of solution for weight matrix can be then found as

$$weight = x + e. \tag{3-27}$$

3.1.4 Efficiency order for the connection topology

Based on the performance of the genetic algorithm strategy the chronological order for all the combination according to their time and memory complexities are GALS (GALS–T1, GALS-T2, GALS-T3) and EALS (EALS-T1, EALS-T2, EALS-T3) respectively. As all the individual subparts for all the combination methodologies are discussed earlier, only the stepwise algorithm for overall EALS-T3 algorithm is discussed in the following section.

EALS-T3 can be described as follows:

Step 1:	*Initialize the input range:* All the inputs are mapped into a range of the open interval (0,1). The method of normalization is based on calculating the mean and standard deviation for each element column and uses these to perform additional scaling if required.
Step 2:	*Start with some number of hidden neurons:* We start the training process using number of hidden neurons obtained from the **findArchitecture** module.
Step 3:	*Initialize all the weights for the hidden layer:* We initialize all the hidden layer weights using a uniform distribution of a closed interval range of [-1, +1]. We also encode the genotype, which represents the weights for the hidden layer with those values for all the population strings. A sample genotype for the lower half gene from the population pool for an n input, h hidden and m output neurons can be written as.
	$\left\|w_{11}\mu_{11} w_{12}\mu_{12} .. w_{1n}\mu_{1n} w_{21}\mu_{21} w_{22}\mu_{22} .. w_{2n}\mu_{2n} .. w_{h1}\mu_{h1} w_{h2}\mu_{h2} .. w_{hn}\mu_{hn}\right\|.$ (3-28)
	Where, range(w) initially is set between the closed interval [-1 +1] μ are the variance vectors, each values of μ is initialized by a Gaussian distribution of mean 0 and standard deviation 1.
	Contd..

Step 4: *Apply Evolutionary algorithm:* We create an intermediate population from the current population using a selection operator.

We use roulette wheel selection. The method creates a region of a wheel based on the fitness of a population string. All the population strings occupy the space in the wheel based on their rank in fitness. A uniform random number is then generated within a closed interval of [0,1]. The value of the random number is used as a pointer and the particular population string that occupies the number is selected for the offspring generation.

Once the intermediate population strings are generated, we randomly choose two parents from the pool of the intermediate population, and apply the genetic operators (mutation only) to generate the offspring with some predefined probability distribution. We continue this process till such time the number of offspring population becomes same as the parent population.

Once the new population has been created, we find the fitness for all the population based on the weights of the hidden weights (obtained from the population string) and the output layer weights (obtained from the least square method). We also normalize the fitness value to force the population string to maintain a pre-selected range of fitness.

Intermediate population generation

$$netOutput = f(hid * weight).$$

Where f is the sigmoid function

$$RMSError = \sqrt{\frac{\sum_{i=1}^{n}\left(netOutput_i - net_i\right)^2}{n * p}}$$

$$popRMSError_i = norm(RMSError_i)$$

norm function normalized the fitness of the individual, so the fitness of each individual population is forced to be within certain range.

Step 5: *Offspring generation*

Each individual population vector $(\mathbf{w_i}, \boldsymbol{\eta_i})$, $i = 1, 2, \ldots, \mu$ creates a single offspring vector $(\mathbf{w_i'}, \boldsymbol{\eta_i'})$

for $j = 1, 2, \ldots, n$

Contd…

$\eta_i'(j) = \eta_i(j)\exp(\tau' N(0,1) + \tau N_j(0,1))$

$w_i'(j) = w_i(j) + \eta_i'(j)\ Nj(0,1)$

Where $w_i(j)$, $w_i'(j)$, $\eta_i(j)$, and $\eta_i'(j)$ denote the jth component of the vectors $\mathbf{w_i}$, $\mathbf{w_i'}$, $\boldsymbol{\eta_i}$, and $\boldsymbol{\eta_i'}$, respectively. N(0,1) denotes a normally distributed one-dimensional random number with mean and variance of 0 and 1 respectively. Nj(0,1) denotes that the random number is generated a new for each value of *j*. The parameter τ and τ' are set to $\left(\sqrt{2\sqrt{n}}\right)^{-1}$ and $\left(\sqrt{2n}\right)^{-1}$

Step 6: *Check the stopping criteria for GALS:*

If the stopping criteria for GALS is satisfied then
goto step 7
else
goto step 5.

Step 7: *Compute the weights for the output layer using a least square based method:*

After applying the evolutionary algorithm for all the weights for hidden and output layer, the best population fitness from the population pool is selected and the lower half gene of the selected population is used to replace its upper half, using least square method. The lower half gene is first used to generate the hidden layer output for all the training pairs data set, generating a matrix *hid*, whose size is P X n (Where P is the number of training pattern). The target output matrix *net* (P X m) is then linearized. As we are using non-linear sigmoidal function, we use the following linearlizing formula

$$netb_i = -\log(\frac{1-net_i}{net_i}). \tag{3-29}$$

We then require to solve the over determined system of equations as given below

$$hid * weight = netb. \tag{3-30}$$

Where *hid* is the output matrix from the hidden layer neurons and *weight* is the weight matrix output neurons. We use least square method, which is based on the QR factorization technique to solve the equation for the weight matrix using the *qr* function

Contd..

$$[Q, R] = qr(hid). \tag{3-31}$$

The *qr* method we used to decompose the **hid** is known as householder decomposition method. The solution matrix can be found from the R matrix using one step iterative process as

$$x = \frac{R}{R^T / (hid^T * netb)}. \tag{3-32}$$

The error **e** can be calculated as

$$r = netb - hid * x \tag{3-33}$$

$$e = \frac{R}{R^T / (hid^T * r)}. \tag{3-34}$$

The final value of solution for weight matrix can be then found as

$$weight = x + e. \tag{3-35}$$

Step 8: *Check the error goal:* After replacing the upper half gene from the best selected candidate from the evolutionary algorithm using least square method described in step 5, the fitness of the new transformed gene is calculated. If the fitness of the new transformed gene meets the error criteria then stop, else goto **findArchitecture** module.

3.1.4.1 Stopping criteria

Stopping criteria for EALS-T3 is based on few simple rules. The rules were same for other algorithms as well for consistency. All the rules are based on current train and test output and the maximum number of generation for the evolutionary algorithm. Following, we describe the stopping criteria for the convergence of the evolutionary algorithm.

If (best_RMS_error[4] < goal_RMS_error) then

[4] The best_RMS_error is the best of the RMS error from the population pool

```
        Stop

Else if (number_of_generation = total_number_of_generation[5]) then
        Stop

Else if (train_classification_error is increased in #m[6] consecutive generation)
then
        Stop
```

3.1.5 Finding optimal number of hidden neurons (findArchitecture)

In Figure 3-7, the flowchart for the dynamics of hierarchical structure for searching architecture (findArchitecture module) and weights (findWeight module) are given. A simple rule base can describe the working of the stopping criteria for the algorithm.

Rule 1:	If the current output is satisfactory, then stop the algorithm, else check rule 2.
Rule 2:	If the stopping criteria for weight search are met and the search is completely exhausted (in terms of number of iterations) then stop else check rule 3.
Rule 3:	If the stopping criteria for weight search are met then go to rule 4 else go to the next generation for the findWeight module.
Rule 4:	If the stopping criteria for findWeight module are met then go to rule 1, else initialize for the next number of hidden neurons.

Henceforth EALS-T3 will be refereed as new proposed algorithm when only finding weight modules is required to be mentioned without the architecture modules.

[5] Total number of generations is considered as 30
[6] m is considered as 3

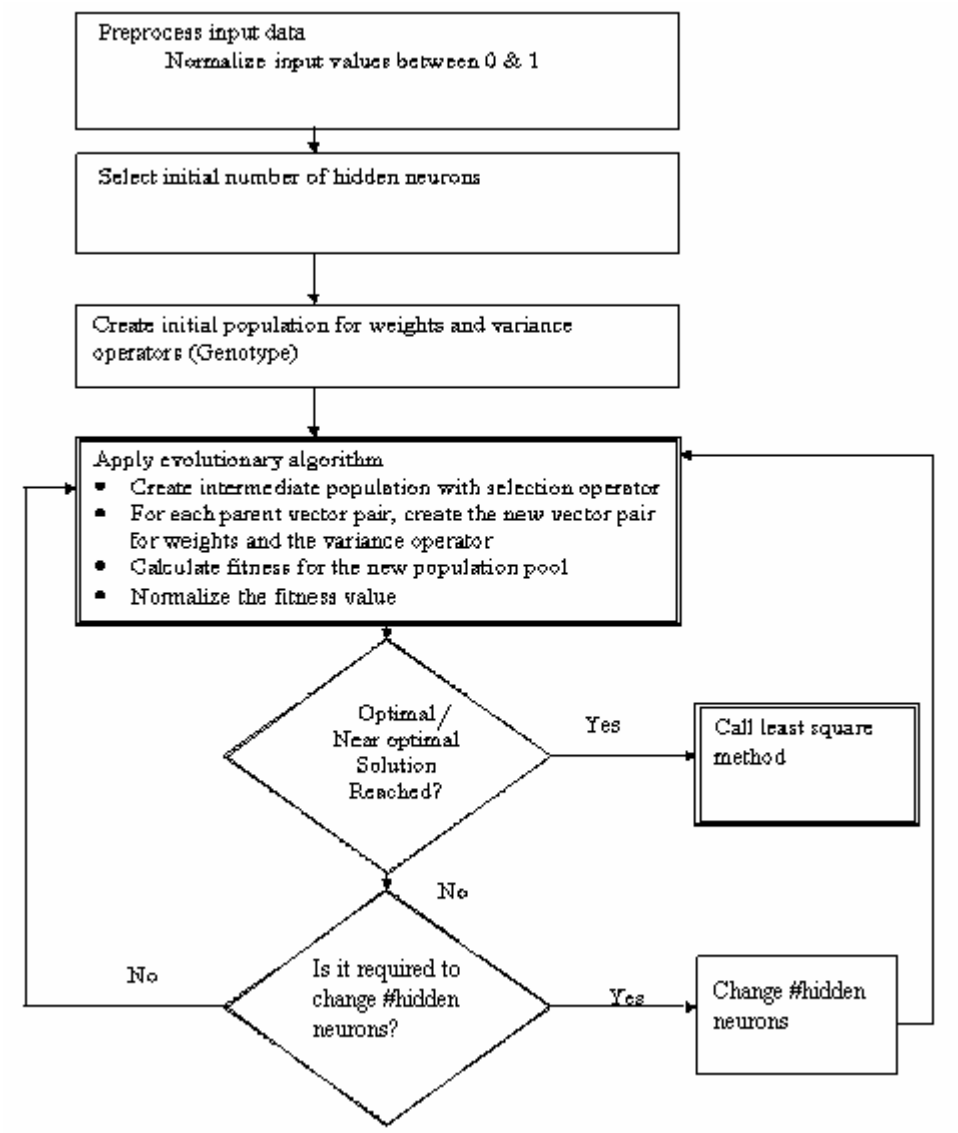

Figure 3-7 Flowchart for EALS-T3

In the following section the details for findArchitecture module when combined with EALS-T3 are given.

3.1.5.1 Combining EALS-T3 and findArchitecture modules

Two different types of experiments – Linear incrementing for EALS (LI-EALS-T3) and binary tree search type for EALS (BT-EALS-T3) to find the number of hidden neurons-

1. Starting with a small number, and then incrementing by 1 (LI-EALS-T3)
2. Using a binary tree search type (BT-EALS-T3)

Experiment A (LI-EALS-T3)

In experiment A, we start with a small number of hidden neurons and then increment the number by one. The stopping criterion for this experiment is as follows:

```
If (train_classification_error  = 0) then
        Stop

Else If (the test classification error is high in #n[7] consecutive generation) then
        Stop
```

Experiment B (BT-EALS-T3)

In experiment B, we use a binary tree search type to find the optimal number. A pseudo-code of the algorithm is given below:

```
Step 1:     Find the % test classification error & train_classification_error
            (error_min ) for #min number of hidden neurons

error_min = (train_classification_error (%)+ test_classification_error (%)) / 2
                                                                        (3-36)

Step 2:     Find the % test classification error & train classification error
            (error_max) for #max number of hidden neurons

error_max = (train_classification_error (%) + test_classification_error (%)) / 2
                                                                        (3-37)

Step 3:     Find the % test classification error & train classification error
            (error_mid) for #mid (mid = (min + max) / 2) number of hidden
            neurons

error_mid = (train_classification_error (%) + test_classification_error (%)) / 2
                                                                        (3-38)
                                                                        Contd..
```

[7] We use n = 3 for our experiments, which is determined by trial and error method

Step 4: If (error_mid < error_min) and (error_mid > error_max) then

min = mid (3-39)

mid = (min + max / 2) (3-40)

else

max = mid (3-41)

mid = (min + max / 2) (3-42)

end if

Step 5: If (mid > min) and (mid < max)

Go to Step1

Else go to Step 6.

Step 6: #number of hidden neurons = mid.

LI-EALS-T3 and BT-EALS-T3 are given the name GV1 and GV2 methods respectively.

4 EXPERIMENTAL RESULTS

In this chapter, all the experimental results are given. The algorithms that are implemented and tested for comparison purposes are – EBP, GAWLS, EAWLS, GALS/T (GALS-T1, GALS-T2, GALS-T3), EALS/T (EALS-T1, EALS-T2, EALS-T3), GV1 (LI-EALS-T3), GV2 (BT-EALS-T3)[8]. The short descriptions for all the algorithms are given in Table 4-1.

Table 4-1 Description for all the algorithms

Algorithm	Description	Algorithm Parameters
EBP	It uses the standard 1st order gradient based back propagation algorithm with adaptive parameters for learning rate and momentum.	Learning rate: 0.2 Momentum: 0.7
GAWLS	It uses the standard GA to evolve for all the weights of the ANN using both crossover and mutation genetic operators.	Mutation probability rate: 0.2 Crossover probability rate: 0.8 Elitism order: 1
EAWLS	It uses the EA to evolve for all the weights of the ANN using only mutation genetic operators.	Mutation probability rate: Not required. Crossover probability rate: Not required. Elitism order: 1
GALS/T	It uses new hybrid learning using GA and the least squares method. The hidden layer weight is evolved using the GA, and the output layer weights are found using the Least square method. The three different variations are due to the connection type for GA with the LS method.	Mutation probability rate: 0.2 Crossover probability rate: 0.8 Elitism order: 1 Continued..

[8] The details for the new algorithms are discussed in detail in chapter 3 (research methodology)

Algorithm	Description	Algorithm Parameters
EALS/T	It uses new hybrid learning using EA and the least squares method. The hidden layer weight is evolved using the EA, and the output layer weights are found using the Least square method. The three different variations are due to the connection type for EA with the LS method.	Mutation probability rate: Not required. Crossover probability rate: Not required. Elitism order: 1
GV1 (LI-EALS-T3)	It finds dynamically both the weights and architecture. The two modules are connected hierarchically. The findArchitecture module is based on an incremental approach, and the findWeight module is based on EA and LS with T3 connection type.	Mutation probability rate: Not required. Crossover probability rate: Not required. Elitism order: 1
GV2 (BT-EALS-T3)	It finds dynamically both the weights and architecture. The two modules are connected hierarchically. The findArchitecture module is based on a binary search type approach, and the findWeight module is based on EA and LS with T3 connection type.	Mutation probability rate: Not required. Crossover probability rate: Not required. Elitism order: 1

4.1 IMPLEMENTATION DETAILS

All the algorithms were implemented using C language.

4.1.1 Platform used

All the experiments were run on a HPC super computer, which uses a SunFire v880 as a node. The nodes are connected via gigabit optical fiber link. The main configuration of the HPC machine is as follows:

- *1 Front node* - 8 X Ultra Sparc III processors 750 MHz and 8GB memory.
- *8 Back nodes* - each contains 8 X Ultra Sparc III processors 900MHz and each node contains 8GB of memory. Back node consists of 64 X Ultra Sparc III processors 900MHz and 64GB of memory.
- The machine is in a cluster formation.
- External access is only via the front node and jobs are submit via the batching system to predefined queues on the back nodes.

4.1.2 Data set details

All the algorithms are tested on XOR, 10 bit Odd Parity, and some other real-world benchmark data sets such as handwriting character dataset from CEDAR, Breast cancer (Wisconsin) and Heart Disease (Cleveland, Hungary and Switzerland) from UCI Machine Learning repository. Table 4-2 shows the details of the individual data set consider for training and testing for the comparison of all the algorithms.

Table 4-2 Data set information

Data set name	Input details	Attribute information	Data source[9]
XOR	Pattern length = 4 Training pattern = 4	# Input columns = 2 # Output column = 1	X
10 bit odd parity	Pattern length = 1024 Training pattern = 900 Testing pattern = 124	# Input columns = 10 # Output column = 1	X
Handwriting characters[10] (CEDAR)	Pattern length = 350 Training pattern = 300 Testing pattern = 50	# Input columns = 100 # Output column = 29 (26 characters and 3 rejected characters)	CEDAR Data set
Breast cancer (Wisconsin)	Pattern length = 699 Training pattern = 400	# Input columns = 10 # Output column = 1	UCI ML Repository

[9] X in a cell of a table denotes not required

[10] For handwriting character data set features are first extracted using the transitional feature extraction method from the image file, and then the feature vectors are used as an input for the ANN. The length of the feature vector is 100 and the images are considered for upper case English characters (A-Z).

	Testing pattern = 299		
Heart disease[11] (Cleveland)	Pattern length = 303 Training pattern = 250 Testing pattern = 53	# Input columns = 14 # Output column = 1	UCI ML Repository
Heart disease (Hungary)	Pattern length = 294 Training pattern = 200 Testing pattern = 94		
Heart disease (Switzerland)	Pattern length = 123 Training pattern = 100 Testing pattern = 23		

4.2 RESULTS

For simplicity, the results are mainly divided into two different categories – Type I and Type II. The first category (Type I) contains the results for caparison purposes with other algorithms. The second category (Type II) mainly contains the results to obtain certain special properties for the proposed algorithm in details.

4.2.1 Experimental Results – Type I

In the first section of the real data set results, algorithms are run for variation of their parameters like number of hidden neurons, number of population, number of generations etc. Experiments with variations in parameters are conducted for comments on comparison of the convergence properties, sensitivity of the proposed algorithm with number of hidden neurons and population etc. In the second section the best results in terms of the classification error are given for comparison on classification error, time complexity etc.

[11] This database originally contains 76 attributes, but all published experiments refer to using a subset of 14 of them. In case of missing attributes values after the mapping of the input data is done, the missing values were substituted with a value 0.5 for the range (0,1).

4.2.1.1 Results with parameter variations

Experiments in this section were mainly for comments on the comparison purposes for the convergence and generalization ability for all the algorithms and also the sensitivity of the algorithm to the variable parameters. The choices for initial values are made by some trail and error method. The variations for the parameters are similar in all the cases for the purpose of comparisons. In this section the main aim is to compare the results on the basis of some characteristic metrics, hence consistency over variations of parameters are more important than obtaining best results in terms of classification and RMS accuracy.

It is important to be noted here that the initial values for all the variable parameters are chosen arbitrarily. Mainly three different types of variable parameters are used in this section: number of generation, number of hidden neurons and number of population. The first two are used for the EBP and the last two variable parameters are used for all the GA / EA based algorithms. The range of values for number of generations are categorized into three different subset (100+[12];0 < p < 100, 500+; 100 <= p < 500, 1000+; p >= 500). For number of hidden neurons the data sets are categorized into four different subsets (2+; 0 < p < 100, 10+; 100 <= p < 200, 20+; 200 <= p < 500, 30+; p >= 500). For number of population the ranges of values are same for all the data set invariant of their patterns length (20, 30, 40, 50 and 60). Algorithms, which find the architecture automatically (LI-EALS-T3 and BT-EALS-T3) need not be considered in this section.

All the algorithms are run in this section in three different ways as applicable to the particular algorithm:

1. The number of hidden neurons is fixed and we increase the number of generation in predefined step size. Always five different sets of results with five different numbers of generations are reported. The value 5 is chosen arbitrarily for comments on comparison purposes for the convergence ability of all the algorithms. This is applicable only to EBP.

[12] 100+ denotes 100 is the initial value for the variable, with the plus sign denoting the subsequent increment. The increment value is based on the initial value for 100, and is chosen quite arbitrarily.

2. The number of hidden neurons is increased one at a time to test the results. Results are given for 5 different numbers of hidden neurons for comments on the comparison purposes for the generalization ability for all the algorithms. This is applicable for all the algorithms.
3. The population number is varied and the algorithm is run for different number of hidden neurons, keeping fixed the number of population. This experiment is also conducted to comment on the convergence property and sensitivity of the algorithm for number of population and hidden neurons. This is applicable only to GA / EA based algorithms.

4.2.1.1.1 Results for EBP

In the following 7 tables (Tables 4-3 to 4-9) the results from the EBP are given. The classification and RMS error are reported with variations of number of hidden neurons and number of generations. For stopping criteria for EBP in this section the following rules are considered:

1. Initialize the ANN with given number of hidden neurons and the training is started.
2. If the given number of generation is reached then the algorithm is stopped.

Table 4-3 EBP results with different #iteration and #hidden neurons for XOR dataset

#hidden neurons	#iterations	Time (seconds)	Train		Test	
			RMS	Class (%)	RMS	Class (%)
2	100	20	0.4869	0	X	X
	200	40	0.2284	0	X	X
	300	61	0.1388	0	X	X
	400	83	0.1002	0	X	X
	500	102	0.0655	0	X	X
3	100	25	0.4764	0	X	X
	200	44	0.3711	0	X	X
	300	67	0.2616	0	X	X
	400	89	0.2485	0	X	X
	500	106	0.1238	0	X	X
4	100	28	0.4718	0	X	X
	200	46	0.2234	0	X	X
	300	69	0.1967	0	X	X

	400	91	0.1060	0	X	X
	500	108	0.0949	0	X	X
5	100	31	0.4669	0	X	X
	200	52	0.3184	0	X	X
	300	73	0.2321	0	X	X
	400	97	0.2150	0	X	X
	500	112	0.0919	0	X	X
6	100	34	0.4216	0	X	X
	200	57	0.3662	0	X	X
	300	79	0.3632	0	X	X
	400	102	0.3227	0	X	X
	500	121	0.0859	0	X	X

Table 4-4 EBP results with different #iteration and #hidden neurons for 10 bit odd parity dataset

#hidden neurons	#iterations	Time (minutes)[13]	Train		Test	
			RMS	Class (%)	RMS	Class (%)
30	1000	30	0.2943	27.10	0.1421	16.89
	1500	39	0.2834	27.00	0.1359	15.44
	2000	43	0.2803	26.80	0.1321	14.53
	2500	45	0.2753	25.70	0.1260	14.52
	3000	48	0.2635	24.30	0.1174	13.58
31	1000	31	0.2735	26.60	0.1415	16.33
	1500	34	0.2637	25.20	0.1364	15.22
	2000	36	0.2573	24.30	0.1268	14.74
	2500	40	0.2417	23.60	0.1164	14.11
	3000	42	0.2351	22.80	0.0952	14.10
32	1000	32	0.2678	25.90	0.1402	16.07
	1500	36	0.2545	24.20	0.1394	15.42
	2000	39	0.2434	23.80	0.1285	15.18
	2500	42	0.2378	21.40	0.1024	14.39
	3000	47	0.2123	19.60	0.0904	14.04
33	1000	32	0.2627	23.40	0.1324	15.64
	1500	40	0.2532	22.10	0.1214	15.23
	2000	42	0.2475	20.40	0.1001	15.08
	2500	43	0.2158	19.10	0.0934	14.12
	3000	46	0.1876	18.20	0.0893	14.12
34	1000	32	0.2423	22.60	0.0967	14.83
	1500	35	0.2267	20.50	0.0897	14.32
	2000	44	0.2012	19.70	0.0856	14.31
	2500	46	0.1789	17.60	0.0789	13.47
	3000	50	0.1541	15.40	0.0765	13.26

[13] The decimal portion for the seconds values are ignored and only integer values for the minutes results are given.

Table 4-5 EBP results with different #iteration and #hidden neurons for handwriting characters (CEDAR) dataset

#hidden neurons	#iterations	Time (minutes)	Train		Test	
			RMS	Class (%)	RMS	Class (%)
30	1000	30	0.9601	30.20	0.9647	23.11
	1500	33	0.1258	25.60	0.8234	22.07
	2000	48	0.0890	22.10	0.1435	21.70
	2500	50	0.0790	18.45	0.1102	19.00
	3000	50	0.0737	10.67	0.1023	18.62
31	1000	31	0.2784	22.78	0.6453	23.02
	1500	34	0.2731	18.45	0.5123	21.52
	2000	38	0.2493	17.00	0.4435	21.34
	2500	45	0.2012	16.32	0.3701	20.61
	3000	47	0.1756	14.67	0.2506	19.14
32	1000	33	0.2245	21.67	0.2103	23.00
	1500	37	0.2204	18.98	0.1978	22.26
	2000	41	0.2103	14.67	0.1767	20.49
	2500	42	0.2101	12.45	0.1509	20.35
	3000	48	0.2012	8.67	0.1245	18.88
33	1000	34	0.2220	12.56	0.1194	22.68
	1500	37	0.2134	10.09	0.1023	21.92
	2000	43	0.2035	9.23	0.0945	19.74
	2500	48	0.1880	8.50	0.0789	19.22
	3000	50	0.1676	6.70	0.0524	18.99
34	1000	38	0.1897	7.90	0.0563	21.81
	1500	41	0.1785	5.60	0.0447	21.47
	2000	43	0.1325	4.70	0.0345	20.90
	2500	43	0.1156	4.50	0.0234	19.37
	3000	50	0.1146	3.00	0.0140	18.15

Table 4-6 EBP results with different #iteration and #hidden neurons for breast cancer (Wisconsin) dataset

#hidden neurons	#iterations	Time (minutes)	Train		Test	
			RMS	Class (%)	RMS	Class (%)
20	500	25	0.4601	20.52	0.9647	16.65
	600	32	0.1258	18.46	0.8234	15.48
	700	32	0.0890	16.10	0.1435	15.46
	800	32	0.0790	14.50	0.1102	14.29
	900	39	0.0737	12.67	0.1023	13.56
21	500	28	0.2784	18.38	0.6453	16.26
	600	32	0.2731	17.50	0.5123	15.62
	700	34	0.2493	15.80	0.4435	15.01
	800	37	0.2012	13.72	0.3701	13.32
	900	40	0.1756	12.07	0.2506	12.62
22	500	31	0.2245	16.70	0.2103	15.91
	600	35	0.2204	15.87	0.1978	15.03
	700	35	0.2103	13.07	0.1767	14.74
	800	36	0.2101	12.15	0.1509	12.75
	900	38	0.2012	11.67	0.1245	12.69
23	500	32	0.2220	10.66	0.1194	15.89
	600	34	0.2134	9.51	0.1023	14.11
	700	36	0.2035	8.10	0.0945	13.82
	800	38	0.1880	7.50	0.0789	13.25

	900	39	0.1676	6.80	0.0524	12.98
24	500	34	0.1897	5.60	0.0563	13.68
	600	36	0.1785	4.60	0.0447	13.59
	700	37	0.1325	3.80	0.0345	13.39
	800	37	0.1156	3.60	0.0234	13.08
	900	39	0.1146	3.50	0.0140	13.06

Table 4-7 EBP results with different #iteration and #hidden neurons heart disease (Cleveland) dataset

#hidden neurons	#iterations	Time (minutes)	Train		Test	
			RMS	Class (%)	RMS	Class (%)
20	500	25	0.2518	5.30	0.1750	16.81
	600	30	0.2213	5.10	0.1644	14.93
	700	32	0.2012	5.05	0.1606	14.68
	800	33	0.1911	4.90	0.1601	13.26
	900	34	0.1900	4.80	0.1599	12.68
21	500	26	0.2423	5.00	0.1693	16.15
	600	26	0.2103	4.90	0.1645	15.28
	700	30	0.2004	4.50	0.1590	14.63
	800	33	0.1985	4.20	0.1500	14.27
	900	36	0.1900	4.10	0.1480	13.11
22	500	26	0.2139	4.80	0.1645	16.09
	600	29	0.2013	4.70	0.1580	16.07
	700	31	0.1897	4.50	0.1450	14.27
	800	38	0.1856	4.60	0.1400	13.63
	900	38	0.1804	4.80	0.1380	12.57
23	500	27	0.2038	4.70	0.1550	16.02
	600	29	0.1956	4.50	0.1461	14.05
	700	30	0.1903	4.40	0.1380	13.93
	800	30	0.1845	4.30	0.1300	12.88
	900	40	0.1801	4.20	0.1250	12.13
24	500	28	0.1904	4.50	0.1460	15.99
	600	29	0.1832	4.20	0.1280	15.81
	700	29	0.1806	4.10	0.1200	14.61
	800	30	0.1789	4.00	0.1150	14.60
	900	37	0.1742	3.90	0.1090	12.74

Table 4-8 EBP results with different #iteration and #hidden heart disease (Hungary) dataset

#hidden neurons	#iterations	Time (minutes)	Train		Test	
			RMS	Class (%)	RMS	Class (%)
20	500	27	0.9773	3.33	0.2193	16.75
	600	28	0.7946	3.27	0.2074	15.11
	700	31	0.6339	3.38	0.2027	13.60
	800	32	0.5663	2.84	0.2009	13.43
	900	34	0.3447	3.19	0.1986	12.85
21	500	27	0.9746	3.38	0.2043	16.42
	600	30	0.9686	3.17	0.1887	16.20
	700	34	0.8784	2.81	0.1814	14.93
	800	38	0.7227	2.19	0.1765	13.12

	900	40	0.4662	2.30	0.1716	12.41
22	500	27	0.9473	3.39	0.1926	16.38
	600	30	0.8093	2.64	0.1591	16.24
	700	31	0.6441	2.43	0.1539	15.81
	800	34	0.5811	2.70	0.1393	13.34
	900	40	0.2771	3.05	0.1283	12.85
23	500	27	0.9275	2.61	0.1896	16.21
	600	30	0.8048	2.71	0.1852	14.55
	700	32	0.6949	2.70	0.1834	13.09
	800	33	0.5507	2.35	0.1605	12.52
	900	37	0.4433	2.36	0.1517	12.36
24	500	28	0.9210	2.71	0.1867	15.67
	600	29	0.8957	2.66	0.1834	15.57
	700	32	0.6908	2.51	0.1820	14.84
	800	34	0.6325	1.92	0.1792	13.38
	900	38	0.5283	1.89	0.1558	12.83

Table 4-9 EBP results with different #iteration and #hidden heart disease (Switzerland) dataset

#hidden neurons	#iterations	Time (minutes)	Train		Test	
			RMS	Class (%)	RMS	Class (%)
10	500	15	0.9980	3.62	0.4629	17.06
	600	18	0.9727	3.43	0.3909	16.29
	700	22	0.9540	3.77	0.3299	15.81
	800	24	0.6863	2.97	0.3048	15.17
	900	24	0.5603	3.37	0.1638	13.48
11	500	17	0.9946	3.54	0.4298	16.88
	600	18	0.8015	3.37	0.3426	15.36
	700	18	0.5709	2.84	0.3118	15.14
	800	22	0.3738	2.60	0.2914	14.75
	900	23	0.1973	2.32	0.2118	14.61
12	500	18	0.9928	3.76	0.4242	16.51
	600	18	0.9908	2.99	0.3942	15.46
	700	21	0.9202	2.86	0.3613	14.38
	800	22	0.6166	2.81	0.2875	13.15
	900	24	0.5032	3.18	0.2509	12.13
13	500	18	0.9903	3.01	0.4198	15.96
	600	23	0.9340	2.89	0.4158	15.51
	700	25	0.6155	3.01	0.3069	13.52
	800	25	0.5147	2.50	0.3041	13.33
	900	25	0.3334	2.61	0.2294	12.74
14	500	20	0.9384	2.88	0.4058	15.77
	600	21	0.7641	2.79	0.3088	15.70
	700	22	0.6271	2.94	0.3036	14.27
	800	22	0.3687	1.98	0.2613	13.11
	900	23	0.1749	2.00	0.1805	12.98

4.2.1.1.2 Results for GAWLS

In the following 7 tables (Tables 4-10 to 4-16) the results from the GAWLS are given. The classification and RMS error are reported with variations of number of hidden neurons and number of generations. Because in case of genetic based algorithm, the solution is very stable once it is reached (the experimental analysis for this characteristic algorithm are given in section 4.2.2), the number of generations are not used as a variable parameters. The number of generations after which convergence is reached are also given.

Table 4-10 GAWLS results with different #population and #hidden neurons for XOR dataset

#population	#hidden neurons	#generations	Time (seconds)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	2	30	0.8	0.8578	25.00	X	X
	3	32	0.8	0.7540	0.00	X	X
	4	36	0.9	0.6742	0.00	X	X
	5	38	1.3	0.6337	0.00	X	X
	6	40	2.6	0.5175	0.00	X	X
30	2	30	0.8	0.8526	0.00	X	X
	3	32	1.5	0.7917	0.00	X	X
	4	36	1.7	0.7629	25.00	X	X
	5	38	1.8	0.6170	0.00	X	X
	6	40	2.0	0.4750	0.00	X	X
40	2	30	1.3	0.8509	0.00	X	X
	3	32	1.5	0.7507	0.00	X	X
	4	36	1.5	0.6532	25.00	X	X
	5	38	2.2	0.5877	0.00	X	X
	6	40	2.6	0.4633	0.00	X	X
50	2	30	1.3	0.8037	0.00	X	X
	3	32	1.5	0.6720	0.00	X	X
	4	36	1.6	0.5196	0.00	X	X
	5	38	1.8	0.5099	0.00	X	X
	6	40	2.6	0.3809	0.00	X	X
60	2	30	1.7	0.7443	0.00	X	X
	3	32	2.0	0.6417	0.00	X	X
	4	36	2.3	0.6139	0.00	X	X
	5	38	2.3	0.5537	0.00	X	X
	6	40	2.6	0.3806	0.00	X	X

Table 4-11 GAWLS results with different #population and #hidden neurons for 10 bit odd parity dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	30	53	34	0.8971	46.22	0.8659	50.68
	31	50	44	0.8578	44.44	0.4580	50.64
	32	52	48	0.4605	42.96	0.4023	47.16
	33	55	52	0.4210	40.61	0.3445	44.42
	34	47	54	0.3521	39.07	0.2216	41.79
30	30	51	35	0.7457	45.54	0.7101	50.31
	31	56	38	0.6855	44.68	0.6561	47.83
	32	57	40	0.4017	40.24	0.4003	47.76
	33	55	44	0.3108	37.36	0.2304	42.92
	34	51	47	0.2638	35.97	0.2227	41.01
40	30	50	36	0.7365	45.56	0.7050	49.76
	31	49	46	0.6632	44.33	0.6495	48.70
	32	57	48	0.4671	44.16	0.4277	44.75
	33	46	48	0.3955	39.98	0.3705	42.06
	34	46	52	0.2450	34.25	0.2118	41.15
50	30	59	36	0.6297	43.72	0.6347	47.74
	31	54	40	0.5225	41.61	0.5011	47.61
	32	46	50	0.4775	40.55	0.4587	46.11
	33	53	52	0.2315	35.32	0.2017	42.07
	34	52	55	0.2298	31.99	0.1492	41.84
60	30	60	38	0.5995	44.10	0.5512	44.71
	31	55	41	0.4294	39.59	0.4014	43.76
	32	55	48	0.3573	35.41	0.3132	43.45
	33	47	53	0.2147	34.34	0.1801	43.25
	34	49	57	0.1615	30.90	0.1552	42.67

Table 4-12 GAWLS results with different #population and #hidden neurons for handwriting characters (CEDAR) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	30	48	34	0.9512	43.41	0.9045	58.47
	32	52	37	0.7735	40.41	0.7587	55.56
	34	59	55	0.6360	37.38	0.6069	54.69
	36	58	55	0.4575	30.71	0.6061	53.58
	38	59	57	0.3718	28.76	0.5087	50.61
30	30	45	35	0.8384	42.14	0.8501	58.29
	32	58	38	0.6134	34.19	0.6919	57.48
	34	57	43	0.6110	33.83	0.5803	55.41
	36	56	50	0.4354	30.12	0.4007	53.64
	38	60	53	0.3355	26.49	0.3501	51.16
40	30	57	37	0.8030	41.72	0.7775	58.26
	32	52	41	0.6431	34.53	0.6075	55.60
	34	53	46	0.5226	32.16	0.5948	54.61
	36	51	47	0.4164	30.05	0.4085	54.59
	38	56	54	0.3040	26.09	0.3182	51.24
50	30	48	38	0.6694	34.76	0.5814	57.90
	32	48	42	0.4780	32.32	0.4218	57.29
	34	50	48	0.3456	30.49	0.3052	56.72
	36	56	54	0.2165	25.91	0.1800	56.15
	38	59	55	0.1458	23.58	0.1014	50.12

60	30	59	43	0.6543	34.10	0.5443	57.51
	32	49	46	0.5805	30.65	0.5312	56.64
	34	56	47	0.4491	25.64	0.4045	52.72
	36	53	48	0.3005	21.09	0.2793	52.23
	38	58	56	0.2843	18.19	0.2520	50.75

Table 4-13 GAWLS results with different #population and #hidden neurons for breast cancer (Wisconsin) dataset

#population	#hidden neurons	#generations	Time (seconds)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	20	46	28	0.5771	39.68	0.5167	55.09
	22	57	36	0.4811	36.03	0.4045	49.38
	24	57	36	0.3453	34.51	0.3746	48.57
	26	54	37	0.2450	32.42	0.2134	48.55
	28	46	45	0.1111	27.83	0.0782	48.21
30	20	53	31	0.5542	37.46	0.5007	54.74
	22	56	36	0.4417	35.05	0.4293	52.04
	24	56	38	0.3904	34.70	0.3027	48.61
	26	53	42	0.2851	32.58	0.2186	48.41
	28	51	44	0.1068	26.15	0.0884	47.62
40	20	46	35	0.5297	35.04	0.4495	54.46
	22	47	39	0.4630	34.45	0.4017	52.24
	24	55	40	0.3962	33.29	0.3497	50.85
	26	58	40	0.2431	28.70	0.2114	50.75
	28	51	42	0.1029	25.93	0.0780	49.77
50	20	49	36	0.5160	35.72	0.4398	54.20
	22	53	38	0.3994	32.75	0.3290	53.72
	24	58	40	0.2007	27.83	0.1996	53.44
	26	49	43	0.1397	26.93	0.1000	52.31
	28	53	45	0.0850	23.33	0.0618	47.23
60	20	50	39	0.4780	32.98	0.4109	53.19
	22	57	41	0.3814	30.29	0.3126	50.94
	24	47	41	0.2327	27.28	0.1729	48.03
	26	58	42	0.1143	21.18	0.0955	46.79
	28	54	43	0.0696	20.20	0.0401	46.05

Table 4-14 GAWLS results with different #population and #hidden neurons heart disease (Cleveland) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	20	57	28	0.8890	23.25	0.9401	54.90
	22	53	34	0.8578	23.01	0.8132	50.07
	24	50	35	0.7540	20.53	0.2614	49.26
	26	52	37	0.6337	18.69	0.1792	47.68
	28	55	38	0.5742	15.37	0.1512	45.95
30	20	47	29	0.8375	22.30	0.8050	54.77
	22	51	32	0.8181	21.91	0.7705	53.94
	24	56	35	0.7629	20.14	0.5040	53.84
	26	57	42	0.6917	18.89	0.4277	48.85
	28	55	42	0.5175	15.31	0.1495	48.04
40	20	51	29	0.8226	21.37	0.8011	54.20

	22	50	29	0.7530	20.73	0.6587	53.40
	24	49	33	0.6507	18.74	0.4347	52.51
	26	57	37	0.5633	15.89	0.4118	50.00
	28	46	41	0.5177	15.19	0.2017	48.60
50	20	46	30	0.7809	20.47	0.6980	53.53
	22	59	32	0.6720	18.64	0.5659	53.18
	24	54	34	0.6037	16.91	0.4023	52.87
	26	46	34	0.5099	14.55	0.3745	51.24
	28	53	45	0.4896	14.50	0.3624	50.60
60	20	52	32	0.7209	20.49	0.6616	52.80
	22	60	32	0.6444	18.74	0.5101	52.49
	24	55	33	0.5418	14.45	0.4561	49.82
	26	55	33	0.4539	13.74	0.3463	47.66
	28	47	42	0.4037	13.13	0.2827	46.25

Table 4-15 GAWLS results with different #population and #hidden heart disease (Hungary) dataset

#population	#hidden neurons	#generations	Time (seconds)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	20	49	45	0.8065	71	0.9919	55.11
	22	48	90	0.7371	98	0.9807	53.50
	24	52	95	0.4606	148	0.6752	48.92
	26	59	147	0.4216	350	0.5201	46.68
	28	58	156	0.3522	557	0.3703	46.42
30	20	59	49	0.7638	76	0.9045	54.85
	22	45	86	0.6457	108	0.7761	53.57
	24	58	96	0.5856	175	0.7182	51.64
	26	57	114	0.4818	316	0.5987	50.52
	28	56	156	0.3108	333	0.4587	47.98
40	20	60	56	0.6850	159	0.9001	54.70
	22	57	133	0.5365	176	0.7775	54.25
	24	52	138	0.4633	188	0.7575	48.36
	26	53	179	0.3671	484	0.5248	47.61
	28	51	196	0.2955	594	0.4785	46.27
50	20	56	64	0.6515	171	0.7220	51.86
	22	48	138	0.5298	281	0.5814	50.59
	24	48	166	0.4226	374	0.4218	47.22
	26	50	169	0.3775	491	0.3552	47.04
	28	56	190	0.2298	567	0.2300	46.32
60	20	59	88	0.6315	228	0.6593	51.67
	22	59	104	0.5396	473	0.6069	48.08
	24	49	124	0.4294	493	0.5443	46.84
	26	56	133	0.3874	516	0.5312	46.66
	28	53	141	0.1847	538	0.4745	46.03

Table 4-16 GAWLS results with different #population and #hidden heart disease (Switzerland) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	10	58	17	0.8269	43.90	0.9786	54.18
	12	46	20	0.7512	40.41	0.9714	51.52
	14	57	24	0.6360	37.41	0.7793	48.21
	16	57	27	0.4576	34.38	0.6227	47.80
	18	54	27	0.3735	30.71	0.6107	47.65
30	10	46	19	0.8018	43.76	0.9055	54.09
	12	53	20	0.7385	42.14	0.8529	53.84
	14	56	20	0.6134	37.19	0.7126	51.19
	16	56	25	0.4111	33.83	0.7109	46.04
	18	53	25	0.3354	29.92	0.3918	46.04
40	10	51	20	0.7355	42.49	0.8945	53.40
	12	46	20	0.6227	37.72	0.8746	49.21
	14	47	24	0.4104	33.53	0.8384	49.13
	16	55	24	0.3431	28.16	0.8167	48.56
	18	58	27	0.2223	26.05	0.5134	47.16
50	10	51	21	0.6964	40.49	0.8497	51.05
	12	49	26	0.5343	37.76	0.7114	50.44
	14	53	28	0.3806	29.32	0.5017	50.20
	16	58	28	0.2492	26.49	0.2582	50.04
	18	49	29	0.1316	22.91	0.2495	48.88
60	10	53	23	0.6256	37.58	0.7396	50.98
	12	50	23	0.5094	31.10	0.7290	48.62
	14	57	25	0.4181	26.65	0.4598	47.13
	16	47	25	0.2458	23.64	0.4500	46.54
	18	58	26	0.1166	18.09	0.1985	46.12

4.2.1.1.3 Results for EAWLS

In the following 7 tables (Tables 4-17 to 4-23) the results from the EAWLS are given. The classification and RMS error are reported with variations of number of hidden neurons and number of generations.

Table 4-17 EAWLS results with different #population and #hidden neurons for XOR dataset

#population	#hidden neurons	#generations	Time (seconds)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	2	21	0.7	0.4450	25.00	X	X
	3	20	0.8	0.4521	0.00	X	X
	4	19	0.9	0.4443	25.00	X	X
	5	18	2.2	0.4368	0.00	X	X
	6	26	2.8	0.4778	25.00	X	X
30	2	22	0.8	0.4420	25.00	X	X
	3	29	1.1	0.4559	0.00	X	X
	4	17	1.6	0.4756	25.00	X	X
	5	17	1.7	0.4339	25.00	X	X

	6	27	2	0.4533	0.00	X	X
40	2	25	0.9	0.4805	0.00	X	X
	3	24	1	0.4466	0.00	X	X
	4	27	1.1	0.4409	25.00	X	X
	5	16	2	0.4411	0.00	X	X
	6	18	2.3	0.4859	0.00	X	X
50	2	19	1.2	0.4444	0.00	X	X
	3	24	1.5	0.4715	25.00	X	X
	4	18	1.8	0.4583	0.00	X	X
	5	17	2.5	0.4591	0.00	X	X
	6	23	2.9	0.4657	25.00	X	X
60	2	19	0.7	0.4450	25.00	X	X
	3	29	1.4	0.4521	0.00	X	X
	4	22	1.9	0.4443	25.00	X	X
	5	21	2.2	0.4368	0.00	X	X
	6	20	2.7	0.4778	25.00	X	X

Table 4-18 EAWLS results with different #population and #hidden neurons for 10 bit odd parity dataset

#population	#hidden neurons	#generations	Time (seconds)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	30	50	33	0.9418	43.66	0.9534	40.53
	31	35	43	0.9081	42.21	0.7017	40.45
	32	43	48	0.8009	40.90	0.6617	39.75
	33	44	50	0.6870	44.62	0.4733	39.56
	34	48	52	0.5265	35.43	0.3082	37.40
30	30	45	34	0.8756	38.69	0.9321	40.49
	31	49	37	0.7921	42.70	0.8441	38.01
	32	41	39	0.6467	40.81	0.6767	35.05
	33	42	43	0.4471	43.23	0.6167	33.02
	34	45	46	0.4028	32.90	0.1138	30.56
40	30	50	35	0.8269	39.76	0.8470	40.39
	31	39	39	0.7939	38.50	0.7891	39.64
	32	42	42	0.6065	40.46	0.6674	39.09
	33	49	47	0.4469	40.22	0.6557	33.46
	34	39	52	0.2295	35.34	0.5637	32.64
50	30	42	35	0.7994	35.67	0.7496	37.12
	31	45	43	0.5670	38.49	0.3856	36.30
	32	49	45	0.5454	35.25	0.3460	33.34
	33	47	48	0.5312	31.94	0.2797	31.98
	34	43	50	0.2562	32.54	0.2400	30.33
60	30	44	36	0.7160	41.88	0.6092	36.83
	31	49	38	0.6818	38.26	0.5957	36.73
	32	47	48	0.6312	38.01	0.3638	34.83
	33	49	51	0.4572	36.81	0.2787	31.82
	34	48	54	0.3079	32.21	0.2637	30.55

Table 4-19 EAWLS results with different #population and #hidden neurons for handwriting characters (CEDAR) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	30	37	34	0.9959	45.48	0.9991	49.98
	31	38	36	0.7237	44.52	0.9942	48.46
	32	36	52	0.6329	35.20	0.6703	48.23
	33	43	54	0.6178	32.71	0.5762	45.70
	34	40	54	0.4939	21.60	0.1165	42.00
30	30	38	34	0.9910	41.99	0.9691	49.51
	31	49	37	0.9664	36.68	0.9678	47.51
	32	46	42	0.9027	34.16	0.8872	46.88
	33	38	50	0.7096	27.01	0.8313	46.28
	34	37	52	0.4984	27.59	0.6254	41.73
40	30	45	36	0.9829	41.36	0.9457	48.92
	31	45	41	0.8702	38.51	0.8716	48.65
	32	46	45	0.7558	33.56	0.6410	44.03
	33	37	46	0.6002	27.83	0.3471	43.49
	34	36	53	0.2017	28.56	0.2837	40.96
50	30	37	38	0.9432	25.94	0.9329	48.62
	31	45	41	0.8051	29.47	0.9010	48.50
	32	49	47	0.6404	21.88	0.6669	46.88
	33	42	53	0.4535	20.40	0.5738	45.06
	34	45	55	0.3732	18.79	0.4065	42.64
60	30	48	42	0.8007	27.37	0.8132	46.72
	31	43	45	0.6697	17.31	0.7365	45.86
	32	49	46	0.6576	23.76	0.7071	43.88
	33	44	48	0.6381	14.98	0.5441	43.32
	34	49	55	0.4830	21.44	0.4284	40.87

Table 4-20 EAWLS results with different #population and #hidden neurons for breast cancer (Wisconsin) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	20	46	28	0.9976	38.51	0.9903	44.96
	21	49	35	0.6308	31.53	0.8790	44.88
	22	36	35	0.4685	26.41	0.7143	44.65
	23	37	36	0.4331	32.84	0.4040	44.59
	24	45	44	0.0975	30.52	0.3407	37.10
30	20	47	31	0.9634	34.77	0.9839	44.52
	21	48	35	0.9047	36.92	0.8114	44.07
	22	44	38	0.8354	34.10	0.6069	43.99
	23	35	40	0.4887	27.94	0.4510	41.74
	24	43	44	0.2435	26.92	0.2923	39.64
40	20	38	34	0.9346	34.89	0.8818	44.51
	21	37	38	0.7943	33.49	0.6566	43.35
	22	38	39	0.6317	24.21	0.6482	38.98
	23	43	40	0.5902	28.63	0.6289	36.86
	24	43	42	0.4956	30.49	0.5973	36.58
50	20	39	35	0.8516	25.16	0.8181	44.23
	21	43	38	0.7623	28.13	0.5157	43.97
	22	47	40	0.7481	23.57	0.4092	40.08
	23	46	42	0.7190	19.05	0.3143	40.06
	24	38	42	0.6469	18.93	0.2064	37.13

60	20	35	37	0.7274	19.05	0.7849	43.40
	21	42	39	0.7167	23.27	0.4942	42.72
	22	36	40	0.5781	17.43	0.4544	42.60
	23	47	41	0.5428	14.29	0.2569	41.80
	24	44	43	0.4216	22.90	0.2482	41.56

Table 4-21 EAWLS results with different #population and #hidden neurons heart disease (Cleveland) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	20	43	28	0.9923	20.57	0.9534	45.06
	21	50	33	0.7957	12.15	0.8470	44.92
	22	35	35	0.7953	12.78	0.6617	39.55
	23	43	37	0.7085	19.05	0.4733	38.63
	24	44	37	0.5454	15.68	0.3082	38.28
30	20	48	28	0.9367	11.20	0.9321	44.75
	21	45	32	0.9329	17.84	0.8441	43.73
	22	49	34	0.8913	11.56	0.6767	42.59
	23	41	41	0.6572	18.59	0.2400	39.79
	24	42	42	0.5922	18.05	0.1138	37.38
40	20	45	29	0.9273	16.32	0.7891	44.47
	21	50	29	0.8682	13.59	0.6674	41.26
	22	39	33	0.6719	19.07	0.6557	38.77
	23	42	37	0.5527	13.99	0.5957	37.80
	24	49	40	0.5221	13.07	0.5637	36.87
50	20	39	30	0.9005	15.28	0.7496	43.81
	21	42	32	0.8871	15.01	0.7017	40.27
	22	45	33	0.7433	11.44	0.3856	39.74
	23	49	33	0.5985	11.44	0.3460	37.45
	24	47	43	0.5412	12.18	0.2797	37.06
60	20	43	31	0.8469	15.03	0.7018	43.54
	21	44	32	0.7737	11.44	0.6092	43.47
	22	49	32	0.6041	14.51	0.3638	40.98
	23	47	33	0.5990	19.35	0.2787	39.67
	24	49	41	0.5492	14.00	0.2637	38.01

Table 4-22 EAWLS results with different #population and #hidden heart disease (Hungary) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	20	48	29	0.9418	14.09	0.9991	44.47
	21	37	33	0.9081	43.66	0.9942	41.14
	22	38	37	0.7921	42.21	0.8132	39.97
	23	36	42	0.6870	40.90	0.6703	36.30
	24	43	44	0.5265	44.62	0.5762	35.99
30	20	40	29	0.8756	35.43	0.9691	44.21
	21	38	33	0.7224	38.69	0.9329	40.42
	22	49	35	0.6467	42.70	0.9010	40.26
	23	46	36	0.4471	40.81	0.6669	38.41
	24	38	41	0.4028	43.23	0.5738	36.08

40	20	37	30	0.8269	32.90	0.9678	43.74
	21	45	33	0.8009	39.76	0.8872	42.97
	22	45	33	0.7939	38.50	0.8313	39.58
	23	46	37	0.6065	40.46	0.6254	38.73
	24	37	44	0.2295	40.22	0.6167	36.70
50	20	36	30	0.7160	35.34	0.9457	43.16
	21	37	30	0.6818	35.67	0.8716	41.01
	22	45	34	0.4572	38.49	0.6410	38.52
	23	49	35	0.4469	35.25	0.2837	37.61
	24	42	37	0.3079	31.94	0.1165	36.15
60	20	45	31	0.6312	32.54	0.7365	41.35
	21	48	32	0.5670	41.88	0.7071	40.02
	22	43	35	0.5454	38.26	0.5441	38.32
	23	49	37	0.5312	38.01	0.4284	37.02
	24	44	41	0.2562	36.81	0.4065	35.94

Table 4-23 EAWLS results with different #population and #hidden heart disease (Switzerland) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	10	49	17	0.9959	32.21	0.9903	44.56
	11	46	20	0.8702	45.48	0.7143	43.73
	12	49	25	0.7237	44.52	0.4040	38.28
	13	36	26	0.6329	35.20	0.3471	38.19
	14	37	27	0.4939	32.71	0.3407	38.17
30	10	45	19	0.9910	21.60	0.9839	44.11
	11	47	20	0.9664	41.99	0.8114	43.16
	12	48	20	0.9027	36.68	0.7849	41.04
	13	44	25	0.7994	34.16	0.6069	39.44
	14	35	25	0.4984	27.01	0.4510	39.06
40	10	43	19	0.9829	27.59	0.8818	44.11
	11	38	20	0.7558	41.36	0.8181	42.76
	12	37	23	0.6404	38.51	0.5157	41.58
	13	38	24	0.6002	33.56	0.4092	41.01
	14	43	26	0.2017	27.83	0.3143	40.60
50	10	43	20	0.9432	28.56	0.8790	44.09
	11	39	25	0.8051	25.94	0.6566	40.32
	12	43	27	0.4830	29.47	0.6482	39.69
	13	47	27	0.4535	21.88	0.6289	38.22
	14	46	27	0.3732	20.40	0.5973	36.87
60	10	38	22	0.8007	18.79	0.4942	43.92
	11	35	23	0.7096	27.37	0.4544	43.87
	12	42	24	0.6697	17.31	0.2569	43.87
	13	36	24	0.6576	23.76	0.2482	42.95
	14	47	26	0.6381	14.98	0.2064	39.89

4.2.1.1.4 Results for GALS-T1

In the following 4 tables (Tables 4-24 to 4-27) the results from the GALS-T1 are given. The classification and RMS error are reported with variations of number of hidden neurons and number of generations. The algorithm is not able to run for parity and handwriting character problems, because of the memory complexity. It is found that with T1 connection the maximum number of data patterns for the CEDAR data set, that could be tested with was 250.

Table 4-24 GALS-T1 results with different #population and #hidden neurons for XOR dataset

#population	#hidden neurons	#generations	Time (seconds)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	2	21	1	0.4748	0.00	X	X
	3	25	1.6	0.4776	25.00	X	X
	4	18	1.8	0.4482	25.00	X	X
	5	21	2.5	0.4377	0.00	X	X
	6	25	2.9	0.4347	25.00	X	X
30	2	29	1.1	0.4426	25.00	X	X
	3	26	1.3	0.4737	0.00	X	X
	4	28	2.4	0.4489	25.00	X	X
	5	18	2.8	0.440	0.00	X	X
	6	27	2.9	0.4400	25.00	X	X
40	2	23	0.8	0.4371	25.00	X	X
	3	24	1.4	0.4352	0.00	X	X
	4	21	2.1	0.4327	25.00	X	X
	5	17	2.8	0.4695	0.00	X	X
	6	15	2.8	0.49	0.00	X	X
50	2	21	1.6	0.4778	25.00	X	X
	3	18	2.1	0.4596	0.00	X	X
	4	27	2.3	0.4422	25.00	X	X
	5	23	2.6	0.4773	0.00	X	X
	6	28	2.8	0.4716	25.00	X	X
60	2	28	1	0.4719	0.00	X	X
	3	19	1.8	0.4381	25.00	X	X
	4	24	1.9	0.4370	0.00	X	X
	5	21	2.1	0.4674	0.00	X	X
	6	25	2.8	0.4483	0.00	X	X

Table 4-25 GALS-T1 results with different #population and #hidden neurons heart disease (Cleveland) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	20	18	24	0.9344	31.21	0.9649	44.93
	21	18	29	0.9071	34.86	0.8508	42.27
	22	17	31	0.7747	34.00	0.2924	41.00
	23	19	32	0.6797	27.37	0.2087	40.09

	24	19	33	0.5955	21.12	0.1749	36.48
30	20	17	25	0.8802	32.74	0.8534	44.92
	21	20	25	0.8396	31.52	0.8117	43.23
	22	17	29	0.8078	27.30	0.5435	42.86
	23	19	32	0.7380	25.45	0.4678	41.20
	24	15	34	0.5529	28.73	0.1967	38.58
40	20	18	25	0.8472	27.71	0.8404	43.43
	21	16	28	0.7782	26.79	0.7063	41.48
	22	19	30	0.6938	29.79	0.4811	41.21
	23	18	37	0.5856	22.07	0.4467	40.82
	24	19	37	0.5510	25.52	0.2243	38.89
50	20	17	26	0.8172	32.13	0.7374	40.58
	21	19	28	0.7071	28.27	0.5886	39.93
	22	18	29	0.6258	27.82	0.4302	39.56
	23	16	29	0.5503	28.63	0.4140	38.62
	24	16	39	0.5344	26.44	0.3834	36.23
60	20	18	27	0.7596	31.87	0.6896	39.72
	21	16	28	0.6646	31.98	0.5398	39.68
	22	18	28	0.5764	20.79	0.4982	39.39
	23	18	29	0.4898	22.61	0.3923	37.03
	24	16	36	0.4470	26.22	0.3261	36.25

Table 4-26 GALS-T1 results with different #population and #hidden heart disease (Hungary) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	20	18	26	0.8446	59.08	1.0401	45.10
	21	17	29	0.7743	54.49	1.0204	41.04
	22	19	31	0.5068	48.05	0.7179	40.47
	23	17	32	0.4472	48.80	0.5567	38.80
	24	17	36	0.3850	44.58	0.4024	38.05
30	20	17	26	0.8131	50.75	0.9504	44.55
	21	18	29	0.6708	50.36	0.8103	44.54
	22	17	30	0.6241	57.35	0.7532	44.00
	23	16	33	0.5080	51.10	0.6374	39.46
	24	18	38	0.3471	51.42	0.5086	36.44
40	20	18	26	0.7306	52.81	0.9419	44.51
	21	17	29	0.5748	56.36	0.8198	43.68
	22	18	33	0.4984	54.23	0.7904	42.38
	23	16	36	0.4091	50.64	0.5587	41.13
	24	20	39	0.3384	48.29	0.5078	38.50
50	20	17	26	0.6783	53.08	0.7719	43.26
	21	16	27	0.5656	55.02	0.6252	38.51
	22	17	30	0.4488	47.83	0.4469	38.03
	23	16	31	0.4217	45.93	0.3783	37.72
	24	15	32	0.2573	43.43	0.2687	37.67
60	20	20	27	0.6730	57.19	0.6951	41.96
	21	16	28	0.5829	55.26	0.6553	40.81
	22	17	31	0.4783	45.50	0.5792	39.66
	23	18	33	0.4269	48.21	0.5548	38.35
	24	15	37	0.2260	44.69	0.5128	37.48

Table 4-27 GALS-T1 results with different #population and #hidden heart disease (Switzerland) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	10	15	14	0.8753	57.53	1.0133	44.43
	11	16	17	0.7722	53.34	1.0080	43.92
	12	18	21	0.6708	46.07	0.8015	39.29
	13	20	23	0.5041	44.41	0.6481	38.26
	14	18	23	0.4008	37.43	0.6352	38.07
30	10	20	16	0.8429	54.88	0.9424	44.21
	11	15	18	0.7832	52.75	0.8912	43.55
	12	20	18	0.6401	49.54	0.7523	43.07
	13	18	21	0.4437	39.46	0.7352	40.07
	14	16	22	0.3674	35.95	0.4218	36.55
40	10	20	17	0.7786	56.69	0.9335	42.15
	11	19	17	0.6660	47.16	0.9126	39.99
	12	16	20	0.4537	44.72	0.8731	38.81
	13	18	21	0.3685	35.11	0.8452	36.82
	14	16	23	0.2524	37.56	0.5517	36.33
50	10	19	17	0.7303	48.66	0.8989	41.83
	11	16	22	0.5759	52.30	0.7323	41.76
	12	19	24	0.4187	38.19	0.5329	40.91
	13	16	24	0.2768	33.92	0.3006	40.17
	14	15	24	0.1681	35.15	0.2928	38.63
60	10	19	19	0.6504	47.55	0.7664	39.36
	11	20	20	0.5533	40.97	0.7527	38.47
	12	15	21	0.4417	32.29	0.4892	38.03
	13	17	21	0.2758	32.13	0.4859	37.72
	14	20	22	0.1397	27.37	0.2362	37.22

4.2.1.1.5 Results for GALS-T2

In the following 5 tables (Tables 4-28 to 4-32) the results from the GALS-T2 are given. The classification and RMS error are reported with variations of number of hidden neurons and number of generations. The algorithm is not able to run for parity and handwriting character problems, because of the memory complexity. It is found that with T2 connection the maximum number of data patterns for the CEDAR data set, that could be tested with was 450.

Table 4-28 GALS-T2 results with different #population and #hidden neurons for XOR dataset

#population	#hidden neurons	#generations	Time (seconds)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	2	28	1.6	0.45075	0.00	X	X
	3	28	1.7	0.46622	0.00	X	X
	4	28	2.3	0.44799	25.00	X	X
	5	23	2.5	0.47753	0.00	X	X
	6	20	2.7	0.44079	25.00	X	X
30	2	25	1	0.49040	0.00	X	X
	3	22	1.9	0.47106	0.00	X	X
	4	18	2.6	0.48445	0.00	X	X
	5	26	2.6	0.48991	25.00	X	X
	6	28	3	0.43702	25.00	X	X
40	2	27	1.4	0.43867	0.00	X	X
	3	21	1.8	0.46433	25.00	X	X
	4	17	2.2	0.46581	25.00	X	X
	5	19	2.5	0.46954	25.00	X	X
	6	23	2.6	0.47082	25.00	X	X
50	2	29	1.1	0.46140	25.00	X	X
	3	25	1.2	0.49257	25.00	X	X
	4	22	1.4	0.46516	25.00	X	X
	5	22	2.1	0.48491	0.00	X	X
	6	20	2.6	0.43851	25.00	X	X
60	2	17	1.4	0.45075	0.00	X	X
	3	19	1.5	0.46623	0.00	X	X
	4	17	1.6	0.44799	25.00	X	X
	5	28	1.8	0.47753	0.00	X	X
	6	28	2.3	0.44079	25.00	X	X

Table 4-29 GALS-T2 results with different #population and #hidden neurons for breast cancer (Wisconsin) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	20	18	24	0.9635	35.08	0.9555	45.12
	21	22	30	0.5846	30.33	0.7840	44.97
	22	21	30	0.4344	28.41	0.5630	39.79
	23	18	31	0.3942	31.27	0.4104	38.91
	24	19	38	0.0742	35.46	0.2694	38.29
30	20	26	27	0.9423	34.58	0.9543	44.16
	21	20	31	0.8599	40.82	0.8321	39.15
	22	22	33	0.8099	33.61	0.6793	37.59
	23	25	36	0.4406	27.83	0.3772	36.33
	24	16	38	0.2036	22.35	0.3089	36.33
40	20	27	30	0.8920	33.78	0.8413	43.52
	21	22	33	0.7606	34.51	0.6126	42.15
	22	29	34	0.5965	25.60	0.6050	39.68
	23	24	34	0.5417	29.13	0.5878	37.46
	24	28	36	0.4686	31.07	0.5704	37.07
50	20	21	31	0.8060	30.03	0.7698	42.14
	21	18	32	0.7323	29.67	0.4846	40.39

	22	20	34	0.7167	20.46	0.3672	40.02
	23	29	36	0.6959	15.29	0.2740	38.83
	24	22	37	0.6231	20.01	0.1843	36.93
60	20	21	32	0.7057	18.83	0.7531	41.69
	21	28	34	0.6751	28.08	0.4589	40.26
	22	23	35	0.5317	22.25	0.4150	39.78
	23	18	36	0.4976	17.66	0.2142	39.15
	24	22	38	0.3760	18.39	0.1986	38.24

Table 4-30 GALS-T2 results with different #population and #hidden neurons heart disease (Cleveland) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	20	17	24	0.9320	34.59	0.9856	44.24
	21	17	29	0.8782	31.55	0.8629	42.92
	22	16	30	0.7805	34.22	0.3030	42.60
	23	18	32	0.6550	29.15	0.2009	37.49
	24	16	33	0.5987	20.83	0.1879	37.35
30	20	15	25	0.8853	35.50	0.8479	44.04
	21	16	25	0.8618	33.16	0.7983	43.15
	22	17	28	0.7962	25.31	0.5335	42.76
	23	19	32	0.7170	26.12	0.4650	41.27
	24	16	34	0.5405	21.15	0.1755	40.62
40	20	16	25	0.8568	30.41	0.8347	43.92
	21	17	28	0.7769	25.80	0.6928	41.50
	22	17	30	0.6733	27.88	0.4624	39.29
	23	15	36	0.5922	22.52	0.4603	38.12
	24	18	37	0.5387	20.79	0.2343	37.51
50	20	17	26	0.8254	34.69	0.7363	43.74
	21	18	28	0.7127	25.93	0.5865	43.09
	22	19	29	0.6386	23.09	0.4470	41.57
	23	17	29	0.5529	26.54	0.4208	40.08
	24	18	38	0.5294	21.98	0.4080	38.22
60	20	16	27	0.7572	34.83	0.6877	42.11
	21	18	28	0.6906	30.51	0.5594	40.73
	22	20	28	0.5632	23.96	0.4854	38.42
	23	19	29	0.4860	21.40	0.3948	38.32
	24	15	36	0.4246	18.48	0.3273	37.82

Table 4-31 GALS-T2 results with different #population and #hidden heart disease (Hungary) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	20	17	26	0.8482	52.96	1.0348	44.93
	21	16	28	0.7649	52.27	1.0235	38.93
	22	18	30	0.4982	56.80	0.7069	38.83
	23	19	33	0.4572	47.61	0.5578	38.54
	24	18	38	0.3861	52.02	0.4132	38.17
30	20	20	26	0.7959	51.87	0.9537	44.92

	21	19	29	0.6677	53.09	0.7972	44.62
	22	18	33	0.6256	49.50	0.7410	44.33
	23	16	37	0.5217	54.29	0.6451	38.26
	24	18	38	0.3361	43.10	0.4901	36.23
40	20	19	26	0.7210	54.58	0.9285	44.76
	21	17	29	0.5735	55.74	0.8270	38.99
	22	16	31	0.5079	46.57	0.7948	38.47
	23	17	32	0.3989	46.36	0.5524	36.85
	24	17	36	0.3280	43.37	0.5102	36.10
50	20	16	26	0.6973	48.70	0.7635	44.52
	21	20	27	0.5613	50.11	0.6111	44.00
	22	19	30	0.4563	45.70	0.4601	40.03
	23	20	31	0.4082	44.75	0.3926	37.97
	24	19	33	0.2734	50.22	0.2623	36.32
60	20	16	27	0.6814	49.89	0.6884	44.45
	21	17	28	0.5831	51.02	0.6440	44.43
	22	20	31	0.4727	48.55	0.5810	43.87
	23	17	32	0.4172	47.62	0.5651	41.87
	24	19	36	0.2242	47.08	0.4953	40.60

Table 4-32 GALS-T2 results with different #population and #hidden heart disease (Switzerland) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	10	16	14	0.8589	49.26	1.0175	44.87
	11	15	17	0.7801	54.73	1.0090	43.43
	12	19	21	0.6734	49.69	0.8245	43.31
	13	20	23	0.4933	41.88	0.6533	42.35
	14	15	23	0.3989	43.20	0.6485	37.79
30	10	20	16	0.8313	53.56	0.9418	44.46
	11	16	17	0.7619	49.51	0.8893	43.76
	12	17	17	0.6342	50.44	0.7415	40.54
	13	15	21	0.4446	45.44	0.7401	38.99
	14	19	22	0.3740	36.76	0.4185	37.91
40	10	15	17	0.7598	49.15	0.9328	44.22
	11	18	17	0.6694	43.51	0.9215	38.46
	12	20	20	0.4419	43.20	0.8656	38.27
	13	18	21	0.3812	42.67	0.8369	37.72
	14	15	23	0.2657	37.44	0.5507	37.25
50	10	18	17	0.7178	51.59	0.8959	43.46
	11	16	22	0.5792	43.79	0.7413	42.80
	12	15	24	0.4281	42.11	0.5298	42.24
	13	18	24	0.2813	40.43	0.3072	38.27
	14	19	24	0.1761	31.89	0.2827	36.09
60	10	15	19	0.6529	49.65	0.7833	42.98
	11	18	20	0.5574	39.39	0.7571	42.31
	12	17	21	0.4416	35.49	0.5040	39.81
	13	18	21	0.2900	37.51	0.4738	37.43
	14	15	22	0.1654	25.30	0.2441	36.24

4.2.1.1.6 Results for GALS-T3

In the following 7 tables (Tables 4-33 to 4-39) the results from the GALS-T3 are given. The classification and RMS error are reported with variations of number of hidden neurons and number of generations.

Table 4-33 GALS-T3 results with different #population and #hidden neurons for XOR dataset

#population	#hidden neurons	#generations	Time (seconds)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	2	19	0.8	0.4416	25.00	X	X
	3	19	1.1	0.4661	25.00	X	X
	4	26	1.7	0.4417	25.00	X	X
	5	18	2	0.4697	25.00	X	X
	6	18	2.5	0.4907	0.00	X	X
30	2	24	1	0.4320	25.00	X	X
	3	15	1.5	0.4773	0.00	X	X
	4	21	1.7	0.4464	25.00	X	X
	5	16	2.1	0.4673	25.00	X	X
	6	22	2.2	0.4387	25.00	X	X
40	2	29	1	0.4574	25.00	X	X
	3	21	2	0.4496	0.00	X	X
	4	20	2.1	0.4417	0.00	X	X
	5	22	2.8	0.4689	25.00	X	X
	6	23	2.9	0.4819	25.00	X	X
50	2	17	1.2	0.4877	25.00	X	X
	3	22	1.8	0.4598	25.00	X	X
	4	25	2.4	0.4562	25.00	X	X
	5	20	2.6	0.4465	0.00	X	X
	6	21	2.6	0.4355	0.00	X	X
60	2	17	0.8	0.4416	25.00	X	X
	3	24	2.5	0.4661	25.00	X	X
	4	19	2.5	0.4417	25.00	X	X
	5	19	2.8	0.4697	25.00	X	X
	6	19	2.8	0.4907	0.00	X	X

Table 4-34 GALS-T3 results with different #population and #hidden neurons for 10 bit odd parity dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	30	17	28	0.8698	39.39	0.8176	40.75
	31	16	37	0.4993	50.52	0.4850	38.43
	32	20	40	0.4495	56.70	0.4225	36.53
	33	17	43	0.3822	53.36	0.3741	33.70
	34	16	45	0.3027	48.06	0.2510	30.16
30	30	20	30	0.7930	53.34	0.7387	40.51
	31	19	32	0.7187	58.66	0.6766	40.51
	32	15	34	0.4376	52.63	0.4429	37.44
	33	18	38	0.3329	45.69	0.2695	31.14
	34	16	40	0.2714	41.80	0.2663	31.12
40	30	18	30	0.7805	52.38	0.7384	38.25

	31	20	34	0.7013	58.62	0.6831	33.83
	32	20	37	0.5116	55.01	0.4685	33.36
	33	18	40	0.4285	48.90	0.4180	33.22
	34	20	44	0.2647	49.01	0.2415	31.46
50	30	15	31	0.6574	55.51	0.6578	38.25
	31	20	38	0.5605	51.54	0.5369	37.84
	32	15	39	0.5178	47.62	0.4863	37.28
	33	18	41	0.2756	43.94	0.2320	33.46
	34	18	44	0.1961	43.44	0.1887	33.29
60	30	19	31	0.6430	57.41	0.5998	34.87
	31	18	34	0.4544	53.00	0.4376	34.28
	32	15	41	0.3986	48.85	0.3336	32.08
	33	17	43	0.2548	48.74	0.2070	30.59
	34	16	47	0.1736	42.53	0.1770	30.48

Table 4-35 GALS-T3 results with different #population and #hidden neurons for handwriting characters (CEDAR) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	30	20	28	0.9865	54.64	0.9461	48.16
	31	18	31	0.8022	52.78	0.8006	46.06
	32	15	44	0.6736	45.07	0.6404	45.89
	33	19	46	0.4893	36.51	0.6370	45.74
	34	20	47	0.4086	41.91	0.5392	42.38
30	30	16	29	0.8768	55.92	0.8954	48.08
	31	19	32	0.6555	42.42	0.7354	47.36
	32	19	36	0.6351	43.37	0.6097	45.83
	33	18	42	0.4726	37.54	0.4313	43.33
	34	20	44	0.3599	38.84	0.3918	42.79
40	30	17	31	0.8273	49.27	0.8244	48.08
	31	16	34	0.6905	41.35	0.6411	47.20
	32	16	38	0.5428	44.90	0.6259	43.61
	33	16	40	0.4475	41.98	0.4455	40.94
	34	19	46	0.3400	36.97	0.3675	40.41
50	30	17	31	0.7176	40.00	0.6120	47.34
	31	17	34	0.5118	39.99	0.4585	45.56
	32	16	40	0.3898	37.33	0.3277	44.22
	33	18	45	0.2482	37.20	0.2030	41.70
	34	19	48	0.1858	34.08	0.1240	40.60
60	30	17	35	0.6912	46.26	0.5743	47.04
	31	18	38	0.6086	40.01	0.5708	45.54
	32	17	40	0.4950	35.42	0.4257	45.41
	33	17	41	0.3323	32.23	0.3173	43.70
	34	16	46	0.3214	27.22	0.2811	43.34

Table 4-36 GALS-T3 results with different #population and #hidden neurons for breast cancer (Wisconsin) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	20	19	23	0.6159	48.64	0.5524	44.95
	21	18	30	0.5069	45.51	0.4302	41.18
	22	18	30	0.3712	46.25	0.4219	41.14
	23	19	30	0.2776	45.18	0.2560	40.00
	24	15	37	0.1422	40.58	0.1177	38.46
30	20	16	27	0.5947	51.97	0.5277	44.51
	21	17	29	0.4881	43.44	0.4519	43.47
	22	16	32	0.4372	44.64	0.3303	42.53
	23	18	35	0.3139	45.52	0.2597	40.12
	24	17	38	0.1360	31.93	0.1255	38.11
40	20	18	30	0.5791	47.48	0.4990	44.48
	21	18	32	0.4899	40.93	0.4504	38.72
	22	18	33	0.4325	40.81	0.3937	38.16
	23	17	34	0.2649	37.31	0.2516	37.51
	24	20	35	0.1258	38.34	0.1237	37.37
50	20	16	30	0.5368	48.18	0.4620	43.89
	21	18	32	0.4294	40.82	0.3675	43.65
	22	16	34	0.2414	38.55	0.2351	43.39
	23	17	36	0.1845	36.98	0.1323	39.80
	24	19	36	0.1335	37.91	0.1079	39.78
60	20	15	32	0.5021	38.03	0.4417	43.70
	21	19	33	0.4270	41.00	0.3614	43.52
	22	16	34	0.2685	42.05	0.2119	39.48
	23	20	35	0.1597	29.21	0.1308	37.31
	24	19	37	0.1138	27.99	0.0665	36.87

Table 4-37 GALS-T3 results with different #population and #hidden neurons heart disease (Cleveland) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	20	16	24	0.9326	36.24	0.9867	45.07
	21	19	28	0.9003	33.10	0.8600	44.83
	22	16	30	0.7852	28.89	0.3003	41.00
	23	18	31	0.6540	30.83	0.2165	38.09
	24	20	32	0.5955	24.88	0.1898	36.99
30	20	18	24	0.8845	28.15	0.8530	44.73
	21	16	25	0.8412	33.32	0.8145	43.58
	22	18	29	0.8067	25.45	0.5478	39.19
	23	16	30	0.7158	26.97	0.4622	36.66
	24	16	34	0.5387	22.28	0.1929	35.94
40	20	17	25	0.8559	33.80	0.8454	44.65
	21	18	27	0.7896	31.37	0.6906	43.77
	22	19	30	0.6939	27.35	0.4651	41.22
	23	16	35	0.6016	23.89	0.4493	38.06
	24	17	35	0.5580	24.05	0.2280	37.92
50	20	16	25	0.8285	35.29	0.7250	44.58
	21	17	27	0.7070	25.48	0.5965	41.43
	22	20	28	0.6247	25.95	0.4442	40.51
	23	19	29	0.5491	29.14	0.4095	39.18
	24	16	37	0.5308	20.54	0.4008	37.46

60	20	16	26	0.7529	26.63	0.6852	43.04
	21	16	28	0.6791	26.47	0.5394	42.48
	22	19	28	0.5893	21.43	0.4796	40.63
	23	18	28	0.4812	27.57	0.3690	38.66
	24	20	35	0.4349	24.56	0.3070	37.69

Table 4-38 GALS-T3 results with different #population and #hidden heart disease (Hungary) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	20	20	25	0.8453	53.50	1.0255	45.09
	21	15	29	0.7628	56.70	1.0195	43.02
	22	17	30	0.4852	48.63	0.7025	41.39
	23	20	33	0.4482	55.80	0.5500	39.86
	24	17	37	0.3913	48.43	0.4083	37.75
30	20	15	25	0.7853	50.53	0.9532	44.90
	21	16	28	0.6783	53.85	0.8156	43.56
	22	16	30	0.6341	57.57	0.7386	40.03
	23	19	31	0.5160	53.88	0.6363	39.76
	24	19	35	0.3492	49.63	0.4832	38.46
40	20	19	25	0.7122	54.14	0.9332	44.85
	21	17	26	0.5753	48.89	0.8100	43.76
	22	18	29	0.4946	50.13	0.7967	41.27
	23	19	30	0.4087	52.81	0.5459	40.23
	24	19	32	0.3450	50.82	0.5147	38.02
50	20	18	26	0.7005	56.52	0.7621	44.50
	21	19	28	0.5665	54.35	0.6015	42.58
	22	16	33	0.4500	54.76	0.4640	41.06
	23	18	35	0.4271	50.32	0.3990	39.68
	24	19	38	0.2540	42.75	0.2708	36.58
60	20	18	26	0.6710	56.96	0.6959	44.30
	21	18	28	0.5781	53.81	0.6312	37.65
	22	16	30	0.4533	53.13	0.5776	37.29
	23	16	32	0.4323	50.67	0.5714	36.45
	24	17	36	0.2317	40.60	0.5177	36.22

Table 4-39 GALS-T3 results with different #population and #hidden heart disease (Switzerland) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	10	17	14	0.8475	55.54	1.0248	43.96
	11	18	17	0.7976	52.58	0.9931	41.53
	12	15	21	0.6831	46.88	0.8045	38.88
	13	17	22	0.4827	49.19	0.6619	36.76
	14	19	22	0.4202	42.55	0.6527	36.48
30	10	19	16	0.8267	50.81	0.9337	43.57
	11	17	17	0.7640	47.48	0.8904	42.85
	12	16	17	0.6582	43.99	0.7539	41.93
	13	17	21	0.4425	43.09	0.7500	39.18
	14	17	21	0.3575	43.44	0.4346	39.15
40	10	19	17	0.7644	52.57	0.9245	43.04

	11	16	22	0.6552	51.76	0.9116	41.78
	12	16	23	0.4430	38.55	0.8764	40.14
	13	18	23	0.3786	38.85	0.8612	37.34
	14	18	24	0.2438	37.42	0.5556	36.77
50	10	17	17	0.7263	48.17	0.8896	42.74
	11	17	17	0.5632	51.28	0.7378	41.14
	12	18	20	0.4295	39.47	0.5410	38.63
	13	19	21	0.2697	33.41	0.2839	38.52
	14	15	23	0.1695	35.55	0.2796	37.19
60	10	17	19	0.6623	42.89	0.7839	40.61
	11	18	20	0.5409	39.11	0.7496	38.64
	12	18	20	0.4605	32.41	0.4819	38.45
	13	16	21	0.2759	36.68	0.4777	38.21
	14	17	23	0.1608	31.63	0.2371	37.06

4.2.1.1.7 Results for EALS-T1

In the following 5 tables (Tables 4-40 to 4-44) the results from the EALS-T1 are given. The classification and RMS error are reported with variations of number of hidden neurons and number of generations. The algorithm is not able to run for parity and handwriting character problems, because of the memory complexity.

Table 4-40 EALS-T1 results with different #population and #hidden neurons for XOR dataset

#population	#hidden neurons	#generations	Time (seconds)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	2	23	225	0.4416	25.00	X	X
	3	23	238	0.4661	25.00	X	X
	4	16	282	0.4417	25.00	X	X
	5	29	345	0.4697	25.00	X	X
	6	20	642	0.4907	0.00	X	X
30	2	29	233	0.4320	25.00	X	X
	3	17	238	0.4773	0.00	X	X
	4	25	423	0.4464	25.00	X	X
	5	18	700	0.4673	25.00	X	X
	6	23	743	0.4387	25.00	X	X
40	2	19	188	0.4574	25.00	X	X
	3	17	293	0.4496	0.00	X	X
	4	23	350	0.4417	0.00	X	X
	5	19	366	0.4689	25.00	X	X
	6	21	409	0.4819	25.00	X	X
50	2	23	277	0.4877	25.00	X	X
	3	22	293	0.4598	25.00	X	X
	4	28	357	0.4562	25.00	X	X
	5	19	446	0.4465	0.00	X	X

	6	17	481	0.4355	0.00	X	X
60	2	28	294	0.4416	25.00	X	X
	3	20	391	0.4661	25.00	X	X
	4	17	563	0.4417	25.00	X	X
	5	23	658	0.4697	25.00	X	X
	6	23	678	0.4907	0.00	X	X

Table 4-41 EALS-T1 results with different #population and #hidden neurons breast cancer (Wisconsin) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	20	29	24	0.9829	44.02	0.9290	20.82
	21	21	30	0.8232	47.30	0.7904	19.79
	22	21	30	0.6751	42.72	0.6419	18.35
	23	21	30	0.4798	37.52	0.6342	18.11
	24	21	37	0.3925	33.19	0.5439	17.53
30	20	24	27	0.8798	45.26	0.8708	20.81
	21	22	30	0.6439	37.81	0.7338	20.71
	22	23	32	0.6423	37.54	0.6260	18.95
	23	16	35	0.4559	36.93	0.4321	17.14
	24	25	38	0.3662	26.71	0.3886	16.68
40	20	27	30	0.8314	49.58	0.8274	18.47
	21	24	33	0.6754	36.33	0.6416	16.79
	22	24	33	0.5467	33.41	0.6207	16.15
	23	23	34	0.4558	33.25	0.4370	15.32
	24	17	36	0.3446	33.88	0.3514	15.22
50	20	25	31	0.6905	39.37	0.6304	17.99
	21	29	33	0.5028	37.15	0.4465	17.83
	22	19	35	0.3727	39.97	0.3529	17.79
	23	16	35	0.2658	25.99	0.2201	17.63
	24	17	37	0.1836	29.30	0.1268	16.82
60	20	21	33	0.6811	40.18	0.5919	17.94
	21	23	33	0.6149	30.83	0.5644	16.10
	22	15	35	0.4871	33.63	0.4296	16.00
	23	29	35	0.3447	21.28	0.3130	15.42
	24	21	37	0.3194	24.79	0.2921	15.39

Table 4-42 EALS-T1 results with different #population and #hidden neurons heart disease (Cleveland) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	20	17	24	0.9006	21.49	0.9435	20.35
	21	18	28	0.8830	25.24	0.8244	20.05
	22	15	31	0.7334	27.06	0.2720	17.30
	23	16	31	0.6523	17.39	0.1611	17.11
	24	18	32	0.5678	15.04	0.1429	16.31
30	20	16	25	0.8553	24.17	0.8227	19.53
	21	19	25	0.8078	21.99	0.7660	18.38
	22	18	29	0.7738	22.14	0.4947	18.17
	23	18	32	0.7068	19.08	0.4255	16.66
	24	18	34	0.5201	21.60	0.1693	15.38
40	20	16	25	0.8204	21.79	0.8068	19.09

	21	18	27	0.7574	19.24	0.6575	17.91
	22	17	29	0.6585	23.01	0.4518	17.47
	23	18	35	0.5516	12.30	0.3970	16.38
	24	16	35	0.5099	15.81	0.1866	15.18
50	20	20	26	0.7755	22.31	0.6922	18.66
	21	17	28	0.6599	19.82	0.5685	16.95
	22	17	28	0.6004	20.88	0.3826	16.93
	23	16	28	0.5156	21.25	0.3672	16.06
	24	18	38	0.4943	20.52	0.3504	15.55
60	20	16	27	0.7253	26.75	0.6437	18.34
	21	15	27	0.6264	26.08	0.5046	17.77
	22	17	28	0.5351	13.39	0.4761	16.65
	23	18	28	0.4572	14.17	0.3579	16.19
	24	18	36	0.4145	18.01	0.3005	16.13

Table 4-43 EALS-T1 results with different #population and #hidden heart disease (Hungary) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	20	16	25	0.8134	51.58	0.6707	20.88
	21	16	28	0.7497	45.10	0.5180	19.22
	22	20	30	0.4688	38.80	1.0168	18.60
	23	17	31	0.4056	40.09	0.3661	17.14
	24	16	35	0.3593	37.71	0.9704	16.87
30	20	15	25	0.7753	41.55	0.9082	20.52
	21	20	27	0.6350	41.38	0.7904	18.59
	22	20	29	0.5878	51.61	0.7585	17.14
	23	19	31	0.4656	44.51	0.5222	15.93
	24	17	32	0.3036	46.19	0.4821	15.58
40	20	19	25	0.7090	43.92	0.7167	20.28
	21	18	29	0.5513	49.40	0.4747	18.31
	22	16	33	0.4579	44.98	0.7828	18.03
	23	18	36	0.3775	41.68	0.6104	16.04
	24	17	38	0.3013	39.53	0.9013	15.93
50	20	15	26	0.6512	46.34	0.6223	19.72
	21	18	29	0.5347	49.46	0.5546	18.95
	22	18	29	0.4208	42.58	0.5278	17.20
	23	17	32	0.3834	39.35	0.4680	16.85
	24	19	37	0.2087	37.99	0.6596	16.12
60	20	17	27	0.6353	50.22	0.7329	19.20
	21	17	28	0.5408	45.54	0.5911	19.01
	22	18	30	0.4367	37.27	0.4051	17.33
	23	19	32	0.3829	38.41	0.3511	17.05
	24	18	37	0.1866	36.29	0.2396	16.58

Table 4-44 EALS-T1 results with different #population and #hidden heart disease (Switzerland) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	10	16	14	0.8307	48.88	0.9850	20.34
	11	16	17	0.7468	47.30	0.9674	16.53
	12	19	21	0.6252	40.66	0.7663	16.27
	13	19	23	0.4744	39.41	0.6020	15.93
	14	17	23	0.3713	27.62	0.6019	15.78
30	10	17	16	0.7942	45.38	0.9112	20.12
	11	18	17	0.7470	43.25	0.8667	18.85
	12	17	17	0.6064	40.32	0.7261	18.66
	13	19	21	0.3966	32.81	0.6923	18.09
	14	19	22	0.3411	26.15	0.3803	15.65
40	10	19	17	0.7472	51.53	0.8881	19.62
	11	19	22	0.6277	39.07	0.8710	19.28
	12	17	23	0.4324	35.05	0.8325	18.25
	13	18	24	0.3226	26.73	0.8127	16.75
	14	17	24	0.2043	28.72	0.5071	16.15
50	10	18	17	0.7019	42.50	0.8512	19.33
	11	17	17	0.5288	46.30	0.6907	18.62
	12	19	20	0.3901	29.10	0.5129	17.00
	13	17	21	0.2425	26.07	0.2551	16.86
	14	16	23	0.1245	28.69	0.2507	16.10
60	10	15	19	0.6221	40.82	0.7396	18.15
	11	15	20	0.5121	34.82	0.7042	18.02
	12	16	21	0.3975	25.54	0.4630	16.23
	13	16	22	0.2412	23.23	0.4583	15.88
	14	17	23	0.1197	21.29	0.1918	15.76

4.2.1.1.8 Results for EALS-T2

In the following 5 tables (Tables 4-45 to 4-49) the results from the EALS-T2 are given. The classification and RMS error are reported with variations of number of hidden neurons and number of generations. The algorithm is not able to run for parity and handwriting character problems, because of the memory complexity.

Table 4-45 EALS-T2 results with different #population and #hidden neurons for XOR dataset

#population	#hidden neurons	#generations	Time (seconds)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	2	28	1.5	0.4622	25.00	X	X
	3	20	1.8	0.4632	0.00	X	X
	4	20	1.9	0.4878	0.00	X	X
	5	27	2.4	0.4846	25.00	X	X
	6	26	2.9	0.4471	25.00	X	X

30	2	17	0.9	0.4619	0.00	X	X
	3	30	1	0.4545	25.00	X	X
	4	16	1.6	0.4372	25.00	X	X
	5	16	1.9	0.4893	0.00	X	X
	6	21	2.9	0.4604	0.00	X	X
40	2	19	1.4	0.4596	25.00	X	X
	3	24	1.8	0.4889	0.00	X	X
	4	19	2.4	0.4519	25.00	X	X
	5	16	2.7	0.4621	25.00	X	X
	6	16	2.9	0.4381	25.00	X	X
50	2	21	0.8	0.4393	0.00	X	X
	3	30	0.9	0.4529	0.00	X	X
	4	22	1.1	0.4767	25.00	X	X
	5	17	1.2	0.4915	25.00	X	X
	6	16	2.3	0.4630	0.00	X	X
60	2	18	1.5	0.4622	25.00	X	X
	3	19	1.6	0.4632	0.00	X	X
	4	22	1.9	0.4878	0.00	X	X
	5	28	2.2	0.4846	25.00	X	X
	6	25	2.9	0.4471	25.00	X	X

Table 4-46 EALS-T2 results with different #population and #hidden neurons for breast cancer (Wisconsin) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	20	20	24	0.9416	25.31	0.9272	20.88
	21	19	28	0.5527	23.84	0.7847	17.84
	22	18	30	0.3892	21.34	0.6591	17.43
	23	17	31	0.3724	25.58	0.3532	16.56
	24	17	37	0.0466	27.47	0.2811	15.72
30	20	20	26	0.8943	25.43	0.9136	20.70
	21	16	31	0.8216	34.07	0.7570	20.17
	22	17	32	0.7690	28.58	0.5394	16.55
	23	20	34	0.4022	20.32	0.3747	16.20
	24	20	38	0.1651	13.80	0.2203	15.17
40	20	16	30	0.8608	24.44	0.7930	20.15
	21	18	32	0.7194	25.47	0.5834	19.89
	22	17	33	0.5522	17.92	0.5670	18.92
	23	16	33	0.5188	19.22	0.5404	17.26
	24	16	35	0.4212	24.22	0.5266	16.48
50	20	18	30	0.7644	22.35	0.7340	20.13
	21	15	32	0.7024	21.21	0.4472	19.76
	22	17	33	0.6898	12.91	0.3401	18.86
	23	17	36	0.6627	6.83	0.2307	16.56
	24	15	36	0.6023	10.70	0.1362	16.51
60	20	17	33	0.6592	10.83	0.7032	18.93
	21	16	34	0.6402	22.54	0.4378	18.31
	22	20	34	0.5092	13.53	0.3769	17.69
	23	18	35	0.4709	10.50	0.1905	16.53
	24	17	36	0.3446	11.33	0.1652	16.33

Table 4-47 EALS-T2 results with different #population and #hidden neurons heart disease (Cleveland) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	20	20	24	0.9059	27.25	0.9378	20.66
	21	18	28	0.8485	24.58	0.8184	18.67
	22	18	30	0.7318	27.44	0.2696	18.48
	23	18	31	0.6268	23.30	0.1563	16.63
	24	17	32	0.5658	15.23	0.1424	16.06
30	20	19	24	0.8623	29.15	0.8113	19.85
	21	16	25	0.8176	25.97	0.6588	19.65
	22	20	28	0.7711	16.63	0.4325	17.68
	23	20	31	0.6727	21.07	0.4232	17.35
	24	17	35	0.5200	11.36	0.1846	16.39
40	20	20	24	0.8100	23.40	0.8070	19.72
	21	19	28	0.7513	18.03	0.7530	19.35
	22	20	29	0.6432	18.94	0.4906	17.74
	23	19	35	0.5643	14.06	0.4292	16.71
	24	19	36	0.5135	15.32	0.1427	16.68
50	20	19	26	0.8015	27.00	0.7073	19.44
	21	16	27	0.6732	19.97	0.5368	19.31
	22	20	28	0.5957	16.94	0.4260	18.36
	23	17	28	0.5242	18.62	0.3964	16.25
	24	20	38	0.4801	12.38	0.3728	15.26
60	20	17	25	0.7206	26.84	0.6395	18.85
	21	18	26	0.6648	20.87	0.5143	18.64
	22	17	27	0.5148	15.93	0.4585	16.48
	23	16	28	0.4567	11.69	0.3533	16.35
	24	17	35	0.3845	12.52	0.3038	15.84

Table 4-48 EALS-T2 results with different #population and #hidden heart disease (Hungary) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	20	19	25	0.8010	43.78	0.9951	20.54
	21	17	28	0.7441	45.23	0.9808	19.93
	22	16	32	0.4551	49.98	0.6732	18.57
	23	16	35	0.4360	39.94	0.5204	16.67
	24	15	37	0.3603	46.81	0.3882	16.58
30	20	16	26	0.7638	46.19	0.9329	20.49
	21	18	28	0.6377	43.44	0.7482	20.36
	22	17	30	0.5845	40.78	0.6923	18.30
	23	16	32	0.4797	47.75	0.6089	17.56
	24	20	38	0.3098	33.34	0.4490	15.86
40	20	19	26	0.6736	45.70	0.8844	19.47
	21	20	26	0.5349	48.73	0.7784	18.64
	22	17	29	0.4091	39.13	0.7550	18.41
	23	19	30	0.3594	40.72	0.5042	18.24
	24	18	32	0.2447	36.27	0.4802	15.84
50	20	20	26	0.6714	40.49	0.7316	18.06
	21	20	28	0.5358	42.08	0.5770	17.37
	22	17	30	0.4798	36.78	0.4284	17.33
	23	19	31	0.3681	35.45	0.3534	17.06

	24	19	35	0.2828	41.25	0.2356	15.71
60	20	17	26	0.6548	44.51	0.6672	17.90
	21	19	28	0.5418	43.82	0.6094	17.14
	22	19	30	0.4297	38.73	0.5480	17.11
	23	19	32	0.3937	41.34	0.5428	16.06
	24	17	35	0.1772	37.79	0.4462	15.41

Table 4-49 EALS-T2 results with different #population and #hidden heart disease (Switzerland) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	10	20	14	0.8126	39.29	0.9946	20.53
	11	19	17	0.7516	48.73	0.9842	18.73
	12	18	20	0.6446	43.77	0.7931	17.96
	13	17	22	0.4608	36.69	0.6329	17.36
	14	17	23	0.3775	33.50	0.6041	16.97
30	10	20	16	0.7995	47.31	0.9008	20.50
	11	16	17	0.7391	43.79	0.8441	20.15
	12	17	17	0.6018	40.55	0.7129	20.04
	13	20	21	0.4117	39.92	0.6994	18.42
	14	20	21	0.3449	30.62	0.3805	15.67
40	10	16	17	0.7145	42.53	0.8899	19.20
	11	16	17	0.6493	33.69	0.8807	18.96
	12	16	20	0.3925	35.26	0.8348	17.91
	13	16	21	0.3543	35.28	0.8048	17.91
	14	16	23	0.2177	28.19	0.5011	16.01
50	10	18	17	0.6701	45.54	0.8521	18.68
	11	15	22	0.5534	34.28	0.7078	17.74
	12	17	23	0.4047	34.61	0.4950	15.86
	13	17	23	0.2426	33.29	0.2750	15.55
	14	15	24	0.1552	24.48	0.2546	15.21
60	10	17	19	0.6187	40.76	0.7456	18.51
	11	16	20	0.5085	32.10	0.7229	17.12
	12	20	20	0.3944	26.00	0.4740	17.12
	13	18	21	0.2406	29.47	0.4256	16.96
	14	17	22	0.1375	17.89	0.2230	15.24

4.2.1.1.9 Results for EALS-T3

In the following 7 tables (Tables 4-50 to 4-56) the results from the GALS-T1 are given. The classification and RMS error are reported with variations of number of hidden neurons and number of generations.

Table 4-50 EALS-T3 results with different #population and #hidden neurons for XOR dataset

#population	#hidden neurons	#generations	Time (seconds)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	2	27	1.4	0.4642	0.00	X	X
	3	18	1.5	0.4293	0.00	X	X
	4	27	1.7	0.4644	0.00	X	X
	5	29	1.8	0.4573	0.00	X	X
	6	20	2.1	0.4176	0.00	X	X
30	2	21	0.8	0.4254	0.00	X	X
	3	22	1.6	0.4677	0.00	X	X
	4	22	2	0.4581	0.00	X	X
	5	17	2.4	0.4258	0.00	X	X
	6	22	2.6	0.4427	0.00	X	X
40	2	17	1.4	0.4600	0.00	X	X
	3	16	1.5	0.4464	0.00	X	X
	4	24	1.8	0.4180	0.00	X	X
	5	20	1.9	0.4235	0.00	X	X
	6	21	2.1	0.4638	0.00	X	X
50	2	29	1.3	0.4723	0.00	X	X
	3	20	1.6	0.4473	0.00	X	X
	4	29	2.7	0.4617	0.00	X	X
	5	22	2.8	0.4547	0.00	X	X
	6	15	3.4	0.4703	0.00	X	X
60	2	27	1.1	0.4642	0.00	X	X
	3	23	1.4	0.4293	0.00	X	X
	4	24	1.4	0.4644	0.00	X	X
	5	27	1.7	0.4573	0.00	X	X
	6	18	2.4	0.4176	0.00	X	X

Table 4-51 EALS-T3 results with different #population and #hidden neurons for 10 bit odd parity dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	30	17	27	0.8864	7.34	0.9230	15.10
	31	20	35	0.8357	5.82	0.8011	13.13
	32	16	38	0.7352	6.33	0.2709	12.22
	33	16	40	0.6528	9.85	0.1672	11.69
	34	19	43	0.5671	8.30	0.1484	11.52
30	30	17	27	0.8569	5.14	0.8200	14.90
	31	19	30	0.7905	6.54	0.7694	14.28
	32	15	32	0.7666	8.22	0.4970	14.11
	33	17	36	0.7151	8.20	0.4460	11.94
	34	20	37	0.5134	9.79	0.1584	11.55
40	30	19	28	0.8100	5.34	0.8194	14.79
	31	17	31	0.7380	9.35	0.6708	14.76
	32	17	39	0.6635	9.67	0.4548	14.28
	33	16	41	0.5366	5.97	0.4237	12.60
	34	16	45	0.5166	6.19	0.1932	11.06
50	30	20	29	0.7898	6.35	0.7037	14.53
	31	19	32	0.6671	5.01	0.5685	13.50
	32	18	35	0.5759	6.72	0.4084	13.03
	33	17	37	0.5031	5.41	0.3927	12.70
	34	15	42	0.5013	9.48	0.3573	12.18

60	30	15	29	0.7284	6.51	0.6601	14.13
	31	17	36	0.6338	7.35	0.5123	12.84
	32	20	37	0.5504	9.77	0.4515	12.20
	33	15	38	0.4607	8.82	0.3685	12.02
	34	17	41	0.4197	9.39	0.2777	11.43

Table 4-52 EALS-T3 results with different #population and #hidden neurons for handwriting characters (CEDAR) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	30	17	27	0.8114	8.93	1.0062	20.92
	31	18	30	0.7287	5.23	0.9862	19.35
	32	18	43	0.4840	5.07	0.6778	16.82
	33	17	44	0.4230	8.78	0.5286	16.34
	34	19	45	0.3527	7.61	0.3650	16.05
30	30	18	28	0.7818	9.69	0.9084	20.76
	31	18	30	0.6243	6.14	0.7853	20.07
	32	16	34	0.5960	7.28	0.7113	18.49
	33	15	40	0.4674	6.20	0.5922	17.14
	34	16	41	0.3174	8.92	0.4807	15.72
40	30	16	29	0.6925	9.79	0.8977	20.16
	31	17	33	0.5407	9.46	0.7831	19.46
	32	17	37	0.4609	5.63	0.7508	18.21
	33	19	37	0.3852	5.02	0.5167	17.38
	34	18	43	0.3114	8.29	0.4809	16.20
0	30	17	31	0.6472	8.08	0.7356	20.10
	31	18	33	0.5457	9.28	0.6039	19.55
	32	18	39	0.4486	5.55	0.4040	18.05
	33	18	43	0.3862	5.49	0.3577	15.92
	34	20	44	0.1990	5.86	0.2365	15.45
60	30	17	34	0.6390	9.25	0.6650	19.64
	31	17	37	0.5197	6.14	0.6247	18.50
	32	17	38	0.4124	5.08	0.5542	17.28
	33	17	38	0.3841	9.07	0.5191	16.20
	34	15	44	0.2176	9.73	0.4716	16.03

Table 4-53 EALS-T3 results with different #population and #hidden neurons for breast cancer (Wisconsin) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	20	19	22	0.6159	48.64	0.9326	14.54
	21	18	28	0.5069	45.51	0.9003	11.57
	22	18	28	0.3712	46.25	0.7852	11.26
	23	19	29	0.2776	45.18	0.6540	10.49
	24	15	35	0.1422	40.58	0.5955	10.29
30	20	16	25	0.5947	51.97	0.8845	14.23
	21	17	29	0.4881	43.44	0.8412	13.81
	22	16	31	0.4372	44.64	0.8067	12.02
	23	18	33	0.3139	45.52	0.7158	11.78
	24	17	36	0.1360	31.93	0.5387	10.58
40	20	18	27	0.5791	47.48	0.8559	13.66

	21	18	31	0.4899	40.93	0.7896	12.65
	22	18	31	0.4325	40.81	0.6939	12.50
	23	17	32	0.2649	37.31	0.6016	10.60
	24	20	34	0.1258	38.34	0.5580	10.46
50	20	16	28	0.5368	48.18	0.8285	13.56
	21	18	31	0.4294	40.82	0.7070	13.32
	22	16	32	0.2414	38.55	0.6247	12.78
	23	17	34	0.1845	36.98	0.5491	12.11
	24	19	35	0.1335	37.91	0.5308	12.01
60	20	15	30	0.5021	38.03	0.7529	13.24
	21	19	32	0.4270	41.00	0.6791	12.47
	22	16	33	0.2685	42.05	0.5893	10.79
	23	20	33	0.1597	29.21	0.4812	10.64
	24	19	35	0.1138	27.99	0.4349	10.38

Table 4-54 EALS-T3 results with different #population and #hidden neurons heart disease (Cleveland) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	20	16	22	0.9326	36.24	0.9867	15.07
	21	19	27	0.9003	33.10	0.8600	14.57
	22	16	29	0.7852	28.89	0.3003	12.78
	23	18	30	0.6540	30.83	0.2165	12.12
	24	20	30	0.5955	24.88	0.1898	10.35
30	20	18	23	0.8845	28.15	0.8530	14.83
	21	16	26	0.8412	33.32	0.8145	13.18
	22	18	28	0.8067	25.45	0.5478	11.64
	23	16	34	0.7158	26.97	0.4622	11.12
	24	16	34	0.5387	22.28	0.1929	10.63
40	20	17	23	0.8559	33.80	0.8454	14.70
	21	18	23	0.7896	31.37	0.6906	13.41
	22	19	26	0.6939	27.35	0.4651	13.38
	23	16	30	0.6016	23.89	0.4493	11.98
	24	17	32	0.5580	24.05	0.2280	11.74
50	20	16	24	0.8285	35.29	0.7250	14.31
	21	17	26	0.7070	25.48	0.5965	13.87
	22	20	26	0.6247	25.95	0.4442	12.77
	23	19	27	0.5491	29.14	0.4095	12.70
	24	16	36	0.5308	20.54	0.4008	12.17
60	20	16	25	0.7529	26.63	0.6852	11.90
	21	16	26	0.6791	26.47	0.5394	11.77
	22	19	26	0.5893	21.43	0.4796	11.00
	23	18	27	0.4812	27.57	0.3690	10.78
	24	20	33	0.4349	24.56	0.3070	10.44

Table 4-55 EALS-T3 results with different #population and #hidden heart disease (Hungary) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	20	17	24	0.8114	8.93	1.0062	14.85
	21	18	25	0.7287	5.23	0.9862	13.77
	22	18	28	0.4840	5.07	0.6778	13.55
	23	17	28	0.4230	8.78	0.5286	13.52
	24	19	30	0.3527	7.61	0.3650	10.59
30	20	18	24	0.7818	9.69	0.9084	14.45
	21	18	27	0.6243	6.14	0.7853	14.08
	22	16	29	0.5960	7.28	0.7113	12.99
	23	15	29	0.4674	6.20	0.5922	12.17
	24	16	33	0.3174	8.92	0.4807	11.74
40	20	16	24	0.6925	9.79	0.8977	13.74
	21	17	27	0.5407	9.46	0.7831	13.45
	22	17	28	0.4609	5.63	0.7508	11.54
	23	19	30	0.3852	5.02	0.5167	11.50
	24	18	36	0.3114	8.29	0.4809	10.72
50	20	17	24	0.6472	8.08	0.7356	13.53
	21	18	27	0.5457	9.28	0.6039	13.41
	22	18	30	0.4486	5.55	0.4040	12.55
	23	18	34	0.3862	5.49	0.3577	11.65
	24	20	36	0.1990	5.86	0.2365	10.42
60	20	17	25	0.6390	9.25	0.6650	13.47
	21	17	26	0.5197	6.14	0.6247	13.33
	22	17	28	0.4124	5.08	0.5542	12.14
	23	17	30	0.3841	9.07	0.5191	11.98
	24	15	34	0.2176	9.73	0.4716	11.06

Table 4-56 EALS-T3 results with different #population and #hidden heart disease (Switzerland) dataset

#population	#hidden neurons	#generations	Time (minutes)	Train		Test	
				RMS	Class (%)	RMS	Class (%)
20	10	19	13	0.8312	8.87	0.9845	15.10
	11	16	16	0.7362	7.78	0.9801	14.87
	12	19	20	0.6310	6.43	0.7655	14.36
	13	17	21	0.4616	7.13	0.6111	11.22
	14	16	21	0.3807	9.41	0.6100	10.44
30	10	17	15	0.8049	8.87	0.8991	14.74
	11	20	16	0.7379	7.71	0.8484	14.45
	12	17	16	0.6024	5.35	0.7149	14.40
	13	16	20	0.4098	8.68	0.7147	14.17
	14	15	20	0.3197	5.08	0.3781	11.65
40	10	19	16	0.7295	7.47	0.8848	14.68
	11	15	16	0.6354	5.40	0.8813	14.47
	12	20	19	0.4292	8.69	0.8317	12.17
	13	16	20	0.3324	5.15	0.8050	11.99
	14	18	22	0.2244	6.86	0.5106	10.32
50	10	16	16	0.7029	5.20	0.8516	13.66
	11	19	20	0.5515	9.59	0.6931	13.56
	12	16	22	0.3935	5.22	0.4985	13.29
	13	17	22	0.2445	9.04	0.2736	13.19

	14	18	22	0.1358	7.25	0.2495	12.34
60	10	19	18	0.6015	7.69	0.7372	12.46
	11	15	19	0.5100	5.68	0.7301	11.99
	12	19	19	0.4093	8.22	0.4617	11.77
	13	18	19	0.2434	6.99	0.4616	11.18
	14	20	20	0.1043	5.05	0.2045	10.59

4.2.1.2 Results with fixed parameters

In this section we report the results from all the algorithms mainly to compare the RMS error, classification error and the time complexity. Based on the performance results from the earlier section we report the results for EBP, EAWLS, LI-EALS-T3 and BT-EALS-T3. The experiments are conducted to obtain the best results in terms of their classification error.

4.2.1.2.1 Results from EBP

Table 4-57 EBP results with fixed parameters

Data set	#hidden neurons	#time[14] (minutes)	Train		Test	
			RMS	Class	RMS	Class
XOR	4	0.66	0.85	0	X	X
10 bit odd parity	34	127	0.1541	15.40	0.0765	8.20
Handwriting characters (CEDAR)	38	238	0.07368	3.00	0.014	18.15
Breast cancer (Wisconsin)	28	121	0.07368	3.50	0.014	11.40
Heart disease (Cleveland)	28	97	0.1742	3.90	0.109	15.01
Heart disease (Hungary)	28	93	0.277091	1.89	0.1283	13.21
Heart disease (Switzerland)	18	71	0.174857	2.00	0.3088	13.31

4.2.1.2.2 Results from EAWLS

Table 4-58 EAWLS results with fixed parameters (population 60)

Data set	#hidden neurons	#time (minutes)	Train		Test	
			RMS	Class	RMS	Class
XOR	5	0.56	0.4368	0.00	X	X
10 bit odd parity	34	147	0.7921	35.43	0.5957	20.76
Handwriting characters (CEDAR)	34	258	0.6178	21.44	0.3471	28.79

[14] The time complexity for the EBP is measured by adding all the time required to obtain the best result in terms of RMS / classification error.

Breast cancer (Wisconsin)	24	125	0.6308	22.90	0.2923	25.06
Heart disease (Cleveland)	22	118	0.6572	14.51	0.6767	21.24
Heart disease (Hungary)	20	113	0.7224	14.09	0.6167	24.90
Heart disease (Switzerland)	14	86	0.4939	14.98	0.6069	25.40

4.2.1.2.3 Results from LI-EALS-T3

Table 4-59 LI-EALS-T3 results with fixed parameters

Data set	#hidden neurons	#time (minutes)	Train		Test	
			RMS	Class	RMS	Class
XOR	3	0.47	0.0015	0	X	X
10 bit odd parity	41	115	0.1272	10	0.0074	5.80
Handwriting characters (CEDAR)	43	219	0.0740	2.80	0.0055	15.16
Breast cancer (Wisconsin)	32	102	0.0159	3.10	0.0113	9.16
Heart disease (Cleveland)	30	85	0.0445	3.50	0.0050	10.52
Heart disease (Hungary)	28	86	0.0349	1.87	0.0018	10.68
Heart disease (Switzerland)	27	56	0.1272	1.90	0.0074	10.28

4.2.1.2.4 Results from BT-EALS-T3

Table 4-60 BT-EALS-T3 results with fixed parameters

Data set	#hidden neurons	#time (minutes)	Train		Test	
			RMS	Class	RMS	Class
XOR	3	0.28	0.0013	0	X	X
10 bit odd parity	39	83	0.1089	10.25	0.0106	7.90
Handwriting characters (CEDAR)	38	141	0.0585	3.12	0.0039	16.75
Breast cancer (Wisconsin)	28	72	0.0299	3.52	0.0137	10.27
Heart disease (Cleveland)	26	63	0.0209	3.72	0.0135	14.74
Heart disease (Hungary)	25	71	0.0386	1.91	0.0010	11.51
Heart disease (Switzerland)	22	34	0.1089	2.12	0.0106	11.53

4.2.2 Experimental results – Type II

In this section, the simulations results are given. Six different type of experiments are conducted and the results for them are reported in this section. Simulation results are conducted to analyze some special characteristics for the EALS-T3 method.

4.2.2.1 Experiment 1

This experiment is conducted to test the effect on the QR factorization method for different initialization of the hidden layer weight matrix and to find the effect on output for the hidden neurons for various range of weight initialization at the start of the evolutionary algorithm. CEDAR handwriting data set is considered as input for this experiment. The length of the pattern is 100. After initializing the hidden layer weights with specific range as closed interval, the output of the hidden neurons are tested. The results are shown below in Table 4–61.

Table 4-61 Experiment 1

Range	Max	Min	Std. dev
0 – 9	1	1	0
0.1 – 0.9	1	0.9979	0.000156
0.01 – 0.09	0.83648	0.64958	0.0337
0.001 – 0.009	0.541417	0.515473	0.00455
0.0001- 0.0009	0.706357	0.5706	0.0237

4.2.2.2 Experiment 2

This experiment is conducted to test the effect on the weights of the output layer for different initialization of the hidden layer weight matrix and to find the effect on weights of the output layers for various range of weight initialization at the start of the evolutionary algorithm. We use handwriting data set as input. The length of the pattern is 100. After initializing the hidden layer weights with specific range as closed interval, the weights of the output layer are tested. 10 hidden neurons are considered. The results are shown below in Table 4–62.

Table 4-62 Experiment 2

Range	Max	Min	Std. dev
0 – 9	NaN[15]	NaN	NaN
0.1 – 0.9	67404.13	-44781.9	6543.086
0.01 – 0.09	57.81	-20.27	12.67
0.001 – 0.009	544.3052	-456.528	135.5742
0.0001- 0.0009	93.505	-70.073	20.199

4.2.2.3 Experiment 3

This experiment is conducted to test the effect on the inverse function of the R matrix from the QR factorization and to find the effect on the calculation of inverse function using the QR decomposition for various distribution of simulated hidden layer output values like a matrix with forcibly various range for its elements. The inverse of R matrix is important to us, because it gives the values of the weights for the output layer. A 10 by 10 matrix is generated by simulation for various ranges to generate different matrix elements. The range is incremented in steps of 0.01. The elements are generated and chosen randomly and the result for the inverse of the R matrix is reported. In the table results are given for increment of 0.1 in every stop for simplicity. The results are shown below in Table 4–63.

Table 4-63 Variation of results for different ranges with matrix element chosen randomly

Range	Max	Min	Std. Dev
0-0.09	NaN	NaN	NaN
0.1-0.19	-1188.94	1156.088	246.1245
0.2-0.29	-436.256	549.1453	118.73
0.3-0.39	-163.114	141.2455	33.627
0.4-0.49	89.907	-148.647	38.486
0.5-0.59	453.13	-457.358	79.521
0.6-0.69	139.896	-184.583	41.901
0.7-0.79	46.561	-43.67	12.38
0.8-0.89	57.89	-49.63	16.58
0.9-0.99	69.559	-66.952	20

[15] NaN denotes not a number

4.2.2.4 Experiment 4

This experiment is conducted to check the effect on the inverse output of the R matrix from the QR decomposition. The main aim is to find out the effect on the elements of the hidden neuron output matrix, depending on the number of elements that are different and the effect on the calculation of inverse function using the QR decomposition for various distribution of simulated hidden layer output values, like a matrix, which is based on the number of elements that are same and other elements that are optionally different. A 10 by 10 matrix is generated by simulation for various ranges and various number of different elements to generate different matrix elements. The elements are generated randomly and the result for the inverse of the R matrix is reported. For simplicity in table results are only given for those rows where there some values are found the remaining rows contains NaN values. The results are shown below in Table 4–64.

Table 4-64 Experiment 4

Number of different elements (row & column)	Min	Max	Std. Dev
9	3.603E+16	3.6E+16	5.37E+15
10	-1.369406	4.485583	0.419016
10	-13.739775	13.099148	2.780696
9	-1.8E+16	1.8E+15	2.69E+15
10	-1.429434	2.809697	0.271014
10	-7.382857	7.758594	2.057559
10	-2.400445	2.945814	0.464327
9	-9.01E+15	9.01E+15	1.34E+09
10	-0.642371	2.404882	0.259341
10	-0.667746	2.279994	0.243792
9	-9E+15	9E+15	1.3E+15
10	-1.392960	2.801738	0.314830
10	-0.242402	2.238973	0.201957
9	-9E+15	9E+15	1.3E+15
10	-67833	85071.7	15568.4

4.2.2.5 Experiment 5

This experiment is conducted to test on the effect of convergence property of the evolutionary algorithm depending on the number of hidden neurons. The main aim is to check the number of generation the evolutionary algorithm takes for convergence and to find out the effect on convergence property of the

evolutionary algorithm depending on the increasing number of hidden neuron for a particular input data set. Only handwriting data set with different training pattern length is considered. The number of generation after which there is no improvement for the best of the population RMS error is reported. The results are shown below in Table 4 – 65.

Table 4-65 Experiment 5

Pattern length	# Hidden neurons	Generation	Best RMS error
100	3	15	0.005954
	4	15	0.006458
	5	14	0.005204
	6	15	0.005336
	7	14	0.004055
	8	12	0.004378
	9	9	0.003444
200	3	14	0.00346
	4	13	0.00341
	5	13	0.00317
	6	13	0.00294
	7	12	0.00289
	8	10	0.00268
	9	10	0.00257
300	3	14	0.00312
	4	14	0.00310
	5	13	0.00297
	6	12	0.00294
	7	11	0.00293
	8	11	0.00286
	9	9	0.00279
400	3	12	0.00386
	4	12	0.00375
	5	11	0.00368
	6	10	0.00355
	7	10	0.00349
	8	9	0.00342
	9	9	0.00339
500	3	12	0.00346
	4	11	0.00337
	5	10	0.00336
	6	10	0.00316
	7	9	0.00327
	8	9	0.00319
	9	8	0.00308
600	3	11	0.00387
	4	11	0.00373
	5	11	0.00369
	6	10	0.00365
	7	10	0.00362
	8	9	0.00358
	9	8	0.00351
700	3	10	0.00357
	4	10	0.00351
	5	10	0.00349

	6	9	0.00351
	7	9	0.00345
	8	8	0.00339
	9	7	0.00346
800	3	9	0.00368
	4	8	0.00358
	5	7	0.00354
	6	7	0.00351
	7	6	0.00349
	8	6	0.00345
	9	6	0.00341
900	3	8	0.00337
	4	8	0.00332
	5	7	0.00328
	6	7	0.00326
	7	6	0.00322
	8	6	0.00319
	9	5	0.00315
1000	3	7	0.00358
	4	7	0.00352
	5	6	0.00351
	6	5	0.00347
	7	5	0.00342
	8	4	0.00337
	9	4	0.00332

4.2.2.6 Experiment 6

The aims of this experiment are to check whether the fitness function of the gene gets highly affected, when its first half characteristic is only considered to call the least square function and to find out the effect of breaking the chromosome for calling the least square solution. CEDAR handwriting data set of pattern length 100 is considered. Next after every generation of the evolutionary algorithm, we calculate the fitness for the population pool, as calculated by our evolutionary algorithm. Then we calculate the fitness of the gene by breaking it into two halves, and taking the first halves and then combining with the output layer weights (by calling the least square function). The comparisons of the fitness values are reported. Only first 10 ranking of the population pool from the set of 50 are considered. For the fitness calculated by the first half of the gene and the least square method, result of those first 10 ranking of the population gene is reported. Also, to verify this properties of the gene, the program is run from the beginning and stop after different generation, so that the same population does not belong to the highest ranking always. The results are shown below in Table 4–66.

Table 4-66 Experiment 6

Generation	**Fitness for full length gene**		**Fitness for half length gene with least square**	
	Population Number	Ranking	Population Number	Ranking
1	10	1	10	2
	11	2	11	1
	9	3	9	3
	33	4	33	5
	37	5	37	7
	1	6	1	4
	41	7	41	6
	36	8	36	8
	17	9	17	9
	19	10	19	14
2	17	1	17	1
	3	2	3	2
	8	3	8	4
	1	4	1	3
	46	5	46	6
	47	6	47	10
	11	7	11	8
	19	8	19	7
	32	9	32	13
	17	10	17	9
3	1	1	1	2
	4	2	4	1
	17	3	17	3
	12	4	12	4
	21	5	21	5
	15	6	15	6
	3	7	3	10
	36	8	36	17
	32	9	32	13
	31	10	31	9

5 ANALYSIS AND DISCUSSION

In this chapter the analysis of all the experimental conducted for the new proposed algorithm are given with a brief discussion at the end of every analysis. The entire analysis can be broadly divided mainly into two sections. In the first section analysis and discussions are given for the comparison purposes of the proposed algorithm with the other existing algorithm. In the second section specific properties of the proposed algorithm are analyzed in details.

5.1 COMPARISON ANALYSIS

The comparison is made mainly on the following properties – classification accuracy (for the testing data set), time and memory complexity, required number of hidden neurons, sensitivity of the result to the initial condition.

5.1.1 Comparison of classification accuracy

The following two figures (Figures 5-1 and 5-2) show the percentage classification accuracy for all the algorithms. To make analysis easier, results from the data set are divided into two different graphs. Only best classification accuracy results from the rest data set are considered.

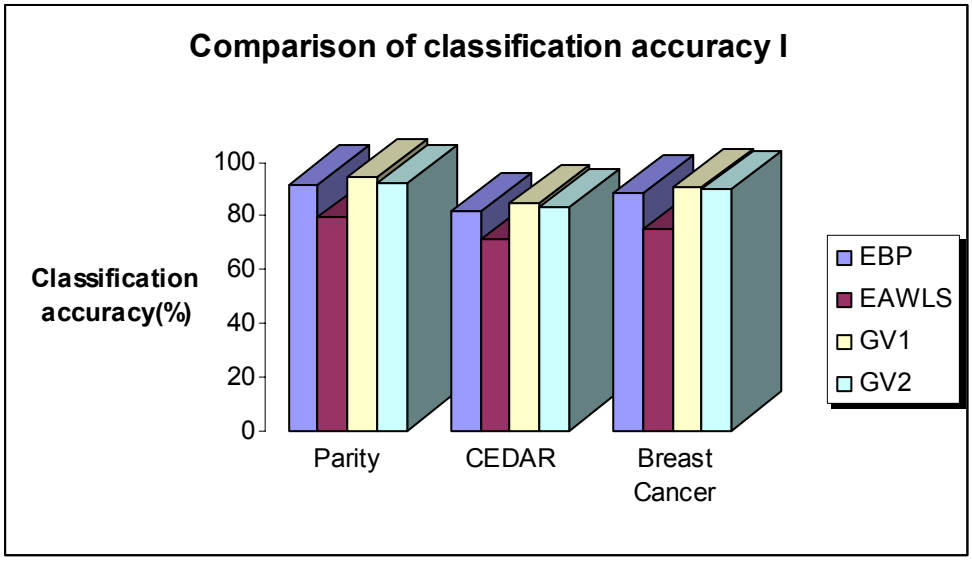

Figure 5-1 Comparison of classification accuracy I

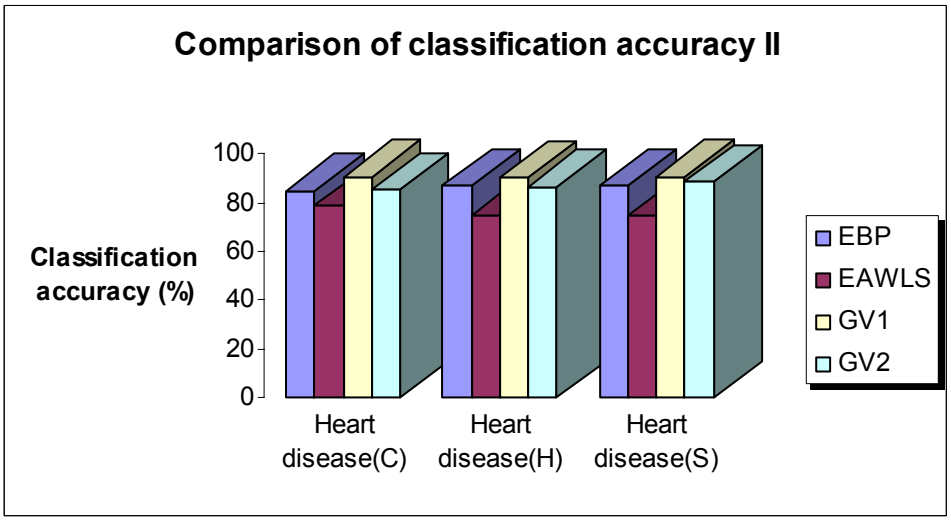

Figure 5-2 Comparison of classification accuracy II

The following two figures (Figures 5-3 and 5-4) show the improvement of test classification accuracy in percentage over the standard EBP and the EAWLS methods.

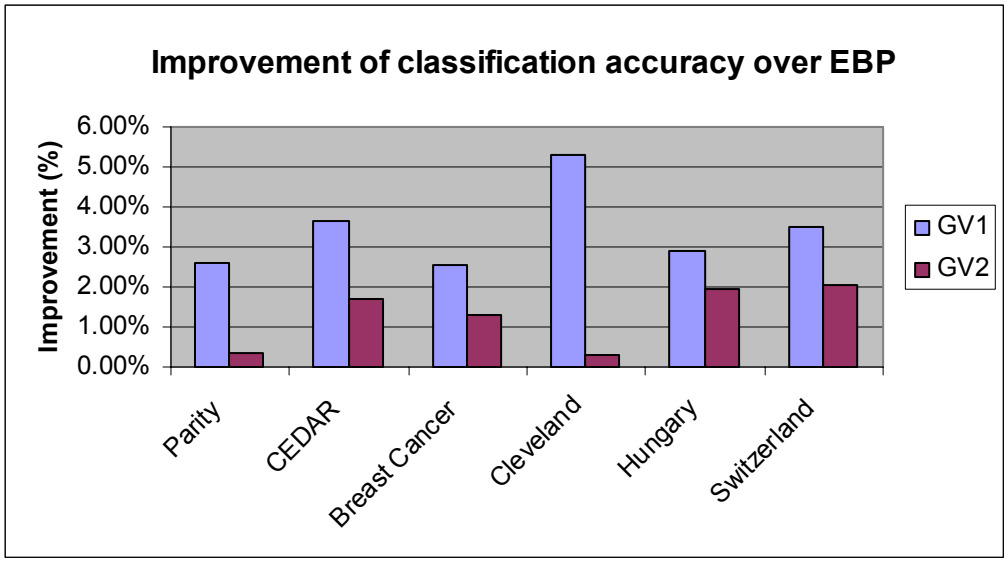

Figure 5-3 Improvement of classification accuracy over EBP

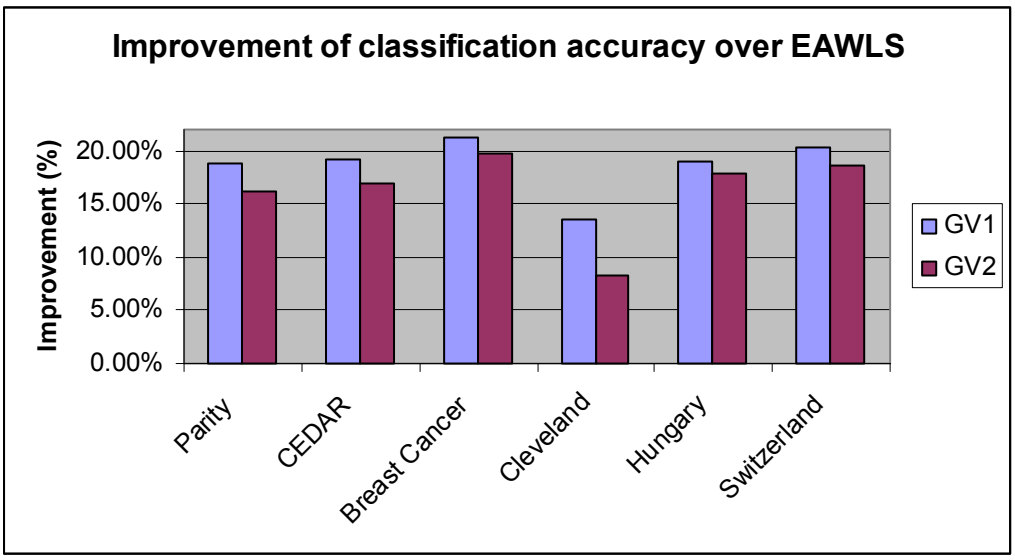

Figure 5-4 Improvement of classification accuracy over EAWLS

From the Figures 5-3 and 5-4, it shows that in cases for the proposed algorithm the test classification accuracies are higher than the standard EBP and EAWLS methods. In all the cases both the GV1 and GV2 gives better classification accuracy than their counterpart. Whereas in case of EBP the improvement range varies from 0.5% to 6%, the results improve a lot when compared with standard EAWLS method. In latter case, the improvement range varies from 8% to 21%. One interesting thing that is observed here that the improvement of classification accuracy over EBP is very large specially in cases for all the heart disease data

set, whereas the improvement is within a limited range for the other three data set (Odd parity, CEDAR and breast cancer data set).

5.1.2 Comparison of time complexity

The following two figures (Figures 5-5 and 5-6) show the time complexity for all the algorithms. To make analysis easier, results from the data set are divided into two different graphs.

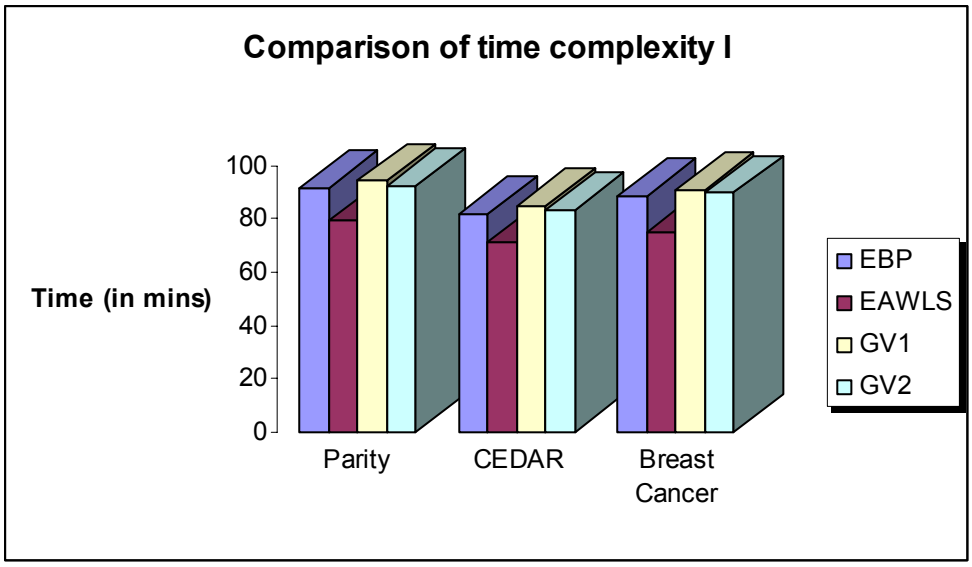

Figure 5-5 Comparison of time complexity I

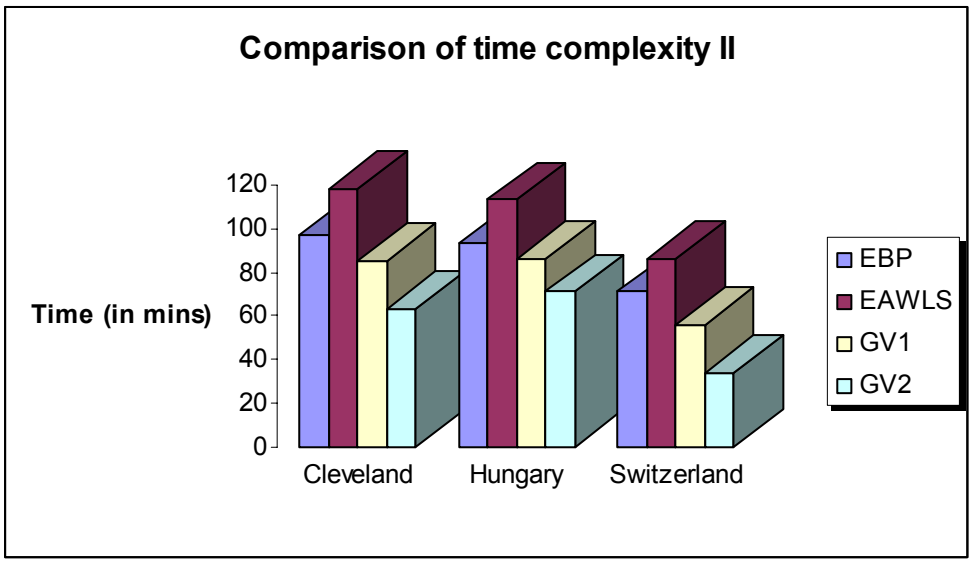

Figure 5-6 Comparison of time complexity II

The following two figures (Figures 5-7 and 5-8) show the improvement of time complexity of training dataset in percentage over the standard EBP and the EAWLS methods.

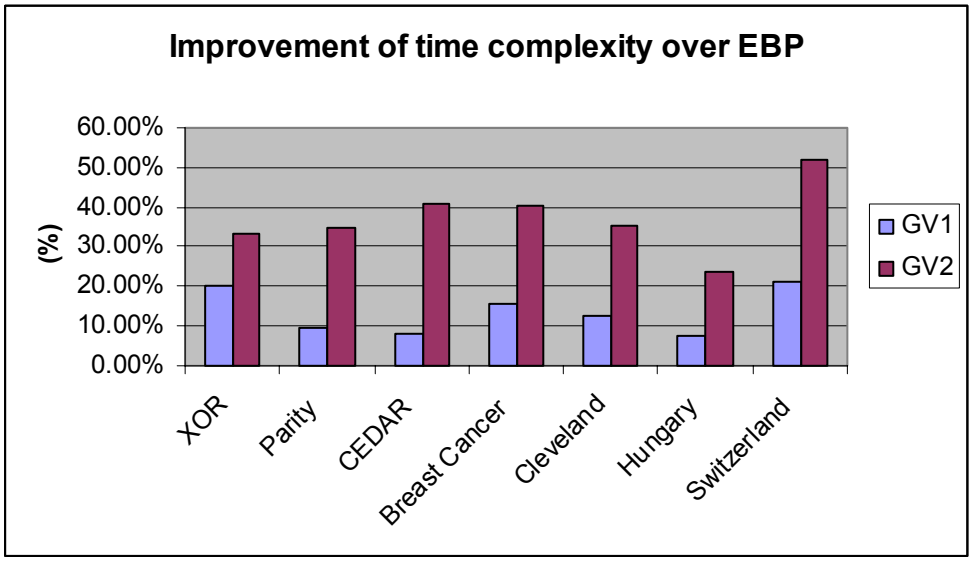

Figure 5-7 Improvement of time complexity over EBP

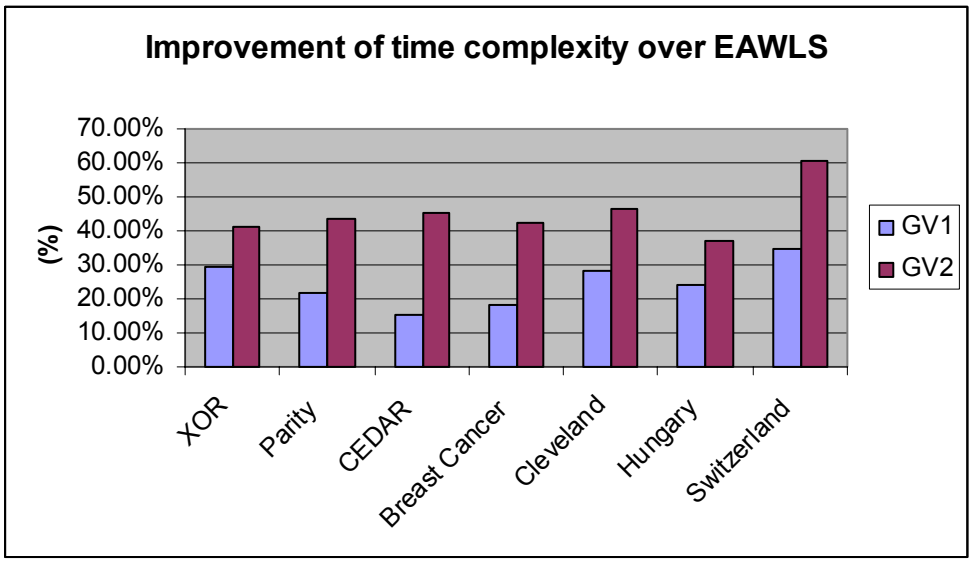

Figure 5-8 Improvement of time complexity over EAWLS

The following three figures (Figures 5-9, 5-10, and 5-11) show the time complexity property of the proposed algorithm. The results are divided into three

figures. The first figure shows the time complexity[16] for increasing number of hidden neurons with fixed number of population and the second figure shows the time complexity for increasing number of population with fixed number of hidden neurons. Because other than the length of the data set, there are only two variables, number of population and number of hidden neurons that affect the time complexity of the program. Only the results from 10 bit odd parity problem are analyzed for the first two figures.

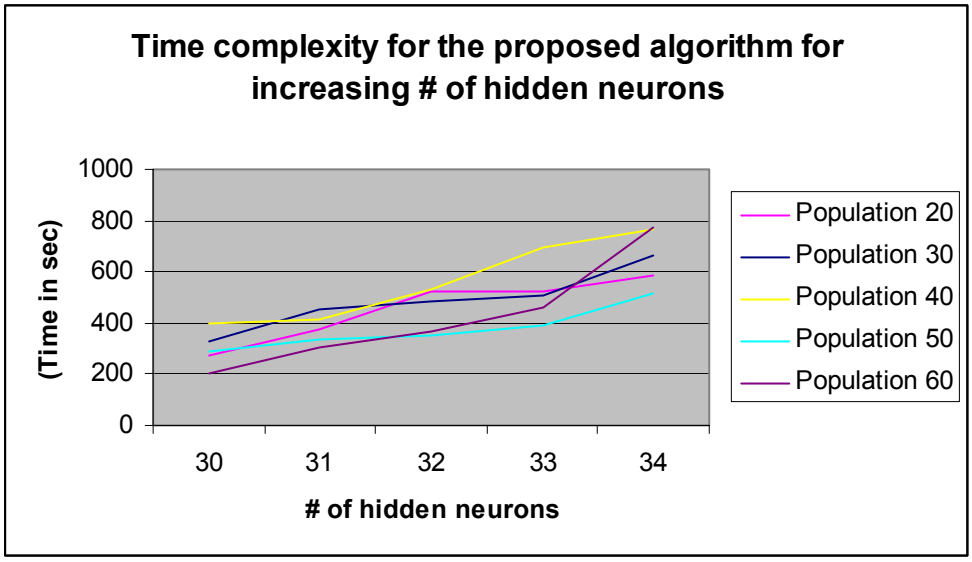

Figure 5-9 Time complexity for the proposed algorithm for increasing number of hidden neurons

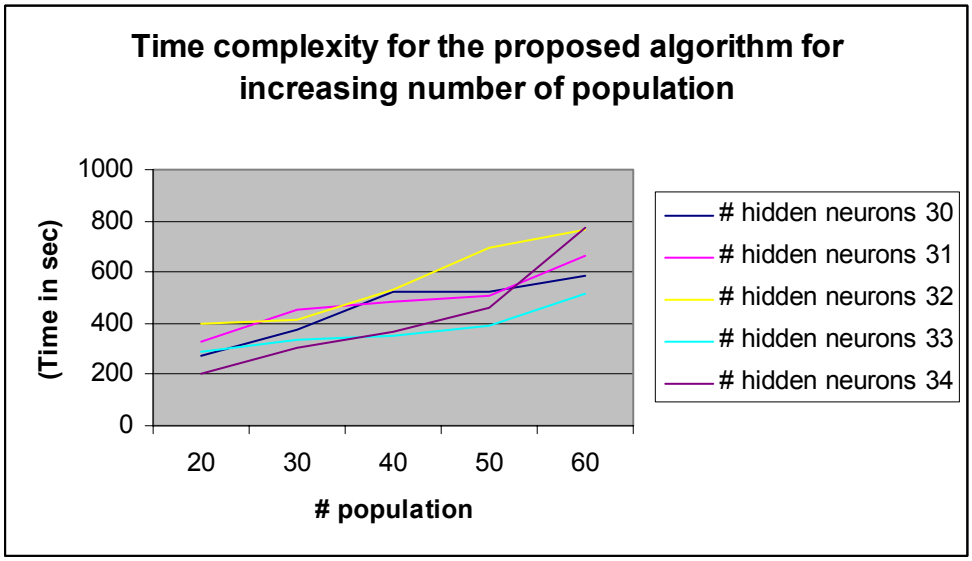

Figure 5-10 Time complexity for the proposed algorithm for increasing number of populations

[16] To obtain the relationship between the time complexity of the proposed algorithm with other variable parameters, the time is measured in secs rather than minutes for more accuracy.

The third figure (Figure 5-11) shows the time complexity for various length of data set keeping number of hidden neurons and number of population fixed. For the third figure we analyze the CEDAR dataset with various lengths of training data.

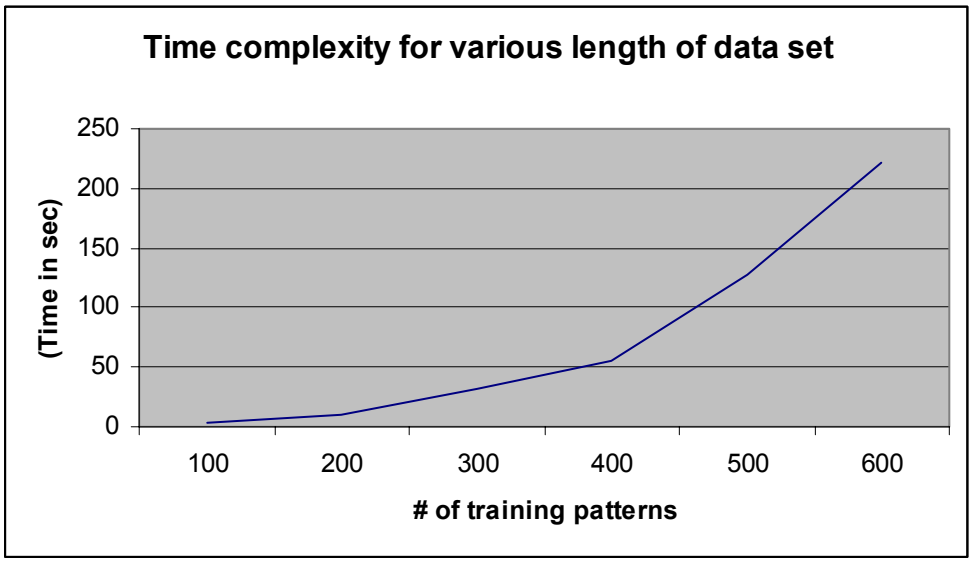

Figure 5-11 Time complexity for the proposed algorithm for increasing number of training data pattern

From the Figures 5-7 and 5-8, it shows that in cases for the proposed algorithm the time complexity are lower than the standard EBP and EAWLS methods. In all the cases both the GV1 and GV2 gives better time complexity than their counterpart. In case of EBP the improvement range varies from 10% to 20% for GV1 and 24% to 52% for GV2 In case of EAWLS, the improvement range varies from 15% to 35% for GV1 and 37% to 60% for GV2. Also, GV2 has a better time complexity than GV1. From figures (Figure 5-9 and 5-10), it shows that the proposed algorithm has a polynomial time complexity O(n). Because in both the graphs the time complexity graph is polynomial with both the variables, number of population and number of hidden neurons. The third figure (Figure 5-11) shows that the time complexity for the variable length of training data set is a higher order polynomial than 1. For CEDAR data set the time complexity with various data length can be represented as $O(n^2)$.

It is interesting to analyze why the time complexity of the new proposed algorithm is less than the EBP especially when the time complexity for EAWLS is higher than that of EBP. In the proposed algorithm, the number of generation

used by EA was all most 25% of total number of generation used by EAWLS. Hence the total time for the new algorithm was 25% of the total time of EAWL + time complexity of the LS method. It is seen that the time complexity of this combination is less than that of EBP and obviously less than EAWLS.

5.1.3 Comparison of required number of hidden neurons

The following two figures (Figures 5-12 and 5-13) show the number of hidden neurons used for all the algorithms. To make analysis easier, results from the data set are divided into two different graphs.

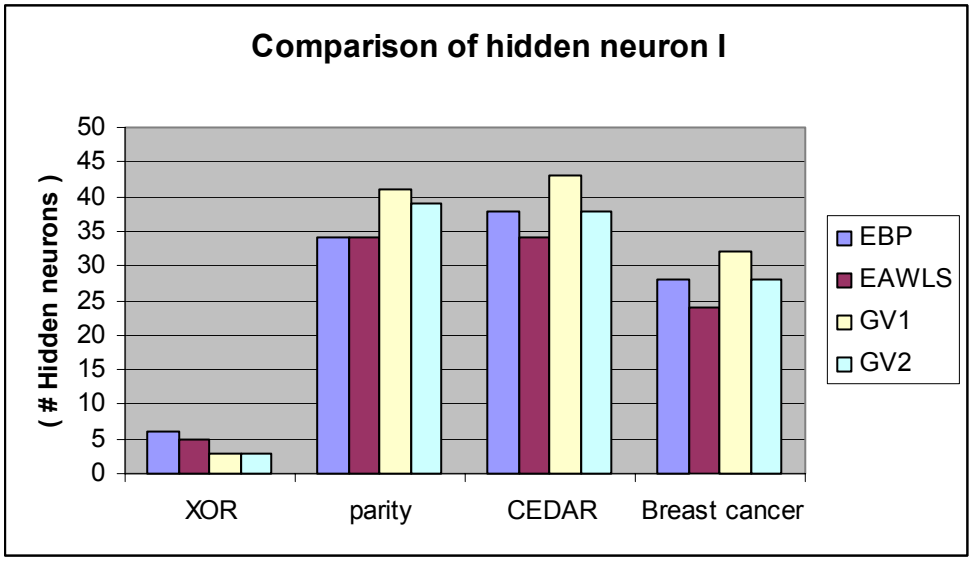

Figure 5-12 Comparison of hidden neuron I

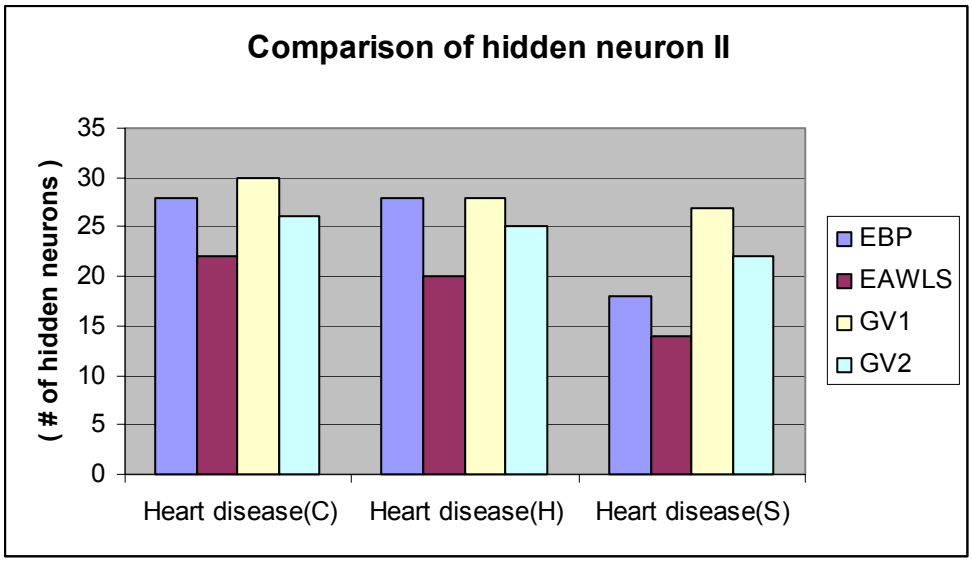

Figure 5-13 Comparison of hidden neuron II

From the above two figures (Figures 5-12 and 5-13), it shows that the required number of hidden neurons are slightly more than the number required by the standard EBP but much higher than EAWLS.

5.1.4 Comparison of memory complexity

The following figure (Figure 5-14) shows the comparison of memory usage for the EBP, EWALS and the new algorithm[17]. Only the CEDAR dataset for the analysis with different data size is considered.

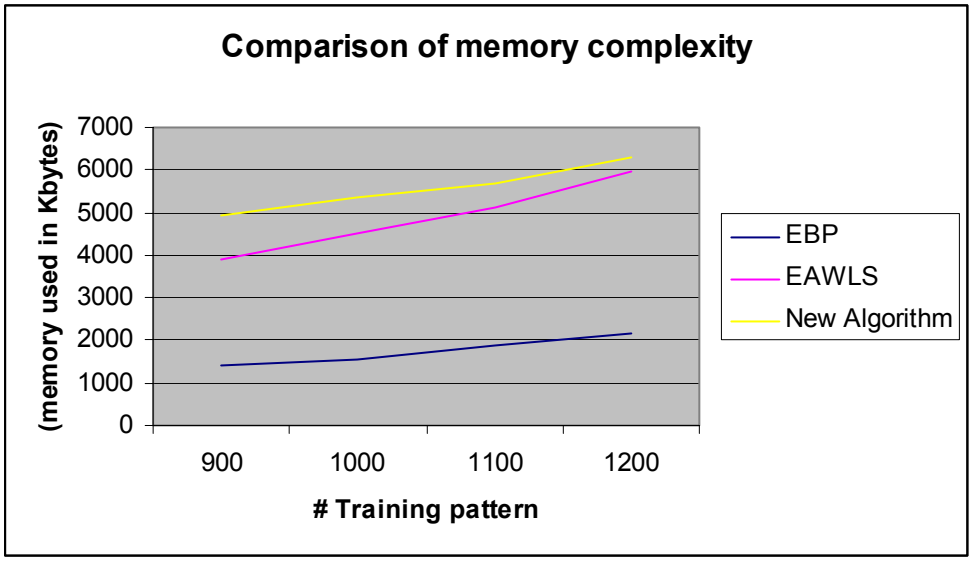

Figure 5-14 Comparison of memory complexity

The following figure (Figure 5-15) shows the increment of memory usage over EBP and EAWLS. Only the CEDAR dataset with variable data length is considered for the analysis.

[17] Proposed algorithm refers only to the findWeight module, as the findArchitecture module is not required to add in those cases. Based on the results the proposed algorithm basically refers to the EALS-T3 algorithm.

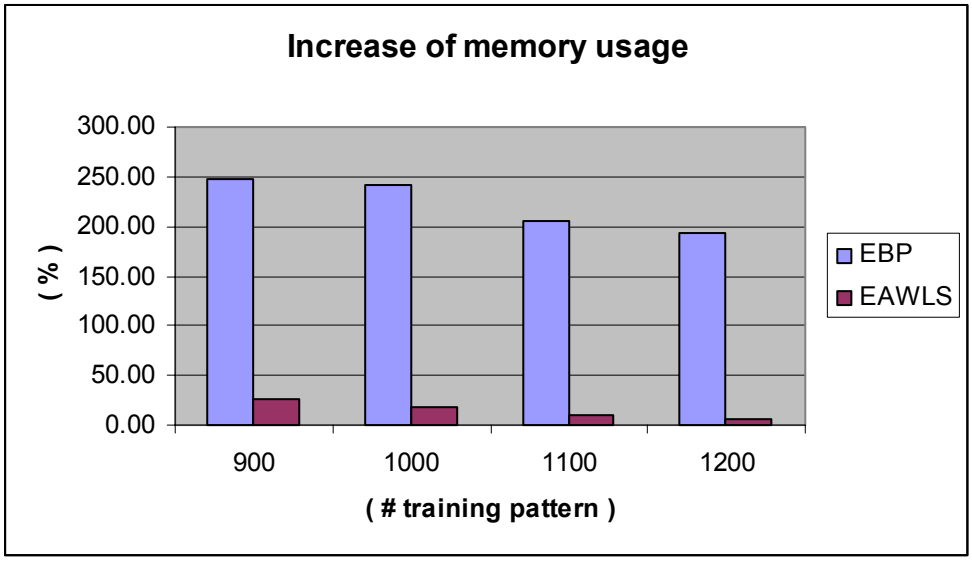

Figure 5-15 Increment of memory usage

From the above figure (Figure 5-15), it shows that, in case of EBP the increment is from 193% to 246%, where as for the EAWLS, the increment is from 5% to 23%. The memory usage is slightly more than the memory usage in EAWLS but much higher than standard EBP.

5.1.5 Comparison of sensitivity analysis

As initialization is done randomly in many training algorithms, hence it is important to show the behavior of the algorithm with in same environment but for different runs. Hence all the algorithms, which use random initialization for its initial weights are trained with different weight matrices keeping all the other variables fixed.

The following figure (Figure 5–16) shows the comparison of sensitivity analysis for the three algorithms. Only the CEDAR dataset with different data length is considered for the analysis. The algorithms are executed 100 times with same data and the test classification accuracy is stored. Then the minimum, maximum and the standard deviation for all the algorithms are shown.

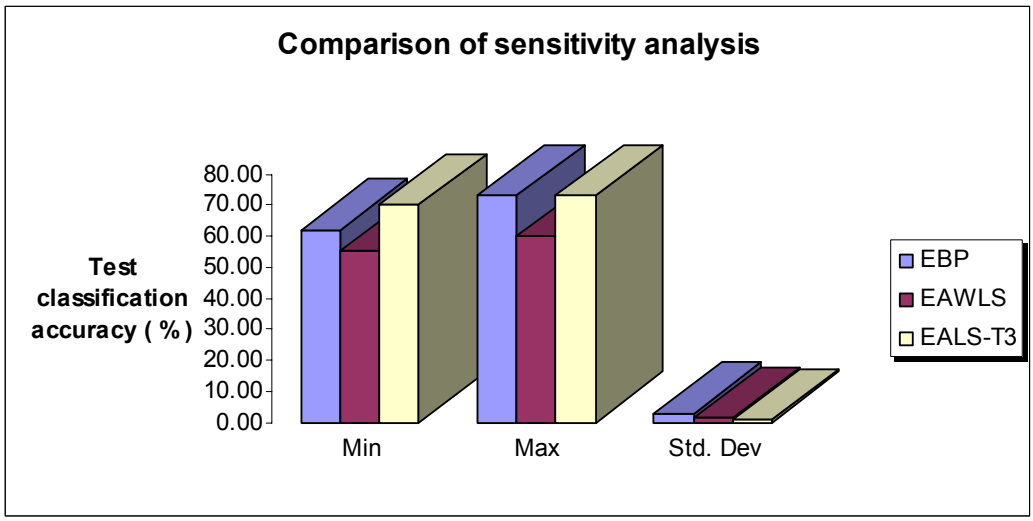

Figure 5-16 Comparison of sensitivity analysis

From the above figure (Figure 5-16) it is clear that the new algorithm is the least sensitive to the initial condition. The variation of EAWLS with respect to the new algorithm is 40% but for the standard EBP with respect to the proposed algorithm it is 73%. The least sensitivity gives the new algorithm to have a better predictable ability with the most confidence

5.2 ANALYSIS OF THE SPECIAL PROPERTIES FOR THE PROPOSED LEARNING ALGORITHM

In this section, analysis are given for finding out some special properties of the algorithm such as convergence property, comments on obtained result in terms of its optimum nature etc. Some of the experiments are simulations based, and some are based on the benchmark data set.

5.2.1 Analysis of convergence property for the new algorithm

The following figures show the four basic convergence properties –

- how the error is decreased over the time (is there a steady decrease of training RMS error)

- how the test classification error is decreased over the training time
- how stable is the solution, or is the result obtained is a optimal solution in its local neighborhood
- what is the effect of RMS and test classification error from the least square output after the end of the generations from EA

The following figure (Figure 5-17) shows the RMS error during training over the generation for all the dataset. Only first 19 generations are considered.

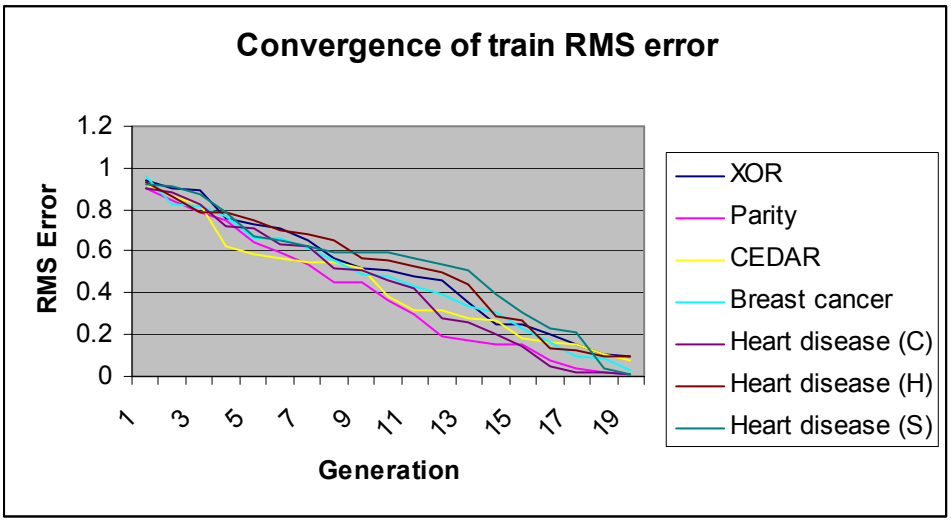

Figure 5-17 Convergence of train RMS error

The following figure (Figure 5-18) shows the classification error during testing over the generation for all the dataset. Only first 19 generations are considered.

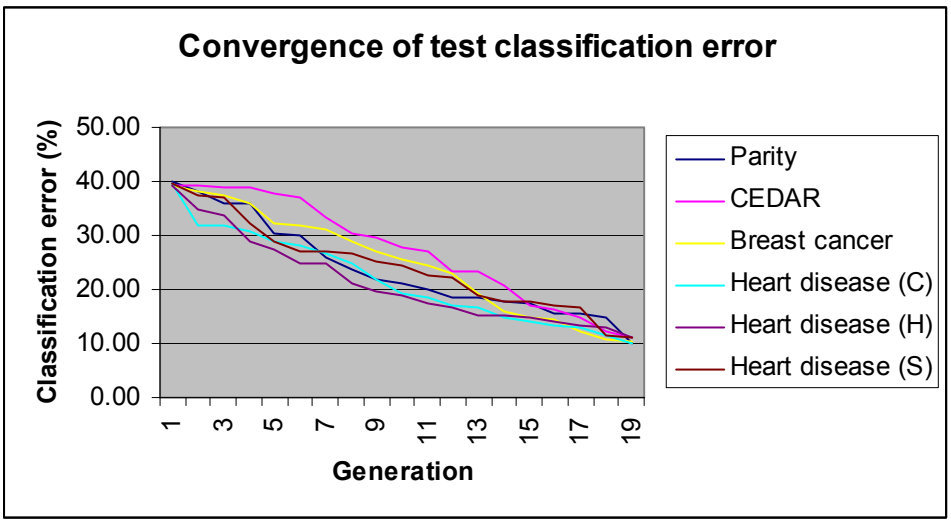

Figure 5-18 Convergence of test classification error

The solution is optimal if it is minima in the local neighborhood. It reaches the local minima where classification error increases in both the cases for increasing and decreasing the number of hidden neurons from the solution point .The following figure (Figure 5-19) shows the solution is the minima in the local neighborhood.

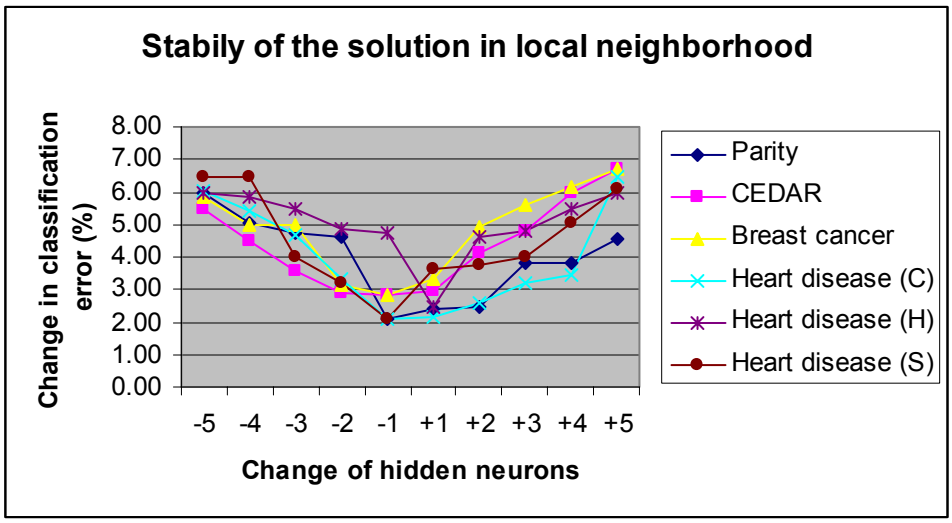

Figure 5-19 Stability of solution in local neighborhood

The proposed algorithm is combination of both evolutionary algorithm and least square method, so it is necessary to show the contribution of both the methods in convergence. Figure 5-20 shows the combined output of RMS error in both the case (evolutionary and LS method). Figure 5-21 shows the combined output of classification error in both the case (evolutionary and LS method).

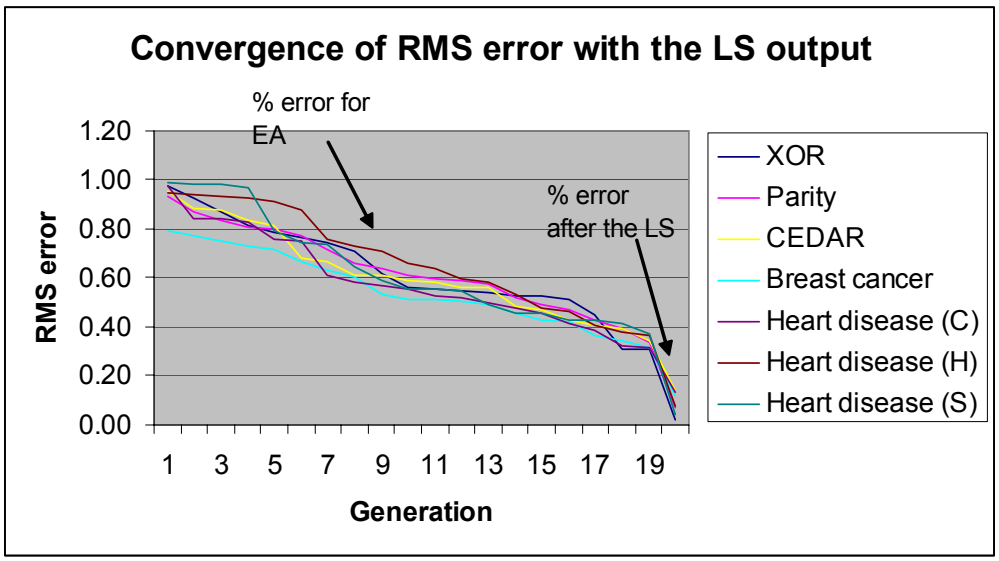

Figure 5-20 Convergence of RMS error with the lest square output

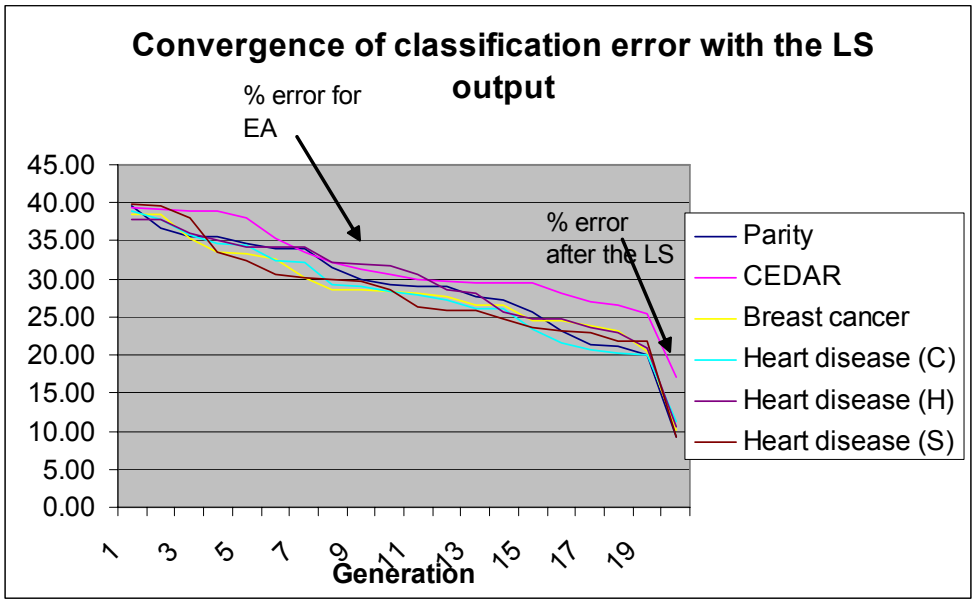

Figure 5-21 Convergence of classification error with the LS output

The following figure (Figure 5–22) shows the comparison of required number of generation to converge for all the dataset.

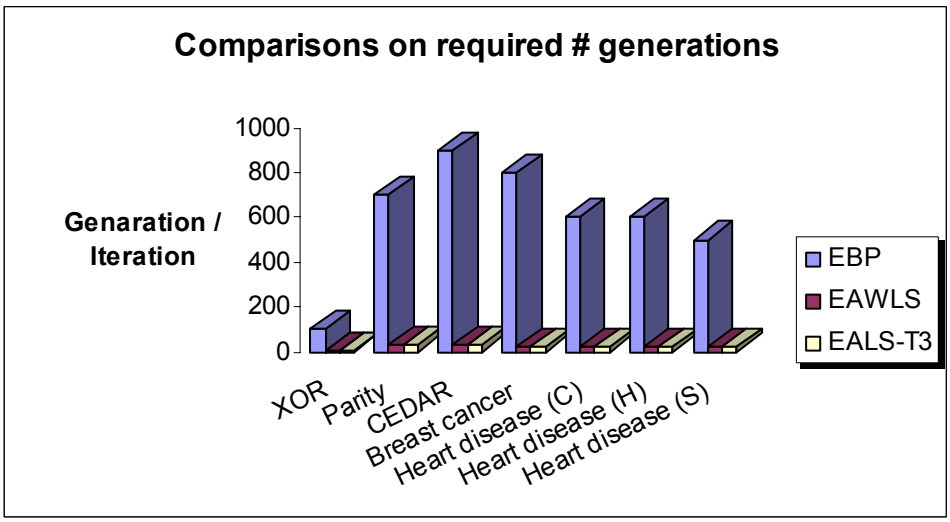

Figure 5-22 Comparisons on required # generations

Figures 5-17 and 5-18 show that both RMS and classification error decrease steadily over the generation. Figure 5-19 shows that the solution point reached is the local minima. The classification error increases with increase of number of hidden unit form the solution point. It also increases with decrease in number of hidden unit from the solution point. Hence the solution is the minima in the local neighborhood in the search space. Hence the solution is optimal/near optimal. Figure 5-20 and Figure 5-21 show the combined effect of both the method

(evolutionary and LS) in convergence. In both the cases there are a steep decent in the error when the algorithm jumps form evolutionary to least square method. Figure 5-22 shows the comparison result for required number of generations to converge. It shows that the new algorithm takes less number of generations than both the EBP and EAWLS in almost al the cases.

5.2.2 Variations of hidden layer output for various weight range

As described in the previous chapter, the variations of output from the hidden layers, is very important for using the QR decomposition. More the variations, better the decomposition can be done, hence the final weight matrix can be solved for the output layer. Section 5.2.3 shows in details what happens when the hidden layer output variation is very less.

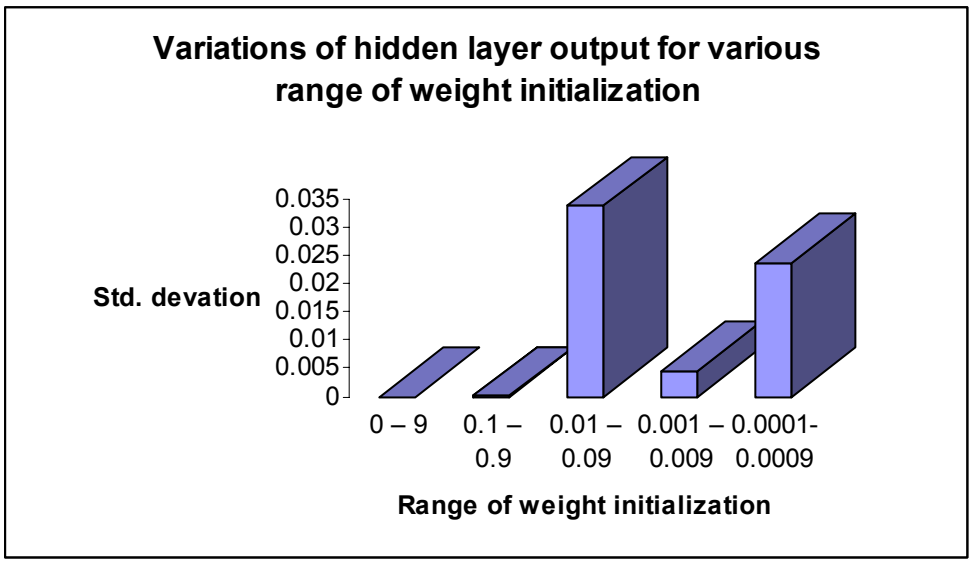

Figure 5-23 Variations of hidden layer output for various range of weight initialization

The figure (Figure 5-23) shows that, initial weight range of 0.01 – 0.08 gives the maximum variations for the hidden layer output. Hence this range is chosen to be the range for the proposed algorithm.

5.2.3 Variations of inverse of R Matrix

As discussed in the chapter 4, the inverse of the R matrix is very important to calculate the weight matrix for the output layers. Because more the deviations of the R Matrix, more the solution is accurate to find the output weight matrix. We only consider first few values from that table to analyze the results for simplicity. The following figure (Figure 5-24) shows the result of effect for first few increments for the various range of hidden layer output.

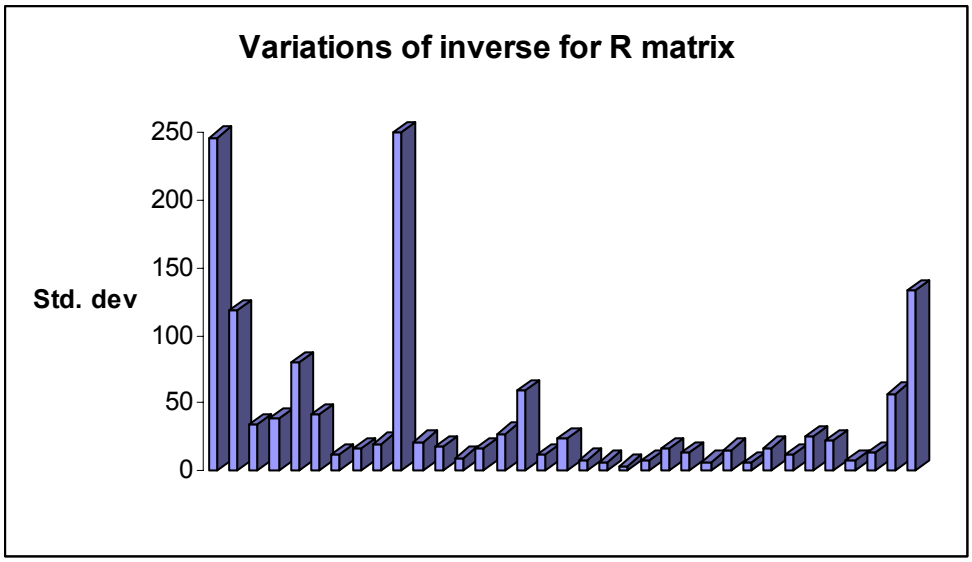

Figure 5-24 Variations of inverse output for R Matrix with range element chosen randomly

The following figure (Figure 5-25) shows the effect of variations for the inverse of the R matrix for various ranges with the changing element keeping fixed.

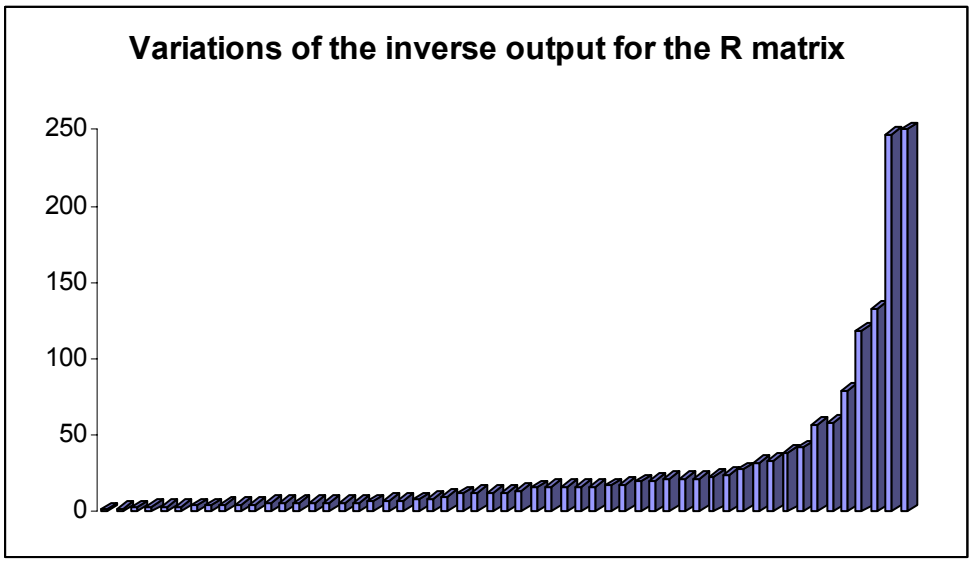

Figure 5-25 Variations of inverse output for R Matrix with range element chosen was fixed

The following figure (Figure 5-26) shows the effect on the inverse matrix when the number of elements of the hidden layer output matrix are different from each other. The 10 X 10 matrix is changed for different matrix elements and ranges and the inverse matrix is calculated.

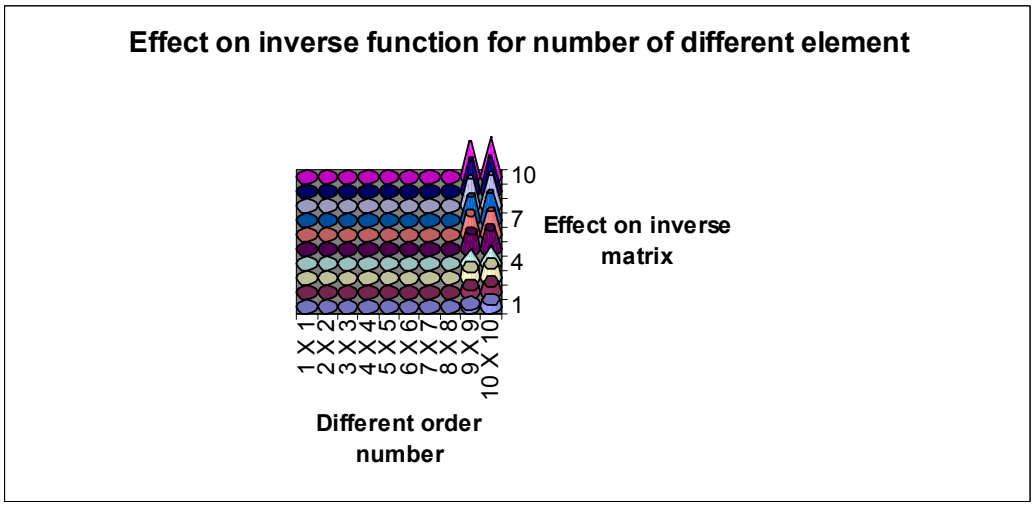

Figure 5-26 Effect of inverse function for number of different element

From the figure (Figure 5-24), it shows that no clear picture emerges for the variations with the range of the hidden layer output. It is because one element from the hidden layer matrix are chosen randomly and the inverse matrix is calculated. In the figure (Figure 5-25), it shows that the variation for the inverse

matrix is increased with the range when the element is fixed. Another properties that can be analyzed from the figure (Figure 5-26), is that the more the number of element are different from each other for the hidden layer output matrix, the inverse function gives better result. More the elements are similar for the hidden layer matrix, worse then the inverse function calculation. Hence it is quite clear that the variations for the hidden layer matrix should have not only more elements that are different from each other but possibly elements with greater range. The figure shows that only for difference in elements of order 9 X 9, and 10 X 10 the inverse matrix exist and the matrix gives the maximum variations with more ranges as already sees in the Figure 5-26.

5.2.4 Need of bias for the proposed algorithm (EALS-T3)

In this analysis, the solution region of the hidden neurons is discussed for the proposed algorithm with or without the application of the bias input. As earlier analysis showed that an initial range with close interval [0.01 0.09] is a good choice for the weight initialization for the proposed algorithm, however it dose not mention any bias term earlier. Mathematically, the output of the hidden neuron or the input for the activation function g(x) can be given as

$$\sum_{\substack{i=1 \\ j=1}}^{\substack{n \\ h}} x_i w_j \,, \tag{5-1}$$

where x is the input vector ($1 <= i <= n$) and w is the weight vector ($1 <= j <= h$) for a n length input vector and h length hidden neuron vector output.

If we consider $x_i > 0$ for $1 <= i <= n$ and

$w_j > 0$ for $1 <= j <= h$ then

the value of equation 5.1 is always positive. In that case the activation output is bounded only for the positive region of the sigmoidal function as shown in figure (Figure 5-27). To avoid this problem, we need an unbounded region for the input for the sigmoidal activation function. Hence a bias value of -0.5 is needed for

adding to the scalar output of every individual hidden neuron, so that the activation output can be obtained for all the range of its input.

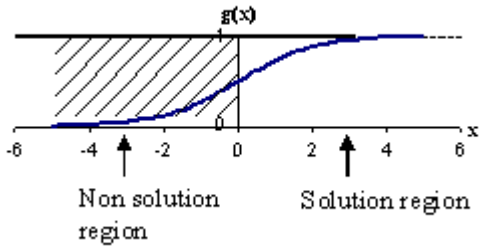

Figure 5-27 Solution region without the bias

In figure (Figure 5-28), the negative bias has been added to obtain unbounded region.

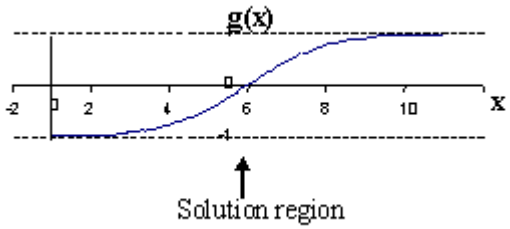

Figure 5-28 Solution region with the bias

5.2.5 Is the hybrid nature of the proposed algorithm (EALS-T3) a combination of global and local search

This analysis is done to test the exact nature of the proposed hybrid algorithm. As the new algorithm does a completely new thing by combining EA and least square methods, it must be analyzed whether the proposed hybrid algorithm is another hybrid model for a combination of a global and local search. There is obviously no doubt about the nature of global search nature of the EA, and also the least square method sometimes can be considered as a local search method. But the analysis in this section is done to answer, does the least square method always exhibit the local search nature. The answer can only be given by analyzing the solution region space that can be explored by the least square

method. If the least square method is a local search procedure then obviously the solution points is bounded within the local neighborhood area. In that situation the only way to support this claim is to formulate the mathematical structure to answer this question whether the solution can be outside the local neighborhood. Another way of looking at the solution could be to analyze the quality of the solution obtained from the results. If the train classification result[18] is analyzed after the EA and after the LS, then the change in the result should give some idea of how much in an average the result is improved. If the average improvement of EA in each generation is very much less than the average improvement from the least square, then obviously it cannot be claimed that the solution point is bounded in the local neighborhood region, because there must be a large jump from the EA solution point to the LS solution point. The following figure (Figure 5–29) shows the percentage improvement after the LS method for different range of classification accuracy from the EA. The result was obtained from the various accuracy obtained from both the EA and LS from all the algorithms, and have been summarized.

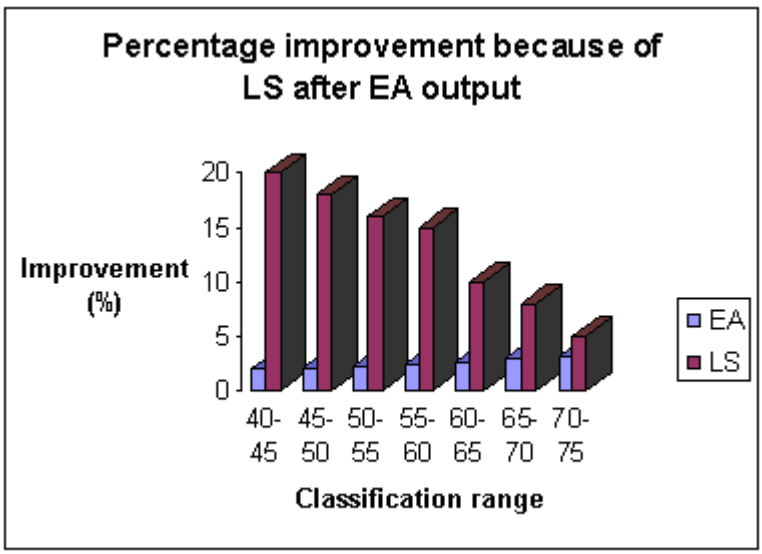

Figure 5-29 Percentage of improvement because of LS after EA output

[18] It is important here to note that in this particular case train classification result has been considered, because solution region during the training is the point of concern here.

The figure (Figure 5-29) shows that for all the results the classification improvement for every generation EA (around 2%), there has been an improvement in the result varied from 4 - 20%. The result seems to be quite interesting, because it shows that the LS became a local search if the EA gives a very good solution point for the LS, and LS becomes again a global search if the EA cannot produce a good solution to it. Hence the LS method in the new hybrid algorithm is capable of producing both a global and local search nature. This gives the new methodology the power to improve the result to quite some extent. Also, it should be noted here that the most of the time the improvement from the LS was very high.

5.2.6 Analysis of fitness breaking

Figure 5-30 shows the fitness distribution for the 10 best populations before least square methods is applied. Figure 5-31 shows the fitness distribution for the same population after applying the least square method. After every generation of the evolutionary algorithm, the fitness for the population pool is calculated by the evolutionary algorithm. Then the fitness of the gene is calculated by breaking it into two halves, and taking the first half and then combining with the output layer weights (by calling the least square function). The comparisons of the fitness values are reported.[19]

[19] Population number is given based on the rank, rather than the raw number of the population obtained from the program.

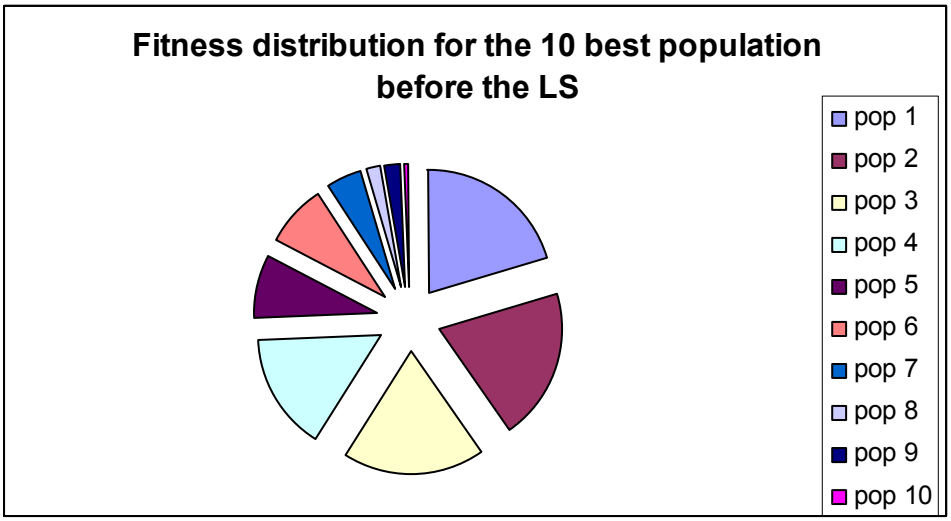

Figure 5-30 Fitness distribution before LS

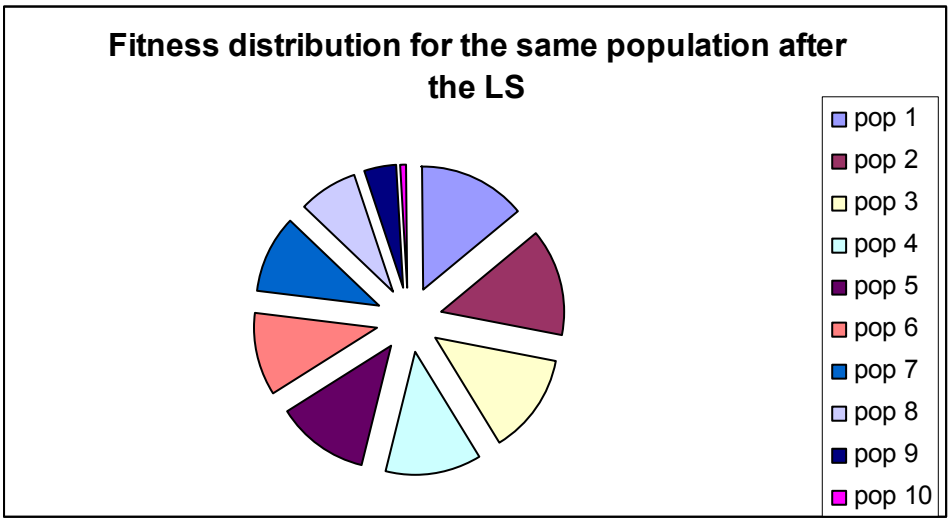

Figure 5-31 Fitness distribution after the LS

The following figure (figure 5-32) shows the rank and their fitness for the 10 best populations before and after calling the least square method.

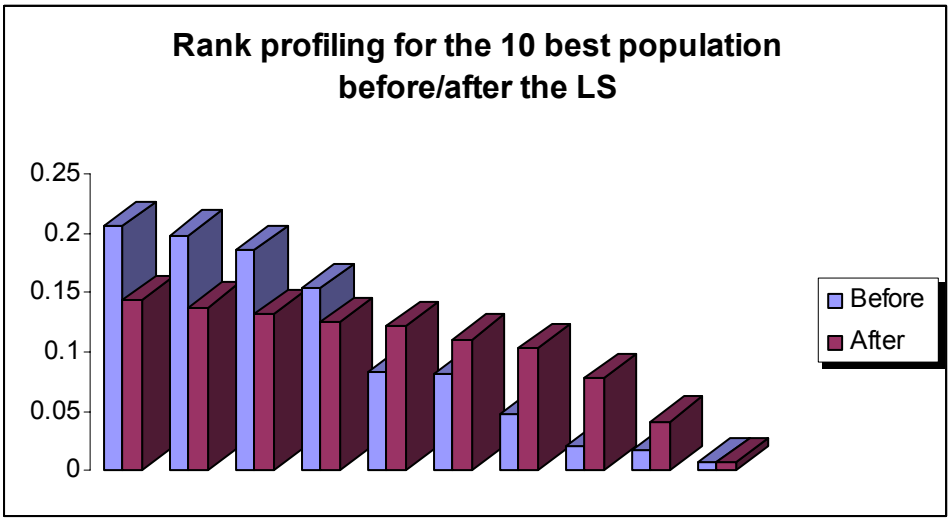

Figure 5-32 Rank profiling before/after the LS

The Figures 5-31 and 5-32 show that the fitness of the gene (specially for the chromosome with very high fitness) are not affected much when it is broken and combined with the least square method. In further Figure 5-32 shows that the ranking based on their fitness value remains unaffected. So, even when the upper half characteristic is replaced by the least square weights, and only the lower half characteristic is considered, it does not break the fitness and affect their rank. The fitness based on the lower half characteristic of the gene has almost the same rank in the population pool, with the fitness based on the full length gene.

5.2.7 Comparison of results for different T connections

In the following graphs, the analysis for different T connections for EALS algorithm is given. The comparison is based on test classification accuracy, time complexity and memory complexity.

The following figure (Figure 5-33) shows the comparison for classification accuracy results for different T connections. Only data set which are run by all the three different T connections, are considered. All algorithms are run for the four data set and the best result is reported.

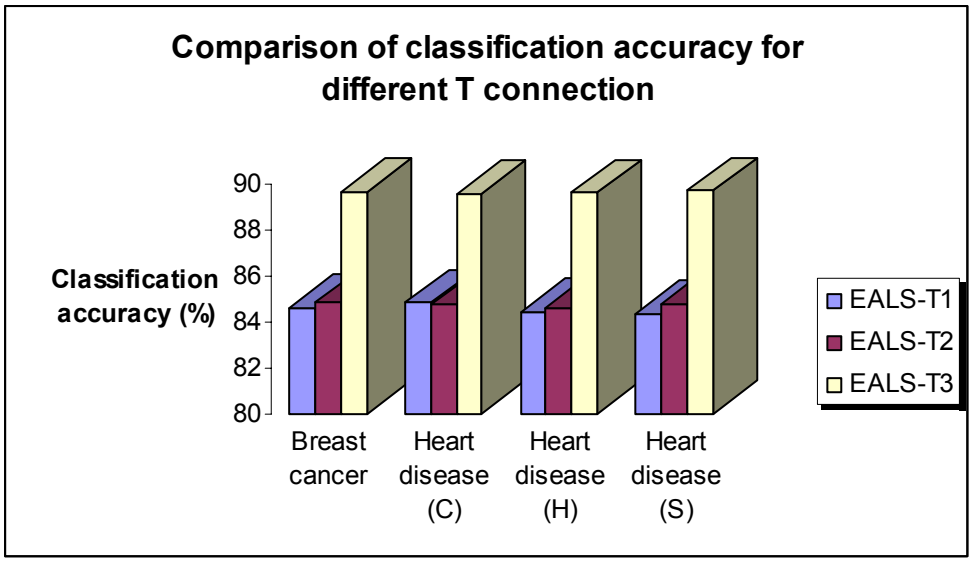

Figure 5-33 Comparison of classification accuracy for different T connections

The following graph shows the time complexity for all the different T connections.

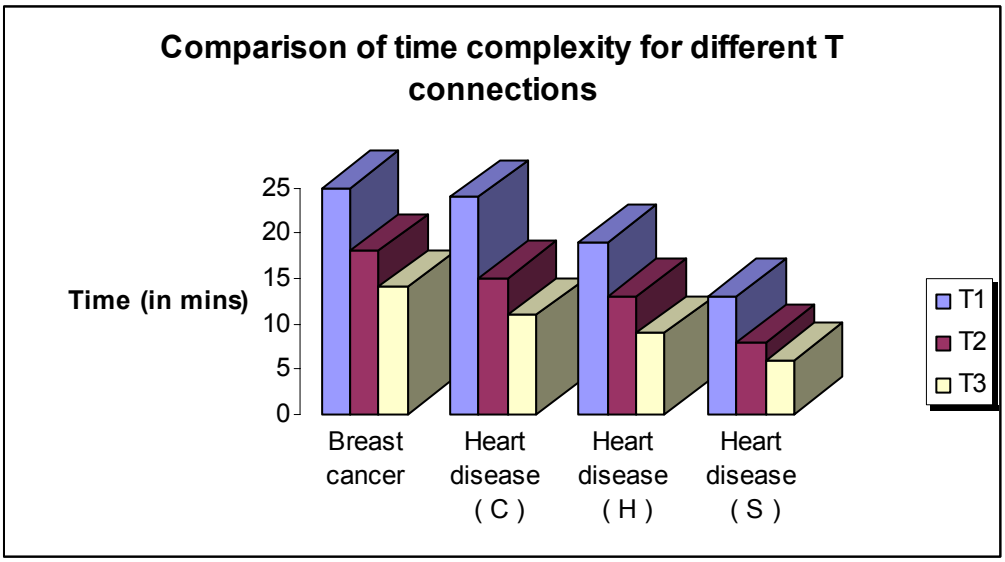

Figure 5-34 Comparison of time complexity for different T connections

The comparison of memory complexity graph is not given, as it is already seen that the memory complexity of T1 and T2 are much higher than that of T3. Algorithm based on T1 and T2 are not able to run for all the data set (for input patterns length > 500 and feature vector of length 100).

From the above two figures, it shows that T3 connection is the best among the three connection in terms of both the classification accuracy and time complexity.

5.2.8 Comparison of results for different GA strategies

The following graphs (Figure 5-35 and 5-36) show the result and analysis based on the different GA strategies that are used. The two different strategies GA and EA are compared on the basis of classification accuracy and time complexity. The results are given after the best connection strategy T3 with LS are used for both the strategies.

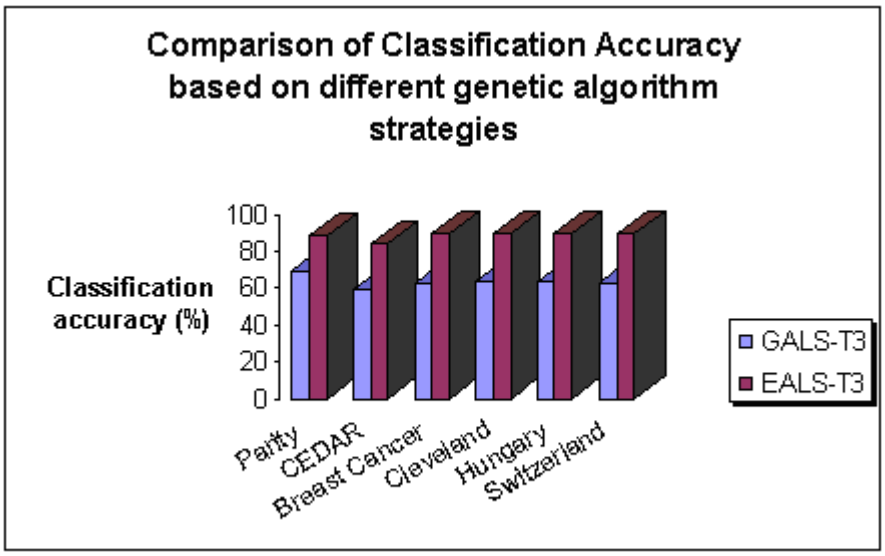

Figure 5-35 Comparison of classification accuracy based on different genetic algorithm strategies

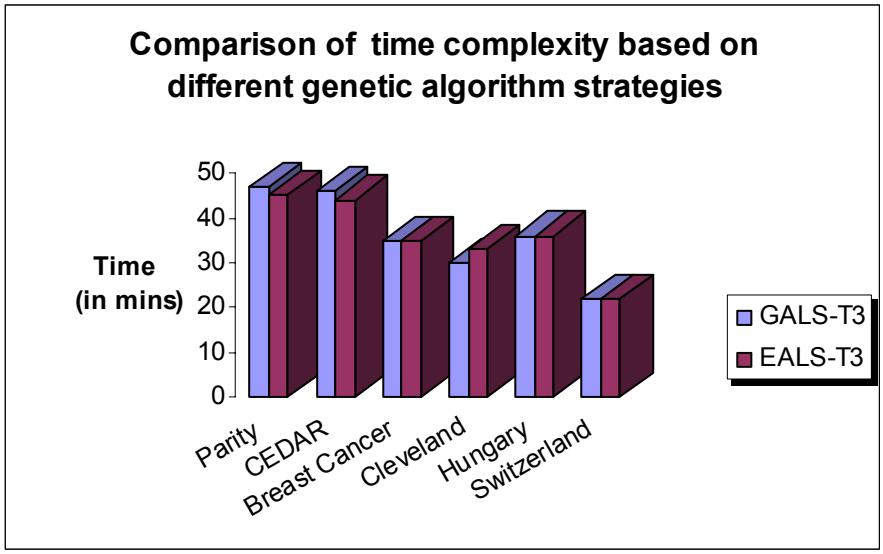

Figure 5-36 Comparison of classification accuracy based on different genetic algorithm strategies

From the figures, it shows that based on the classification accuracy EA is better than GA. However the difference in time complexity is very minimal.

6 CONCLUSION & FURTHER RESEARCH

6.1 CONCLUSION

In this thesis a novel hybrid learning algorithm for ANN has been proposed, which is able to find a suitable architecture and the weight values for a 2 layer feed forward perceptron model. It has been shown that architecture search and the weight search can be considered as separate modules, and with good combination dynamics the naïve architecture search for faster learning can be combined with the weight search module, which is a combination of an evolutionary learning and a least square based learning. Result shows that the proposed hybrid method with two different learning methods for different layers of the ANN produces much better results in terms of classification accuracy, time complexity etc, when compared with the same learning strategy applied for both the layers. It was shown that the global search nature of evolutionary learning and the less time complexity order of the least square technique, when combined together with appropriate combination dynamics displays the good characteristics for both the learning algorithms.

In this thesis various combination strategies for evolutionary learning and least square based learning are explored. Mainly three different types of connection strategies are devised – T1, T2 and T3. The result shows that in terms of classification accuracy and time complexity T3 connection is the best among them. To be more precise, the main difference in result is in the order of time and memory complexity. It is shown that the difference in classification accuracy among the three different connections is very less, but the order of time and memory complexity is very much different. In fact, in case of T1 and T2 the maximum input data size is a matrix of order 300 X 100 (300 training pattern length and 100 is the length of the feature vector).

In this thesis two different types of GA are considered. One is the traditional GA with direct encoding scheme that uses linear interpolation crossover and a gaussian mutation and the other one that uses only mutation operator and evolve

quite differently than their GA counterpart. It was seen that in terms of the classification accuracy the results based on EA strategy are much better than those of GA based strategy. Also the time complexity for EA based strategy is slightly less than that of GA.

In this thesis only two types of naïve architecture search are combined with the weight search module. It is shown that architecture search based on the type of binary search technique takes much less time than the linear architecture search counter part. This looks at first quite obvious but considering that the test or train error does not always behave in a consistent way, it is quite interesting to note that the combination of train and test error as is done in the binary search type module, averages out most of the time the inconsistency of the result. This is further proved by the fact that the classification accuracy from this search is only slightly worse than the linear architecture search.

When the proposed algorithm is compared with the traditional EBP and other genetic algorithm based search such as GAWLS and EAWLS, it is shown that the results in terms of classification accuracy, time complexity, sensitivity to the initial condition, stability of the results, required number of generations etc are much better in case of the proposed algorithm than the other three. The improvement in test classification accuracy ranges from 2% – 20% on an average. The improvement in time complexity is 4% - 10% on an average. It is shown that the proposed algorithm gives more consistent result in terms of classification accuracy as compared to the other three algorithms. The standard deviation in classification accuracy for the proposed algorithm is 0.12, whereas for EBP it is an average of 2.3. The number of generations required by the proposed algorithm is almost 25% less than that of GAWLS and EAWLS. This is also the reason why the time complexity for the proposed algorithm is much less than all the other three algorithms.

However the result shows that the proposed algorithm requires more number of hidden neurons and also the memory complexity for this algorithm is quite high when compared to the traditional EBP, GAWLS and EAWLS. The result shows that the order of memory complexity on an average could be 4-5 times higher than EBP. However the memory complexity is not of a high order because of the required number of hidden neurons (the range for required number of hidden

neurons is 10-20% more in case of the proposed algorithm), but the high memory complexity order is because of the extensive matrix operations which take lots of memory to solve the linear equations to find output layer weights.

Results show that the convergence property of the proposed algorithm is quite steady, and the hybrid algorithm gives a very consistent result of high classification accuracy. For example, when the initial region for the least square learning is not a good starting point or close to the solution region obtained by evolutionary learning, the hybrid algorithm behaves as a mixture of two global search modules. When the initial region for the least square learning is a good starting point or close to the solution region obtained by evolutionary learning, hybrid learning behaves as a mixture of a global and a local search. This is obviously the power of new hybrid learning, because of using evolutionary learning at the beginning the stopping criteria had to be limited to call the least square search, and the consistency of results could only be obtained by putting a stop to the number of generation at the very best. Also, it was seen that most of the time the hybrid algorithm is a mixture of two global search techniques.

The analysis of the result shows that good initialization (not in terms of the quality of the solution region, but in terms of range of the random numbers and also the values of the matrix elements which should be different from each other) of the random numbers for the chromosomes is very important. It was shown the result was consistent in terms of classification accuracy; hence the starting pointy is not of as much concern as is the difference in values for the matrix elements. Because, if the values are very similar for the output matrix of the hidden layer, the equations become unsolvable.

One of the most powerful features of the proposed technique is that the solution is guaranteed, no matter where the starting point is. So whatever be the values for the weight chromosomes, the least square methods will find a solution which is quite acceptable. In those cases the values of the weights for output layer may be quite high, but that does not affect the quality of the solution. And the high values for the output layer in this particular case do not concern us much, because no weight updating is done with those weight values. If the learning is started again, a completely different set of weights is obtained.

Another interesting feature obtained from the new hybrid learning algorithm is that breaking the weight chromosome into two halves does not effect the fitness values for the chromosome when combined with the least square search. Hence it seems that the fitness for the weight chromosomes, which represent both the layer weights, is an individual combination of ranking for both the layers.

6.2 FURTHER RESEARCH

Further research should concentrate mainly on two things that are needed to improve the existing algorithm. The first thing is that because the memory complexity is quite high for the proposed hybrid learning, there needs to be further research to improve this shortcoming. One way of improvement could be using unsupervised clustering methods, and using individual feature for different class to train the ANN. In that case if there are c number of classes, then at the most c number of hidden neurons are required with n number of training pattern length. Quite obviously the value c is much less than the n number of initial training data length. Only important thing here is that classification accuracy in that case could be reduced, unless the unsupervised clustering method could cluster the training data pattern very well. In other words the linear/nonlinear discriminant vector for every class should represent the average class very well and also the distance of the vector from its individual training vector that belongs to a particular class should be very less.

Another improvement for this hybrid algorithm will be to reduce its time complexity further. It is quite clear that the different architecture search for different weight sets is totally independent. Hence, making a parallel implementation of the code will reduce the time complexity to a large extent. The possibility of reducing time complexity with parallel implementation of separating architecture and weight search will be an order of 1/p1 (where p1 is the number for different architecture) of the time taken currently. Also, further reduction of time complexity is possible by introducing further parallelism by evolving different weight chromosome separately and then combining the best fitness values with the least square method. So a total reduction of time complexity could be 1 / (p1 * p2), where p2 is the number of population for the weight chromosomes.

Reference

[1] L. A. Zadeh, "From computing with numbers to computing with words: From manipulations of measurements to manipulation of perceptions," 3rd International Conference on Application of Fuzzy Logic and Soft Computing, pp. 1-2, Wiesbaden, 1998.

[2] D. E. Knuth, "The art of computer programming", vol.2, Reading, MA, Addision –Wesley, 1981.

[3] R. J. Williams and D. Zipset, "A learning algorithm for continually running fully recurrent neural networks," Neural Computation, vol. 1, pp. 279-280, 1989.

[4] T. Back, "Evolutionary algorithms in theory and practice", New York, Oxford university press, 1996.

[5] D. B. Fodgel, "Evolving artificial intelligence", PhD thesis, University of California, San Diego, 1992.

[6] V. Petridis, S. Kazarlis, A. Papaikonomu and A. Filelis, "A hybrid genetic algorithm for training neural network," Artificial Neural Networks, vol. 2, pp. 953-956, 1992.

[7] S. Wright, "The role of mutation, inbreeding, crossbreeding, and selection in evolution," Proceedings of 6th International Congress of Genetics, Ithaca, NY, vol. 1, pp. 356-366, 1932.

[8] D. R. Hush and B. G. Horne, "Progress in supervised neural networks," IEEE Signal Processing Magazine, vol. 10, no. 1, pp. 8-39, 1993.

[9] M. F. Moller, "A scaled conjugate gradient algorithm for fast supervised learning," Neural Networks, vol. 6, pp. 525-523, June 1993.

[10] R. K. Belew, J. McInerney, and N. N. Schraudolph, "Evolving networks: using genetic algorith, with connectionist learning," Technical Report #CS90-174 (Revised), Computer Science & Engineering Department (C-014), University of California at San Diego, La Jolla, CA 92093, USA, 1991.

[11] A. P. Topchy and O. A. Lebedko, "Neural network training by means of cooperative evolutionary search," Nuclear Instruments & Methods in Physics Research, Section A: Accelerators, Spectometers, Detectors and Associated Equipment, vol. 389, no. 1-2, pp. 240-241, 1997.

[12] J. B. Pollack, "Connectionism: past, present, and future," Artificial Intelligence Review, vol. 3, pp. 3-20, 1989.

[13] D. A. Medler, "A brief history of connectionism," Neural Computing Surveys, vol. 1, pp. 61-101, 1998.

[14] D. O. Hebb, "The organization of behavior," John Wile & Sons, New York, 1949.

[15] S. Grossberg, "Some networks that can learn, remember, and reproduce any number of complicated space-time patterns, I," Journal of Mathematics and Mechanics, vol. 19, pp. 53-91, 1969.

[16] S. Grossberg, "Embedding fields: Underlying philosophy, mathematics, and applications to psychology, physiology, and anatomy," Journal of Cybernetics, vol. 49, pp. 28-50, 1970.

[17] J. A. Anderson, "A simple neural network generating an interactive memory," Mathematical Biosciences, vol. 14, pp. 197-220, 1972.

[18] D. J. Willshaw, and V. D. Malsburg, "How patterned neural connections can be set up by self-organization," Proceedings of Royal Society London, vol. B194, pp. 431-445, 1976.

[19] C. Klimasauskas, "Applying neural networks: Parts I-VI," PC AI, January-December 1991.

[20] T. Kohonen, "Self-Organization formation of topologically correct feature maps," Biological Cybernetics, vol. 43, pp. 59-69, 1982.

[21] M. L. Minsky and S. A. Papert, "Perceptrons-An Introduction to Computational Geometry," Expanded Edition, Cambridge, MA, MIT Press, 1988.

[22] J. McCarthy, "Programs with Common Sense," M. Minsky (Editor), Semantic Information Processing, MIT Press, Cambridge, MA, 1968.

[23] W. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," Mathematical Biophysics , vol. 5, pp. 18-27, 1943.

[24] K. Fukunaga, "Introduction to statistical pattern recognition," 2nd Edition, Academic Press Inc., San Diego CA, 1990.

[25] K. H. Becks, F. Block, J. Drees, P. Langefeld and F. Seidel, "B-quark tagging using neural networks and multivariate statistical methods - a comparison of both techniques," Nuclear Instrument Methods A329, pp. 501-517, 1993.

[26] R. P. Brent, "Fast training algorithms for multilayer neural nets," IEEE Transaction on Neural Networks, vol. 2, pp. 346-354, 1991.

[27] M. D. Richard and R. P. Lippmann, "Neural network classifiers estimate bayesian a posterior probabilities,", Neural Computation, vol. 3, pp. 461-483, 1991.

[28] A. Doering, M. Galicki and H. Witte, “Structure optimization of neural networks with the A* algorithm,” IEEE Transaction on Neural Networks, vol. 8, pp. 307-317, 1997.

[29] K. Kumar, C. Tan, and R. Ghosh, "Detecting chaos in financial market," Computational Intelligence for Modeling and Automation, CIMCA99 Vienna, IOS Press, vol. 55, pp. 148-53, 1999.

[30] K. Kumar, C. Tan, and R. Ghosh, “Using chaotic component and ANN for forecasting exchange rates in foreign capital market,” Proceedings of the Advancement Investment Technology 1999 Conference, Ed. E. Tonkes et al., pp. 147-162, 2000.

[31] K. Kumar, C. Tan, and R. Ghosh., “Detecting chaos in capital market and forecasting using ARIMA and ANN”, 14th Australian Statistical Society Conference, Gold Coast, pp. 107, 1998.

[32] K. Kumar, C. Tan, and R. Ghosh., “Detecting non-linearity for Australian stock,” Quantitative Finance, Sydney, p. 41, 1998.

[33] K. Kumar, C. Tan, and R. Ghosh., “Using chaos and artificial neural network to predict foreign exchange market," QFC Conference, Queensland, 1999.

[34] K. Kumar, C. Tan, and R. Ghosh., “Detection of chaos in financial time series," ANZIAM 98, Gold Coast, 1998.

[35] K. Kumar, C. Tan, and R. Ghosh, “A hybrid system for financial trading and forecasting,” South Asian Journal of Management, vol. 5, pp. 1-8, 1999.

[36] R. P. Lipman, “An introduction to computing with neural nets,” IEEE ASAP magazines, April 1987.

[37] S. I. Gallant, “Perceptron based learning algorithm,” IEEE Transaction on Neural Networks, vol. 1, no. 2, pp. 179 -191, 1990.

[38] B. Widrow, “ADALINE and MADALINE – 1963,” Proceedings of IEEE 1st International Conference on Neural Networks, vol 1, pp. 143-157, Plenary Speech, 1987.

[39] B. Widrow and M. E. Hoff, “Adaptive switching circuits,” 1960 IRE WESCON Convention Record, New York, NY, Part 4, pp. 96-104, 1960.

[40] F. C. Chen and C. C. Liu, “Adaptively controlling nonlinear continuous – time system using multilayer neural networks,” IEEE Transaction on Automatic Control, vol. 39, no. 6, pp. 1306-1310, 1994.

[41] J. A. K. Suykens, J. Vandewalle, and B. De Moor, ”NL_q theory checking and imposing stability of recurrent neural networks for nonlinear modeling,” IEEE

Transaction on Signal Processing (Special issue on Neural Networks for Signal Processing), vol. 45, no. 11, pp. 2682-2691.

[42] B. Widrow and M. A. Lehr, "30 years of adaptive neural networks: Perceptron, MADALINE, and backpropagation," IEEE Proceedings, vol. 78, no. 9, pp. 1415 -1441, 1990.

[43] R. Winter and B. Widrow, "MADALINE Rule II: a training algorithm for neural networks," Neural Networks, vol. 1, suppl. no. 1, p. 148, 1988.

[44] W. Schiffmann, M. Joost and R. Werner, "Comparison of optimized backpropagation algorithms," Proceedings of The European Symposium on Artificial Neural Networks, Brussels, pp. 97-104, 1993.

[45] D. E. Rumelhart et al., "Parallel distributed processing," Cambridge, MA, MIT Press, vol. 1, 1988.

[46] P. J. Werbos, "Beyond regression: New tools for prediction and analysis in the behavioral sciences," PhD dissertation, Harvard University, Cambridge, MA 1974.

[47] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representation by error propagation," Parallel Distributed Processing, Exploring the Macro Structure of Cognition, Cambridge, MA: MIT Press, 1986.

[48] E. M. Johansson, F. U. Dowla, and D. M. Goodman, "Backpropagation learning for multi-layer feed-forward neural networks using the conjugate gradient method," International Journal of Neural Systems, vol. 2, no. 4, pp. 291-301, 1991

[49] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," Psychological Review, vol. 65, pp. 386-408, 1958.

[50] R. A. Jacobs, "Increased rates of convergence through learning rate adaptation," Neural Networks, vol. 1, pp. 295-307, 1988.

[51] Y. Yamamoto and P. N. Nikiforuk, "A new learning algorithm and its application to adaptive control," Proceedings of IASTED Conference Control Application, pp. 489-492, 1999.

[52] B. Verma, "Fast training of multi layer perceptron," IEEE Transaction on Neural Networks, vol. 8, no. 6, pp. 1314-1321, 1997.

[53] J. Hertz, A. Krogh, B. Lautrup, and T. Lehmann, "Nonlinear backpropagation: Doing backpropagation without derivatives of the activation function," IEEE Transaction on Neural Network, vol. 8, no. 6, pp. 1321-1327, 1997.

[54] R. D. Lene, R. Capparuccia, and E. Merelli, "A successive over relaxation backpropagation algorithm for neural network training," IEEE Transaction on Neural Networks, vol. 9, no. 3, pp. 381- 388, 1998.

[55] M. Riedmiller and H. Braun, "A direct adaptive method for faster backpropagation learning: The RPROP algorithm," Proceedings of International Conference on Neural Networks, San Fransisco , pp. 586-591, 1993.

[56] Veitch, A. Craig, G. Holmes, "A modified quickprop algorithm," Neural Computation, vol. 3, no. 3, pp. 310-311, 1991.

[57] T. Tollenaere, "SuperSAB: Fast adaptive backpropagation with good scaling properties," Neural Networks , vol. 3, pp. 561-573, 1990.

[48] S. E. Fahlman, "An empirical study of learning speed in back-propagation networks," Carnegie-Mellon Computer science Rpt. CMU-CS-88-162, 1993.

[59] R. Battiti, "First and second-order methods for learning: between steepest descent and newton's method," Neural Computation, vol. 4, 1992, pp. 141-166, 1992.

[60] E. M. Johansson, F. U. Dowla and D. M. Goodman, "Backpropagation learning for multilayer feed-forward neural networks using the conjugate gradient method," International Journal of Neural Systems, vol. 2, no. 4, pp. 291-301, 1991.

[61] M. F. Moller, "A scaled conjugate gradient algorithm for fast supervised learning," Neural Networks, vol. 6, pp. 525-533, 1993.

[62] W. H. Press, B. P. Flannery, S. A. Teukolsky and W. T. Vetterling, "Numerical recipes in C: The art of scientific computing," Cambridge University Press, Cambridge UK, 1986.

[63] Wolf, R. Paul, and D. Ghilani, "Adjustment Computations," John Wiley and Sons Inc, New York, ISBN 0-471-16833-5, 1997.

[64] Nash, G. Stephan, Sofer, and Ariela, "Linear and Nonlinear Programming," McGraw-Hill Companies, Inc, New York, ISBN 0-07-046065-5, 1996.

[65] P. Whittle, "Prediction and regularization by linear least square methods," Van Nostrand, Princeton, N.J., 1963.

[66] S. D. Goggin, K.E. Gustafson, and K. M. Johnson, "An asymptotic singular value decomposition analysis of nonlinear multilayer neural networks," International Joint Conference on Neural Networks, vol. 1, pp 85 – 790, 1991.

[67] S. A. Burton, "A matrix method for optimizing a neural network," Neural Computation, vol. 3, no. 3, pp. 450 – 459, 1991.

[68] Zi-Qin Wang, M. T. Manry, J. L. Schiano, "LMS learning algorithms: misconceptions and new results on convergence," IEEE Transactions on Neural Networks, vol. 11, no. 1, pp. 47 – 56, January 2000.

[69] X Yao and Y Liu, "Evolutionary artificial neural networks that learn and generalise well", IEEE International Conference on Neural Networks, Sheraton Washington Hotel, Washington DC, USA, IEEE Press, New York, NY, 3-6 June 1996.

[70] K. C. Tsui and M. D. Plumbley, "A new hillclimber for classifier systems," Proceedings of IEE/IEEE Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications (GALESIA'97), 1-4 September, Glasgow, 1997. url = "citeseer.nj.nec.com/tsui97new.html".

[71] K. C. Tsui and M. D. Plumbley, "A genetic-based adaptive hillclimber," H.-J. Zimmerman Editor, Proceedings of 5th European Congress on Intelligent Techniques and Soft Computing, EUFIT'97, vol. 1, pp. 799-803. ELITE Foundation, Aachen, Germany, 1997.

[72] E. J. W. Boers and H. Kuiper, "Biological metaphors and the design of modular artificial neural network," Master's thesis, Leiden University, 1992.

[73] J. M. Baldwin, "A new factor in evolution," American Naturalist, vol. 30, pp. 441-451, 1896.

[74] P. Prusinkiewicz and J. Hanan, "Lindenmayer systems, fractals and plants," Lecture notes in Biomathematics, Springer-Verlag, New York , vol. 79, pp. 17-21, 1989.

[75] I. Rechenberg, "Cybernatic solution path of an experimental problem," Royal Aircraft Establishment, Library Translation no. 1122, Farnborough, Hants, U.K, Aug, 1965.

[76] D. Whitley, T. Starkweather, and C. Bogart, "Genetic algorithms and neural networks - optimizing connections and connectivity," Parallel Computing, vol. 14, pp. 347-361, 1990.

[77] D. Montana and L. Davis, "Training feed forward neural networks using genetic algorithms," Proceedings of 11th International Joint Conference on Artificial Intelligence IJCAI-89, vol. 1, pp. 762-767, 1989.

[78] D. E. Goldberg, "Genetic algorithms in search, optimization, and machine learning," Addison-Wesley, Reading, MA, 1989.

[79] J. H. Holland, "Adaptation in natural and artificial systems," Ann Arbor, MI: The University of Michigan Press, 1975.

[80] D. Whitley, "The GENITOR algorithm and selective pressure: Why rank based allocation of reproductive trials is best," Proceedings of 3rd International

Conference on Genetic Algorithms and their Application (J.D. Schaffer, Editor), Morgan Kaufmann, San Maeteo, CA, pp. 116-121, 1989.

[81] T. P. Caudell and C. P. Dolan, "Parametric connectivity: training of constrained networks using genetic algorithm," Proceedings of 3rd International Conference on Genetic Algorithms and their Application (J.D. Schaffer, Editor), Morgan Kaufmann, San Maeteo, CA, pp. 370- 374, 1989.

[82] R. Collins and D. Jefferson, "Selection in massively parallel genetic algorithms," Proceedings of 4th International Conference on Genetic Algorithms, Morgan-Kaufmann, pp. 249-256, 1991.

[83] K. Dejong, "An analysis of the behavior of a class of genetic adaptive systems," PhD Dissertation, Department of Computer and Communication Sciences, University of Michigan, Ann Arbor, 1975.

[84] L. Eshelman, "The CHC adaptive search algorithm," Foundations of Genetic Algorithms, G. Rawlins, Editor Morgan-Kaufmann, pp. 256-283, 1991.

[85] D. Goldberg and K. Deb, "A comparative analysis of selection schemes used in genetic algorithms," Foundations of Genetic Algorithms, G. Rawlins, Editor Morgan-Kaufmann, pp. 69-93, 1991.

[86] G. Liepins and M. Vose, "Representation issues in genetic algorithms," Journal of Experimental and Theoretical Artificial Intelligence, vol. 2, pp. 101-115, 1990.

[87] S. Wright, "The roles of mutation, inbreeding, crossbreeding and selection in evolution," Proceedings of 6th International Congress on Genetics, pp. 356-366, 1932.

[88] L. Booker, "Improving search in genetic algorithms," Genetic Algorithms and Simulating Annealing, L. Davis, Editor Morgan-Kaufman, pp. 61-73, 1987.

[89] H. Kitano, "Empirical studies on the speed of convergence of neural network training using genetic algorithms," Proceedings of 8th National Conference on Artificial Intelligence (AAAI-90), MIT Press, Cambridge, MA 1990.

[90] A. P. Topchy and O. A. Lebedko, "Neural network training by means of cooperative evolutionary search," Nuclear Instrumentation and Methods in Physics Research, Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, vol. 389, no. 1-2, pp. 240-241, 1997.

[91] S. L. Hung and H. Adeli, "A Parallel genetic/neural network learning algorithm for MIMD shared memory machines," IEEE Transactions on Neural Networks, vol. 5, no. 6, pp. 900-909, 1994.

[92] J. M Yang, C. Y. Kao, and J. T. Horng, "Evolving neural induction regular language using combined evolutionary algorithms," Proceedings of 1st joint

conference on Intelligent Systems/ISAI/IFIS, (Piscataway, NJ, USA), IEEE Press, 1996.

[93] P. Zhang, Y. Sankai, and M. Ohta, "Hybrid adaptive learning control of nonlinear system," Proceedings of American Control Conference, vol. 4, pp. 2744-2748, 1995.

[94] X. Yao, "Evolution of connectionist networks," Preprints of the Symposium on Artificial Intelligence, Reasoning, and Creativity, Editor T. Dartnall, Brisbane, pp. 49-52, 1991.

[95] M. Frean, "The upstart algorithm: A method for constructing and training feedforward neural networks," Neural Computation, vol. 2, pp. 198-209, 1990.

[96] A. Roy, L. S. Kim, and S. Mukhopadhyay, "A polynomial time algorithm for the construction and training of a class of multiplayer perceptrons," Neural Networks, vol. 6, pp. 535 – 545, 1993.

[97] S. W. Wilson, "Perception redux: Emergence of structure," Physica D, vol. 42, pp 249-256, 1990.

[98] E. Mjolsness, D. H .Sharp, and B. K .Alpert, "Scaling, machine learning, and genetic neural nets," Advances in Applied Mathematics, vol . 10, pp. 137 – 163, 1989.

[99] D. G .Stork, S. Walker, M. Burns, and B. Jackson, "Preadaptation in neural circuits," Proceedings of International Joint Conference on Neural Networks, vol. 1, pp. 202- 205, Washington DC, Lawrence Erlbaum Associates, Hillsdale, NJ, 1990.

[100] Y. Liu and X. Yao, "Evolutionary design of artificial neural networks with different node transfer functions," Proceedings of 3rd IEEE International Conference on Evolutionary Computation (ICEC'96), Nagoya, Japan, pp. 670-675, 20-22 May 1996.

[101] S. A. Harp, T. Samad, and A. Guha, "Towards the genetic synthesis of neural networks," Proceedings of 3rd International Conference on Genetic Algorithms and their Applications (J.D.Schaffer, Editor), Morgan Kaufmann, San Mateo, CA, pp. 360– 369, 1989.

[102] D. L. Prados, "New learning algorithm for training multilayered neural networks that uses genetic algorithm techniques," Electronic Letters , vol. 28, no. 16, pp. 1560 – 1561, July 1992

[103] D. L. Prados, "Training multilayered neural networks by replacing the least fit hidden neurons," Proceedings of IEEE SOUTHEASTCON , pp. 634 – 637, 1992.

[104] J. J. Spofford, and K. J. Hintz, "Evolving sequential machines in amorphous neural networks," Artificial Neural Networks (T. Kohonen, K. Makisara, O. Simula and J. Kangas Editor), Elsevier Science Publishes, B.V (North Holland), vol. 1, pp. 973 – 978, 1991.

[105] X. Yao and Y. Liu, "EPNet for chaotic time-series prediction," Proceedings of the First Asia-Pacific Conference on Simulated Evolution and Learning (SEAL'96), Taejon, Korea, pp. 331-342, November, 1996.

[106] M. Koeppen, M. Teunis, and B. Nickolay, "Neural network that uses evolutionary learning," Proceedings of IEEE International Conference on Neural Networks, vol. 5, pp. 635-639, IEEE press, Piscstaway, NJ, USA, 1994.

[107] A. Likartsis, I. Vlachavas, and L. H. Tsoukalas, "New hybrid neural genetic methodology for improving learning," Proceedings of 9th IEEE International Conference on Tools with Artificial Intelligence, Piscataway, NJ, USA, pp. 32-36, IEEE Press, 1997.

[108] S. Omatu and S. Deris, "Stabilization of inverted pendulum by the genetic algorithm," Proceedings of IEEE Conference on Emerging Technologies and Factory Automation, ETFA'96., Piscataway, NJ, USA, vol. 1, pp. 282-287, IEEE Press, 1996.

[109] S. Omatu and M. Yoshioka, "Self tuning neuro PID control and applications," Proceedings of IEEE International Conference on Systems, Man, and Cybernatics, Picatasway, NJ, USA, vol. 3, pp. 1985-1989, IEEE Press, 1997.

[110] A. Abraham, "Neuro-Fuzzy Systems: State-of-the-Art Modeling Techniques," Connectionist Models of Neurons, Learning Process, and Artificial Intelligence, Springer-Verlag Germany, LNCS 2084, Jose Mira and Alberto Prieto (Editors), Spain, pp. 269-276, 2001.

[111] A. Abraham and B. Nath, "Is Evolutionary design the solution for optimising neural networks?" Proceedings of 5th International Conference on Cognitive and Neural Systems (ICCNS 2001), Published by Boston University Press, Boston, USA, 2001.

[112] G. Beliakov and A. Abraham, "Global optimization of neural networks using deterministic hybrid approach," Hybrid Information Systems, Proceedings of 1st International Workshop on Hybrid Intelligent Systems, HIS 2001, Springer Verlag, Germany, pp 79-92, 2002.

[113] S. Ergenziger and E. Thompsen, "An accelerated learning algorithm for multilayer perceptorn: Optimizing layer by layer," IEEE Transaction on Neural Networks, vol. 6, pp. 33-42, 1995.

[114] S. R. Nickolai, "The layer-wise method and the back propagation hybrid approach to learning a feedforward neural network," IEEE Transaction on neural networks, vol. 11, no. 2, pp. 295 – 305, 2000.

[115] S. Singhal and L. Wu, "Training feedforward networks with the extended Kalman algorithm," INASSP-89, International Conference Acoustics, Speech, Signal Processing, vol. 2, pp. 1187-1190, 1989.

[116] G. V. Puskorius, and L. A. Feldamp, "Neurocontrol of nonlinear dynamical systems with Kalman filter trained recurrent networks," IEEE International Conference on Neural Networks, vol. 5, pp. 279-297, 1994.

[117] O. Stan, and E. Kamen, "A local linearized least squares algorithm for training feedforward neural betworks," IEEE Transaction on neural networks, vol. 11, no. 2, pp. 487 – 495, 2000.

[118] D R. Hush, B. Horne, "Efficient algorithms for function approximation with piecewise linear sigmoidal networks," IEEE Transaction on Neural Networks, vol. 9, no. 6, pp. 1129-1141.

[119] G. Zhou, "Advanced neural network training algorithm with reduced complexity based Jacobian deficiency", IEEE Transaction on Neural Networks, vol. 9, no. 3, pp. 448 – 453, 1998.

[120] Y. Xia, H. Leung, E. Bosse, "Neural data fusion algorithms based on a linearly constrained least square method," IEEE Transaction on Neural Networks, vol. 13, no. 2, pp. 320-329, 2002.

[121] J. A. Benediktsson, P. H. Smith, and O. K. Ersoy, "Neural network approaches versus statistical method in classification of multisource sensing data," IEEE Transaction on Geoscience and Remote Sensing, vol. 28, no. 2, pp. 540-550, 1992.

[122] F. Wang, J. Litva, T. Lo, E. Bosse, and H. leung, "Feature mapping data fusion," Proceedings of Inst. Electrical Engineering Radar, Sonar, and Navigation, vol. 143, no. 2, pp. 65-70, 1996.

[123] S. Shah, F. Palmieri, and M. Datum, "Optimal filtering algorithm for fast learning in feedforward neural networks," Neural Networks, vol. 5, pp. 779–787, 1992.

[124] C .S. Leung, K. W. Wong, J. Sum and L. W. Chan, "Online training and pruning for RLS algorithms," Electronics Letter, vol. 32, pp. 2152-2153, 1996.

[125] C .Leung, A. Tsoi, and L. W. Chan, "Two regularizer for recursive least square algorithms in feedforward multilayered neural networks," IEEE Transcation on Neural Networks, vol. 12, no. 6, pp. 1314-1332, 2001.

[126] S. Lawrence, C. Lee Giles, A. C. Tsoi, "What size neural network gives optimal generalization? Convergence properties of backpropagatioin," UMIACS-TR-96-22, 1996.

[127] Marcus Reginald Gallagher, "Multilayer perceptron error surfaces: Visualization, structure and modeling," Department of Computer Science and Electrical Engineering, University of Queensland, St Lucia 4072, Australia, A thesis submitted for the degree of Doctor of Philosophy, June 30, 1999.

[128] L. G. C. Hamey, "XOR has no local minima: A case study in neural network error surface analysis," Neural Networks, vol. 11, issue. 4, pp. 669-681, 1998.

[129] I. G. Sprinkhuizen-Kuyper and E. J. W. Boers, "The error surface of the 2-2-1 XOR network: the finite stationary points," Neural Networks, vol. 11, no. 4, pp. 683-690, 1998.

[130] M. Lehr, "Scaled stochastic methods for training neural networks," PhD thesis, Stanford University, January 1996.

[131] G. L. Martin and J. A. Pittman, "Recognizing hand printed letters and digits using backpropagation learning," Neural Computation, vol. 3, pp. 258-267, 1991.

[132] R. Caruana, S. Lawrence, L. Giles, "Overfitting in neural nets: Back propagation, conjugate gradient, and early stopping," Neural Information Processing Systems, Dehver, Colorado, pp. 402- 408, November 2000.

[133] V. N. Vapnik and A Ya. Chervonenkis, "On the uniform convergence of relative frequencies of events to their probabilities," Theory of Probability and Its Applications, vol. 16, no. 2, pp. 264-280, 1971.

[134] R. Ghosh and B. Verma, "Least square method based evolutionary neural learning algorithm," IEEE International Joint Conference on Neural Networks, pp. 2596 -2601, Washington, IEEE Computer Society Press, USA, 2001.

[135] B. Verma and R. Ghosh, "A novel evolutionary neural learning algorithm," IEEE International Conference on Evolutionary Computation 2002, pp1884-89, Honolulu, USA, 2002.

[136] R. Ghosh and B. Verma, "Finding optimal architecture and weights using evolutionary least square based learning," 9th International Conference on Neural Information Processing, vol. 1, pp. 528 – 532, Singapore 2002.

[137] R. Ghosh and B. Verma, "Finding architecture and weights for ANN using evolutionary based least square algorithm," International Journal on Neural Systems, vol. 13, no. 1, pp. 13-24, 2003.