Deanship of Graduate Studies
Al-Quds University

Meta-Learning Evolutionary Artificial Neural Networks: Using Cellular Configurations for Designing Network Architecture

Asma Hilmi Yousef Abu Salah

M.Sc. Thesis

Jerusalem, Palestine

1427 / 2006

Meta-Learning Evolutionary Artificial Neural Networks: Using Cellular Configurations for Designing Network Architecture

Prepared by:
Asma Hilmi Yousef Abu Salah

B.Sc.: Computer Engineering
Palestine Polytechnic University
Hebron, Palestine

Supervisor: Dr. Yahya Al-Salqan

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Computer Science Department
College of Science and Technology
Al-Quds University

1427 / 2006

Al-Quds University
Deanship of Graduate Studies
Master of Computer Science / Computer Science Department

Thesis Approval

Meta-Learning Evolutionary Artificial Neural Networks: Using Cellular Configurations for Designing Network Architecture

Prepared by: Asma Hilmi Yousef Abu Salah
Registration No: 20111391

Supervisor: Dr. Yahya Al-Salqan

Master thesis submitted and accepted, Date: ……
The names and signatures of the examining committee members are as follows:

1-Head of Committee:…….....Signature: ……..
2-Internal Examiner:……..Signature: ……..
3-External Examiner:...……..... .....Signature: ……..
4-Committee Member:……..……Signature: ……...……....

Jerusalem, Palestine

1427 / 2006

Dedication

To my parents, my husband, my brothers, and my sisters, for their support, care, and love.

Asma Hilmi Yousef Abu Salah

Declaration:

I certify that this thesis submitted for the degree of Master is the result of my own research, except where otherwise acknowledged, and that this thesis (or any part of the same) has not been submitted for a higher degree to any other university or institution.

Signed,

Asma Hilmi Yousef Abu Salah

15 July 2006

Acknowledgment:

I would like to thank my thesis advisor Dr. Yahya Al-Salqan for his valuable guidance and support.

I am very thankful to the Department of Computer Science of Al-Quds University, and to the master program supervisor and committee for their support and encouragement.

I am very grateful to the program chair of the international conference on computational intelligence for modelling, control and automation (CIMCA 2005), to the program chair of the IASTED international conference on artificial intelligence and applications (AIA 2006), and to the reviewers committee for their valuable comments and suggestions on the paper I had published based on my thesis.

Abstract

In this thesis, a meta-learning evolutionary artificial neural network by means of cellular automata (MLEANN-CA) is proposed. It is an adaptive computational framework based on evolutionary learning and local search procedures for automatic design of optimal artificial neural networks using direct and indirect encoding methods. In this proposed framework, the evolutionary cellular configurations are used to, first, design small feed-forward network architectures, and then all the generated architectures are trained and evolved separately using the meta-learning algorithm with the direct evolutionary approach, where four different learning algorithms are used in parallel mode. The neural network architecture, activation function, connection weights, and the learning algorithm with its parameters are adapted according to the problem. The performance of the MLEANN-CA framework is tested and explored, experimentally, using NeuroSolutions and NeuroGenetic Optimizer toolboxes, and two famous chaotic time series. Moreover, the performance of different neural network learning algorithms (backpropagation algorithm, conjugate gradient algorithm, quasi-Newton algorithm and Levenberg–Marquardt algorithm) is explored and evaluated for the two chaotic time series when the architecture was changed. The performance of the MLEANN-CA framework is compared with the previous MLEANN, which used the direct encoding methods for designing architectures, and with the conventional design of ANNs. The results showed how effective and scalable is the proposed MLEANN-CA framework to obtain an efficient design of feed-forward neural network that is smaller, faster and with better generalization performance.

ملخّص

التعليم والتدريب المتعدد للشبكات العصبيّة الصناعيّة المتطورة: استخدام المحاكيات والنظم الخلويّة في تصميم هيكلية الشبكات

في هذه الرّسالة، قمت باقتراح نموذج للتعليم والتدريب المتعدد للشبكات العصبيّة الصناعيّة المتطورة بواسطة المحاكيات الخلويّة (MLEANN-CA). هذا النموذج عبارة عن بناء محوسب ومهيئ يعتمد على عملية تعليم متطورة و إجراءات بحث محلية لعمل تّصميم أوتوماتيكيّ لافضل الشّبكات العصبيّة الصّناعيّة باستخدام طّرق الترميز المباشرة و الغير مباشرة . في هذا البناء والنموذج المقترح، تم استخدام المحاكيات والنظم الخلويّة المتطورة (طريقة ترميز غير مباشرة) لعمل تصميم صغير الحجم لتركيبة وهيكلية الشبكات العصبية (الامامية التغذية). ثمّ بعد ذلك تم عمل تدريب و تطوير لهذه التراكيب الجديدة باستخدام خوارزمية التعليم المتعدد والمتكرر حيث تم تطبيق واستخدام اربع خوارزميات مختلفة بشكل متوازي ومنفصل وبالاستعانة بطريقة الترميز المباشرة المتطورة. ولقد تم تحديد هيكلية الشبكة العصبية الاساسية، الوسيلة المنشطة، اوزان الروابط بين عناصر الشبكة، و خوارزمية التعليم حسب طبيعة المشكلة . في هذه الدراسة، قمت بفحص وتقييم اداء النموذج المقترح من خلال تجارب عديدة استخدمت فيها اثنتين من اشهر المتسلسلات الزمنية الفوضوية و اثنتين من أدوات الفحص: NeuroSolutions و NeuroGenetic Optimizer. اضافة الى ذلك، قمت بعمل فحص واستكشاف لأداء عدد من خوارزميات التعليم الخاصة بالشبكات العصبية وذلك اثناء تغيير شكل وحجم هيكلية الشبكة وتطبيق المتسلسلات الزمنية عليها. واخيرا"، قمت بمقارنة النت ائج الخاصة بالنموذج المقترح مع نموذج سابق استخدم فيه طريقة الترميز المباشرة فقط ومع التصميم التقليدي للشبكات العصبية الصناعية. و قد اظهرت النتائج مدى كفاءة و فاعلية واتزان النموذج الذي اقترحناه في الحصول على تصاميم فعالة لشبكات عصبية صناعية صغيرة الحجم، سريعة التعلم، وباداء افضل واشمل.

Table of Contents

List of Figures

List of Tables

CHAPTER ONE

INTRODUCTION

The human brain provides proof of the existence of massive neural networks that can succeed at those cognitive, perceptual, and control tasks in which humans are successful. The brain is capable of computationally demanding perceptual acts and control activities. The advantage of the brain is its effective use of massive parallelism, the highly parallel computing structure, and the imprecise information-processing capability. The human brain is a collection of more than 10 billion interconnected neurons. Each neuron, as shown is figure (1.1) (Jain, Mao, & Mohiuddin, 1996), is a cell that uses biochemical reactions to receive, process, and transmit information. Treelike networks of nerve fibers called *dendrites* are connected to the cell body or soma, where the cell nucleus is located. Extending from the cell body is a single long fiber called the *axon*, which eventually branches into strands and substrands, and is connected to other neurons through synaptic terminals or synapses. The transmission of signals from one neuron to another at synapses is a complex chemical process in which specific transmitter substances are released from the sending end of the junction. The effect is to raise or lower the electrical potential inside the body of the receiving cell. If the potential reaches a threshold, a pulse is sent down the axon and the cell is 'fired'.

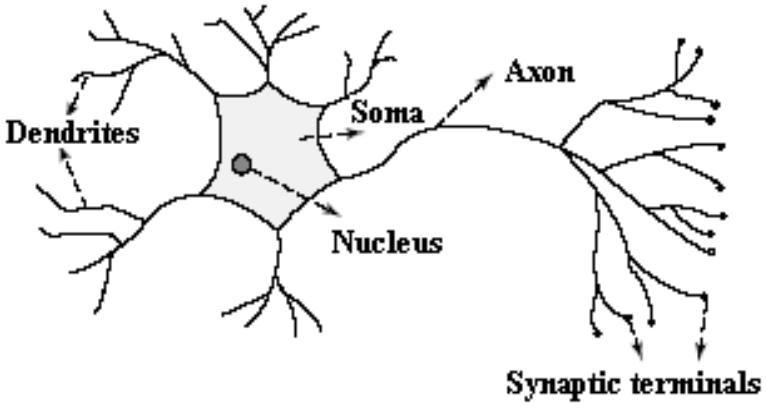


Figure 1.1: Mammalian neuron

Artificial neural networks (ANN) have been developed as generalizations of mathematical models of biological nervous systems. The basic processing elements of neural networks are called *artificial neurons*, or *simply neurons* or *nodes*. In a simplified mathematical model of the neuron, the effects of the synapses are represented by connection weights that modulate the effect of the associated input signals, and the nonlinear characteristic exhibited by neurons is represented by a transfer function. The neuron impulse is then computed as the weighted sum of the input signals, transformed by the transfer function. The learning capability of an artificial neuron is achieved by adjusting the weights in accordance to the chosen learning algorithm (Jain, Mao, & Mohiuddin, 1996), (Lippmann,1987).

Such artificial neural networks are currently one of the most popular techniques that are successfully used in many applications such as: pattern classification, pattern recognition, task

of optimization, medical diagnosis, financial modeling, etc. Many of the conventional ANNs being designed are statistically quite accurate but they still leave a bad taste with users who expect computers to solve their problems accurately. The important drawback is that the designer has to specify manually the number of neurons, their distribution over several layers and interconnection between them. As the complexity of the problem domain increases, manual design becomes more difficult and unmanageable. Several methods have been proposed to automatically construct ANNs for reduction in network complexity and the evolutionary algorithms (EAs) are one of these methods. The interest in evolutionary algorithms for designing ANN architecture has been growing in recent years as they can evolve towards the optimal architecture without outside interference, thus eliminating the tedious trial and error work of manually finding an optimal network (Korning, 1995), (Yoon, Holmes, & Langholz, 1994), (Yao, 1999), (Caudell & Dolan, 1989), (Abraham, 2004), (Abraham & Nath, 2001), (Binos, 2003), (Braun & Weisbrod, 1993), (Belew, McInerney, & Schraudolph, 1991), (Yao & Liu, 1998), (Andersen & Tsoi, 1993). The advantage of the automatic design over the manual design becomes clearer as the complexity of ANN increases.

Despite many advantages in using evolutionary algorithms for designing artificial neural network architectures, some aspects require improvements. A notable problem is that the evolutionary algorithms are inefficient in fine tuning local search, although they are good at global searches (Yao, 1999), (Abraham, 2002), (Yao, 1993). This is especially true for genetic algorithms (GA's). The efficiency of evolutionary algorithms can be improved significantly by using a hybrid learning approach that incorporates the local search procedure into the evolution. Evolutionary algorithms are used to, first, locate a good region in the space and then a local search procedure is used to find a near optimal solution in this region. Several hybrid learning approaches had been successfully used for evolving neural network topology and/or weights (Abraham & Nath, 2000), (Abraham, 2004), (Abraham, 2002), (Yao & Liu, 1997), (Belew, McInerney, & Schraudolph, 1991), (Wong, Chung, & Wong, 1998), (Magoulas, Plagianakos, & Vrahatis, 2001), (Hendtlass & Podlena, 1995). One of these hybrid learning approaches is called meta-learning evolutionary artificial neural networks (MLEANN) (Abraham, 2004), (Abraham, 2002). It can be considered as an automatic computational framework that used a direct encoding method for the adaptive optimization of ANNs. The main aim of using the MLEANN framework is to improve the learning process and to obtain an efficient design of neural networks with faster convergence.

1.1 Justification

Until now, the MLEANN framework (Abraham, 2004) uses only the direct encoding methods for optimizing the neural network architectures. These direct encoding methods base on the codification of the complete network into the chromosome. They are relatively simple and straightforward to implement but requires much larger chromosomes especially for ANNs with complex architectures (Caudell & Dolan, 1989), (Yao, 1993), (Branke, 1995), (Koza & Rice, 1991), (Yao & Liu, 1998), (Braun & Weisbrod, 1993). This could end in a too huge space search that could make the method impossible in practice. On the other hand, implementation of crossover operator for the chromosome is often difficult due to production of non-functional offsprings. An alternative more interesting for optimizing the ANN architecture are the indirect encoding methods such as cellular automata (Gutierrez, Isasi,

Molina, Sanchis, & Galvan, 2001), (Harp, Samad, & Guha, 1989), (Kitano, 1990), (Molina, Galván, Isasi, & Sanchis, 2000-A), (Gruau & Whitley, 1993), (Gruau, Whitley, & Pyeatt, 1995), (Harp, Samad, & Guha, 1990), (Koza & Rice, 1991), (Molina, Galván, Isasi, & Sanchis, 2000-B), (Chval, 2002), (Hussain & Browse, 1998), (Jacob & Rehder, 1993), (Luke & Spector, 1996). These methods concentrate on codifying a compact representation of the networks reducing the length of the genotype and avoiding the scalability problem. In this thesis, an automatic computational framework: meta-learning evolutionary artificial neural network by means of cellular automata (MLEANN-CA) is proposed. This proposed framework combines the local search methods with the evolutionary learning in order to obtain an efficient design of neural networks. The MLEANN-CA framework was explored and simulated using NeuroSolutions and NeuroGenetic Optimizer toolboxes, and two famous chaotic time series.

1.2 Thesis Objective

The primary objective of this thesis is to propose and design an efficient and effective framework: meta-learning evolutionary artificial neural network by means of cellular automata (MLEANN-CA). It is an adaptive computational framework based on direct and indirect evolutionary computation and local search methods for automatic design of optimal ANN. Using this framework significantly improves the learning process, increase the scalability, enhance the predictive accuracy of the results, and obtain a small and efficient design of neural networks with faster convergence and better generalization performance.

1.3 Contribution

In this thesis, we propose a hybrid meta-heuristic learning approach (MLEANN-CA) combining evolutionary learning and local search methods using direct and indirect evolutionary approaches (Abu Salah & Al-Salqan, 2006-A), (Abu Salah & Al-Salqan, 2006-B). This thesis work moves forward the research on proposing and designing the MLEANN-CA framework in the following way:

- We use the evolutionary cellular configurations for designing small feed-forward neural network architectures.

- We apply the meta-learning algorithm with the direct evolutionary approach for training and evolving, separately, the new generated architectures of neural networks (after optimized by cellular configurations) using different learning algorithms in parallel mode.

- We test, investigate, and explore the performance of the MLEANN-CA framework using Neurosolution and NeuroGenetic Optimizer toolboxes and two famous chaotic time series. We also explore and evaluate the performance of different neural network learning algorithms for the two chaotic time series when the architecture was changed.

- We compare the tested results of the MLEANN-CA with the previous MLEANN framework that used the direct encoding methods, and with the conventional design of

ANNs. The empirical results should indicate that the proposed MLEANN-CA framework is important and efficient in designing optimal ANNs that are smaller, faster, and with better generalization performance.

1.4 Thesis Overview

The rest of this thesis is organized as follows:

In chapter two, Background Knowledge, an overview about the artificial neural networks and the learning algorithms is provided. Moreover, the evolutionary and genetic algorithms are presented and discussed. The definition of the cellular automata and the meta-learning concept is also introduced.

In chapter three, Literature Review, a survey of the related works is presented. It is organized in two subtopics: Pure Evolutionary Algorithms for Training and Evolving Artificial Neural Networks, and Hybrid Evolutionary-Gradient Search Algorithms for Training and Evolving Artificial Neural Networks.

In chapter four, Meta-Learning Evolutionary Artificial Neural Networks: By Means of Cellular Automata, the proposed framework (MLEANN-CA) is discussed and presented in details.

In chapter five, Experiments and Results, the proposed framework is simulated and tested using two efficient toolboxes and two famous chaotic time series. The performance of the MLEANN-CA is explored and evaluated, and the results are discussed and compared with other approaches (i.e. MLEANN and the conventional design of ANNs).

Finally in chapter six, Conclusions and Future Works, the main conclusions are listed and some recommended suggestions and ideas are provided as future works.

We provide short description about the Neurosolution and NeuroGenetic Optimizer toolboxes that are used for the experiments simulations beside number of windows and screens that appeared during the experiments in appendix A.

CHAPTER TWO

BACKGROUND KNOWLEDGE

This chapter describes the paradigm of artificial neural network and number of its learning algorithms. Beside that, it introduces the concepts of evolutionary algorithms and the genetic algorithms. The description of the cellular automata is also presented. Furthermore, this chapter introduces the meta-learning concept and its benefits. The material here is general; it is intended to clarify the concepts and paradigms used throughout this thesis. The topics related directly to the subject of the thesis, i.e. the Evolutionary Artificial Neural Networks, are discussed in details in subsequent chapters.

2.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) are a component of artificial intelligence that simulates real brain's neurons. Also known as *parallel distributed processing*, or *connectionist models,* artificial neural networks are information processors inspired by the way the highly interconnected structures of the brain process information (Jain, Mao, & Mohiuddin, 1996), (Lippmann,1987). Artificial neural networks are mathematical models that emulate some properties observed from the biological neural network: the knowledge is acquired by the network through a learning process and the synaptic weight is used to store the knowledge. Computations are performed through the passing of signals within a structured arrangement of highly interconnected processing units (neurons) in response to a given input signal. The artificial neural network model was introduced by McCulloch and Pitts, after the definition of the computational model for the traditional perceptron in 1943. This is an artificial neuron with a hard-limiting activation function. Since that, artificial neural networks have been implemented to solve a variety of problems involving pattern classification, pattern recognition, task of optimization, medical diagnosis, and financial modeling.

2.1.1 Artificial Neural Networks Components:

2.1.1.1 Artificial Neurons:

The basic element of an artificial neural network is the artificial neuron which simulates some of the operations the natural neuron can perform. This artificial neuron is shown in the following figure (2.1) (Lippmann,1987), (Jain, Mao, & Mohiuddin, 1996):

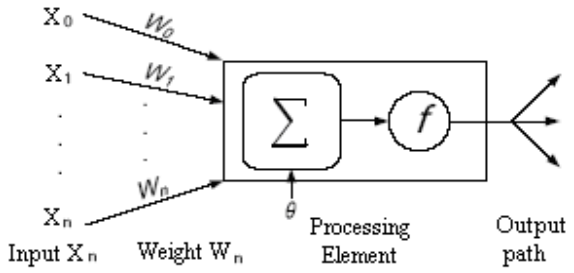


Figure 2.1: The artificial neuron model

The neuron receives as inputs the outputs from other neurons, if the combined strength of the signal reaches a specific threshold; the neuron sends a signal to all the neurons waiting for the output. This process can be described by the following equation (Yao, 1999):

$$y = f\left(\left[\sum_{i=1}^{n} w_i . x_i\right] - \theta\right)$$

where symbols x_i, represent the strength of the input signals, w_i, represent the connection strengths of the given input signal, and the output is represented by the symbol y, θ is a threshold value (or bias), and f is the neuron's activation function. Different activation functions (also known as transfer functions) were found and the most commonly ones are shown in the following figure (2.2) (Jain, Mao, & Mohiuddin, 1996):

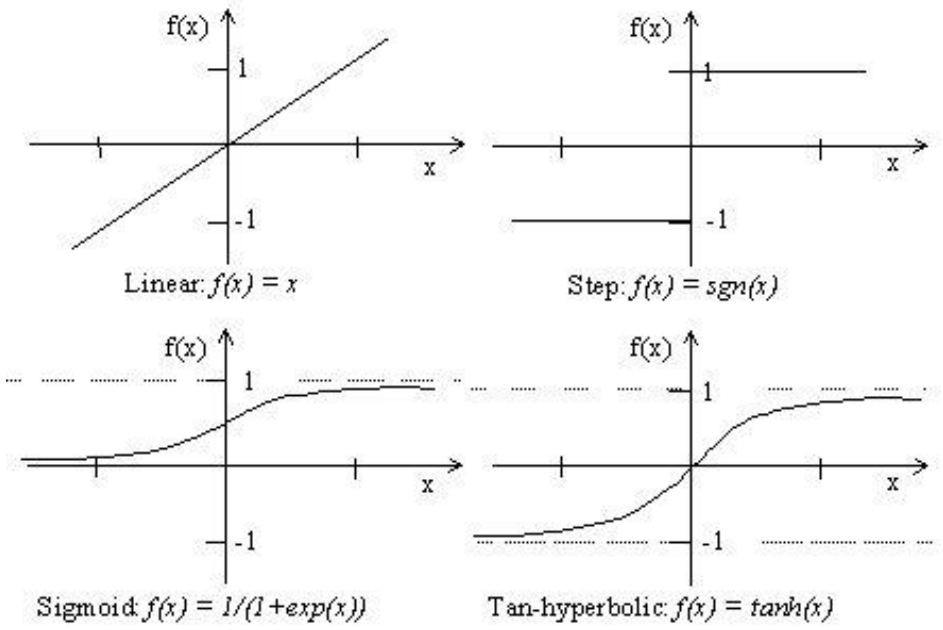


Figure 2.2: Most commonly used transfer functions

2.1.1.2 Architectural Elements of an Artificial Neural Network:

The basic components of neural network architecture are neurons, the layers, and neuron connection. A neural network consists of a set of neurons highly interconnected, grouped into three types of layers: the input layer, output layer and the hidden layers. The behavior of the neural network depends on the interaction between the neurons. Interaction between network components depends on the type of connection that is used to pass messages between neurons. There are two major types of synaptic connections: feed forward and feedback connections (Jain, Mao, & Mohiuddin, 1996), (Rumelhart, Hinton, & Williams, 1986). It is important to highlight that synaptic connections may be fully interconnected or partially interconnected. In feed-forward neural networks, connections are used to propagate the output from the neurons

of a lower layer to neurons of an upper layer as shown in figure (2.3) (a) (Jain, Mao, & Mohiuddin, 1996). They have the property of being static, producing only one output pattern for each input pattern. The feed forward networks could be single-layer or multi-layer (Rumelhart, Hinton, & Williams, 1986). Single-layer feed forward networks consist of input and output layers only, where the multi-layer feed forward network contains at least one hidden layer of nodes that receives connections from the previous adjacent layer of nodes.

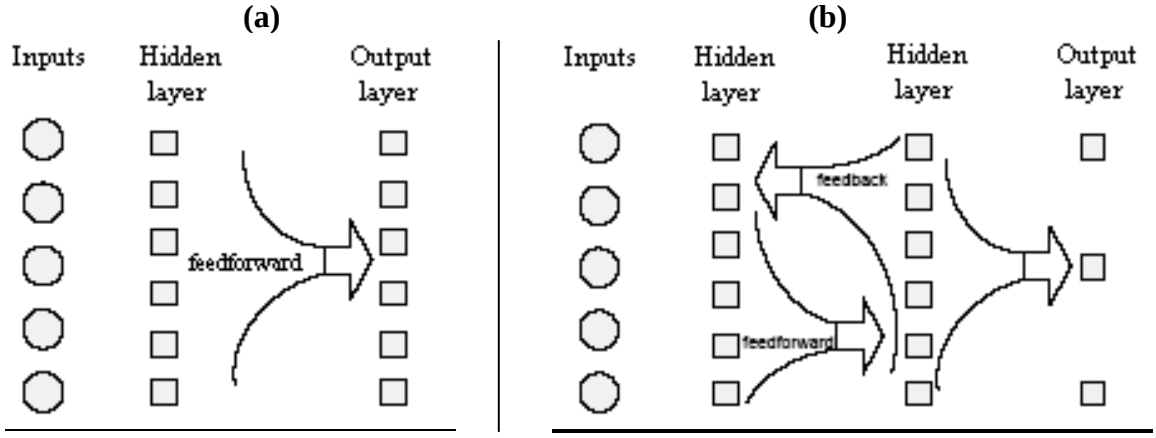


Figure 2.3: (a) Feed Forward connections **(b)** Feedback connections

Artificial neural networks with an architecture that includes feedback connections are recurrent or feedback neural networks. The feedback connections are used to send the output from neurons of an upper layer back to neurons of a lower layer, as shown in figure (2.3) (b). As a result, feedback neural networks are dynamic systems, entering more than one state for each new input pattern (Jain, Mao, & Mohiuddin, 1996). In this research, the feed forward networks with multilayer perceptron (MLP) will be used in the experiments as in Ajith's work (Abraham, 2004) for easy comparisons.

2.1.2 Learning Process for Artificial Neural Networks:

The purpose of neural network training is to produce appropriate output patterns for corresponding input patterns. It is achieved by an iterative learning process that updates the connection weights based on the neural network response to a set of training input patterns. Learning algorithms (processes) in an artificial neural network are classified into: supervised, reinforcement, and unsupervised learning (Jain, Mao, & Mohiuddin, 1996), (Lippmann,1987). Supervised learning occurs when the correct output pattern is known and used during training. It is based on direct comparison between the actual output and the desired correct output. Reinforcement learning is a special case of supervised learning where the exact desired output is unknown. It is based only on the information of whether or not the actual output is correct. Unsupervised learning does not require a correct output to be available during training. It is based on the correlations among input data. The essence of a learning algorithm is the learning rule, i.e., a weight-updating rule which determines how connection weights are changed. Examples of popular learning rules include the delta rule, the Hebbian rule, the anti-Hebbian rule, the competitive learning rule, etc.

2.1.3 Artificial Neural Network Learning Algorithms:

This subsection discusses the most popular supervised learning algorithms that we will use in our research for training multi-layer feed forward neural networks. These are: Backpropagation, Conjugate Gradient Descent, Scaled Conjugate Gradient, Quasi-Newton, and Levenberg-Marquardt.

2.1.3.1 Training Multilayer Perceptrons:

For any MLP, once the number of layers and number of units in each layer has been selected, the network's weights and thresholds must be set (or adjusted) so as to minimize the prediction error made by the network (Molar, 1997). This is the role of the *training / learning algorithms*. The Learning algorithms differ from each other in the way in which the adjustment $\Delta \boldsymbol{w}_{kj}$ to the synaptic weight $\boldsymbol{w}_{kj}$ is formulated (Battiti, 1992). The error of a particular configuration of the network can be determined by running all the training cases through the network, comparing the actual or predicted output generated with the desired or target outputs. The differences are combined together by a cost (error) function to give the network error. The most common cost function is the sum of the squared differences between the networks actual output and the desired output. This is commonly known as the mean-squared error (MSE) cost function.

$$j = E\left[\frac{1}{2}\sum_{k} \mathrm{e}_k^2(\mathrm{n})\right], \text{ where } \mathrm{e}_k(\mathrm{n}) = \mathrm{t}_k(\mathrm{n}) - \mathrm{y}_k(\mathrm{n})$$

where $\mathbf{t_k}$ (*n*) denotes the desired outcome (response) for the $\mathbf{k^{th}}$ neuron at time **n**, $\mathbf{y_k}$ (*n*) is the actual response of the neuron, and $\mathbf{e_k}$ (n) is the difference between the desired response and the actual response (error signal). Here, summation runs over all neurons in the output layer of the network. This method has the task of continually search for the bottom of cost function in iterative manner. Minimization of the cost function $\boldsymbol{j}$ with respect to free parameters of the network leads to so-called method of Gradient Descent $\mathbf{g_n}$ (the first derivative of the cost function) (Molar, 1997). In practice, there are four types of optimization algorithms that are used to minimize the cost function $\boldsymbol{j}$. These algorithms are: back-propagation (gradient descent), conjugate gradients, quasi-Newton, and Levenberg Marquardt. A common feature of these training algorithms is the requirement of repeated efficient calculation of gradients.

A plot of the cost function versus the synaptic weights characterizes the neural network consists of a multidimensional surface called error surface (as shown in figure 2.4) (Burney, Jilani , & Ardil, 2004). The neural network consists of cross-correction learning algorithm to start from **n** arbitrary point on the error surface (initial weights) and then move towards global minima, in step-by-step fashion.

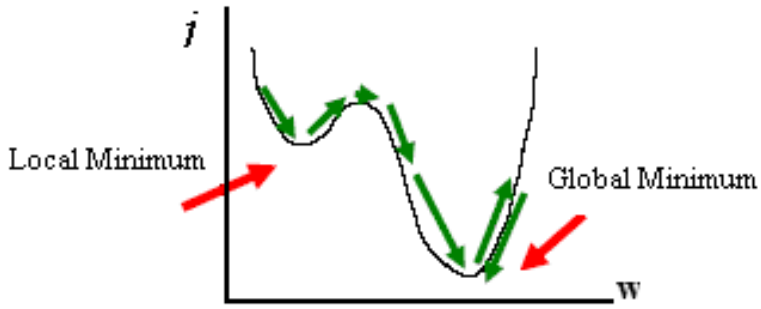


Figure 2.4: Quadratic error surface with local and global minima

In a linear model with sum squared error function, this error surface is a parabola (a quadratic), which means that it is a smooth bowl-shape with a single minimum. It is therefore "easy" to locate the minimum. In case of non-linear model (neural network), the error surfaces are much more complex, and are characterized by a number of unhelpful features, such as many local minima, flat-spots and plateaus, saddle-points, and long narrow ravines (Burney, Jilani , & Ardil, 2004). Therefore, it is not possible to analytically determine where the global minimum of the error surface is, and so neural network training is essentially an exploration of the error surface. In presence of many plateaus, training will get slow. To overcome this situation momentum is introduced that forces the iterative process to cross saddle-points and small landscapes(Molar, 1997).

2.1.3.2 Back-Propagation:

Back propagation is a training algorithm used for training multi-layer feed forward neural networks that have nonlinear differentiable activation (transfer) functions. Based on the generalized delta rule, backpropagation is a gradient descent algorithm that updates the network weights and biases in the direction in which the performance function decreases most rapidly - the negative of the gradient of the cost function (Hinton, 1989), (Rumelhart, Hinton, & Williams, 1986), (Battiti, 1992), (Burney, Jilani , & Ardil, 2004). The gradient of the cost function (the first derivative of the network error with respect to the weights) and the weights updating are given by:

$$W_{n+1} = W_n + \Delta W_n$$

$$\Delta W_n = -\alpha \frac{\partial E}{\partial W} = -\alpha \nabla j_n$$

$$W_{n+1} = W_n - \alpha \; g_n$$

where $\mathbf{w_n}$ is a vector of current weights and biases, $\mathbf{g_n}$ is the current gradient, and $\boldsymbol{\alpha}$ is the learning rate (step-size) that controls how big a step is taken in the negative gradient direction (defines the proportion of error for weight updating). The learning parameter has a profound impact on the performance of convergence of learning (Burney, Jilani , & Ardil, 2004). The negative sign indicates that the new weight vector $\mathbf{w}_{n+1}$ is moving in a direction opposite to that of the gradient. A momentum term ($\boldsymbol{\mu}$) can also be added to stabilize the learning in the

algorithm. The momentum encourages movement in a fixed direction, so that if several steps are taken in the same direction, the algorithm "picks up speed", which gives it the ability to escape local minimum, and also to move rapidly over flat spots and plateaus.

$$W_{n+1} = W_n - \alpha\, g_n + \mu\, W_{n-1} \quad ,0 \le \mu < 1$$

An outline of the back-propagation algorithm is given as follows (Lippmann,1987):

1. Initialize weights of the network with small random values.
2. Choose an input and desired output pair.
3. Propagate the activation of the input layer to the hidden layer, and calculate the activation of the hidden nodes using sigmoid function.
4. Propagate the activation of the hidden nodes to the output layer, and calculate the activation of the actual output using sigmoid function.
5. Calculate the errors (deltas) of the output layer.
6. Compute the errors (deltas) for the hidden layer.
7. Adjust the weights between the hidden layer and output layer.
8. Adjust the weights between the input layer and hidden layer.
9. Repeat steps 4 to 8 until the total error of the network is small enough for each of the training-vector pairs in the training set.

2.1.3.3 Faster Training Algorithms:

Earlier in the previous paragraphs, we discussed how the *back propagation* algorithm performs gradient descent on the error surface. This method is often too slow for practical problems. It does not produce the fastest convergence (Burney, Jilani , & Ardil, 2004), (Battiti, 1992), (Schiffmann, Joost, & Werner, 1993). In the following paragraphs, we discuss other learning algorithms that can converge from ten to one hundred times faster than the backpropagation algorithm. Theses algorithms are: Conjugate Gradient Descent, Scaled Conjugate Gradient, Quasi-Newton, and Levenberg-Marquardt which are very successful forms of two types of algorithm: line search and model-trust region approaches. They are collectively known as second order training algorithms / faster training algorithms.

❖ **Line Search and Trust-region Models**

A line search algorithm (Burney, Jilani , & Ardil, 2004) works as follows: pick a sensible direction to move in the multi-dimensional landscape. Then, project a line in that direction (how far to move), locate the minimum of the error function along that line in the weight space, and repeat (Schraudolph & Grapple, 2003). An obvious choice of the direction in this context is the direction of steepest descent (the same direction that would be chosen by *back propagation*). Actually, this intuitively obvious choice proves to be rather poor. Having minimized along one direction, the next line of steepest descent may spoil the minimization along the initial direction. A better approach is to select conjugate or non-interfering directions i.e. conjugate gradient descent and quasi-newton.

A model-trust region approach (Hunt & Deller, 1995), (Burney, Jilani , & Ardil, 2004) works as follows: instead of following a search direction, assume that the surface is a simple shape such that the minimum can be located (and jumped to) directly - if the assumption is true. The model typically assumes that the surface is a nice well-behaved shape (e.g., a parabola), which will be true if sufficiently close to a minima. Elsewhere, the assumption may be grossly violated, and the model could choose wildly inappropriate points to move to. The model can only be trusted within a region of the current search point, and the size of this region isn't known. Therefore, choose new points to test as a compromise between that suggested by the model and that suggested by a standard gradient-descent jump. If the new point is good, move to it, and strengthen the role of the model in selecting a new point; if it is bad, don't move, and strengthen the role of the gradient descent step in selecting a new point (and make the step smaller). *Levenberg-Marquardt* uses a model-trust region that assumes the underlying function modeled by the network is locally linear.

❖ Conjugate Gradient Descent

Conjugate gradient descent (Hestenes & Stifle, 1952), (Burney, Jilani , & Ardil, 2004) works by constructing a series of line searches across the error surface. It starts out by searching in the steepest descent direction (negative of the gradient -**g**) on the first iteration, just as back propagation would do. The initial search direction (**P**) is given by:

$$P_0 = -g_0$$

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 (Moller, 1993), (Hestenes & Stifle, 1952). So, conjugate gradient descent converges faster than steepest descent. This gives the next values for the weight vector ($\mathbf{w}_{n+1}$) as:

$$W_{n+1} = W_n + \lambda_n P_n$$

where the parameter $\boldsymbol{\lambda}$ is chosen to minimize error (***E***)

$$E(\lambda) = E(W_n + \lambda P_n)$$

The next search direction is determined so that it is conjugate to previous search directions. The general procedure for determining the new search direction ($\mathbf{P}_{n+1}$) is to combine the new steepest descent direction with the previous search direction (Schraudolph, 1993):

$$P_{n+1} = -g_{n+1} + \beta_{n+1}.P_n$$

The various versions of conjugate gradient are distinguished by the manner in which the constant $\boldsymbol{\beta}_{\mathbf{n+1}}$ is computed. $\boldsymbol{\beta}_{\mathbf{n}}$ is a time varying parameter. For the Fletcher-Reeves update the procedure is:

$$\beta_{n+1} = \frac{g_{n+1}^{T}.g_{n+1}}{g_n^{T}.g_n}$$

where $\mathbf{g}_n$ and $\mathbf{g}_{n+1}$ are the gradient vectors. For the Polak-Ribiére update, the constant $\boldsymbol{\beta}_{\mathbf{n+1}}$ is computed by:

$$\beta_{n+1} = \frac{g_{n+1}^{T} \cdot (g_{n+1} - g_n)}{g_n^{T} \cdot g_n}$$

The conjugate gradient algorithms are usually much faster than backpropagation, although the results will vary from one problem to another. They are often a good choice for networks with a large number of weights (more than a few hundred) and/or multiple output units (Burney, Jilani , & Ardil, 2004), (Schiffmann, Joost, & Werner, 1993). The conjugate gradient algorithms require only a little more storage than the simpler algorithms (it has memory requirements proportional to the number of weights). One of the common variations of conjugate gradient algorithms is the Scaled Conjugate Gradient, which we will present next.

❖ Scaled Conjugate Gradient

Each of the conjugate gradient algorithms requires a line search at each iteration. This line search is computationally expensive, since it requires that the network response to all training inputs be computed several times for each search. The scaled conjugate gradient algorithm (SCG), developed by Moller (Moller, 1993), was designed to avoid the time-consuming line search per learning iteration by using a step size scaling mechanism. This makes the algorithm faster and inexpensive than other algorithms. The scaled conjugate gradient algorithm is too complex to explain in a few lines, but the basic idea is to combine the model-trust region approach with the conjugate gradient approach. The Scaled Conjugate Gradient routine may require more iterations to converge than the other conjugate gradient algorithms, but the number of computations in each iteration is significantly reduced because no line search is performed. The storage requirements for the scaled conjugate gradient algorithm are about the same as those of conjugate gradient algorithms (Fletcher-Reeves).

❖ Quasi-Newton

Quasi-Newton (Battiti, 1992), (Burney, Jilani , & Ardil, 2004) is an advanced method of training multilayer perceptions. It is the most popular algorithm in nonlinear optimization, with a reputation for fast convergence. 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 (Newton's direction):

$$-H^{-1}g \qquad \textit{where} \quad H = \frac{\partial^2 E}{\partial^2 W}$$

It is a calculation involving the Hessian matrix **H** (the matrix of the second derivative for the cost function – i.e. second derivative of the network error with respect to the weights **E / W**) (Pearlmutter, 1994), (Zhou & Si, 1998). The weights are updated at each iteration as follows:

$$W_{n+1} = W_n - H^{-1}g_n$$

where $\mathbf{H}^{-1}$ is the inverse of the Hessian matrix **H**. Any error surface is approximately quadratic "close to" a minimum. Since, unfortunately, the Hessian matrix is difficult and expensive to calculate, and anyway the Newton step is likely to be wrong on a non-quadratic surface, Quasi-Newton iteratively builds up an approximation to the inverse Hessian. The approximation at first follows the line of steepest descent, and later follows the estimated Hessian more closely (Battiti, 1992).

Newton's method often converges faster than conjugate gradient methods. It is used as an efficient training method for smaller networks with a small number of weights. Unfortunately, it is complex since it requires computing the analytical derivative of Hessian matrix at each iteration and thus requires more storage (it has memory requirements proportional to the square of the number of weights) (Lippmann,1987), (Pearlmutter, 1994). There is a class of algorithms that is based on Newton's method, but which doesn't require calculation of second derivatives. They update an approximate Hessian matrix at each iteration of the algorithm. The update is computed as a function of the gradient. Among theses general purpose quasi-Newton algorithms that is used to update the search direction is probably the Broydon–Fletcher–Goldfarb–Shanno (BFGS) algorithm. The BFGS algorithm builds upon the earlier and similar Davidon–Fletcher–Powell (DFP) algorithm.

❖ Levenberg-Marquardt

Levenberg-Marquardt (Burney, Jilani , & Ardil, 2004), (Battiti, 1992) is an advanced non-linear optimization algorithm. It is a trust region based method with hyper-spherical trust region. Levenberg-Marquardt appears to be the fastest method for training moderate-sized feed forward neural networks (up to several hundred weights) but it needs enough memory (Schiffmann, Joost, & Werner, 1993).

The Levenberg-Marquardt algorithm is designed specifically to minimize the sum-of-squares error function, using a formula that assumes 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.

Like the quasi-Newton methods, the Levenberg-Marquardt algorithm was designed to approach second-order training speed without having to compute the Hessian matrix **H**. When the performance function has the form of a sum of squares (as is typical in training feed forward networks), then the Hessian matrix can be approximated as

$$\mathrm{H} = \mathrm{J}^{T}\ \mathrm{J}$$

and the gradient (**g**) can be computed as

$$\mathrm{g} = \mathrm{J}^{T}\mathrm{e}$$

Where **J** is the Jacobian matrix that contains first derivatives of the network errors with respect to the weights and biases, and **e** is a vector of network errors. The Jacobian matrix can

be computed through a standard backpropagation technique (Zhou & Si, 1998), (Pearlmutter, 1994) that is much less complex than computing the Hessian matrix. The Levenberg-Marquardt algorithm uses this approximation to the Hessian matrix in the following equation to update weights and biases:

$$W_{n+1} = W_n - \left(J_n{}^T J_n\right)^{-1} J_n{}^T e_n$$

where ($\mathbf{J}^T\ \mathbf{J}$) is positive definite, but if it is not, then, we make some perturbation into it that will control the probability of being non positive definite. Such that the recursion equation is

$$W_{n+1} = W_n - \left(J_n{}^T J_n + \lambda\ I\right)^{-1} J_n{}^T e_n$$

where the quantity $\boldsymbol{\lambda}$ is called the learning parameter, it ensures that $\mathbf{J}^T\ \mathbf{J}$ is positive definite. When the scalar $\boldsymbol{\lambda}$ is zero, this is just Newton's method, using the approximate Hessian matrix. When $\boldsymbol{\lambda}$ is large, this becomes gradient descent with a small step size. Newton's method is faster and more accurate near an error minimum, so the aim is to shift towards Newton's method as quickly as possible (Battiti, 1992). Thus, $\boldsymbol{\lambda}$ is decreased after each successful step (reduction in performance function) and is increased only when a tentative step would increase the performance function. In this way, the performance function will always be reduced at each iteration of the algorithm.

2.2 Evolutionary Algorithms

Evolutionary Algorithms are based on the basic concepts of biological reproduction and evolution that is used as a model to solve problems using computers to emulate the same process (Haupt, 1997). They are a robust heuristic search and optimization mechanism which can be applied to problems where normal solutions are not available or generally lead to unsatisfactory results (Salomon, 1998), (Yao, 1993). All possible solutions for a problem are represented with a particular genetic representation scheme (called chromosome). A set of solutions or individuals is generated to form the initial population of organisms, as shown in figure (2.5). Each organism in this figure is evaluated using a fitness function specific to the problem. The fitness function measures the performance of the organism according to specific characteristics. Using a particular selection algorithm based on the fitness value, some organisms are chosen to be the parents for the next generation. New organisms, also known as offspring are produced after the information contained in the parents is combined using reproduction operators such as crossover and mutation. Finally, some organisms are selected from the old population and from the new offspring to form the population for the next generation. These steps are repeated until a solution that satisfies the selected criteria is found. Evolutionary algorithms encompass: genetic algorithms, evolutionary programming, and evolution strategies.

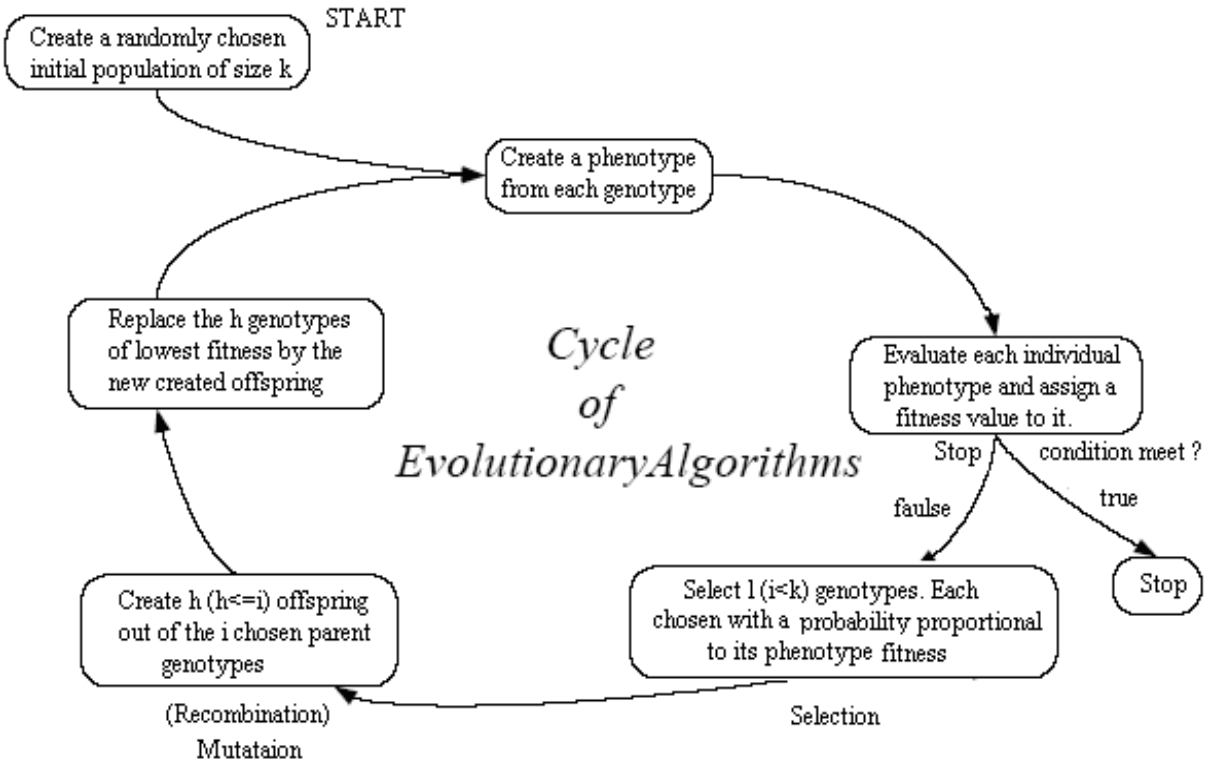


Figure 2.5: Cycle of evolutionary algorithms

2.2.1 Genetic Algorithms (GA):

GA is one of the most popular areas of research in evolutionary algorithms (Whitley, 1994), (Haupt, 1997) particularly useful for multidimensional optimization problems. The goal of optimization problem is to find the best solution where several feasible solutions (know as search space) are available. An evaluation (fitness) function is used for determining how good each particular solution in the population is. In a particular problem, the search space and the evaluation function for the elements in the search space in terms of performance define a landscape. Given a set of possible solutions in the search space, there may be several local minimum or sub-optimal values, but the over all lowest value of the set is considered the optimal value (global minima). If the search space is small, all the possible solutions can be examined, but as the search space grows in size, this exhaustive search becomes impractical.

Traditional search algorithms, such as the gradient descent, examine a point in the search space at the time, and the next point to be examined is obtained based on the current position. Usually, the next point to be examined has better performance than the previous point and it may be a local minimum. However since the new position is based on the previous one, it may not be possible to make a drastic move to go down the slope (gradient) towards the global minima (Burney, Jilani , & Ardil, 2004), (Salomon, 1998), (Sexton, Dorsey, & Johnson, 1998). Another deficiency of this algorithm is that it is possible to get stuck on a plateau. This may happen if the algorithm is unable to move far away from the flat region. Another problem with this algorithm comes from the fact that the final result depends on the starting search

point, it may be possible that different starting points produce different results. In the GA, although some individuals in the search space may reside near local minima, it is less likely to get trapped because the population provides global information about the landscape. There is a better chance that some individual will be near the global minima, and the genetic operators allow the GA to move the population in large jumps to focus the search in the most fruitful regions of the landscape (region around the global minima). For these reasons, GAs are well suited for searching the space of neural networks. Instead of training a network by performing gradient-descent on an error surface, the GA samples the space of networks and recombines those that perform best on the task in question.

The most important factors to consider in genetic algorithms as a search mechanism are: the encoding / representing of a chromosome, the reproduction operators, the selection methods, and the fitness function.

2.2.1.1 Encoding of a Chromosome:

The encoding of the chromosome is one of the important factors to consider in genetic algorithms as a search mechanism (Whitley, 1994). The chromosome should in some way contain information about solution which it represents. The most used way of encoding is a binary string. The chromosome then could look like figure (2.6).

Chromosome 1	10110010110010101110010 1
Chromosome 2	11111110000011000001111 1

Figure 2.6: Example of chromosomes with binary encoding

Each chromosome has one binary string. Each bit in this string can represent some characteristic of the solution. Or the whole string can represent a number – this has been used in the basic GA applet. Of course, there are many other ways of encoding. This depends mainly on the solved problem. For example, one can use the value encoding where every chromosome is a string of some values. Values can be anything connected to problem as integers, real numbers or chars for some complicated objects. Sometimes it is useful to encode some permutations where every chromosome is a string of numbers.

2.2.1.2 Reproduction Operators:

The two most common reproduction operators in GAs are mutation and crossover (Whitley, 1994). When binary genotypic representations are used, crossover is performed by splitting two parent chromosomes at some point, and one part of one parent chromosome is exchanged for the corresponding part of the other parents' chromosome to produce offspring. Crossover can then look like figure (2.7) (a) (| is the crossover point). Mutation involves changing one or more components of a chromosome at random. With binary representations, we can switch the chosen bits from 1 to 0 or from 0 to 1. Real valued genotypic representations implement mutation differently. Mutation is shown in figure (2.7) (b). The mutation depends on the

encoding method as well as the crossover. For example when we use the permutation encoding, mutation could be exchanging two genes.

(a)

Chromosome 1	11011 | 00100110110
Chromosome 2	11101 | 11000011110
Offspring 1	11011 | 11000011110
Offspring 2	11101 | 00100110110

(b)

Original Offspring 1	1101111000011110
Original Offspring 2	1101100100110110
Mutated Offspring 1	1100111000011110
Mutated Offspring 2	1101101100110110

Figure 2.7: (a) Example of crossover operator **(b)** Example of mutation operator

2.2.1.3 Selection Methods:

The chromosomes are selected from the population to be parents to crossover. The problem is how to select these chromosomes. According to Darwin's evolution theory the best ones should survive and create new offspring. There are many methods how to select the best chromosomes, these methods are (Whitley, 1994), (Blickle & Thiele, 1995):

- **Roulette Wheel Selection**. Each individual has a selection probability proportional to its fitness.

- **Tournament Selection**. A group of individuals is chosen from the population and the most fit in the group is selected. The size of the group chosen is called the tournament size. A tournament size of 2 is a binary tournament.

- **Linear Ranking Selection**. The population is sorted by fitness and assigned a rank from best to worst. The selection probability is linearly assigned to the individuals according to their rank.

2.2.1.4 Fitness:

Fitness is determined by a fitness or objective function (Whitley, 1994). The fitness value represents the quality of the chromosome, and is used to grade and order the population. The fitness function is specific to the individual problem and is essential as a driving force for an effective evolutionary search.

2.3 Cellular Automata

In evolving artificial neural networks, several representation methods based on evolutionary computation paradigms are used to automatically determine the appropriate architectures of feed-forward neural networks. Some of those designed methods are based on direct representations of the parameters of the network. These representations become less effective with larger networks because the effects of crossover are often unfavorable for retaining any kind of high level network structure that may have evolved. This is known as the scalability

problem. An alternative more interesting are the indirect encoding methods that codify a compact representation of the neural network. Thus, they avoid the scalability problem and reduce the length of the genotype. In this thesis, we will use one of those indirect constructive encoding methods: cellular automata which provide an efficient way for representing network architectures.

2.3.1 Principles of Cellular Automata:

Cellular Automata (CA) are a class of discrete dynamical systems, consisting of an array of nodes (lattice of cells) of some dimension **n** (Gutowitz, 1991)**,** (Nehaniv, 2002), (Wolfram, 1994). Each cell in the lattice can be in one of *k* different states. At discrete time steps, all cells update and change their states simultaneously, in a way determined by the transition rules of the particular CA. The transition rules describe precisely how a given cell should change states, depending on its current state and the states of its neighbors. The cells that are in the neighborhood of a given cell must be specified explicitly. This process of simultaneously updating the cells in the lattice is repeated over time, starting from some particular (random) initial configuration of cell states. When plotted over time, the lattice as a whole can show a wide variety of behaviors, depending on the particular transition rules that are used.

Cellular automata may differ in the following:

- The set of initial states of all the automata.
- The definition of the set of neighbours to a given grid point.
- The actual finite automaton associated to each point in the grid.
- The shape and size of the grid (lattice), usually square, rectangular or triangular, which may be infinite.

Cellular automata have been successfully used as a simulation tool in several areas such as urban development, ecological systems, and image processing.

2.3.2 Two-Dimensional Cellular Automata:

Two-Dimensional Cellular Automata are a natural extension of the 1-D case (Nehaniv, 2002), (Gutowitz, 1991). The one-dimensional CA can be visualized as having a cell at each integral point on the real number line where the two-dimensional CA have cells at all points in the plane that have only integral coordinates. In 2-D CA, there are large numbers of rows and columns of cells whose states change with time according to transition rules. The iterative process in such 2-D cellular automata is that in each time step the number of neighbours is calculated for each site simultaneously and the automation is updated accordingly. This parallel processing characterizes the CA. This property makes them attractive the modelling of processes where such parallel processing is involved.

The neighbours for each cell in 2-D cellular automata must be specified explicitly. The two most popular choices are the Von Neumann neighborhood and the Moore neighborhood; both are named after their creators (Gutowitz, 1991), (Wolfram, 1994). With the Von Neumann neighborhood, a cell has four neighbors in its north, south, east, and west sides. The Moore

neighborhood includes four additional neighbors in its northeast, southeast, southwest, and northwest corners. Generally, a cell is always part of its own neighbors; therefore there are five neighbors in the Von Neumann neighborhood and nine in the Moor neighborhood. The Von Neumann and the Moore neighborhood are shown in figure (2.8).

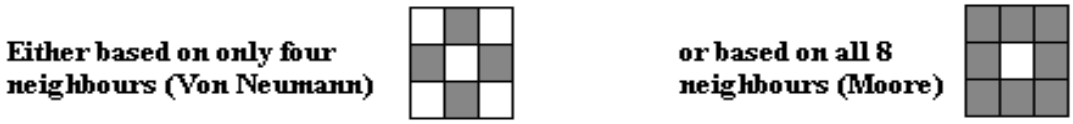


Figure 2.8: Von Neumann and Moore neighborhood

As it is indicated before, a set of transition rules governs each cell's state of being alive (white) or dead (black) based on it and it's immediate neighbour's states in the last time step. Conway's Game of Life is perhaps the most famous of the rule sets and a good place to start (Gardner, 1970). In Conway's game of Life the rules are based on the values of all 8 neighbours and itself in the last time step. Examples of these rules are (see figure 2.9):

1. An alive cell dies from exposure if less than 2 neighbors were alive.
2. An alive cell dies from overcrowding if more than 3 neighbours were alive.
3. A dead cell becomes alive if precisely 3 neighbours were alive.

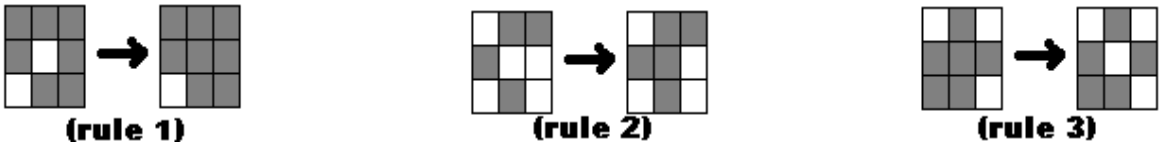


Figure 2.9: Examples of transition rules in Conway's game of Life [23]

2.4 Meta-Learning

Meta-learning is defined as learning from learned knowledge (Chan & Stolfo, 1993). It refers to a general strategy that seeks to learn how to integrate a number of separate learning processes in an intelligent fashion. The basic idea of the meta-learning is to compute a number of independent classifiers by executing number of machine learning processes to a collection of data subsets in parallel mode. These “base classifiers" are then collected and combined into a final classifier by another learning process. A graphical representation of meta-learning process with three different classifiers is depicted in figure (2.10) (Prodromidis, 1999). In this figure, two classifiers are derived from the same data set (either from different samples or from different learning algorithms, or both) while the third is induced from a separate set. The meta-learning algorithm combines the three classifiers into an ensemble meta-classier by “learning” how they predict, i.e., by observing their input/output behavior. Meta-learning addresses the scaling problem for machine learning. It improves efficiency by executing in parallel the base-learning processes on (possibly disjoint) subsets of the training set (a data reduction technique). Meta-learning, is scalable because meta-classifiers can be similarly integrate into higher level meta-classifiers in a distributed fashion and it improves the

predictive performance and accuracy by combining classifiers with different inductive classifiers (Prodromidis & Stolfo, 1998).

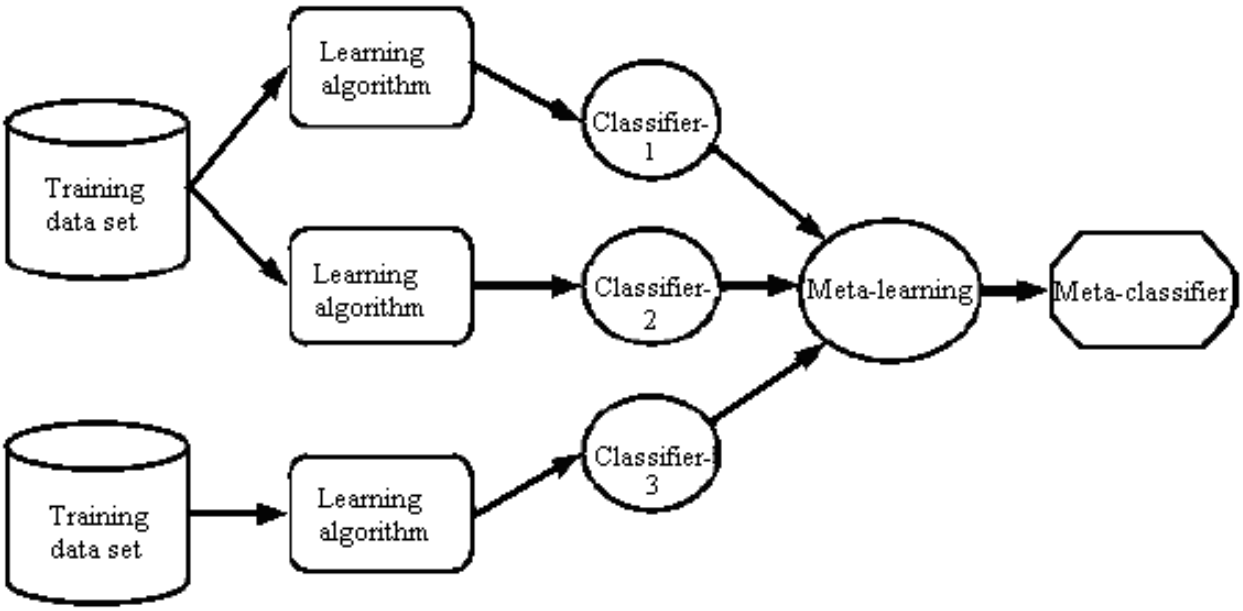


Figure 2.10: Meta-learning process

2.4.1 Meta-Learning Techniques (Integrating multiple classifiers):

The integration problem can be defined as follows (Seppo, Vagan, & Alexey, 1999-B). Let the training set T be:{(Xi, Yi), i=1,...,n}, where n is the number of the training instances, Xi is the vector of the attributes of the i-th training instance (the values of the attributes can be numeric, nominal, or symbolic), and Yi $\in$ {Y1,...,Yk} is the actual class of the i-th instance (k is the number of classes). Let the ensemble C of classifiers be:{C_1,...,C_m}, where m is the number of the available classifiers (component classifiers). Each component classifier is either derived using some learning algorithm or using some heuristic knowledge. Let a new instance e* be an assignment of values to the vector of the attributes {Xi} without known actual classification. The integration problem is to use the ensemble C of the classifiers to classify the new instance e* as accurately as possible. Recently two basic approaches are used to integrate multiple component classifiers of an ensemble. In the first approach, all the component classifiers produce their classification results, which are then combined. In the second approach the best classifier is selected from the base classifiers and then it is used to produce the classification result.

2.4.1.1 Combining Approach for Meta-Learning:

The main idea of meta-learning approach is to learn a global classifier "GC" that combines the output of a number of classifiers. Initially, each learning task, also called a base learner, computes a base classifier, i.e. a model of its underlying data subset or training set. Next, a separate learning task, called a meta learner, combines these independently computed base

classifiers into a higher level classifier, called a meta-classifier, by learning over a meta-level training set. This meta-level training set is composed from the predictions of the individual base-classifiers when tested against a separate subset of the raw training data, also called a validation dataset E. Validation data are extracted from the training set and are not used for classifier training. From these predictions, the meta-learner discovers the properties of the base-classifiers and computes a meta-classifier which models the "global" dataset. To classify an unlabeled instance e*, the base-classifiers present their own predictions to the meta-classifier which then makes the final classification (Prodromidis, 1999), (Seppo, Vagan, & Alexey, 1999-B).

Figure (2.11) (Chan & Stolfo, 1993) depicts the different stages in a simplified meta-classifier scenario:

1. The classifiers (base classifiers) are trained from the initial (base-level) training datasets.
2. Predictions are generated by the learned classifiers on a separate validation dataset.

3. A meta-level training set is composed from the validation set and the predictions generated by the classifiers on the validation dataset.

4. The global classifier "GC" (meta-classifier) is trained from the meta-level training set.

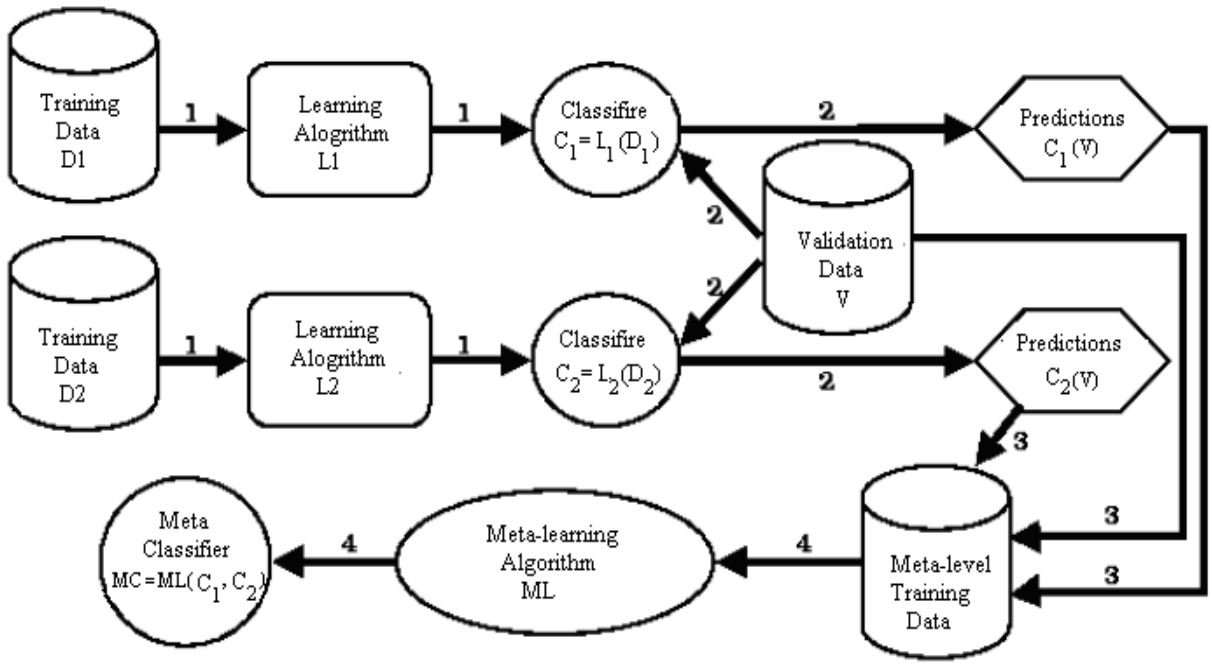


Figure 2.11: The stages in a simplified meta-classifier scenario

Several effective methods have been proposed to combine the results of the ensembled component classifiers. These methods include: Voting, Stacking, Bagging, Boosting, SCANN,

etc. They differ on the way the meta-level training set is formed and the way the final prediction of the meta-classier is synthesized.

2.4.1.2 Selection Approach for Meta-Learning:

Techniques of this approach try to select the best base classifier for the data. So there is no need for a meta-classifier. These techniques can be divided into two subsets: static and dynamic selection. The static approaches select the best classifier for the whole data space, while the dynamic approaches take into account each new instance to be classified (Seppo, Vagan, & Alexey, 1999-A), (Merz, 1996). Usually better results can be achieved if the classifier integration is done dynamically taking into account characteristics of each new instance.

2.4.2 Benefits of Meta-Learning:

- Meta-learning improves predictive performance, efficiency, and scalability. It illustrates two characteristics, parallelism and reduced communication. All base classifiers are generated in parallel and collected at any location without the time-consuming process of writing parallel programs (i.e., using standard o-the-shelf serial code), where the communication overhead is negligible compared to the transfer of entire raw data (Chan & Stolfo, 1993), (Prodromidis, 1999).

- Meta-learning improves predictive accuracy by combining different inductive classifier. By combining separately learned concepts, meta-learning is expected to derive a higher level learned model that explains a large data more accurately than individual learner (Prodromidis & Stolfo, 1998), (Prodromidis, 1999).

- Meta-learning unifies and scales up learning algorithms to very large datasets in wide area computing networks. It is unifying because it is algorithm and representation independent, i.e., it does not examine the internal structure and strategies of the learning algorithms themselves, but only the outputs (predictions) of the individual classifiers, and it is scalable because it can be intuitively generalized to hierarchical multiple level meta-learning (Prodromidis & Stolfo, 1998), (Prodromidis, 1999), (Seppo, Vagan, & Alexey, 1999-B).

- Meta-learning has been applied with success to a number of applications like Distributed Data Mining (DDM), Multiple Classifier Systems, and Information Fusion (Prodromidis, 1999), (Seppo, Vagan, & Alexey, 1999-B).

- Meta-learning can be considered primarily as a method that reduces the size of the data basically due to its data reduction technique and its parallel nature. On the other hand, it is also generic, meaning that it is algorithm and representation independent, hence it can benefit from fast algorithms and efficient relational representations (Chan & Stolfo, 1993), (Prodromidis & Stolfo, 1998), (Prodromidis, 1999).

Summary

This chapter presented the basic concepts that are used in this thesis. The artificial neural networks and genetic algorithms were described, as these are the main techniques used. In addition, we introduced other important paradigms such as meta-learning and cellular automata. In the next chapter we will outline a literature review of some previous works related to the main techniques used in this thesis.

CHAPTER THREE

LITERATURE REVIEW AND PREVIOUS WORK

This chapter outlines a literature review of some previous works related to the main techniques used in this thesis. First, it describes the use of pure evolutionary learning algorithms as a learning tool for traditional neural networks in addition to different evolution trends in evolutionary artificial neural networks including evolution of connection weights, architectures, and learning rules. Next, this chapter presents the hybrid training approaches that combine the global search capability of evolutionary algorithms with the efficient local search of gradient descent algorithms for training ANNs.

3.1 Pure Evolutionary Algorithms for Training and Evolving Artificial Neural Networks

The interest in using the pure evolutionary algorithms for training ANNs has been growing in recent years since they can handle the global search problem efficiently in a vast, complex, multimodal, and non differentiable surface. They can avoid the local minima by searching in several regions simultaneously in contracts with the traditional search algorithms (i.e. gradient descent algorithms) which may get stuck in local minima. Pure evolution in artificial neural networks can be found at three different levels: connection weights, architectures, and learning rules. So, EANN can be seen as a system that adapts to weights, architectures, and rules dynamically without human intervention.

The evolution of connection weights in ANNs provides a global approach to connection weight training, especially when gradient information of the error function is difficult or costly to obtain. The architecture of an Artificial Neural Networks is known before the learning process, and it does not change (fixed) during the evolution of the connections weights. Researches and applications have been conducted on the evolution of connection weights by (Kitano, 1990), (Yao, 1999), (Caudell & Dolan, 1989), (Belew, McInerney, & Schraudolph, 1991), (Fogel, Wasson, & Boughton, 1995), (Koza & Rice, 1991), (Sexton, Dorsey, & Johnson, 1998), (Yoon, Holmes, & Langholz, 1994), (Korning, 1995) because they can deal with very large, complex, not differentiable and multimodal spaces. The evolutionary approach to weight training in ANN's includes a major point which is the ability to decide the representation of connection weights, i.e., whether in the form of binary strings or not. Some of the early work in evolving ANN connection weights used binary strings for representation (Caudell & Dolan, 1989). In such a representation scheme, each connection weight is represented by a number of bits with certain length. Other works in evolving ANN connection weights used real numbers for representation (Fogel, Wasson, & Boughton, 1995). In such a representation scheme, each connection weight is represented by a one real number.

Evolution can also be used to find a near-optimal ANN architecture automatically. This is the second level of evolution in artificial neural networks. The architecture includes its topological structure, i.e., connectivity, and the transfer function of each node in the ANN. The architecture design is crucial in the successful application of ANN's because the architecture has significant impact on a network's information processing capabilities. Recently, a lot of research on evolving ANN architectures has been carried out. Most of the research has

concentrated either on the evolution of artificial neural network topological structures alone (separated from that of the connection weights) (Yao, 1999), (Leung, Lam, & Ling, 2003), (Jacob & Rehder, 1993), or simultaneously with the evolution of ANN connection weights (Binos, 2003), (Yao, 1999), (Koza & Rice, 1991), (Branke, 1995). The transfer function is often assumed to be fixed and the same for all the nodes in an ANN, at least for all the nodes in the same layer. Relatively little has been done on the evolution of node transfer functions.

Similar to the evolution of connection weights, one major phase involved in the evolution of architectures is the genotype representation scheme of architectures (encoding the ANN architectures). One of the key issues in encoding ANN architectures is to decide how much information about an architecture should be encoded in the chromosome. At one extreme, all the details, i.e., every connection and node of an architecture can be specified by the chromosome. This kind of representation scheme is called direct encoding (Yao & Liu, 1997), (Branke, 1995), (Koza & Rice, 1991), (Yao & Liu, 1998), (Braun & Weisbrod, 1993). Direct encoding scheme takes two different approaches, one separates the evolution of architectures from that of connection weights (Yao, 1999), (Leung, Lam, & Ling, 2003) and the other evolves architectures and connection weights simultaneously (Yao, 1999), (Koza & Rice, 1991), (Branke, 1995). At the other extreme, only the most important parameters of an architecture, such as the number of hidden layers and hidden nodes in each layer, are encoded in the chromosome. Other details about each connection in an ANN are left to the training process to decide. This kind of representation scheme is called indirect encoding which can produce more compact genotypical representation of ANN architectures (Gutierrez, Isasi, Molina, Sanchis, & Galvan, 2001), (Harp, Samad, & Guha, 1989), (Kitano, 1990), (Molina, Galván, Isasi, & Sanchis, 2000-A), (Gruau & Whitley, 1993), (Gruau, Whitley, & Pyeatt, 1995), (Harp, Samad, & Guha, 1990), (Koza & Rice, 1991), (Molina, Galván, Isasi, & Sanchis, 2000-B), (Chval, 2002), (Hussain & Browse, 1998), (Jacob & Rehder, 1993), (Luke & Spector, 1996). There are different kinds of indirect encoding schemes including: structural encoding, parametric encoding, and grammar encoding.

Structural encoding defines the structure of the network that is embedded in the chromosome. Koza (Koza & Rice, 1991) applied genetic programming to discover both the architecture and the weights of a neural network. In this work, the neural network was represented as a point-labeled tree. Parametric encoding uses certain important aspects of neural network architecture (such as the number of hidden layers, the number of hidden nodes in each layer, etc.) and is represented by fixed parameters (Harp, Samad, & Guha, 1989), (Harp, Samad, & Guha, 1990), (Gutierrez, Isasi, Molina, Sanchis, & Galvan, 2001). Another technique is grammatical encoding, where the neural network is represented as a sentence of a special language described by a grammar. Two basic approaches to grammar encoding include *developmental grammar encoding*, and *derivation grammar encoding.* Developmental grammar encoding describes the chromosome by grammar rules that will be used to develop a specific neural network structure (Kitano, 1990). Derivation grammar encoding design a single fixed grammar and the chromosome contains the derivation sequence which define the network architecture (Jacob & Rehder, 1993).

Gruau and Whitley (Gruau & Whitley, 1993), Gruau and Whitley and Pyeatt (Gruau, Whitley, & Pyeatt, 1995) had used genetic programming to create the topology for recurrent neural networks. Luke and Spector (Luke & Spector, 1996) presented an edge encoding technique for

evolving graph and network structures via genetic programming. Hussain and Browse (Hussain & Browse, 1998) proposed the use attribute grammars in creating a useful and compact genetic encoding of neural networks. Molina and Galvan (Molina, Galván, Isasi, & Sanchis, 2000-B) used grammars and cellular automata for evolving Neural Networks Architectures. Gutiérrez and Isasi, (Gutierrez, Isasi, Molina, Sanchis, & Galvan, 2001) had used the evolutionary cellular configurations for designing feed-forward neural networks architectures, and they used only the backpropagation algorithm for training neural networks. They did not apply the idea of meta-learning algorithm were different number of learning algorithms are used in parallel mode for training the neural networks separately. Harp *et al.* (Harp, Samad, & Guha, 1989) used a "blueprint" to represent an architecture which consists of one or more segments representing an area (layer) and its efferent connectivity (projections). Andersen and Tsoi (Andersen & Tsoi, 1993) proposed different approach to the evolution of architectures where each individual in a population represents a hidden node rather than the whole architecture. An architecture is built layer by layer, i.e., hidden layers are added one by one if the current architecture cannot reduce the training error below certain threshold. Each hidden layer is constructed automatically through an evolutionary process which employs the GA with fitness sharing. Fitness sharing encourages the formation of different feature detectors (hidden nodes) in the population. The number of hidden nodes in each hidden layer can vary. This approach (Andersen & Tsoi, 1993) could only deal with strictly layered feed forward ANN's.

The third level of evolution in artificial neural networks is evolving the learning rules. Since evolution is one of the most fundamental forms of adaptation, it is not surprising that the evolution of learning rules has been introduced into ANN's in order to learn their learning rules. Adapting a learning rule through evolution is expected to enhance ANN's adaptivity greatly in a dynamic environment. Research into the evolution of learning rules is still in its early stages (Chalmers, 1990), (S. Bengio, Y. Bengio, Cloutier, & Gecsei, 1992), (Baxter, 1992). Other combinations between ANN's and EA's are also presented in: **1)** The evolution of input features (Guo & Uhrig, 1992). **2)** ANN as fitness estimator (Leung, Lam, & Ling, 2003). **3)** Evolving ANN ensembles (Yao & Liu, 1996).

3.2 Hybrid Evolutionary-Gradient Search Algorithms for Training and Evolving ANNs

As indicated before, the pure evolutionary algorithm for training ANN is attractive because it can handle the global search problem better in a vast, complex, multimodal, and non differentiable surface. However, most EA's are rather inefficient in fine-tuned local search (Yao, 1999), (Abraham, 2002), (Yao, 1993), (Salomon, 1998). The efficiency of evolutionary training for ANN can be improved significantly by using a hybrid training approach that incorporates the evolutionary algorithm's global search ability with local search's ability to fine tune. EA's can be used to locate a good region in the space and then a local search procedure is used to find a near-optimal solution in this region. The local search procedure could be backpropagation or any other gradient descent algorithms. Hybrid training approach has been used successfully in many application areas (Abraham, 2004), (Abraham, 2002), (Yao & Liu, 1997), (Belew, McInerney, & Schraudolph, 1991), (Hendtlass & Podlena, 1995), (Magoulas, Plagianakos, & Vrahatis, 2001), (Wong, Chung, & Wong, 1998), (Abraham & Nath, 2000).

Belew (Belew, McInerney, & Schraudolph, 1991) and many other researchers used GA's to search for a near-optimal set of initial connection weights and then used BP to perform local search from these initial weights. Their results showed that the hybrid GA/BP approach was more efficient than either the GA or BP algorithm used alone. If we consider that BP often has to run several times in practice in order to find good connection weights due to its sensitivity to initial conditions, the hybrid training algorithm will be quite competitive. In Wong and Chung work (Wong, Chung, & Wong, 1998), a hybrid approach, combining the global search capability of evolutionary algorithms for training ANN with the backpropagation algorithm has been used to solve the Unit Commitment problem (Wong, Chung, & Wong, 1998). A population of neural networks with a fixed number of nodes is evolved by altering the active connections with a genetic algorithm. High fitness individuals from this process are used as starting points that are then trained via backpropagation.

In (Hendtlass & Podlena, 1995), a modified genetic algorithm is used to evolve neural network topology and weights for character recognition. Architectural mutations are achieved by uniform crossover between two individuals. Weights are modified by mutation only. Good networks resulting from this process undergo fine-tune training with backpropagation. In (Yao & Liu, 1997), the EPNet algorithm is a hybrid algorithm used to evolve feed forward artificial neural networks. It combines the architectural evolution of the network with its weight learning. Neural network node density, connectivity and weights (including biases) are evolved in a series of steps by the application of the five mutation operators: hybrid training using a modified back-propagation algorithm and simulated annealing, node and connection deletion, node and connection addition.

In (Magoulas, Plagianakos, & Vrahatis, 2001), a hybrid algorithm combining a Differential Evolution Strategy and Stochastic Gradient Descent is used for on-line training of large fixed topology neural networks on image classification tasks. There are two stages to this algorithm that operates on a population of weight vectors representing neural network individuals. The first one uses stochastic gradient descent to train the network weights. This is done by modifying the weights using an adaptive step size and the error of the network. The second stage uses a differential evolution to increase the diversity of the population by using a combination of mutation and crossover. The results showed good generalization on two image classification tasks. Alexander (Topchy, Lebedko, & Miagkikh, 1995) proposed another work that combined the global search of EAs with the local search procedures. He developed a fast learning in multilayered neural networks by means of hybrid evolutionary and gradient algorithms. His research described two algorithms based on cooperative evolution of internal hidden network representations and a combination of global evolutionary and local search procedures.

Abraham (Abraham, 2004) proposed a hybrid meta-heuristic learning approach, which is called meta-learning evolutionary artificial neural networks (MLEANN). His proposed approach can be considered as adaptive computational framework based on evolutionary learning and local search procedures for automatic design of optimal artificial neural networks. Abraham used four different learning algorithms in parallel mode for training neural networks. He also used the direct encoding methods for optimizing the neural network architectures and that requires much larger chromosomes especially for ANNs with complex

architectures. This could end in a too huge space search that could make the method impossible in practice.

Summary

In this chapter, a literature review of some works related to my research was outlined. Examining the previous works, we observed that the pure evolutionary learning algorithms, especially genetic algorithms, were used as a learning tool for traditional neural networks. Different evolution trends in evolutionary artificial neural networks were found at three levels: connection weights, architectures, and learning rules. Moreover, hybrid training approaches that combine the global search capability of evolutionary algorithms with the efficient local search of gradient descent algorithms for training ANNs were presented. Also, two types of genotype representation scheme, which were used for encoding the architecture of ANNs in chromosomes, were mentioned and the preferable one that reduced the complexity of the networks was the indirect encoding methods.

In the next chapter, we will present our proposed approach: Meta-learning Evolutionary Artificial Neural Networks using one of the indirect encoding methods, i.e. cellular automata, instead of using the direct ones as in the previous MLEANN framework.

CHAPTER FOUR

META-LEARNING EVOLUTIONARY ARTIFICIAL NEURAL NETWORKS: BY MEANS OF CELLULAR AUTOMATA

This chapter presents the proposed framework: meta-learning evolutionary artificial neural networks by means of cellular automata (MLEANN-CA). It is an adaptive computational framework based on evolutionary learning and local search procedures for automatic design of optimal artificial neural networks using direct and indirect encoding methods. We start this chapter by presenting the evolutionary artificial neural networks followed by its general framework. Then, we describe the three kinds of evolution in EANNs: evolution of connection weights, architectures, and learning rules. Moreover, we introduce the hybrid meta-learning approach: MLEANN, which uses direct encoding methods in designing the network architecture. Finally, we examine our proposed framework: MLEANN-CA.

4.1 Evolutionary Artificial Neural Networks

Many of the conventional ANNs now being designed are statistically quite accurate. However, the important drawback is that neural network design relies heavily on human experts who have sufficient knowledge about the different aspects of the network and the problem domain. The human experts have to specify manually the number of neurons, their distribution over several layers and interconnection between them. As the complexity of the problem domain increases, manual design becomes more difficult and unmanageable. Several methods have been proposed to automatically construct ANNs for reduction in network complexity. Most of these methods got its own limitations. The interest in using evolutionary algorithms for designing ANN architecture, automatically, has been growing in recent years as they can evolve towards the optimal architecture without outside interference, thus eliminating the tedious trial and error work of manually finding an optimal network (Yao, 1999), (Abraham, 2004), (Abraham, 2002), (Abraham & Nath, 2001), (Yao & Liu, 1997), (Yao & Liu, 1998), (Braun & Weisbrod, 1993).

In Evolutionary Artificial Neural Networks (EANN), evolution has been introduced into ANNs at roughly three different levels: connection weights; architectures; and learning rules. EANNs provide a general framework, as indicated in figure (4.1), where interactions among the three levels of evolution are considered (Yao, 1999), (Abraham & Nath, 2000), (Abraham, 2004), (Abraham, 2002), (Abraham & Nath, 2001), (Yao, 1993), (Liu & Yao, 1998). In this framework, the evolution of connection weights proceeds at the lowest level on the fastest time scale in an environment determined by an architecture, a learning rule, and learning tasks. There are, however, two alternatives to decide the level of the evolution of architectures and that of learning rules either the evolution of architectures is at the highest level and that of learning rules at the lower level or vice versa. The decision on the level of evolution depends on what kind of prior knowledge is available. The lower the level of evolution, the faster the time scale it is on.

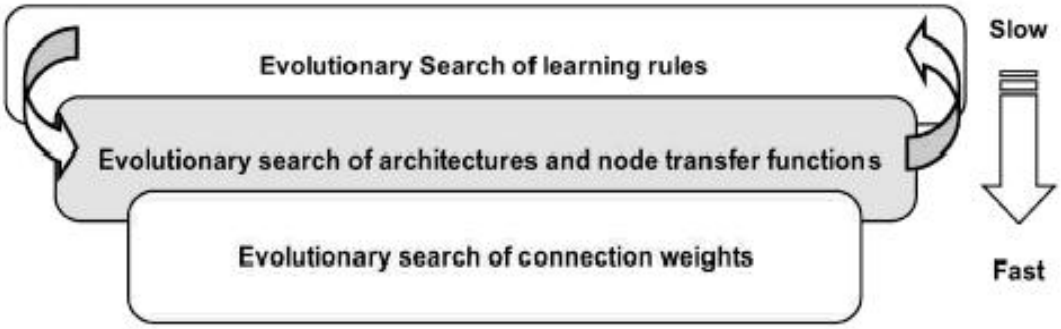


Figure 4.1: A general framework for EANN's

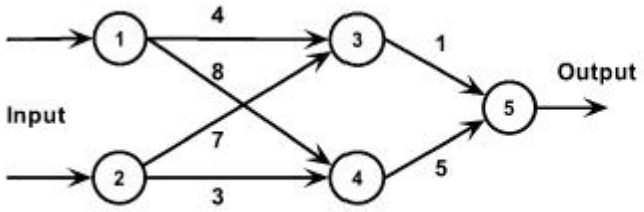
4.2 Evolutionary Search for Connection Weights, Architectures, and Learning Rules

As described before, the evolution has been introduced into ANNs at three different levels: connection weights; architectures; and learning rules. The evolution of connection weights introduces an adaptive and global approach to training, especially in the reinforcement learning and recurrent network learning paradigm where gradient-based training algorithms often experience great difficulties. The evolution of architectures enables ANNs to adapt their topologies to different tasks without human intervention and thus provides an approach to automatic ANN design as both ANN connection weights and structures can be evolved. The evolution of learning rules can be regarded as a process of "learning to learn" in ANNs where the adaptation of learning rules is achieved through evolution. It can also be regarded as an adaptive process of automatic discovery of novel learning rules.

4.2.1 Evolutionary Search of Connection Weights:

Weight training in ANNs is usually formulated as minimization of an error function, such as the mean square error between target and actual outputs averaged over all examples, by iteratively adjusting connection weights. Most training algorithms for ANN, such as BP and conjugate gradient algorithms (Burney, Jilani , & Ardil, 2004), (Schiffmann, Joost, & Werner, 1993), (Rumelhart, Hinton, & Williams, 1986), are based on gradient descent. There have been some successful applications of BP in various areas, but BP has drawbacks due to its use of gradient descent. It often gets trapped in a local minimum of the error function and is incapable of finding a global minimum if the error function is multimodal and/or non-differentiable. One way to overcome gradient-descent-based training algorithms' shortcomings is to adopt EANN's, i.e., to formulate the training process as the evolution of connection weights in the environment determined by the architecture and the learning task. EA's can then be used effectively in the evolution to find a near-optimal set of connection weights globally without computing gradient information. The architecture and the learning rules of the neural networks are pre-defined and fixed during the evolution. A key question in evolving connection weights is to decide the representation of connection weights, i.e., whether in the form of binary strings or real (Caudell & Dolan, 1989), (Fogel, Wasson, & Boughton, 1995), (Yao, 1999), (Koza & Rice, 1991). Thus, the proper genetic operators such as crossover and mutation are to be chosen in conjunction with the representation scheme.

Figure (4.2) illustrates the multilayered feed-forward neural network with its weight matrix, encoded directly, besides the connection weight chromosome using binary representation.



To\From	1	2	3	4	5
1	0	0	0	0	0
2	0	0	0	0	0
3	4	7	0	0	0
4	8	3	0	0	0
5	0	0	1	5	0

Genotype: 0100 0111 1000 0011 0001 0101

Figure 4.2*:* The feed-forward neural network, its weight matrix, and its connection weight chromosome using binary representation

The whole network is encoded directly by concatenation of all the connection weights of the network in the chromosome. A heuristic concerning the order of the concatenation is to put connection weights to the same node together. The representation of the connection weights in ANN using the real numbers could be: (4.0, 7.0, 8.0, 3.0, 1.0, 5.0).

Evolutionary Search of connection weights can be formulated as follows (Yao, 1999), (Abraham, 2004), (Abraham, 2002):

(1) Generate an initial population of N weight chromosomes. Evaluate the fitness of each EANN depending on the problem.
(2) Depending on the fitness and using suitable selection methods reproduce a number of children for each individual in the current generation.
(3) Apply genetic operators (crossover, mutation) to each child individual generated above and obtain the next generation.
(4) Check whether the network has achieved the required error rate or the specified number of generations has been reached. Go to Step 2.
(5) End.

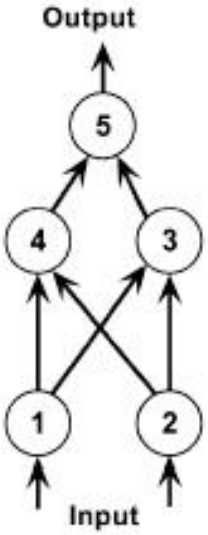
Using evolutionary algorithms to train the weights instead of gradient descent algorithms, which can only find local optimum in a neighborhood of the initial solution, can result in faster and better convergence. Better still, since EAs are good at global search but inefficient at local finely tuned search (Yao, 1999), (Abraham, 2004), (Abraham, 2002), (Abraham & Nath, 2001), a hybrid approach combining EAs and gradient descent could be attractive.

4.2.2 Evolutionary Search of Architectures:

The architecture of an ANN includes its topological structure, i.e., connectivity, and the transfer function of each node in the ANN. Architecture design is crucial in the successful application of ANNs because the architecture has significant impact on a network's information processing capabilities. Up to now, architecture design is still very much a human

expert's job. It depends heavily on the expert experience and a tedious trial-and-error process. There is no systematic way to design a near-optimal architecture for a given task automatically. Research on constructive and destructive algorithms represents an effort toward the automatic (evolutionary) design of neural network architectures (Yao, 1999), (Abraham, 2004), (Abraham, 2002). A constructive algorithm starts with a minimal network (network with minimal number of hidden layers, nodes, and connections) and adds new layers, nodes, and connections when necessary during training while a destructive algorithm does the opposite, i.e., starts with the maximal network and deletes unnecessary layers, nodes, and connections during training.

A key issue in evolving neural network architecture is to determine how to encode the architectures and how much information should be encoded in the chromosome. There are two types of encoding methods for finding the optimum architecture: direct and indirect encoding methods (Yao & Liu, 1997), (Branke, 1995), (Koza & Rice, 1991), (Gutierrez, Isasi, Molina, Sanchis, & Galvan, 2001), (Harp, Samad, & Guha, 1989), (Kitano, 1990), (Molina, Galván, Isasi, & Sanchis, 2000-A), (Gruau & Whitley, 1993), (Gruau, Whitley, & Pyeatt, 1995), (Harp, Samad, & Guha, 1990), (Molina, Galván, Isasi, & Sanchis, 2000-B). In direct encoding, all the details, i.e., every connection and node of an architecture, can be specified by the chromosome. The direct encoding method is relatively simple and straightforward to implement but requires much larger chromosomes. In indirect encoding, important parameters such as the number of hidden layers and hidden nodes in each layer of the network are represented and the details of the exact connectivity are left to developmental rules. Using the indirect encoding scheme will minimize the size of the genotype string and improve scalability. Several indirect encoding methods were successfully used in many applications for optimizing the neural network architecture such as: graph generation system, symbiotic adaptive neuro-evolution, marker based genetic coding, L-systems, cellular encoding, fractal representation, etc. The following figure (4.3) demonstrates how typical neural network architecture could be directly encoded by a square connectivity matrix and how the genotype is represented in a chromosome.



From / To	1	2	3	4	5
1	0	0	0	0	0
2	0	0	0	0	0
3	1	1	0	0	0
4	1	1	0	0	0
5	0	0	1	1	0

Genotype: 00000 00000 11000 11000 00110

Figure 4.3*:* Architecture chromosome using binary coding (direct encoding)

In evolving the NN architecture, the node transfer function is often assumed to be fixed and pre-defined before the architecture is decided. For an optimal network, the required node transfer function can be formulated as a global search problem, which is evolved simultaneously with the search for architectures.

Evolutionary search of ANN architecture can be formulated as follows (Yao, 1999), (Abraham, 2004), (Abraham & Nath, 2001)***:***

(1) The evolution of architectures has to be implemented such that the evolution of weight chromosomes are evolved at a faster rate, i.e. for every architecture chromosome, there will be several weight chromosomes evolving at a faster time scale.
(2) Generate an initial population of N architecture chromosomes. Evaluate the fitness of each EANN depending on the problem.
(3) Depending on the fitness and using suitable selection methods reproduce a number of children for each individual in the current generation.
(4) Apply genetic operators to each child individual generated above and obtain the next generation.
(5) Check whether the network has achieved the required error rate or the specified number of generations has been reached. Go to Step 3.
(6) End.

4.2.3 Evolutionary Search of Learning Rules:

An ANN learning algorithm may have different performance when applied to different architectures. The design of learning algorithms, more fundamentally the learning rules used to adjust connection weights (weight-updating rule), depends on the type of architectures under investigation and the task to be performed. In other words, ANN should learn its learning rule dynamically rather than have it designed and fixed manually. Evolving learning rules expected to enhance ANN's adaptivity greatly in a dynamic environment.

The key issue in evolving learning rules is how to encode the dynamic behavior of a learning rule into static chromosomes. Trying to develop a universal representation scheme which can specify any kind of dynamic behaviors is clearly impractical, let alone the prohibitive long computation time required to search such a learning rule space. Constraints have to be set on the type of dynamic behaviors, i.e., the basic form of learning rules being evolved in order to reduce the representation complexity and the search space. Two basic assumptions which have often been made on learning rules are (Yao, 1999), (Yao, 1993):

1) Weight updating depends only on local information such as the activation of the input node, the activation of the output node, the current connection weight, etc.,
2) The learning rule is the same for all connections in an ANN. A learning rule is assumed to be a linear function of these local variables and their products.

The learning rule can be described by the following function (Yao, 1999), (Abraham, 2004), (Abraham & Nath, 2001):

$$\Delta w(t) = \sum_{k=1}^{n} \sum_{i_1, i_2, \ldots i_k = 1}^{n} \left(\theta_{i_1, i_2, \ldots i_k} \prod_{j=1}^{k} x_{ij}(t-1) \right)$$

Where t is time, Δw is the weight change, $x_1, x_2, \ldots, x_n$ are local variables, and θ's are real-valued coefficients which will be determined by evolution. In other words, the evolution of learning rules in this case is equivalent to the evolution of real-valued vectors of θ's. Different θ's determine different learning rules.

Evolutionary Search of learning rules can be formulated as follows (Yao, 1999), (Abraham, 2004), (Yao, 1993)***:***

(1) The evolution of learning rules has to be implemented such that the evolution of architecture chromosomes are evolved at a faster rate i.e. for every learning rule chromosome, there will be several architecture chromosomes evolving at a faster time scale.
(2) Generate an initial population of N learning rules. Evaluate the fitness of each EANN depending on the problem.
(3) Depending on the fitness and using suitable selection methods reproduce a number of children for each individual in the current generation.
(4) Apply genetic operators to each child individual generated above and obtain the next generation.
(5) Check whether the network has achieved the required error rate or the specified number of generations has been reached. Go to Step 3.
(6) End.

Several researches have been working on formulating different optimal learning rules (Yao, 1999), (Abraham, 2002), (Abraham & Nath, 2001), (Baxter, 1992), (Chalmers, 1990), (S. Bengio, Y. Bengio, Cloutier, & Gecsei, 1992). The adaptive adjustment of BP algorithm's parameters, such as the learning rate and momentum, through evolution could be considered as the first attempt of the evolution of learning rules (Harp, Samad, & Guha, 1989), (Belew, McInerney, & Schraudolph, 1991).

4.3 Meta-Learning Evolutionary Artificial Neural Networks (MLEANN)

Evolutionary algorithms are used to adapt the connection weights, network architecture, and learning algorithms according to the problem environment. Even though evolutionary algorithms are well known as efficient global search algorithms, very often they miss the best local solutions in the complex solution space. In other words, they are inefficiency in fine-tuned local search (Yao, 1999), (Abraham, 2004), (Abraham, 2002), (Yao, 1993). This is especially true for GA's. The efficiency of evolutionary algorithms can be improved significantly by using a hybrid learning approach that incorporates the EA's global search

ability with local search's ability to fine tune. Thus, the EA is used to locate a good region in the space and then a local search procedure, such as BP or other random search algorithm, is used to find a near-optimal solution in this region. Several hybrid learning approaches had been successfully used for evolving neural network topology and/or weights (Abraham & Nath, 2000), (Abraham, 2004), (Abraham, 2002), (Yao & Liu, 1997), (Belew, McInerney, & Schraudolph, 1991), (Hendtlass & Podlena, 1995), (Magoulas, Plagianakos, & Vrahatis, 2001), (Wong, Chung, & Wong, 1998). One of these hybrid learning approaches is called meta-learning evolutionary artificial neural networks (MLEANN) (Abraham, 2004), (Abraham, 2002), (Abraham & Nath, 2001). It can be considered as an automatic computational framework that used a direct encoding method for the adaptive optimization of ANNs. The main aim of using this framework is to improve the learning process and to obtain a small and efficient design of NNs with faster convergence.

It is interesting to consider finding good initial weights as locating a good region in the weight space. Defining that basin of attraction of a local minimum as being composed of all the points, sets of weights in this case, which can converge to the local minimum through a local search algorithm, then a global minimum can easily be found by the local search algorithm if an EA can locate a point, i.e., a set of initial weights, in the basin of attraction of the global minimum. Figure (4.4) illustrates a simple case where there is only one connection weight in the ANN (Yao, 1999), (Abraham, 2004), (Yao, 1993). *WG1* and *WG2* could be considered as the initial weights as located by the evolutionary search and *WA, WB* could be considered as the corresponding final weights fine-tuned by meta-learning technique which is the work of the local search algorithms.

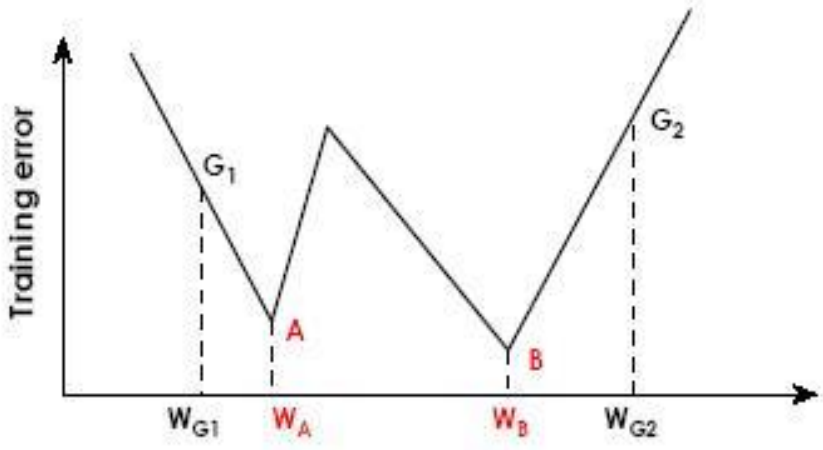


Figure 4.4: Fine tuning of weights using meta-learning

Figure (4.5) (Yao, 1999), (Abraham, 2004), (Abraham, 2002) illustrates the general interaction mechanism with the learning mechanism of the EANN evolving at the highest level on the slowest time scale.

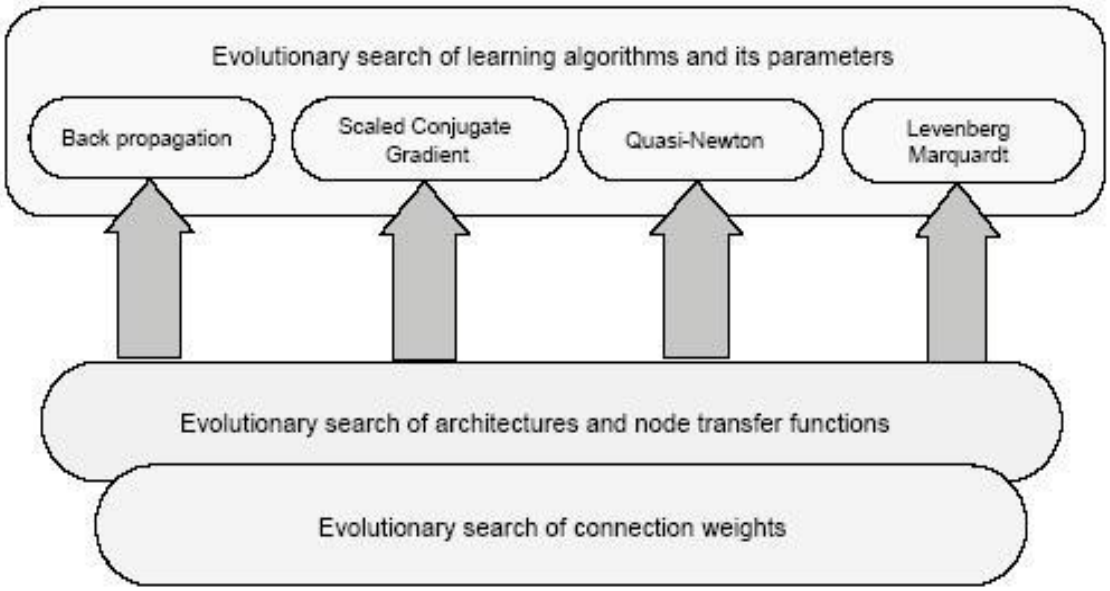


Figure 4.5: Interaction of various evolutionary search mechanism

In the MLEANN, all the randomly generated architecture of the initial population are trained and evolved separately by four different learning algorithms (backpropagation -BP, scaled conjugate gradient -SCG, quasi-Newton algorithm -QNA and Levenberg-Marquardt -LM) in a parallel environment. Parameters controlling the performance of the learning algorithm (as learning rate and momentum for BP) will be adapted according to the problem (Abraham, 2002), (Abraham & Nath, 2001). The basic Meta-learning algorithm for the EANN is as follows (Abraham, 2004), (Abraham, 2002), (Abraham & Nath, 2001):

1. Set t=0 and randomly generate an initial population of neural networks with architectures, node transfer functions and connection weights assigned at random.
2. In a parallel mode, train separately each network and evaluate its fitness using the learning algorithms: BP,SCG,QNA, and LM.
3. Based on fitness value, select parents for reproduction
4. Apply mutation to the parents and produce offspring (s) for next generation. Refill the population back to the defined size.
5. Repeat step 2
6. STOP when the required solution is found or number of iterations has reached the required limit.

The architecture of the chromosome is depicted in figure (4.6) (Abraham, 2004), (Abraham, 2002). For every learning algorithm parameter (LR2), there is the evolution of architectures (AR1, AR2, …, AR7) that proceeds on a faster time scale in an environment decided by the learning algorithm. For each architecture (AR3), the evolution of connection weights (WT1, WT2, ….., WT5) proceeds at a faster time scale in an environment decided by the problem, the learning algorithm and the architecture.

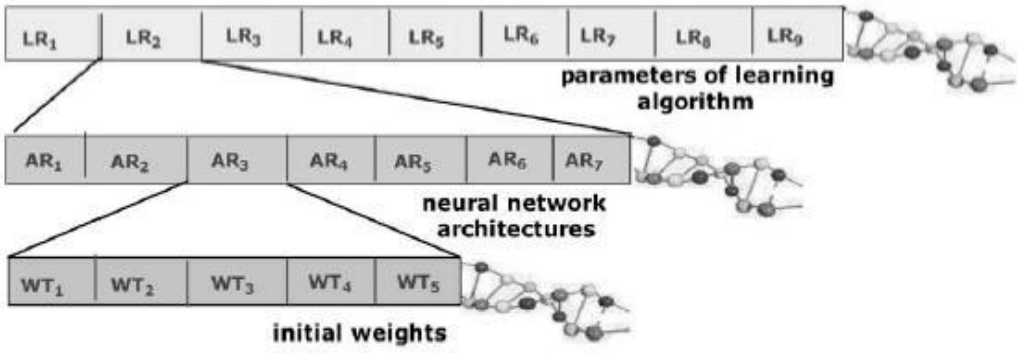


Figure 4.6: Chromosome representation of the MLEANN framework

4.4 Meta-Learning Evolutionary Artificial Neural Networks by Means of Cellular Automata

The previous MLEANN framework that was proposed by Ajith (Abraham, 2004), (Abraham, 2002) used the direct encoding methods for the adaptive optimization of artificial neural network architectures. These direct encoding methods base on the codification of the complete network into the chromosome. They are relatively simple and straightforward to implement but requires much larger chromosomes especially for ANNs with complex architectures (Yao & Liu, 1997), (Branke, 1995), (Koza & Rice, 1991), (Yao & Liu, 1998), (Braun & Weisbrod, 1993), (Yao, 1999). This could end in a too huge space search that could make the method impossible in practice. On the other hand, implementation of crossover operator for the chromosome is often difficult due to production of non-functional offsprings. An alternative more interesting for optimizing the ANN architecture is the indirect encoding methods (Gutierrez, Isasi, Molina, Sanchis, & Galvan, 2001), (Harp, Samad, & Guha, 1989), (Kitano, 1990), (Molina, Galván, Isasi, & Sanchis, 2000-A), (Gruau & Whitley, 1993), (Gruau, Whitley, & Pyeatt, 1995), (Harp, Samad, & Guha, 1990), (Koza & Rice, 1991), (Molina, Galván, Isasi, & Sanchis, 2000-B), (Chval, 2002), (Hussain & Browse, 1998), (Jacob & Rehder, 1993), (Luke & Spector, 1996). These methods concentrate on codifying a compact representation of the networks reducing the length of the genotype and avoiding the scalability problem. One of these indirect encoding methods is the cellular automata (Gutierrez, Isasi, Molina, Sanchis, & Galvan, 2001), (Wolfram, 1994), (Molina, Galván, Isasi, & Sanchis, 2000-B). This method was used by Gutirrez (Gutierrez, Isasi, Molina, Sanchis, & Galvan, 2001) and according to his experiment, he proved that using cellular configurations for designing feed-forward NN architectures is more efficient than using the direct encoding methods.

Therefore, we present an adaptive computational framework: meta-learning evolutionary artificial neural network by means of cellular automata. This framework (MLEANN-CA) combines the local search methods with the evolutionary learning in order to obtain an efficient design of NNs that is smaller, faster and with better generalization performance using direct and indirect encoding methods (Abu Salah & Al-Salqan, 2006-A), (Abu Salah & Al-Salqan, 2006-B). The MLEANN-CA framework is explored and simulated using Neurosolution and NeuroGenetic Optimizer toolboxes, and two famous chaotic time series.

4.4.1 The Proposed Approach -- MLEANN-CA:

Two main stages compose the proposed framework in this research: the cellular configuration stage and the meta-learning stage. The cellular configuration stage includes three main modules for designing small neural network architectures: the genetic algorithm module, the cellular automata module, and the neural network module (Abu Salah & Al-Salqan, 2006-A). The meta-learning stage includes the meta-learning algorithm that is responsible for training and evolving the new generated architectures with the direct codification using different learning algorithms in parallel mode. The system architecture and the modules relationship is shown in the following figure (4.7) (Abu Salah & Al-Salqan, 2006-B).

The MLEANN-CA approach can be summarized as follows**:**

1. Randomly, generate an initial population of neural networks with architectures according to the indicated problem.

2. Apply the indirect encoding technique (CA) for optimizing each NN architecture. This is done by the following steps:

   - ❖ The GA module takes charge of generating initial configurations of the cellular automata, i.e. seeds positions in a two-dimensional grid.
   - ❖ The cellular automata module takes the initial configurations and generates final configurations corresponding to particular NN architectures. This is done using cellular automata rules that allow the convergence of the automata toward a final configuration.
   - ❖ The neural network module translates these final cellular configurations into feed-forward NN with smaller architectures.

3. Use the translated NNs to create the population with architectures, node transfer functions, and weights assigned at random.

4. In parallel mode, train each translated neural network separately and evaluate the fitness value for each one using the four learning algorithms (BP, SCG, QNA, and LM).

5. Based on the fitness value, select parents for reproduction.

6. Apply mutation to the parents and produce offspring (s) for the next generation Refill the population back to the defined size.

7. Repeat step 4.

8. Stop when the required solution is found or number of iterations reached the required limit.

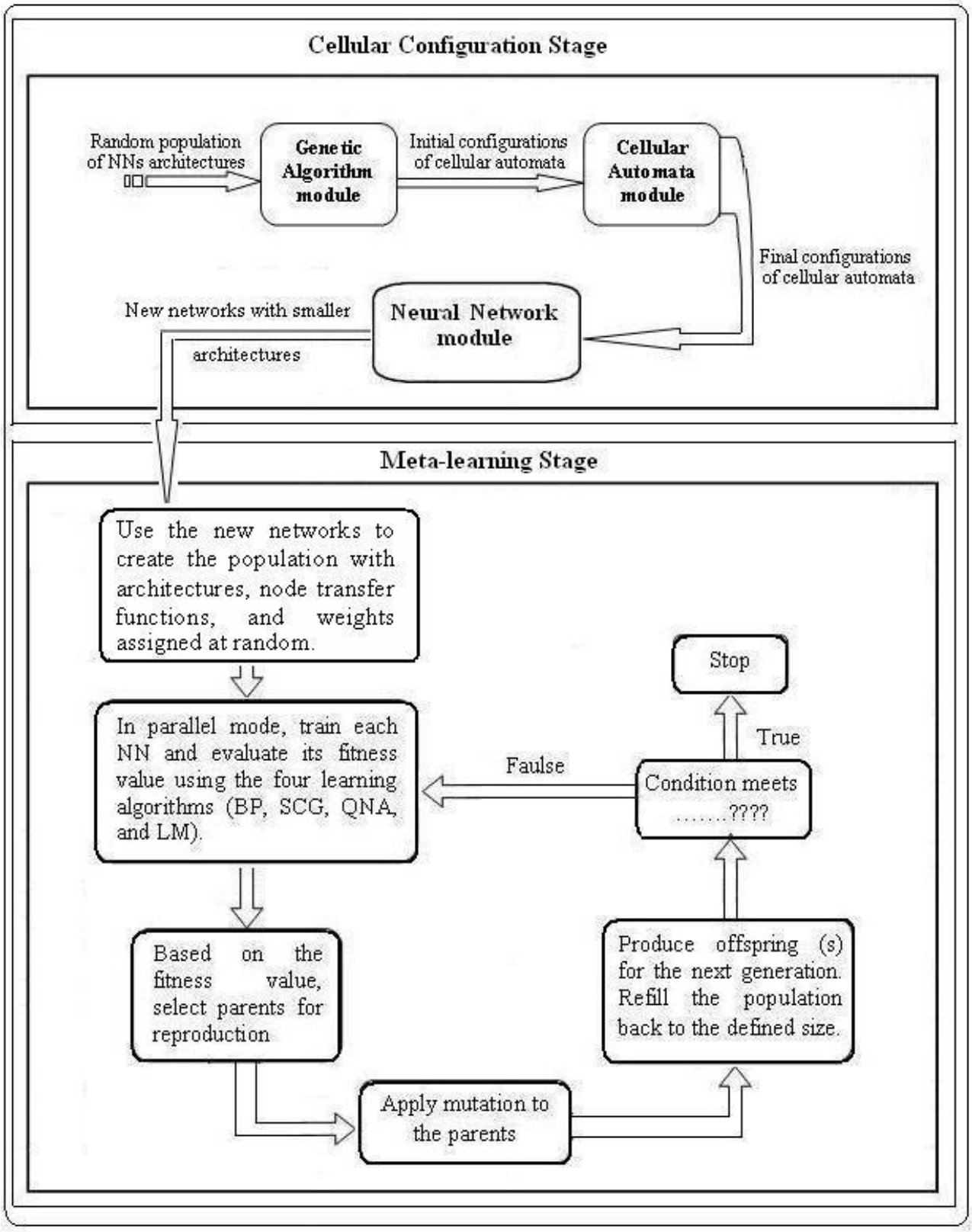


Figure 4.7: System's architecture and modules relationship

4.4.2 Genetic Algorithm Module:

As it was mentioned in Gutirrez research (Gutierrez, Isasi, Molina, Sanchis, & Galvan, 2001), GA module takes charge of generating initial configurations of the cellular automata. This module works with a population of chromosomes that codifies the positions of the seeds (growing and decreasing seeds) in a two-dimension grid. The size of chromosomes in the GA corresponds with the number of seeds, and it codifies all the possible locations of seeds in the grid. Chromosomes have been codified in base b, where b is the number of rows in the grid and is given through the number of inputs plus the number of outputs of the neural network (Gutierrez, Isasi, Molina, Sanchis, & Galvan, 2001), (Molina, Galván, Isasi, & Sanchis, 2000-B). Each seed is determined by a co-ordinate (x, y). A unique gene, indicating the row in which the seed is located, represents the first co-ordinate x. The second co-ordinate y will require more than one gene, if, as usual, the maximal number of hidden neurons is bigger than b. In this particular case, two genes have been used to codify the y coordinate, what allows a maximum of b*b hidden neurons. This could be a good estimation of the maximum number of neurons in the hidden layer, but any other consideration could be taken into account without modifying the proposed method. Hence, the chromosome will have 3 genes for each seed to be placed in the grid. The genetic algorithm module can be described by the following figure (4.8):

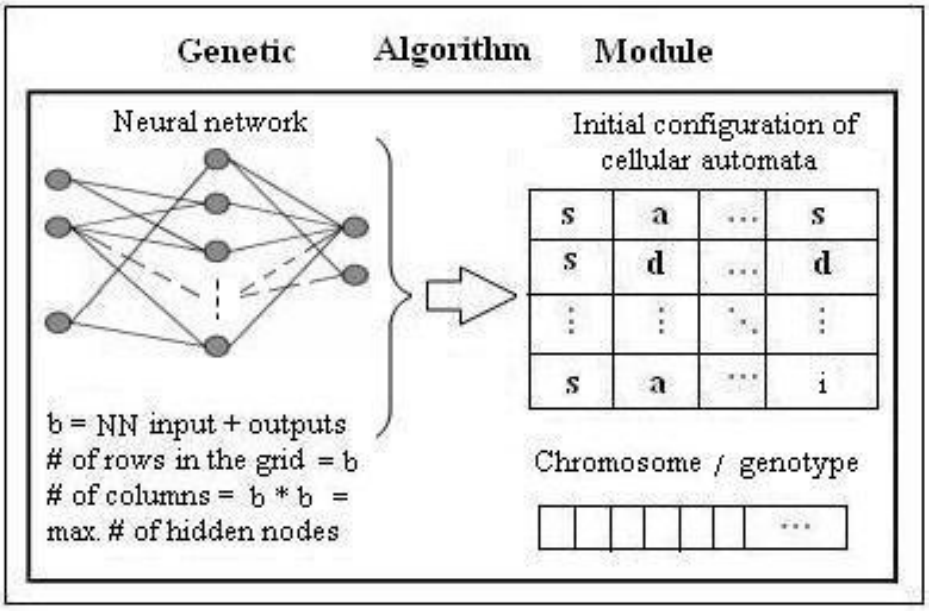


Figure 4.8: Genetic algorithm module

4.4.3 Cellular Automata Module:

The cellular automaton takes the initial configuration and generates a final configuration corresponding to a particular NN architecture. For generating neural networks architectures, a two-dimension CA has been used. The size of the two-dimension grid is defined as follows: the number of rows is equal to the number of input neurons plus the number of output neurons; number of columns corresponds with the maximum number of hidden neurons to be

considered. Each cell in the grid could be in two different states: active (occupied by a seed) or inactive. Two different kinds of seeds have been introduced: growing seeds and decreasing seeds. The first kind allows making connections and the second one removing connections. Each seed type corresponds with a different type of automata rule, so there are two rules called growing rule and decreasing rule respectively. The rules determine the evolution of the grid configuration and they have been designed allowing the reproduction of growing and decreasing seeds. In the description of the rules, **s** is a specific growing seed, **d** is a decreasing seed, **i** is an inactive state for the cell, and **a** means that the cell could be in any state or contains any type of seed (even a decreasing seed).

Growing Rules (Gutierrez, Isasi, Molina, Sanchis, & Galvan, 2001): They reproduce a particular growing seed when there are at least three identical growing seeds in its neighborhood. There are different configurations, growing seeds located in: rows, columns, or in a corner of the neighborhood. In the following figures (4.9) (a) and (b), some of those rules are shown (the others are symmetrical). The growing rules allow obtaining feed-forward NN with a large number of connections.

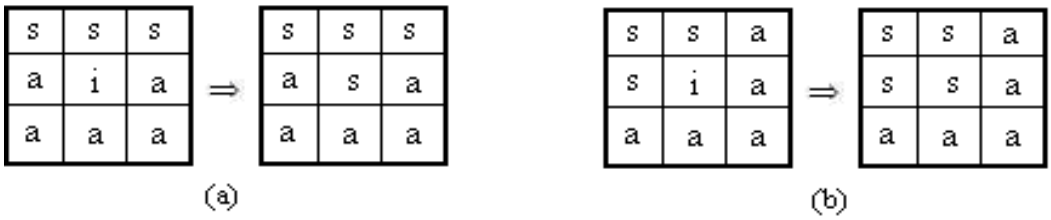


Figure 4.9 (a), (b): Examples of Growing Rules

Decreasing Rules (Gutierrez, Isasi, Molina, Sanchis, & Galvan, 2001): They remove connections in the network deactivating a cell in the grid when the cell has a seed and a cell of its neighborhood contains also a decreasing seed. One situation in which the decreasing rules can be applied is shown in the following figure (4.10), the others can be obtained symmetrically.

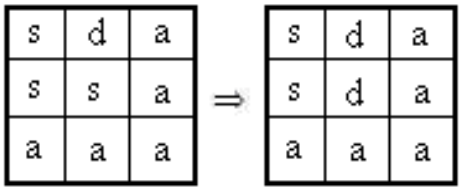


Figure 4.10: Example of Decreasing Rules

The mechanism of expanding the CA is as follows (Gutierrez, Isasi, Molina, Sanchis, & Galvan, 2001), (Molina, Galván, Isasi, & Sanchis, 2000-B):

1) The growing seeds are located in the grid.
2) An expansion of the growing seeds takes place. This expansion consists on replicating each seed in turns, over its quadratic neighborhood, in such a way that if a new seed has to be placed in a position previously occupied by another seed, the first one is replaced.
3) The growing rules are applied until no more rules could be fired.
4) The decreasing seeds are placed in the grid. If there are some other seeds in those places, they are replaced.
5) The decreasing rules are applied until the final configuration is reached.
6) The final configuration of the CA is obtained replacing the growing seeds by a **1** and the decreasing seeds or inactive cells by a **0**.

The cellular automata module can be described by the following figure (4.11)

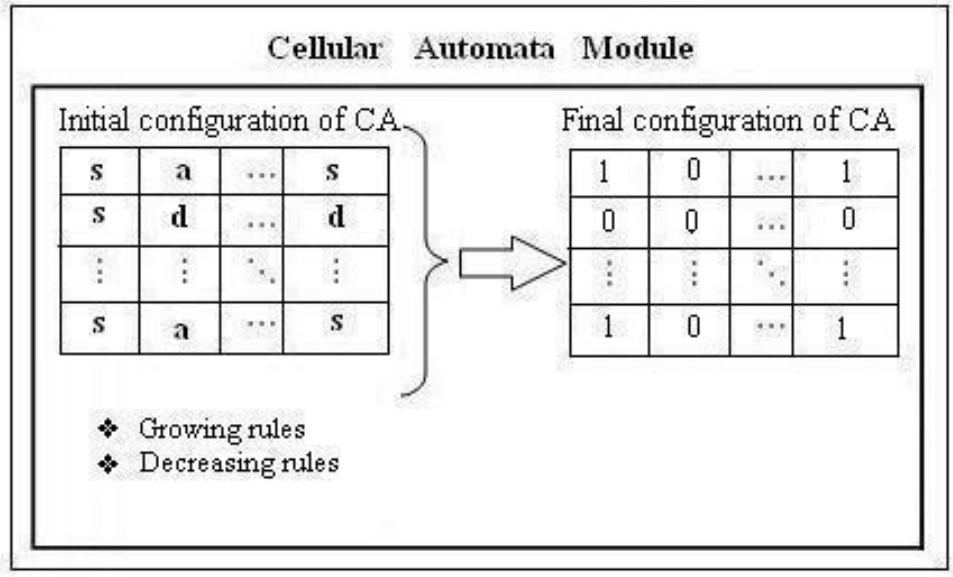


Figure 4.11: Cellular Automata module

4.4.4 Neural Network Module:

The neural network module translates the final cellular configuration into feed-forward NN architecture (Gutierrez, Isasi, Molina, Sanchis, & Galvan, 2001), (Molina, Galván, Isasi, & Sanchis, 2000-B). To relate the final configuration of the cellular automata with an architecture of a neural network, the following meaning for a cell in the (x,y) grid is defined (Gutierrez, Isasi, Molina, Sanchis, & Galvan, 2001):

- If $x < n$, with n the number of input neurons, (x,y) represents a connection between the x-th input neuron and the y-th hidden neuron.
- If $x > n$, (x,y) represents a connection between the y-th hidden neuron and the (x -n) -th output neuron.

In the final configuration, **1** is interpreted as a connection, and **0** as the absence of connection. Thus, the rows and columns in the **matrix** with values 0 are removed. A new and shorter binary matrix (M) is obtained. If Mij=1 then a connection between the i-th input neuron and the j-th hidden neuron is created, or between the j-th hidden neuron and the (i-n)-th output neuron, as is previously described. If Mij=0, there do not exist connection between that neurons.

In this module, when the final matrix connection is obtained from the final configuration of CA there are some special cases take into account, following these steps (Gutierrez, Isasi, Molina, Sanchis, & Galvan, 2001):

- If there is a node in hidden layer without any connection to output, this node is eliminated from the net.

- When a hidden node has no connection from input, but it's connected to output layer, two chances have been considerate: penalizes the net and don't train it, or eliminate that node and is training.

- If an output node has no connection from hidden layer, the net is penalized and is not trained.

The neural network module can be described by the following figure (4.12)

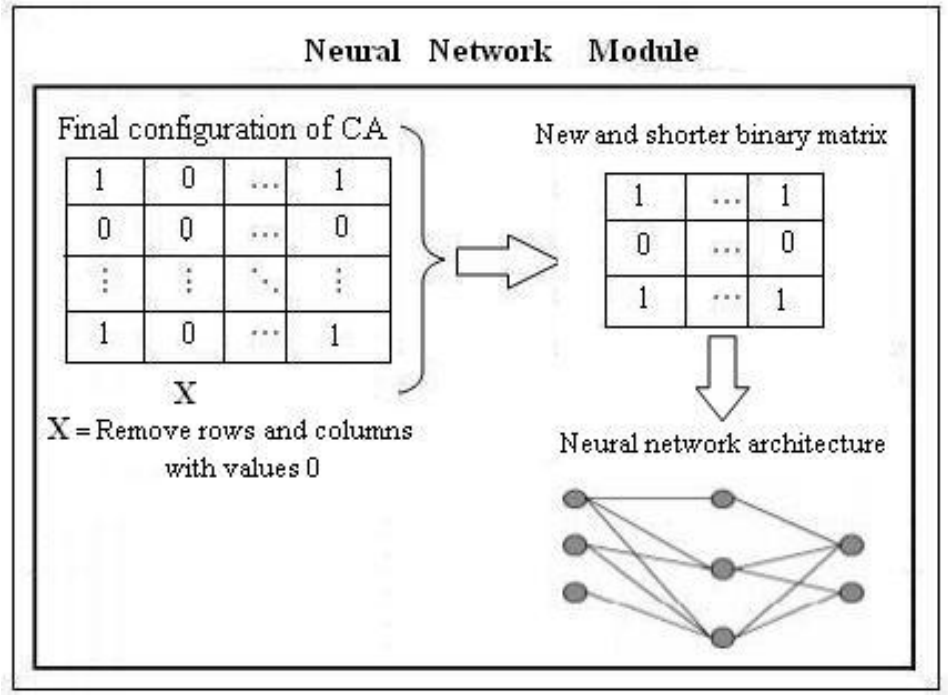


Figure 4.12: Neural network module

After obtaining the new network architectures, they are initialized by random weights and trained and evolved in parallel mode using four different learning algorithms. This is, absolutely, the work of the meta-learning algorithm for the EANN.

Summary

In this chapter, we presented our proposed framework: Meta-Learning Evolutionary Artificial Neural Networks by means of Cellular Automata. It is an adaptive computational framework based on evolutionary learning and local search procedures for automatic design of optimal artificial neural networks using direct and indirect encoding methods. In this proposed framework, the evolutionary cellular configurations are used to, first, design small feed-forward network architectures, and then all the generated architectures are trained and evolved separately using the meta-learning algorithm with the direct evolutionary approach, where four different learning algorithms are used in parallel mode.

We started this chapter by presenting the evolutionary artificial neural networks and its general framework. Then, we introduced three kinds of evolution in EANNs: evolution of connection weights, architectures, and learning rules. After that, we described the previous MLEANN framework that was proposed by Ajith. In that framework, the direct encoding methods were used for the adaptive optimization of neural network architectures. These direct encoding methods based on the codification of the complete network into the chromosome, thus they required much larger chromosomes and this could end in a too huge space search. Finally, we examined our proposed framework (MLEANN-CA) that used indirect encoding methods, i.e. cellular automata, for designing optimal network architectures. These methods concentrate on codifying a compact representation of the networks reducing the length of the chromosomes. Using MLEANN-CA framework will significantly improve the learning process, increase the scalability, and obtain a small and efficient design of neural networks with faster convergence and better generalization performance.

In the next chapter, Experiments and Results, we will test and explore the performance of our proposed framework (MLEANN-CA) and prove its efficiency.

CHAPTER FIVE

EXPERIMENTS AND RESULTS

This chapter deals with the experiments performed to evaluate the proposed MLEANN-CA framework and to compare it with previous approaches. In section 5.1 we describe the test collections (data sets) we will use. In section 5.2 we look at the test environment in which the experiments were conducted and at the parameters used during the experiments. Section 5.3 describes how the experiments were performed and what results are collected from these experiments. The results are analyzed and discussed in this section.

5.1 Test Collections - Data Sets

In our experiments, we used two different time series for training the neural networks and evaluating their performance. These data sets were used before in Ajith's work (Abraham, 2004), so it will be easy to compare our results with those in Ajith experiments. The raw data sets for these two time series could be found in http://neural.cs.nthu.edu.tw/jang/dataset/

- **Mackey-glass Chaotic Time Series**. The Mackey-glass differential equation (Mackey & Glass, 1977) is a well known and widely used benchmark problem in neural network and fuzzy modeling research communities. This time series is chaotic, it will not converge or diverge and the trajectory is highly sensitive to initial conditions.

$$\frac{dx(t)}{dt} = \frac{0.2x(t-\tau)}{1+x^{10}(t-\tau)} - 0.1x(t).$$

  We used all or some of these values: {x(t −18), x(t −12), x(t −6), x(t)} to predict {x(t +6) and / or x(t+12)}. Fourth order Runge-Kutta method was used to generate 1000 data series where data from t = 118 to 1117. The data was sampled every 6 points, as it is usually recommended for the Mackey Glass time series. The time step used in the method is 0.1 and initial condition were x(0) = 1.2, τ = 17, x(t) = 0 for t<0. First 500 data sets were used for training and remaining data for testing.

- **Gas Furnace Time Series**. The gas furnace data from the Box-Jenkins (Box & Jenkins, 1970) is used in our simulations. This time series was used to predict the CO2 (carbon dioxide) concentration. In a gas furnace system, air and methane are combined to form a mixture of gases containing CO2. Air fed into the gas furnace is kept constant, while the methane feed rate can be varied in any desired manner. After that, the resulting CO2 concentration is measured in the exhaust gases at the outlet of the furnace. In this time series, there are originally 296 data points. We are trying to predict {y(t+1) and / or y(t)} based on all or some of these best set values {y(t-1), y(t-2), u(t-3), u(t-4)}, where **y** is the CO2 concentration and **u** is the gas flow rate. This reduces the number of effective data points to 290. The first 50% of data was used for training and remaining for testing.

5.2 Test Environment

The experiments were simulated using two toolboxes: Neurosolution toolbox, and NeuroGenetic Optimizer toolbox. These toolboxes are used for training and optimizing the neural networks. The experiments were carried out on a computer with the following configurations: 1.8 GHz AMD processor, 512 MB RAM, and Windows XP Professional.

In our proposed framework (MLEANN-CA), several parameters can influence the experiments. These parameters can be distinguished into two categories: **(i)** the parameters that are related to the Evolutionary Artificial Neural networks, and **(ii)** the parameters that are related to the learning algorithms. These parameters in tables (5.1) and (5.2) were set to be the same for the two data sets, and were finalized after a few trail and error approaches according to Ajith (Abraham, 2004).

Table 5.1: Parameters used for EANNs

Parameter	Setting
Population size	40 (chromosomes)
Maximum no of generations	40
Number of hidden nodes	3 to 36 neurons
Activation functions	tanh (T), logistic (L), sigmoidal (S).
Output neuron	Linear (Li)
Training epochs	2500 for standard training , 500 for optimizing
Initialization of weights	+ / - 0.3
Ranked based selection	0.50
Mutation rate	0.40
Crossover / one point	0.50

Table 5.2: Parameters for the Learning Algorithms

Learning algorithm	Parameter	Setting
Backpropagation (BP)	Learning rate	0.25-0.05
	Momentum rate	0.25-0.05
Scaled conjugate gradient (SCG)	----------	----------
Quasi Newton algorithm (QNA)	Step size	0.1 - 0.6
Levenberg Marquardt (LM)	Learning rate	0.001 -0.02

5.3 The Experiments Conducted

For each data set mentioned before, three main experimental simulations are carried out (Abu Salah & Al-Salqan, 2006-B). The first one evaluates the performance of the conventional design of artificial neural networks. The second one explores the performance of the MLEANN framework. The third one test and explore the performance of our proposed approach: MLEANN-CA. These experiments use four different learning algorithms (BP, SCG, QNA, LM) in the training process. By applying these experiments, we (a) should know the best solution, we (b) can carefully control various parameters, and we (c) should know the effect of different learning algorithms namely BP, SCG, QNA and LM on different data sets.

5.3.1 Artificial Neural Networks: Experimentation and Simulation Results:

In this subsection we explored the performance of the conventional design of artificial neural networks. We used two different time series, i.e. Mackey-glass and Gas furnace, for training the artificial neural networks and evaluating the performance. The Neurosolution and NeuroGenetic optimizer toolboxes are used for training the ANNs. We used a feed-forward neural network with one hidden layer for the two time series. The number of hidden neurons were varied (3, 5, 8, 14, 16, 18, 24, 36) as indicated in table (5.1). The speed of convergence and generalization error for each of the four learning algorithms was observed. Any required parameter for any learning algorithm is found in table (5.2). Performances of the four different learning algorithms were evaluated when the architecture is changed. The experiments were replicated three times each with a different starting condition (random weights) and the worst errors were reported. No stopping criterion, and no method of controlling generalization is used other than the maximum number of updates (epochs). All networks were trained for an identical number of stochastic updates: 2500 epochs.

5.3.1.1 Mackey-glass Time Series with Different Network Architectures:

This experiment investigates the training and generalization behavior of the networks for the Mackey glass time series when the network architecture was changed. The same architectures were used for the four learning algorithms using same node transfer function for the hidden layer and the output layer: tanh (T) and linear (Li). The node transfer function has an effect on the training speed and generalization performance. Therefore, I used the tanh function after examining its performance and compare it with other activation functions. Tables (5.3 -5.5) summarize the empirical results of training and generalization for the Mackey glass. Figures (5.1–5.6) graphically depict the training and generalization performance for the Mackey glass with different learning methods.

Table (5.3) summarizes the empirical results of training and generalization for different architectures with four inputs and one output in Mackey-glass time series $\{x(t-18), x(t-12), x(t-6), x(t), x(t+6)\}$.

Table 5.3: Training and test performance for Mackey-glass time series for different architectures with four inputs and one output

Mackey-glass time series			
Learning algorithm	**Hidden neurons**	**RMSE**	
		Training data	**Testing data**
BP	3	0.0990	0.0993
	5	0.0971	0.0965
	8	0.0932	0.0924
	14	0.0907	0.0897
	16	0.0838	0.0874
	18	0.0782	0.0768
	24	0.0456	0.0454
	36	0.0408	0.0403
SCG	3	0.0088	0.0095
	5	0.0079	0.0084
	8	0.0066	0.0075
	14	0.0052	0.0063
	16	0.0071	0.0069
	18	0.0070	0.0071
	24	0.0055	0.0055
	36	0.0048	0.0049
QNA	3	0.0076	0.0075
	5	0.0061	0.0062
	8	0.0055	0.0054
	14	0.0042	0.0041
	16	0.0033	0.0032
	18	0.0043	0.0044
	24	0.0037	0.0039
	36	0.0035	0.0034
LM	3	0.0051	0.0060
	5	0.0036	0.0043
	8	0.0020	0.0022
	14	0.0019	0.0019
	16	0.0017	0.0017
	18	0.0017	0.0017
	24	0.0012	0.0012
	36	0.0010	0.0010

Table (5.4) summarizes the empirical results of training and generalization for different architectures with four inputs and two outputs in Mackey-glass time series {x(t −18), x(t −12), x(t −6), x(t), x(t +6), x(t+12)}.

Table 5.4: Training and test performance for Mackey-glass time series for different architectures with four inputs and two outputs

Mackey-glass time series			
Learning algorithm	**Hidden neurons**	**RMSE**	
		Training data	**Testing data**
BP	3	0.1188	0.1192
	5	0.1165	0.1158
	8	0.1118	0.1109
	14	0.1088	0.1076
	16	0.1006	0.1049
	18	0.0938	0.0922
	24	0.0547	0.0545
	36	0.0490	0.0484
SCG	3	0.0106	0.0114
	5	0.0095	0.0101
	8	0.0079	0.0090
	14	0.0062	0.0076
	16	0.0085	0.0083
	18	0.0084	0.0085
	24	0.0066	0.0066
	36	0.0058	0.0059
QNA	3	0.0091	0.0090
	5	0.0073	0.0074
	8	0.0066	0.0065
	14	0.0050	0.0049
	16	0.0040	0.0038
	18	0.0052	0.0053
	24	0.0044	0.0047
	36	0.0042	0.0041
LM	3	0.0061	0.0072
	5	0.0043	0.0052
	8	0.0024	0.0026
	14	0.0023	0.0023
	16	0.0020	0.0020
	18	0.0020	0.0020

	24	0.0014	0.0014
	36	0.0012	0.0012

Table (5.5) summarizes the empirical results of training and generalization for different architectures with three inputs and two outputs in Mackey-glass time series {x(t −12), x(t −6), x(t), x(t +6), x(t+12)}.

Table 5.5: Training and test performance for Mackey-glass time series for different architectures with three inputs and two outputs

Mackey-glass time series			
Learning algorithm	**Hidden neurons**	**RMSE**	
		Training data	**Testing data**
BP	3	0.1426	0.1430
	5	0.1398	0.1390
	8	0.1342	0.1331
	14	0.1306	0.1292
	16	0.1207	0.1259
	18	0.1126	0.1106
	24	0.0657	0.0654
	36	0.0588	0.0580
SCG	3	0.0127	0.0137
	5	0.0114	0.0121
	8	0.0095	0.0108
	14	0.0075	0.0091
	16	0.0102	0.0099
	18	0.0101	0.0102
	24	0.0079	0.0079
	36	0.0069	0.0071
QNA	3	0.0109	0.0108
	5	0.0088	0.0089
	8	0.0079	0.0078
	14	0.0060	0.0059
	16	0.0048	0.0046
	18	0.0062	0.0063
	24	0.0053	0.0056
	36	0.0050	0.0049
LM	3	0.0073	0.0086
	5	0.0052	0.0062

	8	0.0029	0.0032
	14	0.0027	0.0027
	16	0.0024	0.0024
	18	0.0024	0.0024
	24	0.0017	0.0017
	36	0.0014	0.0014

Figures (5.1–5.6) graphically depict the training and generalization performance for the different learning methods with different architectures using Mackey-glass time series.

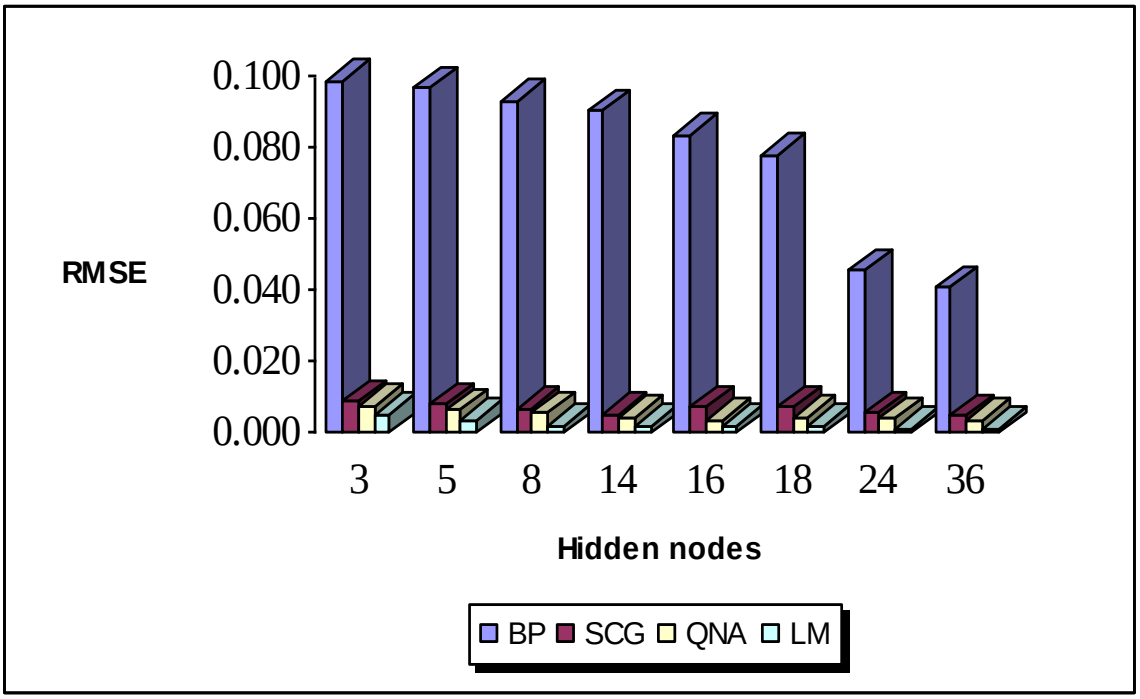


Figure 5.1: Architecture variation: Mackey-glass time series training performance for different training algorithms with 4 inputs and 1 output network

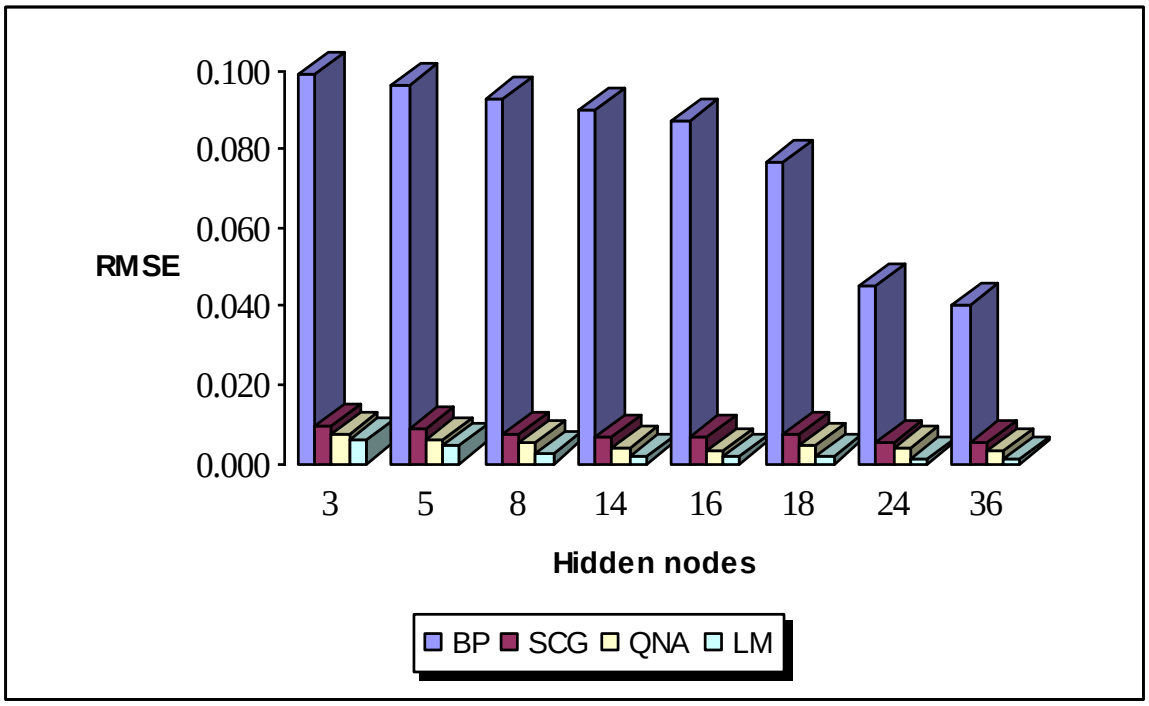


Figure 5.2: Architecture variation: Mackey-glass time series generalization performance for different learning algorithms with 4 inputs and 1 output network

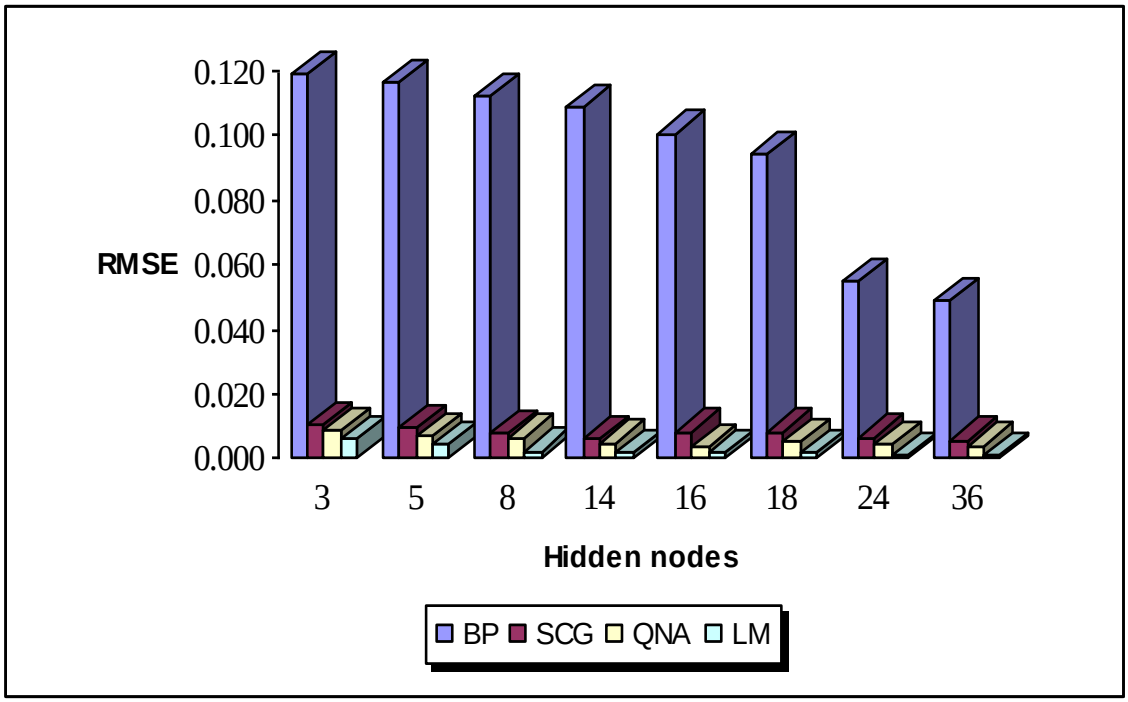


Figure 5.3: Architecture variation: Mackey-glass time series training performance for different training algorithms with 4 inputs and 2 output network

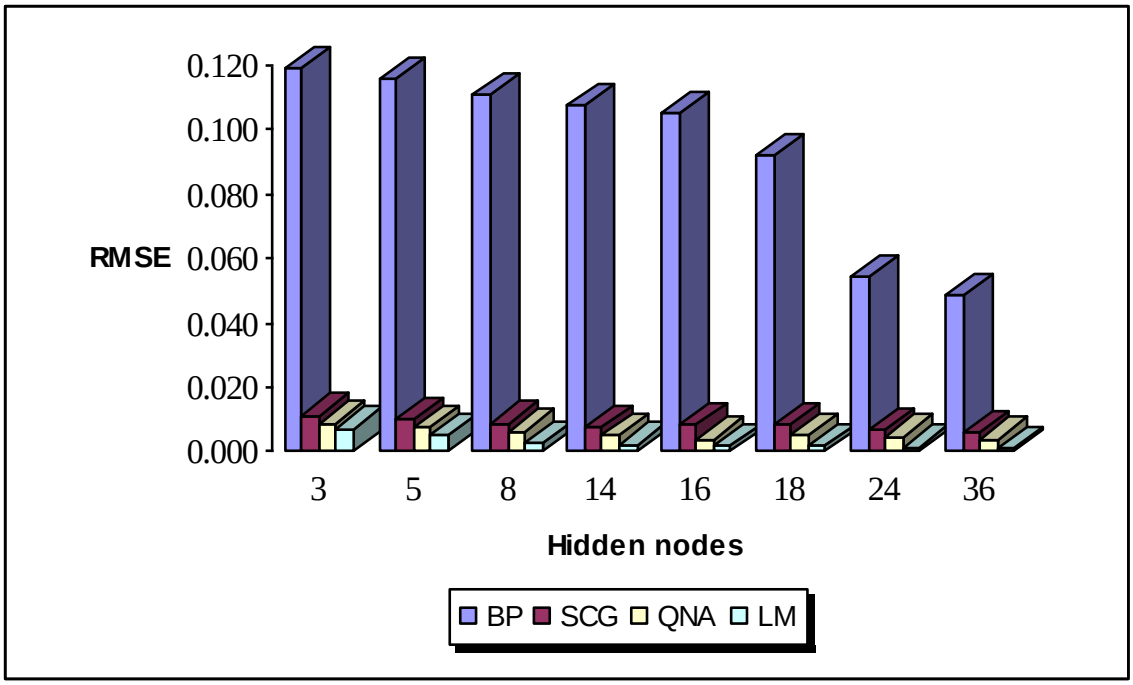


Figure 5.4: Architecture variation: Mackey-glass time series generalization performance for different learning algorithms with 4 inputs and 2 output network

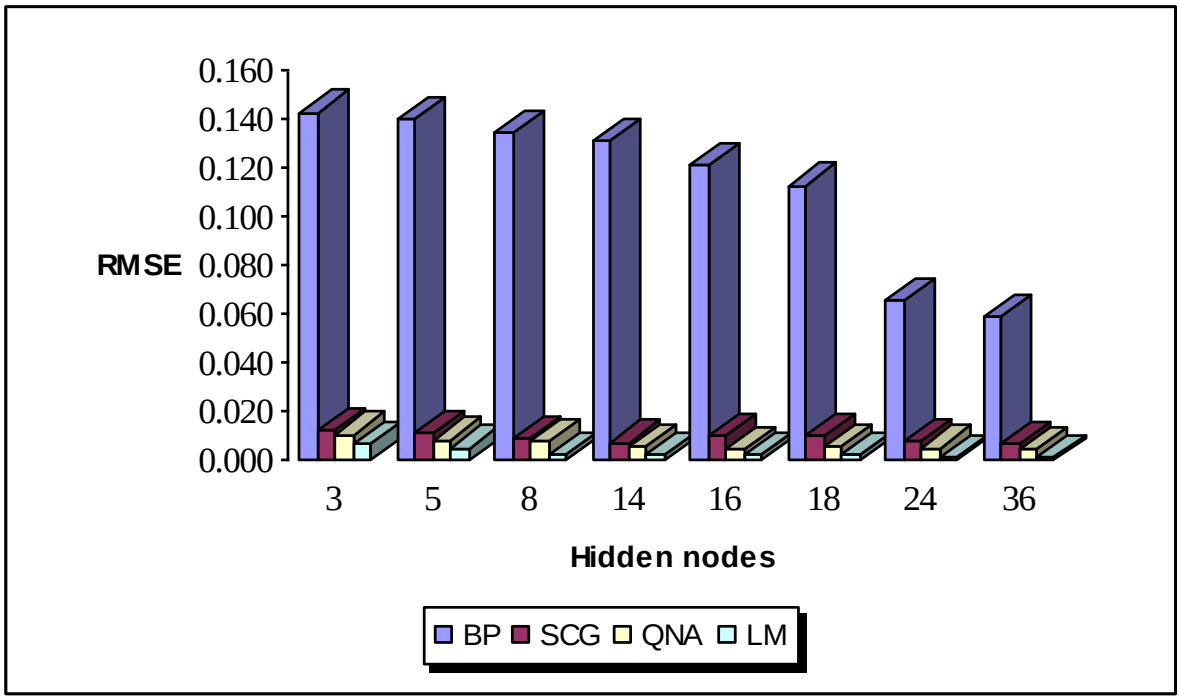


Figure 5.5: Architecture variation: Mackey-glass time series training performance for different training algorithms with 3 inputs and 2 output network

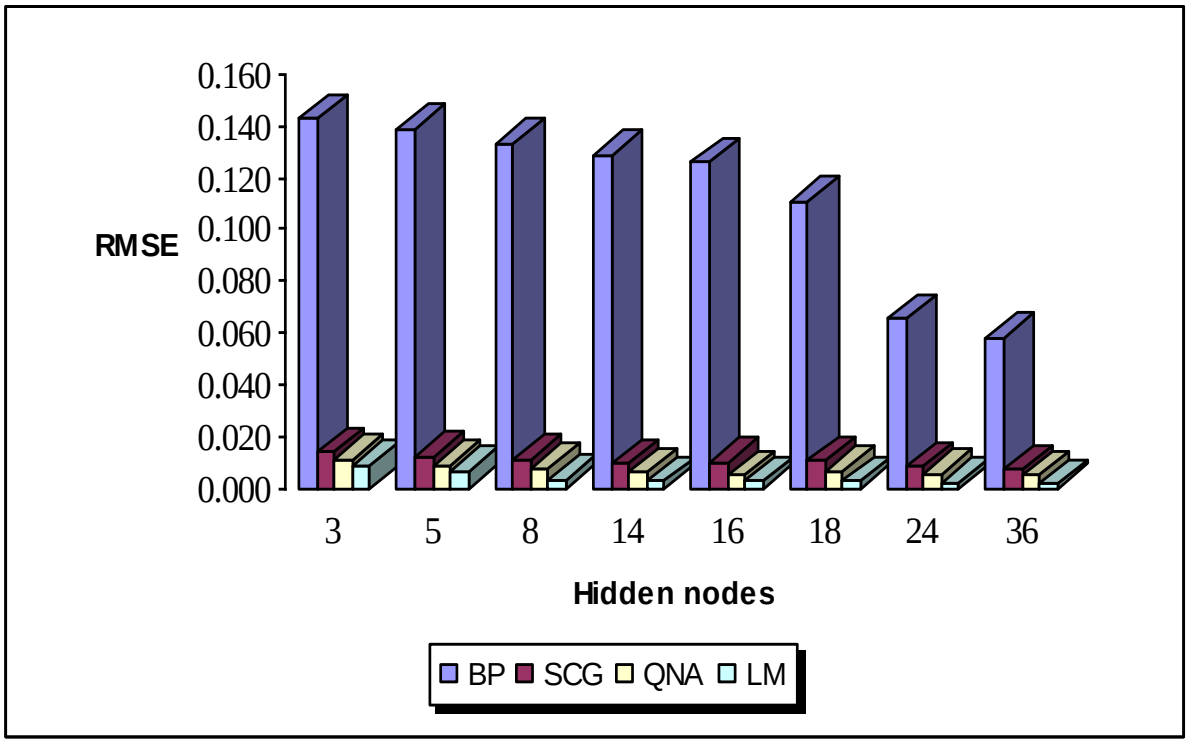


Figure 5.6: Architecture variation: Mackey-glass time series generalization performance for different learning algorithms with 3 inputs and 2 output network

5.3.1.2 Gas Furnace Time Series with Different Network Architectures:

This experiment investigates the training and generalization behavior of the networks for the Gas furnace time series when the network architecture was changed. The same architectures were used for the four learning algorithms using same node transfer function for the hidden layer and the output layer: tanh (T) and linear (Li). Tables (5.6 -5.8) summarize the empirical results of training and generalization for the Gas furnace time series. Figures (5.7–5.12) graphically depict the training and generalization performance for the Gas furnace time series with different learning methods.

Table (5.6) summarizes the empirical results of training and generalization for different architectures with four inputs and one output in Gas furnace time series {y(t-1), y (t-2), u(t-3),u (t-4), y (t)}.

Table 5.6: Training and test performance for Gas furnace time series for different architectures with four inputs and one output

Gas Furnace time series			
Learning algorithm	**Hidden neurons**	**RMSE**	
		Training data	**Testing data**
BP	3	0.0461	0.0496
	5	0.0458	0.0527
	8	0.0451	0.0569
	14	0.0419	0.0807
	16	0.0522	0.0660
	18	0.0448	0.0479
	24	0.0414	0.0606
	36	0.0428	0.0686
SCG	3	0.0123	0.0333
	5	0.0115	0.0326
	8	0.0109	0.0318
	14	0.0100	0.0207
	16	0.0098	0.0206
	18	0.0103	0.0206
	24	0.0096	0.0229
	36	0.0093	0.0308
QNA	3	0.0098	0.0340
	5	0.0093	0.0332
	8	0.0089	0.0326
	14	0.0087	0.0343
	16	0.0083	0.0291
	18	0.0083	0.0235
	24	0.0080	0.0323
	36	0.0082	0.0329
LM	3	0.0081	0.0305
	5	0.0077	0.0313
	8	0.0079	0.0321
	14	0.0074	0.0281
	16	0.0082	0.0607
	18	0.0073	0.0675
	24	0.0069	0.1160
	36	0.0064	0.1342

Table (5.7) summarizes the empirical results of training and generalization for different architectures with four inputs and two outputs in Gas furnace time series {y(t-1), y (t-2), u(t-3),u (t-4), y (t), y(t+1)}.

Table 5.7: Training and test performance for Gas furnace time series for different architectures with four inputs and two outputs

Gas Furnace time series			
Learning algorithm	**Hidden neurons**	**RMSE**	
		Training data	**Testing data**
BP	3	0.0553	0.0596
	5	0.0550	0.0632
	8	0.0541	0.0683
	14	0.0503	0.0968
	16	0.0626	0.0792
	18	0.0538	0.0575
	24	0.0497	0.0728
	36	0.0514	0.0823
SCG	3	0.0148	0.0433
	5	0.0138	0.0424
	8	0.0131	0.0414
	14	0.0120	0.0269
	16	0.0118	0.0268
	18	0.0124	0.0268
	24	0.0115	0.0298
	36	0.0112	0.0401
QNA	3	0.0118	0.0442
	5	0.0112	0.0431
	8	0.0107	0.0424
	14	0.0104	0.0446
	16	0.0100	0.0378
	18	0.0100	0.0306
	24	0.0096	0.0419
	36	0.0098	0.0427
LM	3	0.0097	0.0397
	5	0.0092	0.0407
	8	0.0095	0.0418
	14	0.0089	0.0366
	16	0.0098	0.0789
	18	0.0088	0.0878

	24	0.0083	0.1508
	36	0.0077	0.1744

Table (5.8) summarizes the empirical results of training and generalization for different architectures with three inputs and two outputs in Gas furnace time series {y(t-1), u(t-3), u(t-4), y(t), y(t+1)}.

Table 5.8: Training and test performance for Gas furnace time series for different architectures with three inputs and two outputs

Gas Furnace time series			
Learning algorithm	**Hidden neurons**	**RMSE**	
		Training data	**Testing data**
BP	3	0.0664	0.0715
	5	0.0660	0.0759
	8	0.0649	0.0820
	14	0.0603	0.1162
	16	0.0752	0.0950
	18	0.0645	0.0689
	24	0.0596	0.0873
	36	0.0616	0.0987
SCG	3	0.0177	0.0520
	5	0.0166	0.0509
	8	0.0157	0.0496
	14	0.0144	0.0323
	16	0.0141	0.0322
	18	0.0148	0.0322
	24	0.0138	0.0358
	36	0.0134	0.0481
QNA	3	0.0141	0.0530
	5	0.0134	0.0518
	8	0.0128	0.0509
	14	0.0125	0.0535
	16	0.0120	0.0453
	18	0.0120	0.0367
	24	0.0115	0.0503
	36	0.0118	0.0513
LM	3	0.0117	0.0476

	5	0.0111	0.0488
	8	0.0114	0.0501
	14	0.0107	0.0439
	16	0.0118	0.0947
	18	0.0105	0.1053
	24	0.0099	0.1810
	36	0.0092	0.2093

Figures (7–12) graphically depict the training and generalization performance for the different learning methods with different architectures using Gas furnace time series

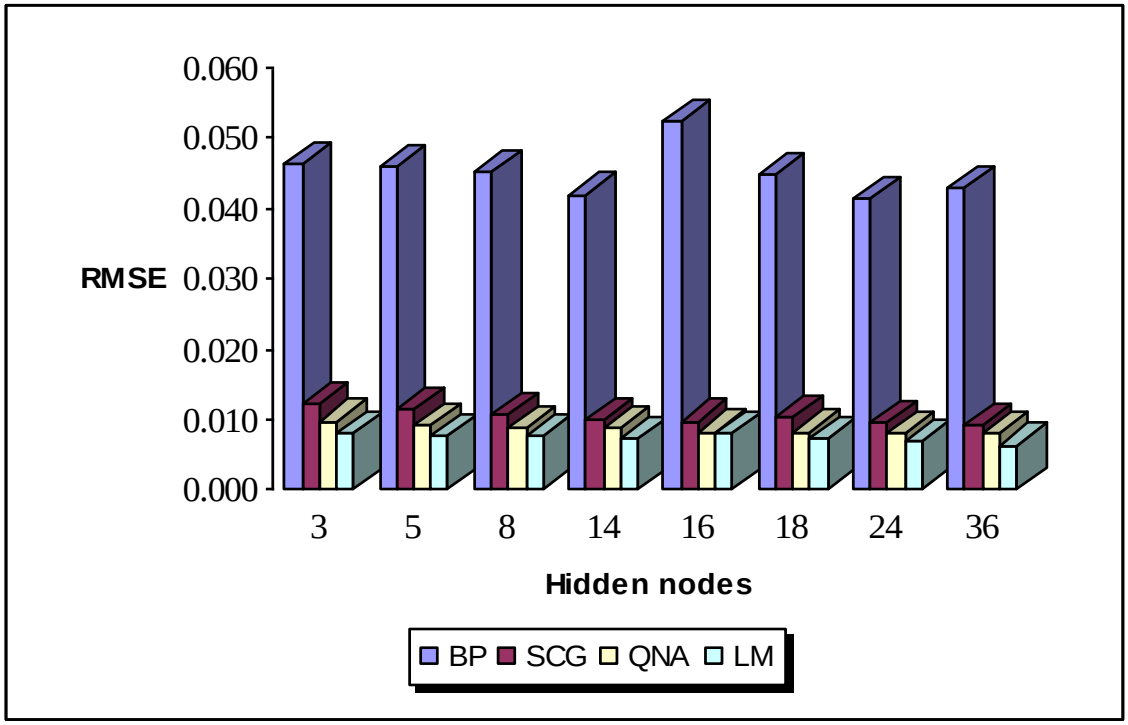


Figure 5.7: Architecture variation: Gas furnace time series training performance for different training algorithms with 4 inputs and 1 output network

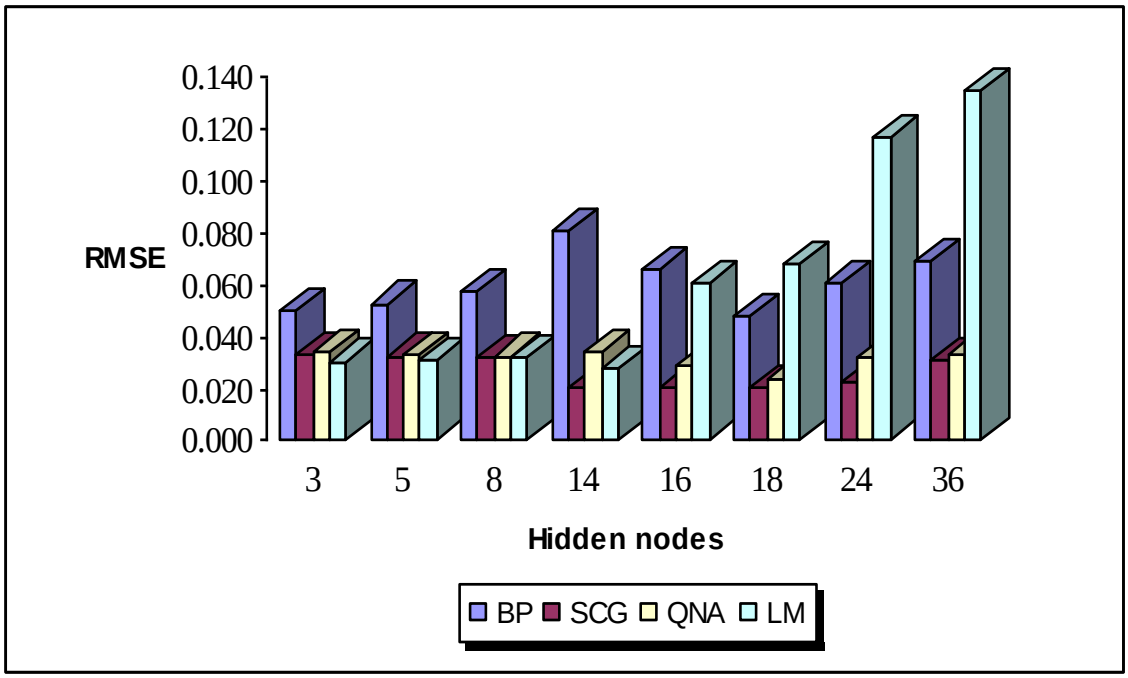


Figure 5.8: Architecture variation: Gas furnace time series generalization performance for different learning algorithms with 4 inputs and 1 output network

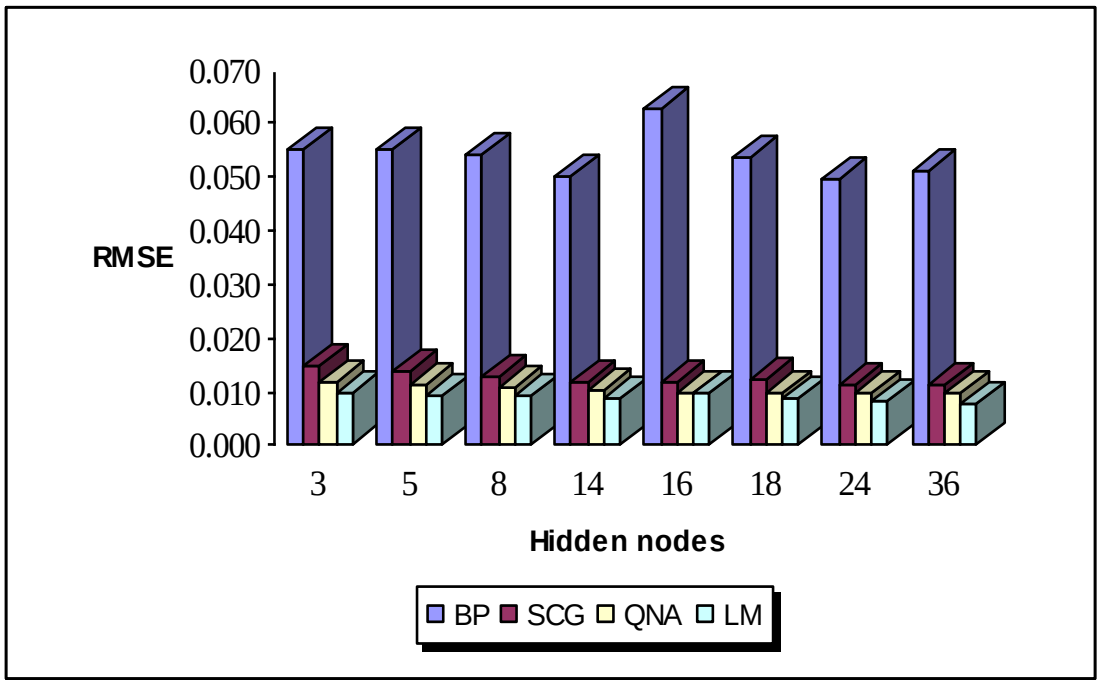


Figure 5.9: Architecture variation: Gas furnace time series training performance for different training algorithms with 4 inputs and 2 output network

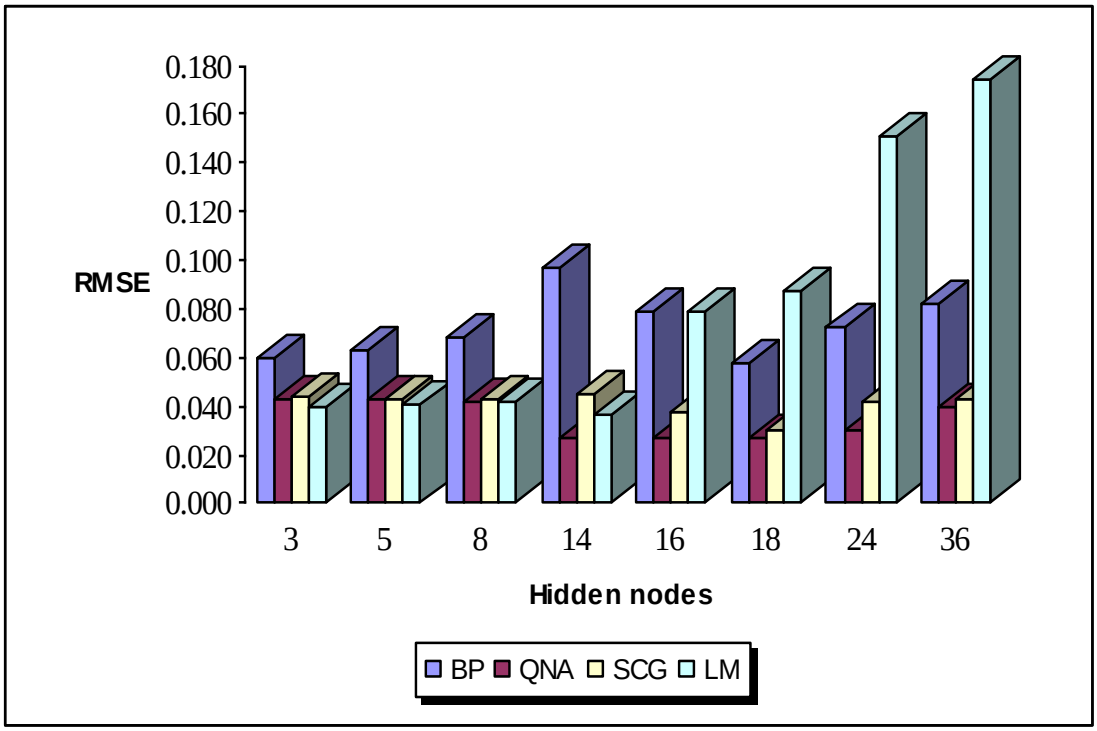


Figure 5.10: Architecture variation: Gas furnace time series generalization performance for different learning algorithms with 4 inputs and 2 output network

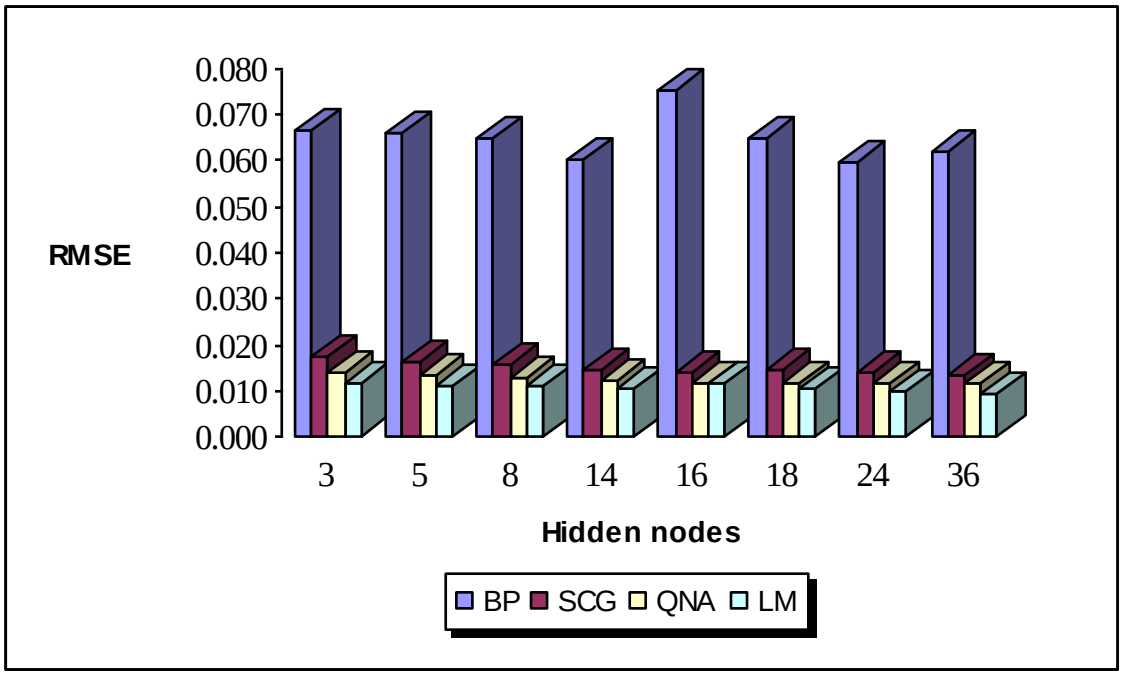


Figure 5.11: Architecture variation: Gas furnace time series training performance for different training algorithms with 3 inputs and 2 output network

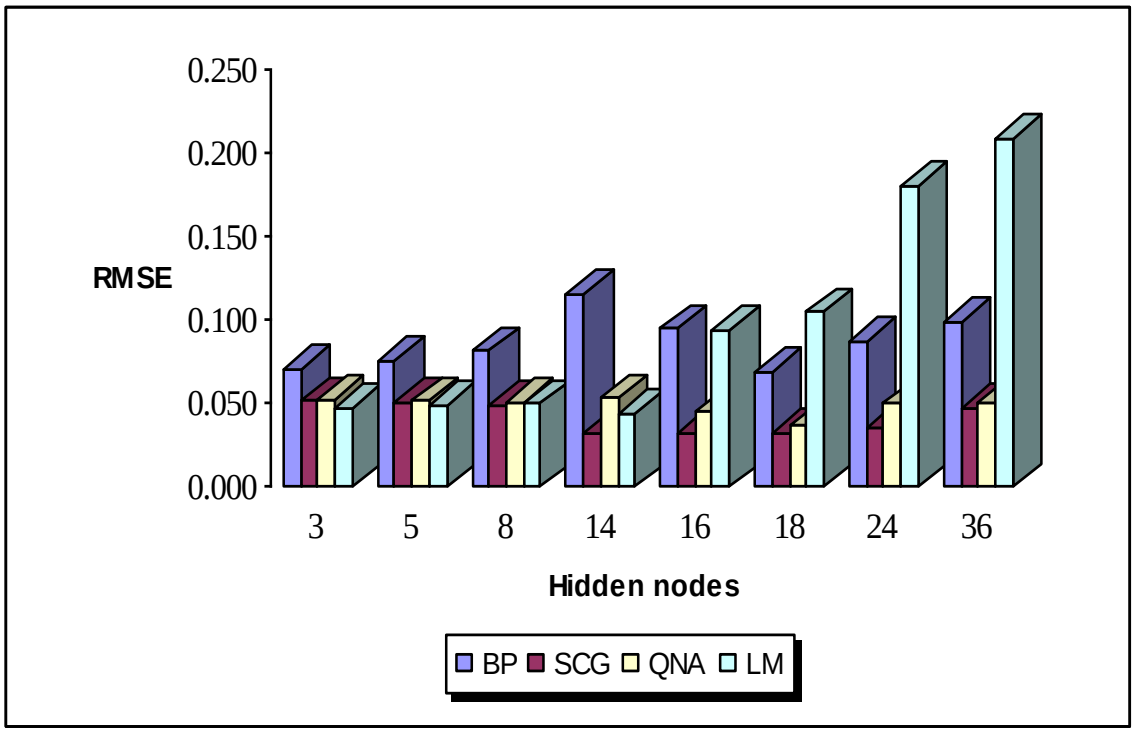


Figure 5.12: Architecture variation: Gas furnace time series generalization performance for different learning algorithms with 3 inputs and 2 output network

5.3.1.3 ANN- Results Discussion:

This subsection includes evaluation and summarization of the experimentations results mentioned in subsections 5.3.1.1 and 5.3.1.2.

For Mackey-glass series (tables 5.3 - 5.5), all the four learning algorithms tend to generalize well (i.e. test set RMSE decreased) as the hidden neurons were increased. The run time also increases for the four learning algorithms as the number of hidden nodes increase. LM showed the fastest convergence regardless of architecture. As an example (in table 5.3), the LM gave the lowest generalization RMSE of 0.0010 with 36 hidden neurons. However, the run time of LM algorithm is the longest in comparison with the other learning algorithms. On the other hand, for every learning algorithm with the same number of hidden nodes and same number of inputs, the RMSE and the run time will increase as the number of outputs increases. Also, the RMSE will increase and the run time will decrease for every learning algorithm as the number of inputs decrease using the same number of hidden nodes and outputs. This is true for Mackey glass and Gas furnace time series.

For Gas furnace series (as shown in tables 5.6 - 5.8), the generalization performance were entirely different for the different learning algorithms. Better generalization does not depend on increasing the hidden neurons. For example (in table 5.6), BP gave the best generalization RMSE of 0.0479 with 18 hidden neurons. RMSE for SCG, QNA and LM were 0.0206 (16

neurons), 0.0235 (18 neurons) and 0.0281 (14 neurons), respectively. However, increasing the number of hidden nodes will cause increasing in execution time for every learning algorithm, and the LM algorithm will have the longest execution time. In spite of execution time, LM performed well for Mackey-glass series. For gas furnace SCG algorithm performed better. However, the speed of convergence of LM in all cases is worth noting.

In general, the training speed and generalization performance of an ANN is totally dependant on the learning algorithm and its parameters, architecture, node transfer function, initial weights, and the data sets used for training and testing. From the above discussion it is clear that the selection of the topology of a network and the best learning algorithm and its parameters is a tedious task for designing an optimal ANN. Evolutionary algorithm is an adaptive search technique based on the principles and mechanisms of natural selection and survival of the fittest from natural evolution. The interest in evolutionary search procedures for designing neural network topology has been growing in recent years as they can evolve towards the optimal architecture without outside interference, thus eliminating the tedious trial and error work of manually finding an optimal network.

5.3.2 MLEANN: Experimentation and Simulation Results:

In this subsection we explore the performance of the MLEANN approach. We applied this MLEANN approach to the two-time series prediction problems discussed before. We used the Neurosolution and NeuroGenetic optimizer toolboxes in training and optimizing processes. For performance comparison, we used the same set of training and test data that were used for experimentations with conventional design of neural networks. We used the same feed-forward neural network with one hidden layer for the two time series. The number of hidden neurons was varied (from 3 to 36) as indicated in table (5.1). For performance evaluation, the parameters used in this experiment were set to be the same for the two problems. Fitness value is calculated based on the RMSE achieved on the test set. In this experiment, we have considered the best-evolved neural network as the best individual of the last generation. As the learning process is evolved separately, user has the option to pick the best neural network (e.g. less RMSE, fast convergence, short run time, or small architecture size, etc.) among the four learning algorithms. All the genotypes were represented using binary coding and the initial populations were randomly generated based on the parameters shown in table (5.1). All networks with different architectures were trained for an identical number of stochastic updates (500 epochs) using the same four learning algorithms. The parameter settings, which were evolved for the different learning algorithms, are found in table (5.2). The experiments were repeated three times and the worst RMSE values are reported.

5.3.2.1 MLEANN: Simulation Results:

Tables (5.9 – 5.14) display empirical values of RMSE on test data for the two time series problems using the meta-learning technique with different architectures. For comparison purposes, test set RMSE values using conventional design techniques are also presented in these tables (adapted from tables 5.3 – 5.8).

Table 5.9: Performance comparison between MLEANN and ANN for Mackey-glass time series with different architectures (four inputs / one output)

Time series	Learning algorithm	MLEANN			ANN	
		RMSE		Architecture	RMSE	Architecture
		Train	Test		Test	
Mackey Glass	BP	0.0173	0.0175	**3:3T:1Li**	0.0993	4 **:** 3T **:** 1Li
		0.0168	0.0170	**3:4T:1Li**	0.0965	4 **:** 5T **:** 1Li
		0.0075	0.0080	**3:10T:1Li**	0.0454	4 **:**24T **:**1Li
		0.0067	0.0071[a]	**3:17T:1Li**	0.0403	4 **:**36T **:**1Li
	SCG	0.0061	0.0065	**3:3T:1Li**	0.0095	4 **:** 3T **:** 1Li
		0.0057	0.0058	**3:5T:1Li**	0.0084	4 **:** 5T **:** 1Li
		0.0036	0.0038	**3:11T:1Li**	0.0055	4 **:**24T **:**1Li
		0.0032	0.0034[a]	**3:19T:1Li**	0.0049	4 **:**36T **:**1Li
	QNA	0.0062	0.0060	**3:3T:1Li**	0.0075	4 **:** 3T **:** 1Li
		0.0050	0.0049	**3:5T:1Li**	0.0062	4 **:** 5T **:** 1Li
		0.0028	0.0030	**3:10T:1Li**	0.0039	4 **:**24T **:**1Li
		0.0022	0.0027[a]	**3:17T:1Li**	0.0034	4 **:**36T **:**1Li
	LM	0.0025	0.0027	**3:3T:1Li**	0.0060	4 **:** 3T **:** 1Li
		0.0019	0.0019	**3:4T:1Li**	0.0043	4 **:** 5T **:** 1Li
		0.0005	0.0005	**3:11T:1Li**	0.0012	4 **:**24T **:**1Li
		0.0004	*0.0004[a]	**3:18T:1Li**	0.0010	4 **:**36T **:**1Li

a : ***Lowest RMSE in each algorithm***
*** :** ***Lowest RMSE in all the algorithms***

Table 5.10: Performance comparison between MLEANN and ANN for Mackey-glass time series with different architectures (four inputs / two outputs)

Time series	Learning algorithm	EANN			ANN	
		RMSE		Architecture	RMSE	Architecture
		Train	Test		Test	
	BP	0.0207	0.0210	**3:3T:2Li**	0.1192	4 **:** 3T **:** 2Li
		0.0201	0.0204	**3:5T:2Li**	0.1158	4 **:** 5 T**:** 2 Li
		0.0091	0.0096	**3:11T:2Li**	0.0545	4 **:**24T **:**2 Li
		0.0081	0.0085[a]	**3:18T:2Li**	0.0484	4 **:**36T **:**2 Li
	SCG	0.0074	0.0079	**3:3T:2Li**	0.0114	4 **:** 3T **:** 2Li

Mackey Glass		0.0068	0.0070	**3:5T:2Li**	0.0101	4 **:** 5 T**:** 2 Li
		0.0044	0.0045	**3:13T:2Li**	0.0066	4 **:**24T **:**2 Li
		0.0040	0.0041[a]	**3:20T:2Li**	0.0059	4 **:**36T **:**2 Li
	QNA	0.0072	0.0071	**3:3T:2Li**	0.0090	4 **:** 3T **:** 2Li
		0.0061	0.0059	**3:5T:2Li**	0.0074	4 **:** 5 T**:** 2 Li
		0.0034	0.0037	**3:11T:2Li**	0.0047	4 **:**24T **:**2 Li
		0.0030	0.0033[a]	**3:18T:2Li**	0.0041	4 **:**36T **:**2 Li
	LM	0.0027	0.0032	**3:3T:2Li**	0.0072	4 **:** 3T **:** 2Li
		0.0021	0.0023	**3:5T:2Li**	0.0052	4 **:** 5 T**:** 2 Li
		0.0006	0.0006	**3:12T:2Li**	0.0014	4 **:**24T **:**2 Li
		0.0005	[*]0.0005[a]	**3:19T:2Li**	0.0012	4 **:**36T **:**2 Li

a : ***Lowest RMSE in each algorithm***

*** :** ***Lowest RMSE in all the algorithms***

Table 5.11: Performance comparison between MLEANN and ANN for Mackey-glass time series with different architectures (three inputs / two outputs)

Time series	**Learning algorithm**	**EANN**			**ANN**	
		RMSE		**Architecture**	**RMSE**	**Architecture**
		Train	**Test**		**Test**	
Mackey Glass	**BP**	0.0249	0.0252	**3:3T:2Li**	0.1430	3**:** 3T **:** 2Li
		0.0241	0.0245	**3:4T:2Li**	0.1390	3**:** 5T **:** 2 Li
		0.0110	0.0115	**3:9T:2Li**	0.0654	3**:**24T **:**2 Li
		0.0100	0.0102[a]	**3:16T:2Li**	0.0580	3**:**36T **:**2 Li
	SCG	0.0090	0.0094	**3:3T:2Li**	0.0137	3**:** 3T **:** 2Li
		0.0081	0.0083	**3:5T:2Li**	0.0121	3**:** 5T **:** 2 Li
		0.0052	0.0054	**3:12T:2Li**	0.0079	3**:**24T **:**2 Li
		0.0047	0.0049[a]	**3:19T:2Li**	0.0071	3**:**36T **:**2 Li
	QNA	0.0087	0.0086	**3:3T:2Li**	0.0108	3**:** 3T **:** 2Li
		0.0070	0.0071	**3:5T:2Li**	0.0089	3**:** 5T **:** 2 Li
		0.0041	0.0044	**3:10T:2Li**	0.0056	3**:**24T **:**2 Li
		0.0036	0.0039[a]	**3:17T:2Li**	0.0049	3**:**36T **:**2 Li
	LM	0.0034	0.0038	**3:3T:2Li**	0.0086	3**:** 3T **:** 2Li
		0.0026	0.0028	**3:4T:2Li**	0.0062	3**:** 5T **:** 2 Li
		0.0008	0.0008	**3:11T:2Li**	0.0017	3**:**24T **:**2 Li
		0.0006	[*]0.0006[a]	**3:18T:2Li**	0.0014	3**:**36T **:**2 Li

a : ***Lowest RMSE in each algorithm***

*** :** ***Lowest RMSE in all the algorithms***

Table 5.12: Performance comparison between MLEANN and ANN for Gas furnace time series with different architectures (four inputs / one output)

Time series	Learning algorithm	EANN			ANN	
		RMSE		Architecture	RMSE	Architecture
		Train	Test		Test	
Gas furnace	BP	0.0121	0.0232	**3:3T:1Li**	0.0496	4 **:** 3T **:** 1Li
		0.0117	0.0246	**3:4T:1Li**	0.0527	4 **:** 5T **:** 1 Li
		0.0100	0.0224[a]	**3:9T:1Li**	0.0479	4 **:**18T **:**1 Li
		0.0108	0.0320	**3:19T:1Li**	0.0686	4 **:**36T **:**1 Li
	SCG	0.0112	0.0191	**3:3T:1Li**	0.0333	4 **:** 3T **:** 1Li
		0.0106	0.0187	**3:4T:1Li**	0.0326	4 **:** 5T **:** 1 Li
		0.0069	*0.0131[a]	**3:11T:1Li**	0.0206	4 **:**16T **:**1 Li
		0.0095	0.0167	**3:22T:1Li**	0.0308	4 **:**36T **:**1 Li
	QNA	0.0098	0.0196	**3:3T:1Li**	0.0340	4 **:** 3T **:** 1Li
		0.0103	0.0190	**3:4T:1Li**	0.0332	4 **:** 5T **:** 1 Li
		0.0072	0.0160[a]	**3:10T:1Li**	0.0235	4 **:**18T **:**1 Li
		0.0085	0.0175	**3:21T:1Li**	0.0329	4 **:**36T **:**1 Li
	LM	0.0115	0.0180	**3:3T:1Li**	0.0305	4 **:** 3T **:** 1Li
		0.0111	0.0184	**3:4T:1Li**	0.0313	4 **:** 5T **:** 1 Li
		0.0075	0.0139[a]	**3:9T:1Li**	0.0281	4 **:**14T **:**1 Li
		0.0083	0.0623	**3:20T:1Li**	0.1342	4 **:**36T **:**1 Li

a : ***Lowest RMSE in each algorithm***
**** : Lowest RMSE in all the algorithms***

Table 5.13: Performance comparison between MLEANN and ANN for Gas furnace time series with different architectures (four inputs / two outputs)

Time series	Learning algorithm	EANN			ANN	
		RMSE		Architecture	RMSE	Architecture
		Train	Test		Test	
	BP	0.0145	0.0278	**3:3T:2Li**	0.0596	4 **:** 3T **:** 2Li
		0.0140	0.0295	**3:5T:2Li**	0.0632	4 **:** 5T **:** 2 Li
		0.0121	0.0269[a]	**3:10T:2Li**	0.0575	4 **:**18T **:**2 Li
		0.0130	0.0385	**3:20T:2Li**	0.0823	4 **:**36T **:**2 Li
	SCG	0.0146	0.0248	**3:3T:2Li**	0.0433	4 **:** 3T **:** 2Li

Gas furnace		0.0138	0.0243	**3:4T:2Li**	0.0424	4 **:** 5T **:** 2 Li
		0.0090	*0.0171[a]	**3:12T:2Li**	0.0268	4 **:**16T **:**2 Li
		0.0124	0.0217	**3:23T:2Li**	0.0401	4 **:**36T **:**2 Li
	QNA	0.0127	0.0255	**3:3T:2Li**	0.0442	4 **:** 3T **:** 2Li
		0.0134	0.0247	**3:5T:2Li**	0.0431	4 **:** 5T **:** 2 Li
		0.0094	0.0208[a]	**3:11T:2Li**	0.0306	4 **:**18T **:**2 Li
		0.0111	0.0228	**3:22T:2Li**	0.0427	4 **:**36T **:**2 Li
	LM	0.015	0.0234	**3:3T:2Li**	0.0397	4 **:** 3T **:** 2Li
		0.0144	0.0239	**3:4T:2Li**	0.0407	4 **:** 5T **:** 2 Li
		0.0098	0.0181[a]	**3:10T:2Li**	0.0366	4 **:**14T **:**2 Li
		0.0108	0.0810	**3:20T:2Li**	0.1744	4 **:**36T **:**2 Li

a : ***Lowest RMSE in each algorithm***

**** : Lowest RMSE in all the algorithms***

Table 5.14: Performance comparison between MLEANN and ANN for Gas furnace time series with different architectures (three inputs / two outputs)

Time series	**Learning algorithm**	**EANN**			**ANN**	
		RMSE		**Architecture**	**RMSE**	**Architecture**
		Train	**Test**		**Test**	
Gas furnace	**BP**	0.0174	0.0334	**3:3T:2Li**	0.0715	3**:** 3T **:** 2Li
		0.0169	0.0355	**3:4T:2Li**	0.0759	3**:** 5T **:** 2 Li
		0.0144	0.0322[a]	**3:8T:2Li**	0.0689	3**:**18T **:**2 Li
		0.0155	0.0461	**3:18T:2Li**	0.0987	3**:**36T **:**2 Li
	SCG	0.0175	0.0298	**3:3T:2Li**	0.0520	3**:** 3T **:** 2Li
		0.0165	0.0292	**3:4T:2Li**	0.0509	3**:** 5T **:** 2 Li
		0.0108	*0.0205[a]	**3:10T:2Li**	0.0322	3**:**16T **:**2 Li
		0.0148	0.0261	**3:21T:2Li**	0.0481	3**:**36T **:**2 Li
	QNA	0.0153	0.0306	**3:3T:2Li**	0.0530	3**:** 3T **:** 2Li
		0.0161	0.0297	**3:4T:2Li**	0.0518	3**:** 5T **:** 2 Li
		0.0112	0.0250[a]	**3:9T:2Li**	0.0367	3**:**18T **:**2 Li
		0.0133	0.0273	**3:20T:2Li**	0.0513	3**:**36T **:**2 Li
	LM	0.0179	0.0281	**3:3T:2Li**	0.0476	3**:** 3T **:** 2Li
		0.0173	0.0287	**3:4T:2Li**	0.0488	3**:** 5T **:** 2 Li
		0.0117	0.0217[a]	**3:8T:2Li**	0.0439	3**:**14T **:**2 Li
		0.0129	0.0972	**3:19T:2Li**	0.2093	3**:**36T **:**2 Li

a : ***Lowest RMSE in each algorithm***

**** : Lowest RMSE in all the algorithms***

Figures (5.13 – 5.18) show the test set RMSE for the two time series problems using the meta-learning technique with different architectures.

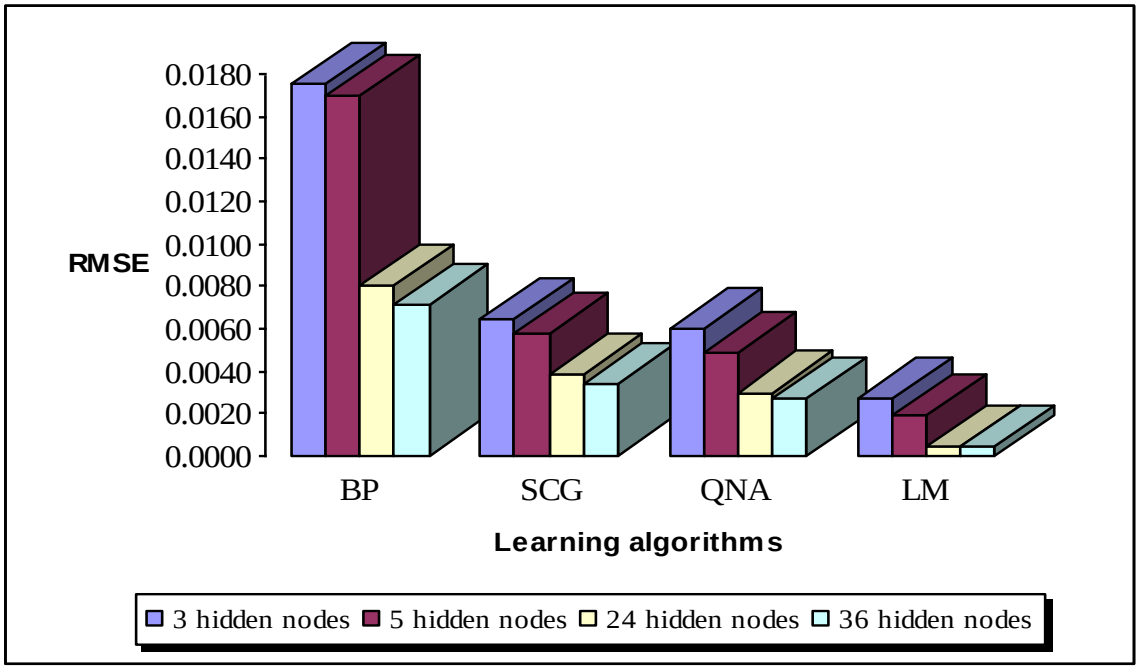


Figure 5.13: test set RMSE for Mackey glass using meta-learning technique (for architectures with 4 inputs – 1 output)

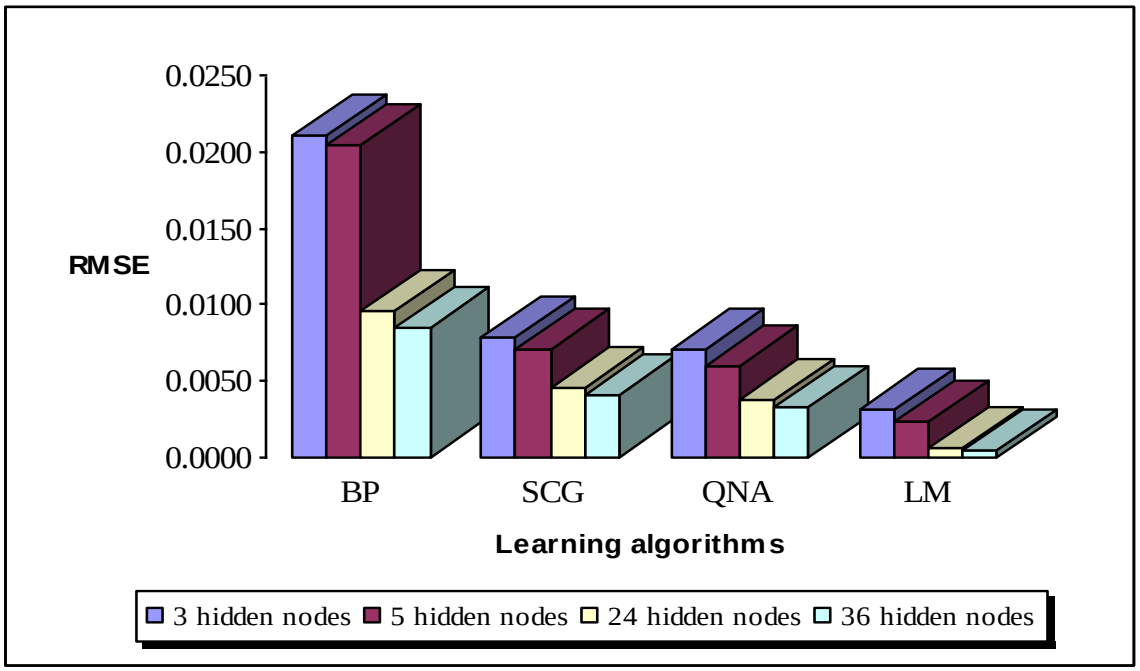


Figure 5.14: test set RMSE for Mackey glass using meta-learning technique (for architectures with 4 inputs – 2 outputs)

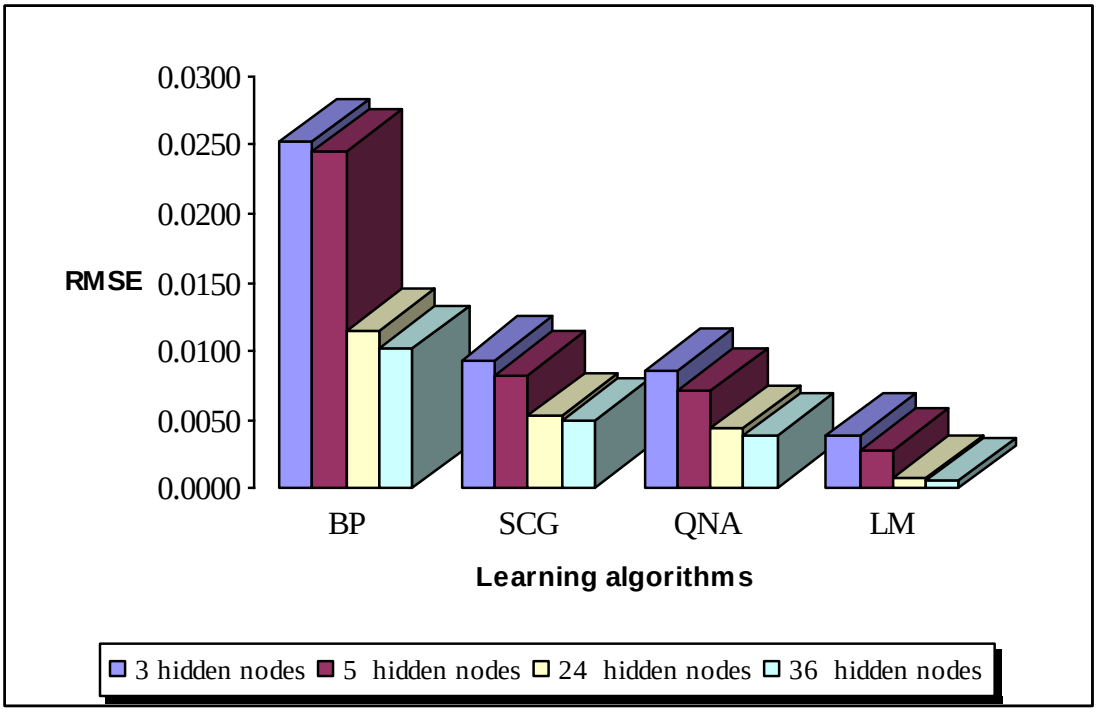


Figure 5.15: test set RMSE for Mackey glass using meta-learning technique (for architectures with 3 inputs – 2 outputs)

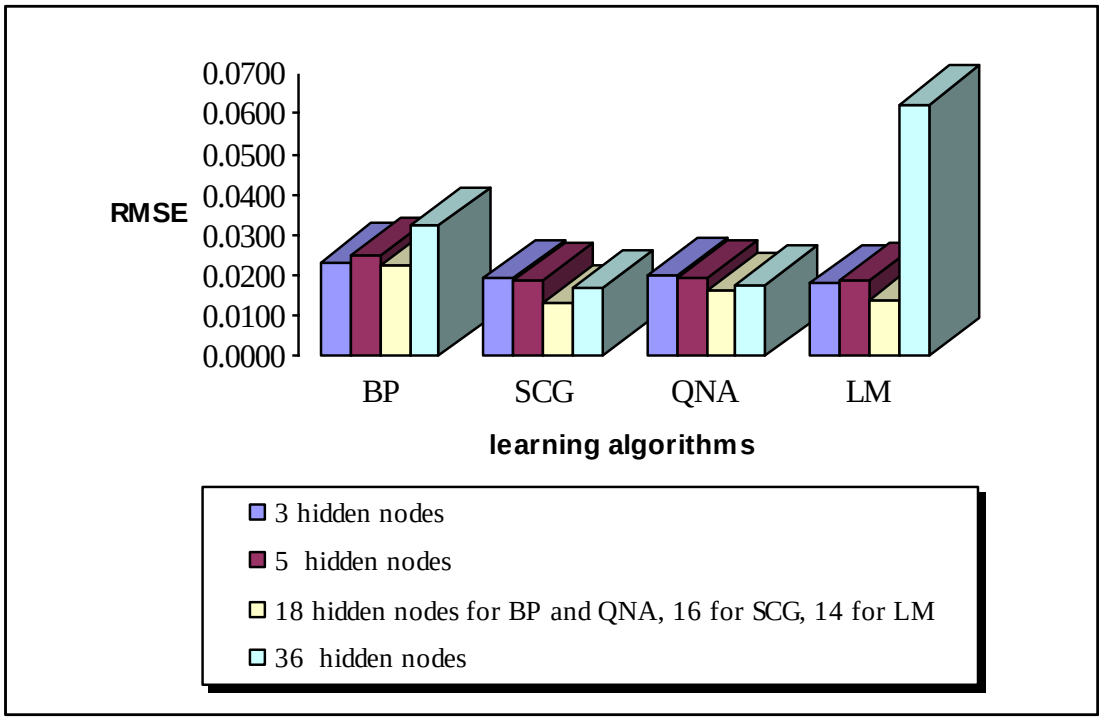


Figure 5.16: test set RMSE for Gas furnace using meta-learning technique (for architectures with 4 inputs – 1 output)

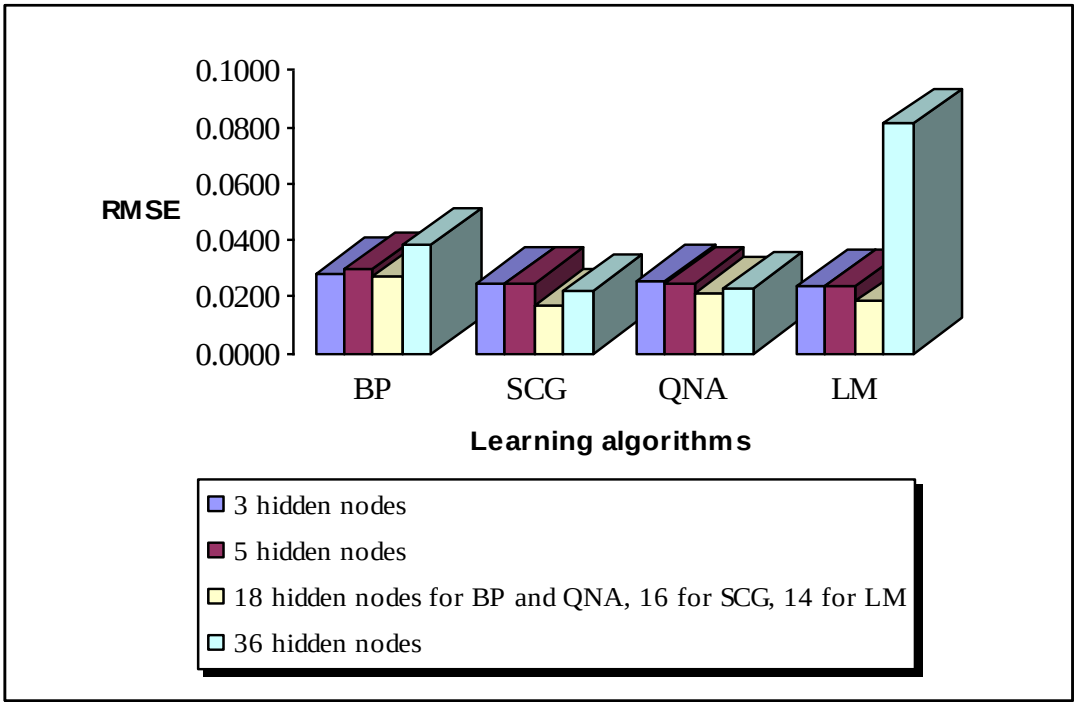


Figure 5.17: test set RMSE for Gas furnace using meta-learning technique (for architectures with 4 inputs –2 outputs)

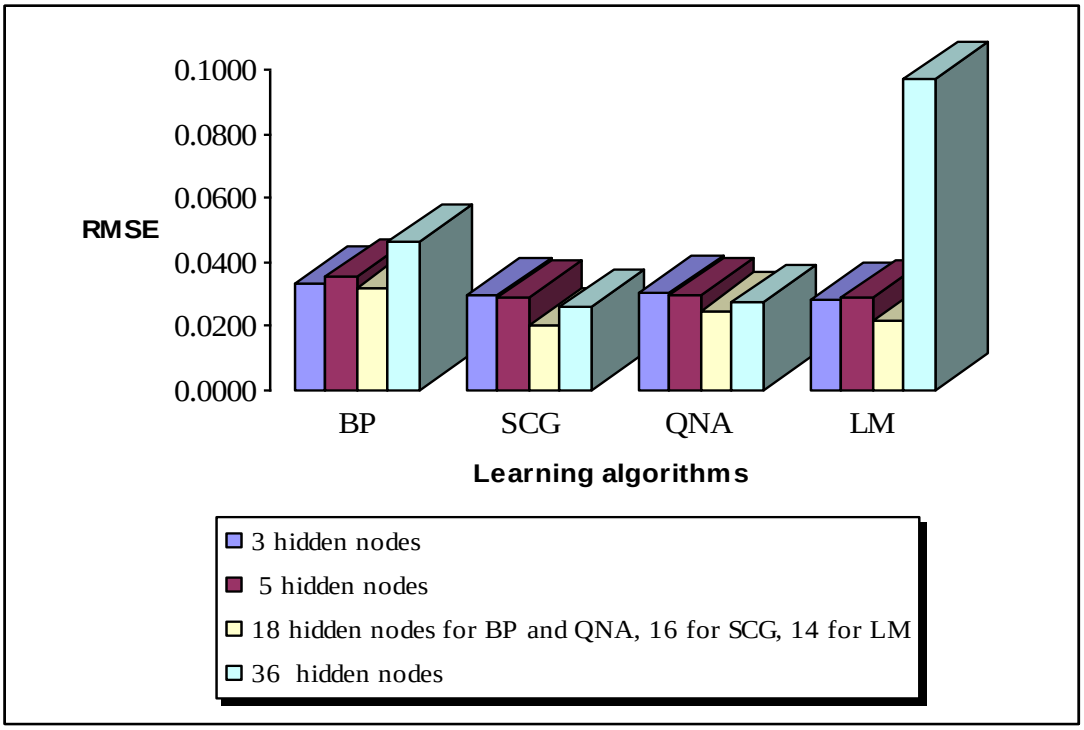


Figure 5.18: test set RMSE for Gas furnace using meta-learning technique (for architectures with 3 inputs –2 outputs)

Tables (5.15, 5.16) illustrate the run times of the MLEANN for the two time series with different architectures

Table 5.15: Run time comparison of MLEANN for Mackey glass time series with different architectures

Time series	Learning algorithm	Hidden neurons	Run time in minutes		
			4 i/p - 1 o/p	4 i/p - 2 o/p	3 i/p - 2 o/p
Mackey Glass	BP	3	240.96	268.80	246.60
		5	262.80	293.40	269.40
		24	295.80	501.00	305.40
		36	376.20[a]	537.60[a]	400.20[a]
	SCG	3	394.80	440.41	404.04
		5	433.20	483.64	444.08
		24	492.60	834.32	508.59
		36	634.20[a]	906.29[a]	674.66[a]
	QNA	3	417.00	465.18	426.76
		5	457.80	511.11	469.30
		24	522.60	885.13	539.56
		36	673.20[a]	962.02[a]	716.15[a]
	LM	3	470.40	524.75	481.41
		5	516.00	576.08	528.96
		24	588.60	996.92	607.70
		36	*760.20[a]	*1086.35[a]	*808.70[a]

a : ***Maximum run time in each algorithm***

**** : Maximum run time in all the algorithms***

Table 5.16: Run time comparison of MLEANN for Gas furnace time series with different architectures

Time series	Learning algorithm	Hidden neurons	Run time in minutes		
			4 i/p -1 o/p	4 i/p -2 o/p	3 i/p -2 o/p
Gas furnace	BP	3	88.20	106.80	91.20
		5	90.60	111.00	100.20
		18	102.00	113.40	108.60
		36	120.00[a]	141.60[a]	132.00[a]
	SCG	3	166.20	201.25	171.85

		5	174.00	213.18	192.44
		16	204.00	226.80	217.20
		36	229.20[a]	270.46[a]	252.12[a]
	QNA	**3**	174.00	210.69	179.92
		5	184.20	225.68	203.72
		18	216.00	240.14	229.98
		36	243.00[a]	286.74[a]	267.30[a]
	LM	**3**	179.40	217.23	185.50
		5	192.00	235.23	212.34
		14	228.60	254.15	243.39
		36	*251.40[a]	*296.65[a]	*276.54[a]

a : ***Maximum run time in each algorithm***
**** :*** ***Maximum run time in all the algorithms***

Figures (5.19 – 5.24) show the run times of the MLEANN for the two time series with different architectures

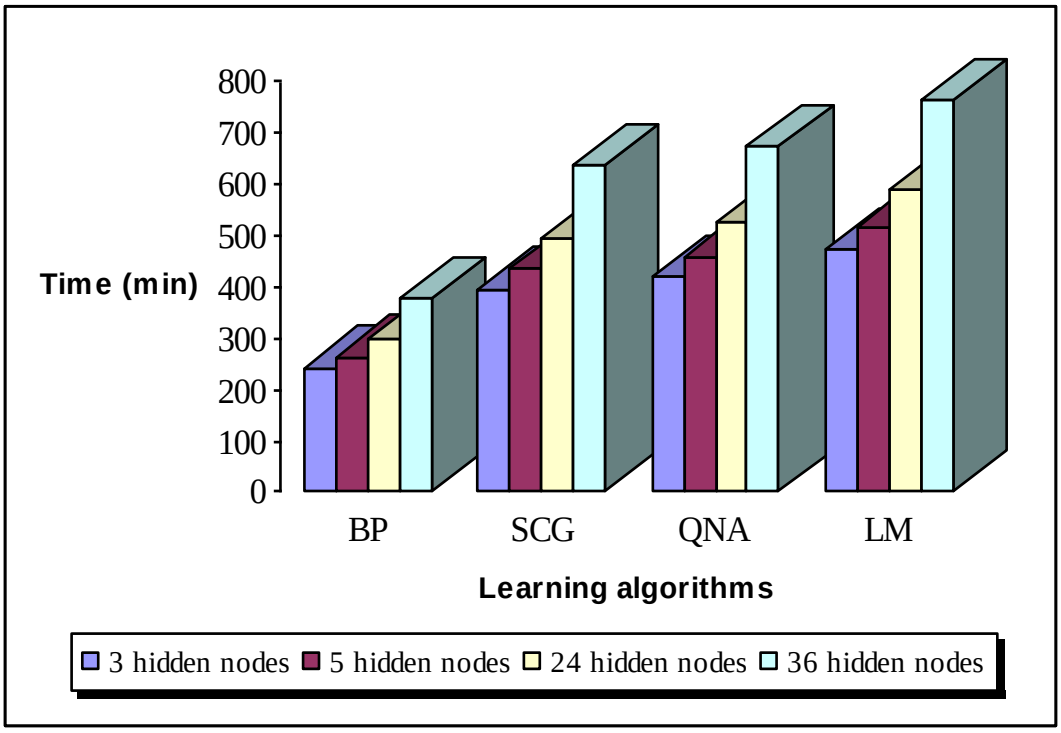


Figure 5.19: Run time of the MLEANN for Mackey glass with different architectures (4 inputs – 1 output)

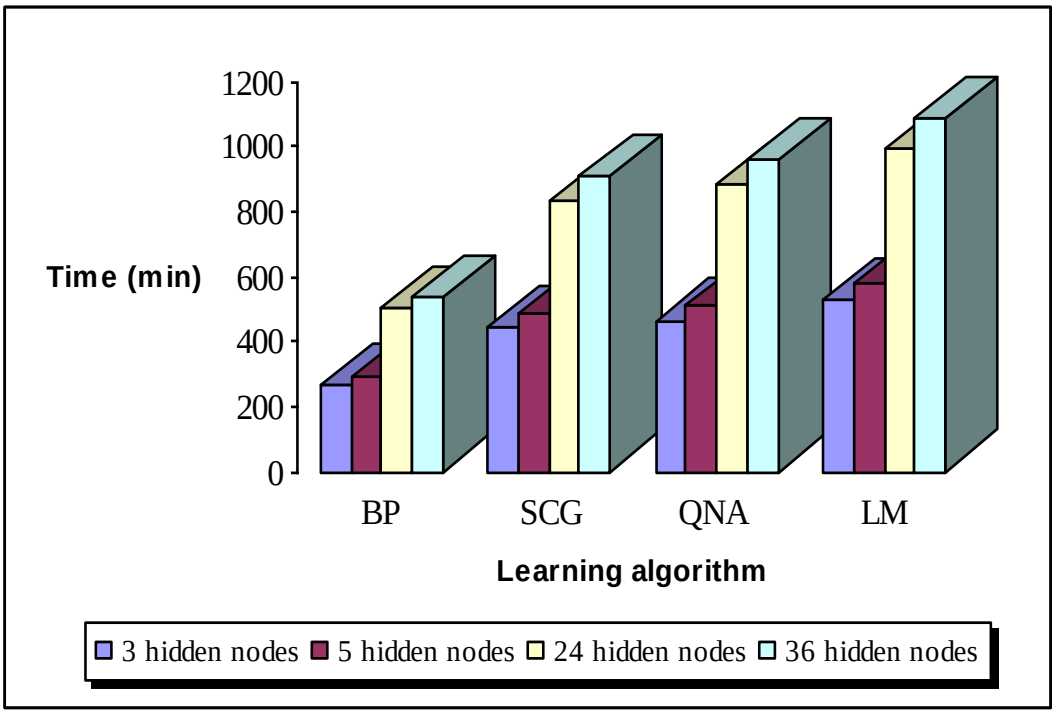


Figure 5.20: Run time of the MLEANN for Mackey glass with different architectures (4 inputs –2 outputs)

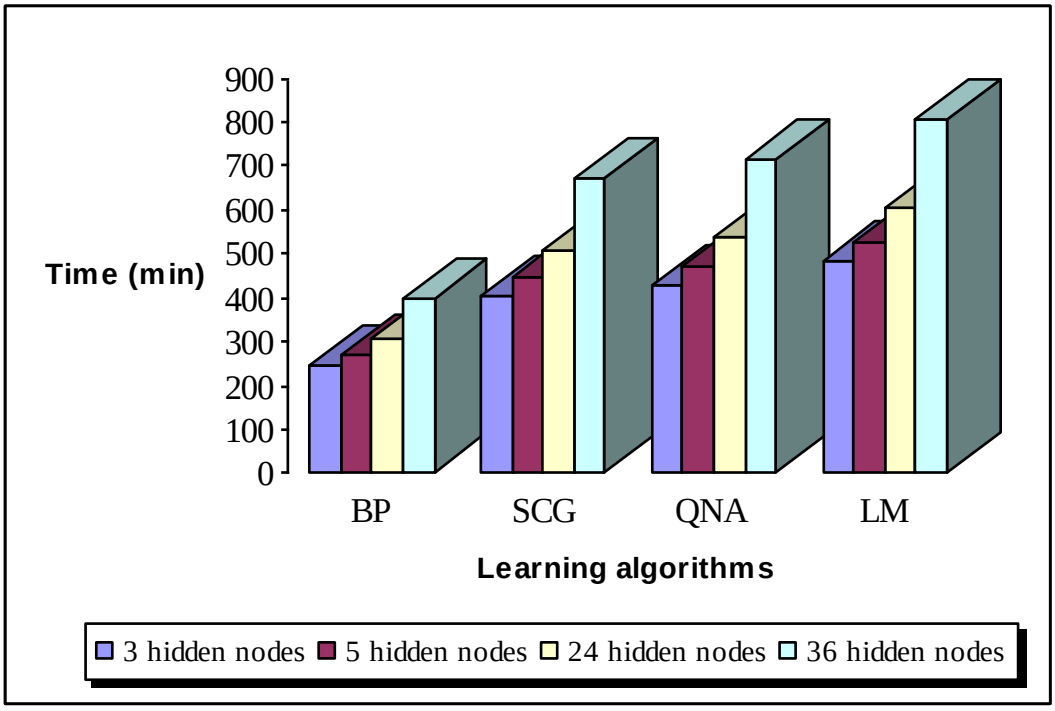


Figure 5.21: Run time of the MLEANN for Mackey glass with different architectures (3 inputs –2 outputs)

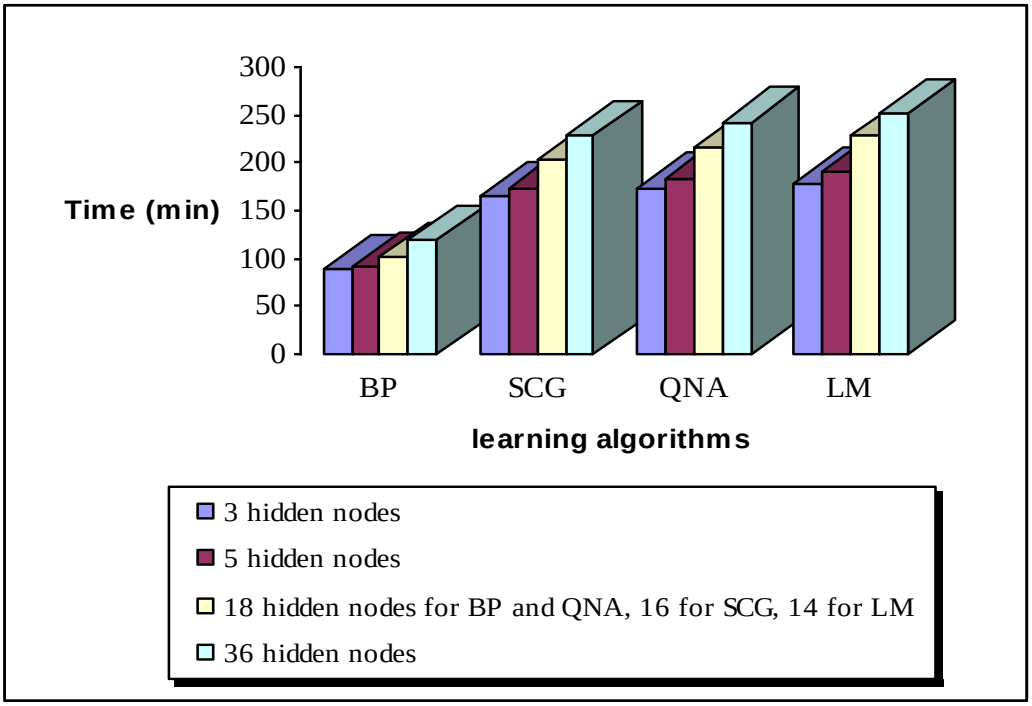


Figure 5.22: Run time of the MLEANN for Gas furnace with different architectures (4 inputs – 1 output)

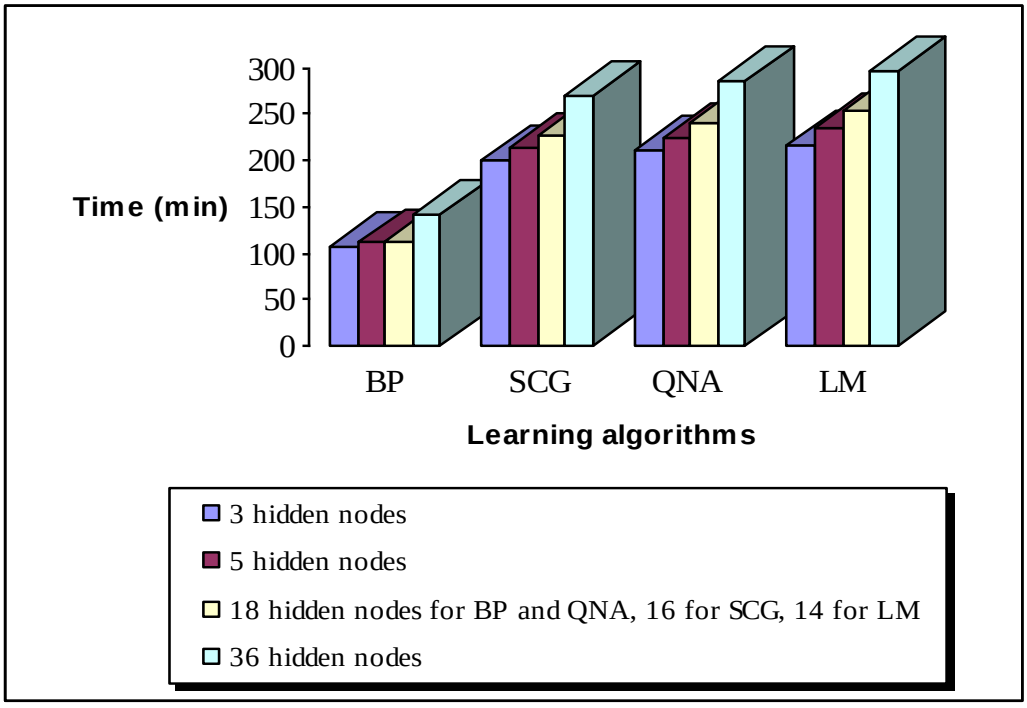


Figure 5.23: Run time of the MLEANN for Gas furnace with different architectures (4 inputs –2 outputs)

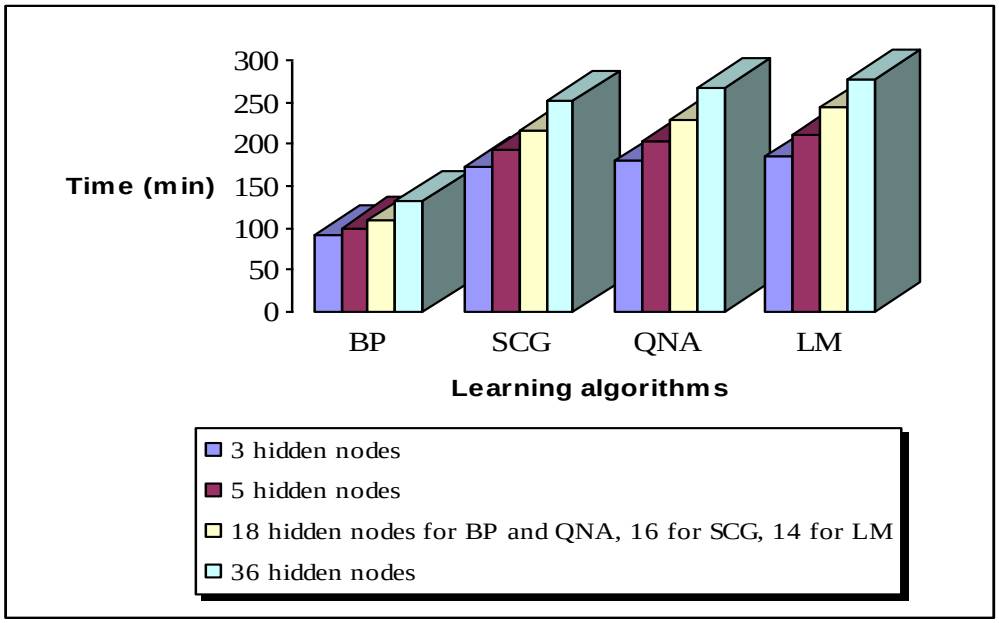


Figure 5.24: Run time of the MLEANN for Gas furnace with different architectures (3 inputs –2 outputs)

Figures (5.25, 5.26) show the test results using 500 epochs BP meta-learning for the two time series (with architecture of 4 inputs and 1 output).

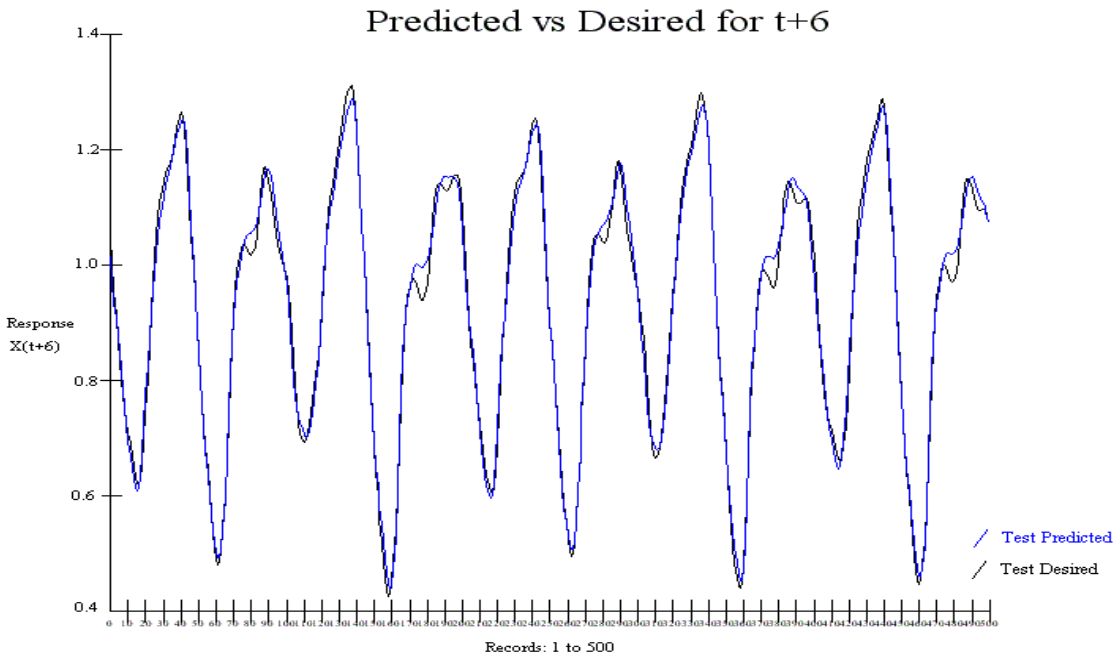


Figure 5.25: Test results using 500 epochs BP meta-learning for Mackey-glass series. (36 hidden nodes)

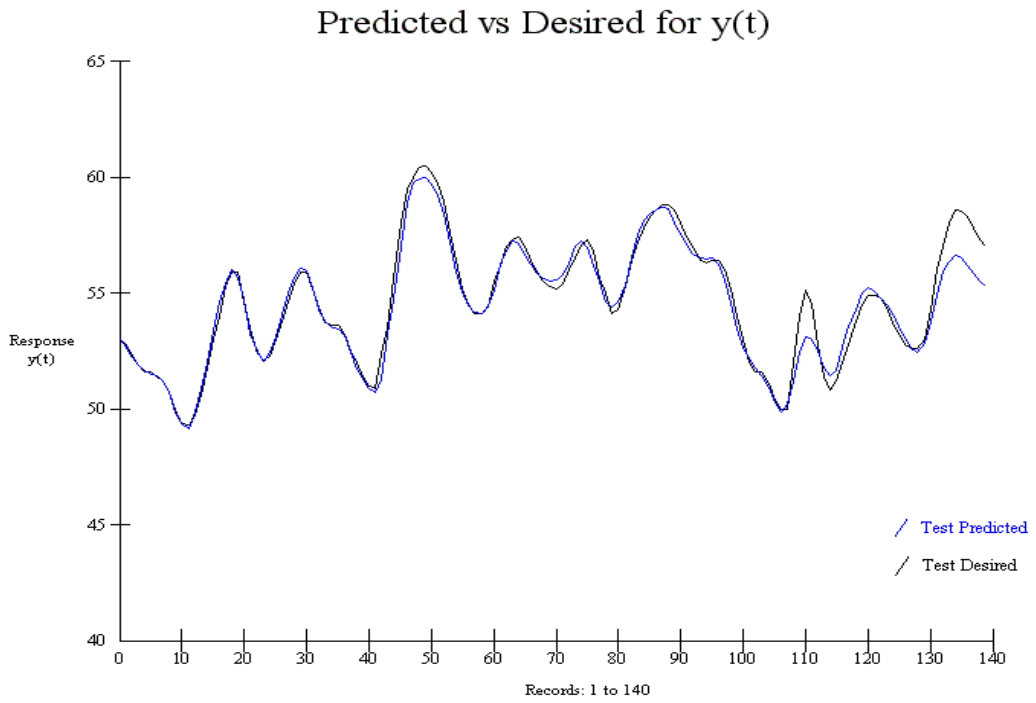


Figure 5.26: Test results using 500 epochs BP meta-learning for gas furnace series (18 hidden nodes)

Convergence of test set RMSE for the two time series is depicted in figures (5.27– 5.28). This is for the architectures with the lowest RMSE.

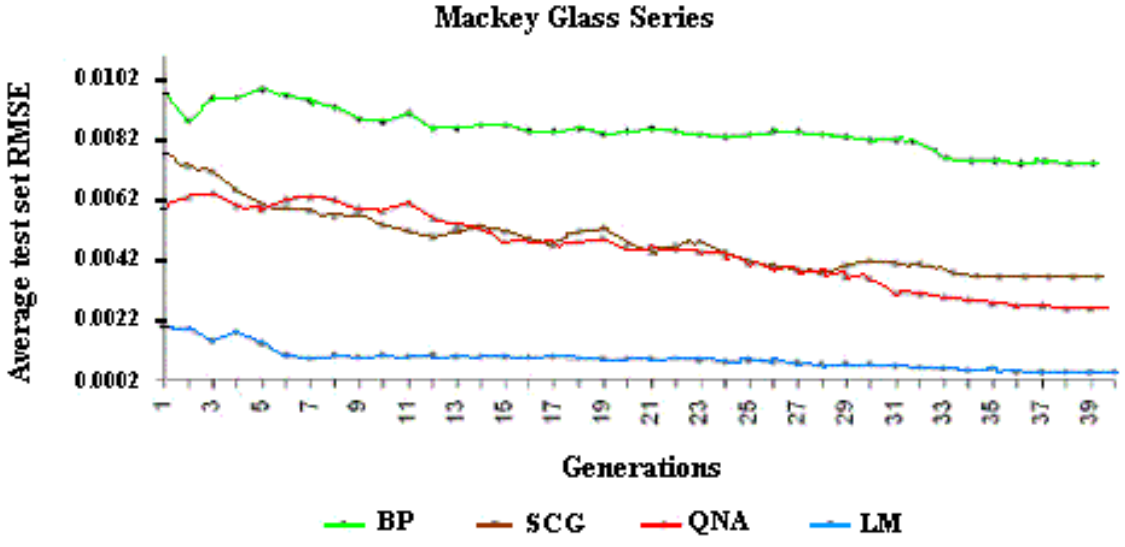


Figure 5.27: Mackey-glass time series: average test set RMSE values during the 40 generations and meta-learning. (4 inputs-36 hidden nodes-1output)

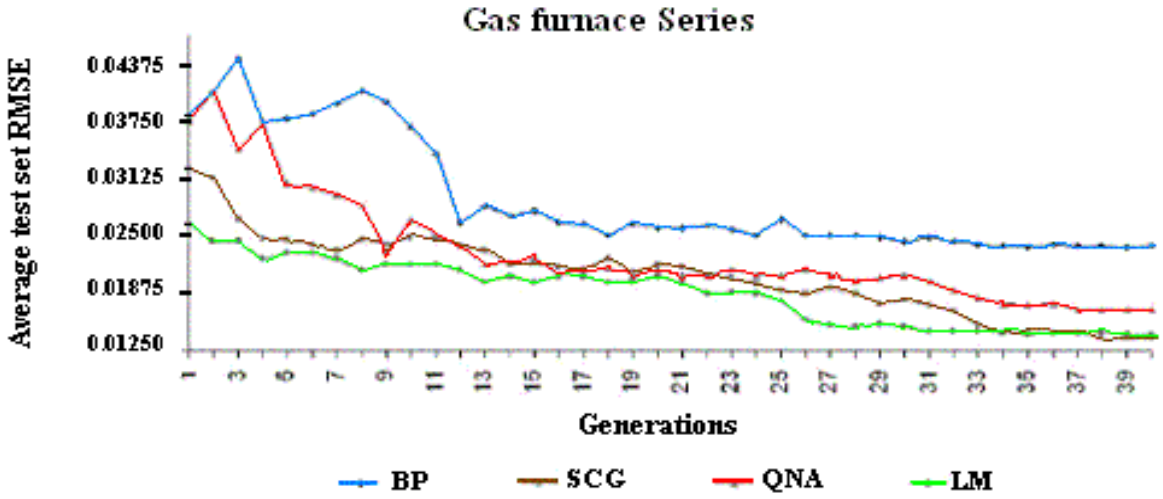


Figure 5.28: Gas furnace time series: average test set RMSE values during the 40 generations and meta-learning. (4 inputs- 1 output - hidden nodes with 18 (BP), 16 (SCG), 18 (QNA), 14 (LM))

5.3.2.2 MLEANN- Results Discussion:

This subsection includes evaluation and summarization of the experimentation results mentioned in subsection 5.3.2.1. Tables (5.9 – 5.14) show comparative performance between MLEANN and a conventional ANN with different architectures for the two time series problems. Performance comparison reveals that EANN design performs more efficiently than conventional ANN design for the two time series.

For Mackey-glass series (figure 5.25), using 500 epochs of BP learning with architecture of 36 hidden nodes (table 5.9), the RMSE on test set was reduced by 82.4% (BP), 30.6% (SCG), 20.6% (QNA) and 60% (LM). At the same time, number of hidden neurons got reduced by approximately 52.8% (BP), 47.2% (SCG), 52.7% (QNA) and 50% for LM. LM algorithm gave the best RMSE error on test set (0.0004) even though it takes long time (760.2 minutes) while the BP algorithm takes the shortest time (376.2 minutes) as shown in table (5.15).

For the gas furnace time series (figure 5.26), using 500 epochs of BP learning with architectures indicated in table 5.12, RMSE on test set was reduced by 53.2% (BP with 18 hidden nodes), 36.4% (SCG with 16 hidden nodes), 31.9% (QNA with 18 hidden nodes) and 50.5% (LM with 14 hidden nodes). Savings in hidden neurons amounted to 50% (BP), 31.3% (SCG), 44.4% (QNA) and 35.7% (LM). SCG training gave the best RMSE value (0.0131) for gas furnace series. To have an empirical comparison, we deliberately terminated the local search after 500 epochs (regardless of early stopping in some cases) for all the training algorithms. In some cases the generalization performance could have been further improved.

As depicted in tables (5.9 - 5.14), our experimentations with small architectures also reveal the efficiency of MLEANN technique. The gas furnace time series could be learned just with 3 or 5 hidden neurons using LM algorithm. LM produced best results with few hidden neurons. However, when the hidden neurons were increased, SCG algorithm marginally preformed better than LM. For Mackey-glass series the results were not that encouraging (using 4 hidden neurons) when compared with the conventional design using 36 hidden neurons. The Mackey-glass series requires more hidden neurons to improve the RMSE values.

In this experiment, the work was mostly concentrated on the evolutionary search of optimal learning algorithms for feed forward neural networks using direct encoding method (fixed chromosome structure) to represent the architecture. As the size of the network increases, the chromosome size grows. Moreover, implementation of crossover is often difficult due to production of non-functional offspring's. Indirect encoding methods overcome the problems with direct encoding although the search of architectures is restricted to layers. Using the cellular configuration as an indirect encoding method to explore the architecture of neural networks is more efficiently. Gutierrez (Gutierrez, Isasi, Molina, Sanchis, & Galvan, 2001) had shown that their cellular automata technique performed better than direct codifications and this is what I used in the following experiment.

5.3.3 MLEANN-CA: Experimentation and Simulation Results:

In this subsection, we test and explore the performance of the proposed MLEANN-CA approach that used the cellular configurations in optimizing networks architectures. According to Gutteriez experiment (Gutierrez, Isasi, Molina, Sanchis, & Galvan, 2001), he applied the evolutionary cellular configurations for designing feed-forward neural network architecture. He used a network with four inputs, 36 hidden nodes, and two outputs. By using the cellular automata technique, the positions of growing and decreasing seeds in the CA grid are codified into the genotype. The length of chromosome is 30, 3 genes for each growing or decreasing seed. The result is an optimized neural network with three inputs, three hidden nodes, and two outputs.

In this experiment, we used the original neural network (4: 36: 2) and the optimized one (3: 3: 2) from Gutteriez experiment. We assumed that the two networks are fully connected as a worst case. We evolved and trained these networks through the Neurosolution and NeuroGenetic optimizer toolboxes using two time series: Mackey-glass and Gas furnace. We used the direct encoding method in training and evolving the networks. We compared the results according to the architecture, RMSE error, and run time. The user has the option to pick the best neural network (e.g. small architecture size, less RMSE, fast convergence, or short run time, etc.) among the four learning algorithms used during the training process.

5.3.3.1 MLEANN-CA: Simulation Results:

Tables (5.17 – 5.18) display empirical values of RMSE on test data using the meta-learning technique for the two time series problems with the network architectures: (4:36:2) and (3:3:2). These tables also include the new optimized architectures generated after applying the meta-learning technique. The results are adapted from tables (5.10, 5.11, 5.13, 5.14). For comparison purposes, test set RMSE values using conventional design techniques are also

presented in these tables (adapted from tables 5.4, 5.5, 5.7, 5.8). In addition, the run times using the meta-learning technique for the two time series are also presented in tables (adapted from tables 5.15- 5.16)

Table 5.17: Performance comparison between MLEANN and ANN for Mackey-glass time series with: original architecture, and the optimized one using cellular configurations

Mackey glass time series					
Learning algorithm	**ANN**		**EANN**		
	Architecture	**RMSE Test**	**Architecture**	**RMSE Test**	**Run time** (minutes)
BP	3:3T:2Li	0.1430	**3:3T:2Li**	0.0252	246.60
	4:36T:2Li	0.0484	**3:18T:2Li**	0.0085	537.60
SCG	3:3T:2Li	0.0137	**3:3T:2Li**	0.0094	404.04
	4:36T:2Li	0.0059	**3:20T:2Li**	0.0041	906.29
QNA	3:3T:2Li	0.0108	**3:3T:2Li**	0.0086	426.76
	4:36T:2Li	0.0041	**3:18T:2Li**	0.0033	962.02
LM	3:3T:2Li	0.0086	**3:3T:2Li**	0.0038	481.41
	4:36T:2Li	0.0012	**3:19T:2Li**	0.0005	1086.35

Table 5.18: Performance comparison between MLEANN and ANN for Gas furnace time series with: original architecture, and the optimized one using cellular configurations

Gas furnace time series					
Learning algorithm	**ANN**		**EANN**		
	Architecture	**RMSE Test**	**Architecture**	**RMSE Test**	**Run time** (minutes)
BP	3:3T:2Li	0.0715	**3:3T:2Li**	0.0334	91.20
	4:36T:2Li	0.0823	**3:20T:2Li**	0.0385	141.60
SCG	3:3T:2Li	0.0520	**3:3T:2Li**	0.0298	171.85
	4:36T:2Li	0.0401	**3:23T:2Li**	0.0217	270.46
QNA	3:3T:2Li	0.0530	**3:3T:2Li**	0.0306	179.92
	4:36T:2Li	0.0427	**3:22T:2Li**	0.0228	286.74
LM	3:3T:2Li	0.0476	**3:3T:2Li**	0.0281	185.50
	4:36T:2Li	0.1744	**3:20T:2Li**	0.0810	296.65

Figures (5.29 – 5.30) show the test set RMSE for the two time series problems using the meta-learning technique with the tested network architectures.

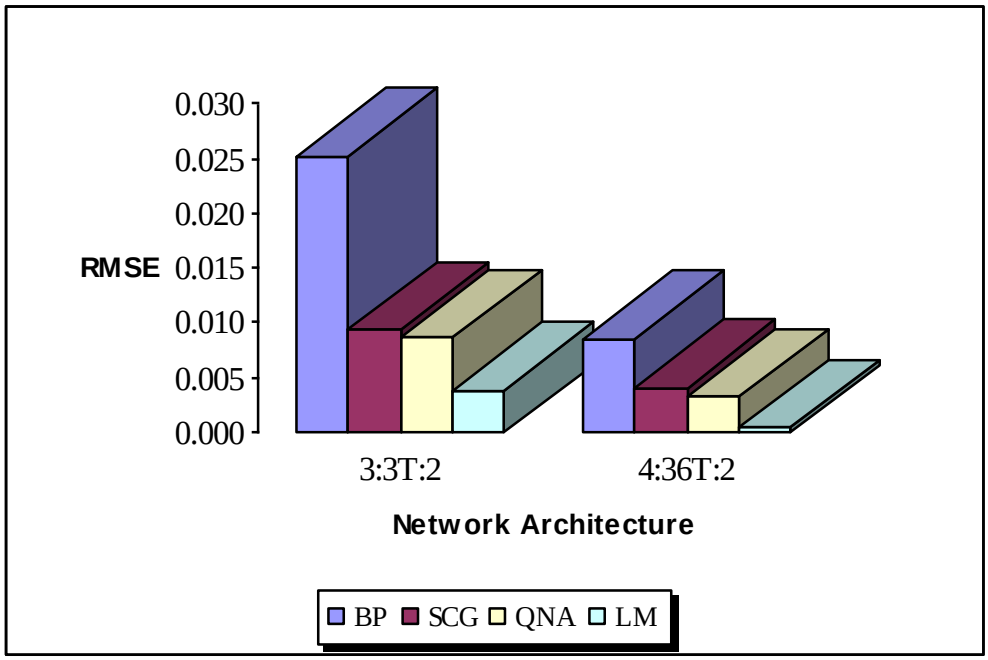


Figure 5.29: test set RMSE for Mackey glass using meta-learning technique with two network architectures: original network (4:36:2), optimized network by cellular (3:3:2)

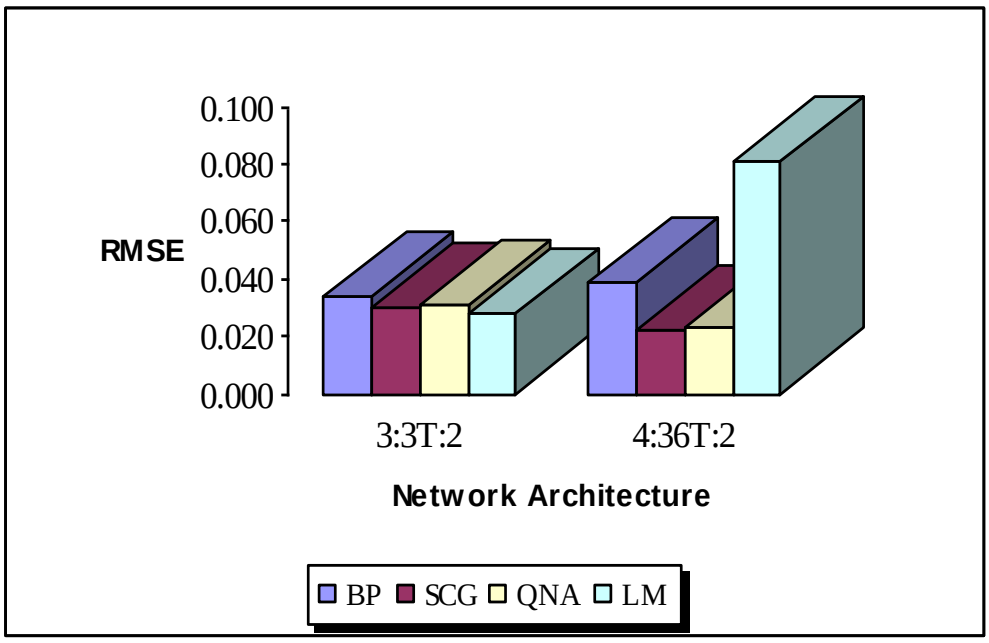


Figure 5.30: test set RMSE for Gas furnace using meta-learning technique with two network architectures: original network (4:36:2), optimized network by cellular (3:3:2)

Figures (5.35 – 5.36) show the run times of the MLEANN for the two time series with the tested network architectures.

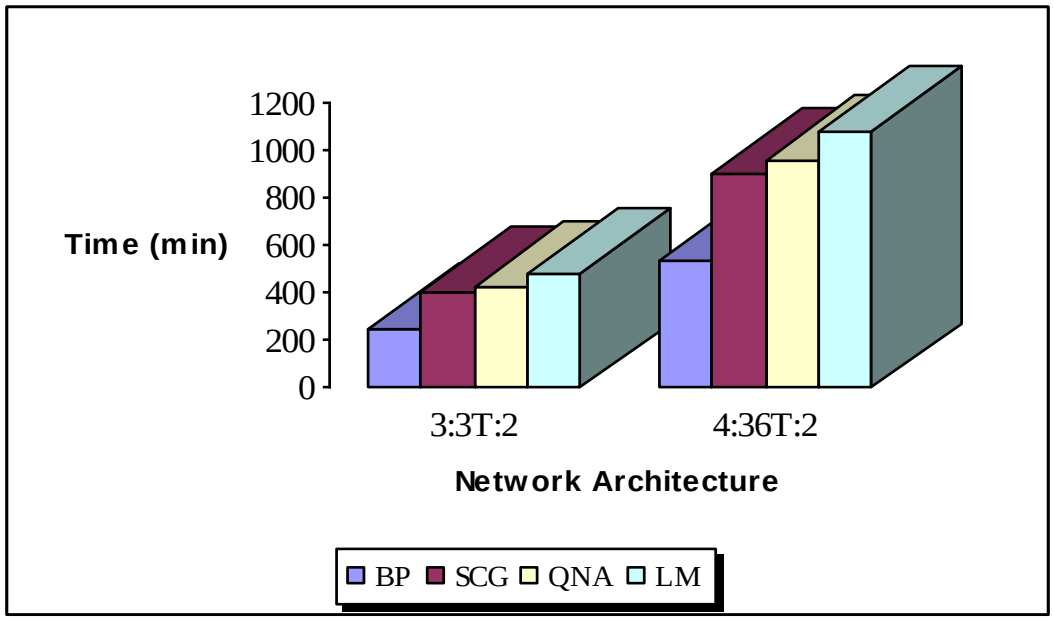


Figure 5.31: Run time of the MLEANN for Mackey glass with two network architectures: original network (4:36:2), optimized network by cellular (3:3:2)

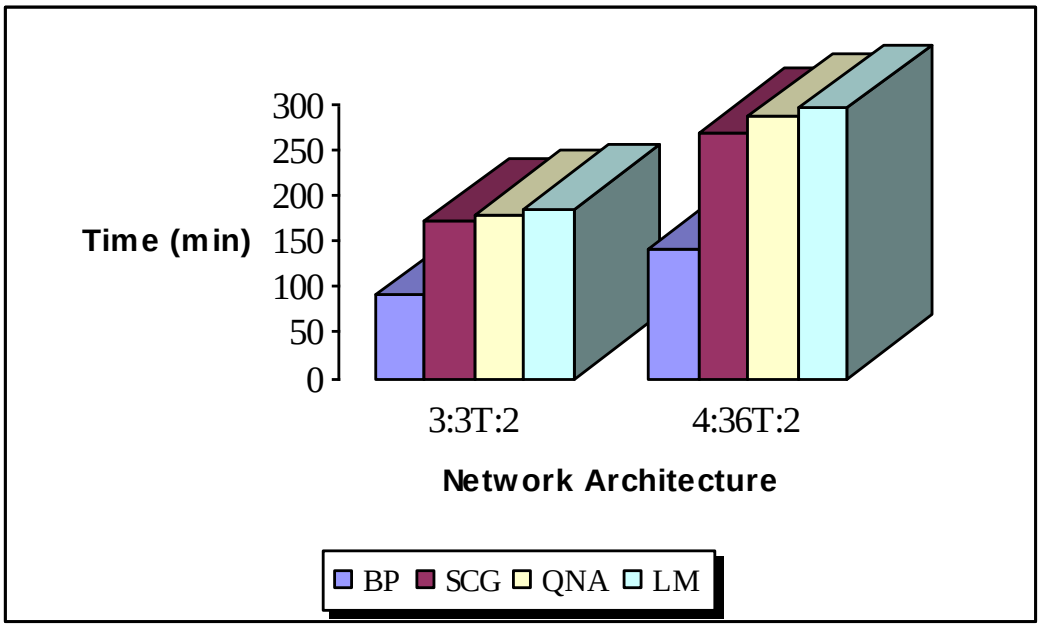


Figure 5.32: Run time of the MLEANN for Gas furnace with two network architectures: original network (4:36:2), optimized network by cellular (3:3:2)

5.3.3.2 MLEANN-CA: Results Discussion:

This subsection includes evaluation and summarization of the experimentation results mentioned in section 5.3.3. The cellular automata technique is able to provide more optimal architectures than direct codification methods. According to Gutteriez experiment result (Gutierrez, Isasi, Molina, Sanchis, & Galvan, 2001), the number of hidden neurons got reduced by approximately 91.7% (from 36 to 3 hidden nodes) using the cellular configurations. Using the direct codification in my experiments, for the Mackey glass (table 5.17) the number of hidden neurons (36 hidden nodes) reduced by 50% (BP), 44.4% (SCG), 50% (QNA), and 47.2% (LM). For the Gas furnace (table 5.18), the number of hidden neurons (36 hidden nodes) reduced by 44.4% (BP), 36.1% (SCG), 38.9% (QNA), and 44.4% (LM). Tables (5.17 – 5.18) also show comparative performance between the MLEANN and the conventional design of ANNs for the two time series problems with architectures: (4:36:2) and (3:3:2). For Mackey-glass series (table 5.17), the results of RMSE on test set were not that encouraging (using 3 hidden neurons) when compared with the architecture design of 36 hidden neurons. The Mackey-glass series requires more hidden neurons to improve the RMSE values. LM algorithm gave the best RMSE error on test set for the architecture of 36 hidden nodes even though it takes the longest time. For the gas furnace time series (table 5.18), the results of RMSE on test set were much better than in Mackey glass results. For BP and LM algorithms, the RMSE for architecture with 3 hidden nodes was less than in architecture with 36 hidden nodes. This is different in SCG and QNA algorithms, since the RMSE for architecture with 36 hidden nodes was less than in architecture with 3 hidden nodes. The LM algorithm produced the best results of RMSE with 3 hidden neurons while SCG algorithm produced the best results with 36 hidden neurons. For the two time series in tables (5.17 – 5.18), all the learning algorithms take short run time for the architecture with 3 hidden nodes in comparison with 36 hidden nodes. For the Mackey glass, the time for evolving and training was much longer than in Gas furnace for all algorithms with different architectures.

Summary

In this chapter, three main experiments were performed and compared together using NeuroSolutions and NeuroGenetic Optimizer toolboxes, and two famous chaotic time series. These experiments used four different learning algorithms in the training process. By applying these experiments, we recognized the best solution and we distinguished the effect of different learning algorithms on different data sets. In the first experiment, the performance of the conventional design of ANNs was tested. We found that the training speed and generalization performance of an ANN is totally dependant on the learning algorithms, architectures, transfer functions, and initial weights. In the second experiment, the performance of the MLEANN approach was explored. In this experiment, the work was concentrated on the evolutionary search of optimal network architectures using direct encoding methods which required much larger chromosomes especially for ANNs with complex architectures. This ended in a too huge space search and thus had longer time in the training process. The third experiment investigated the performance of the MLEANN-CA approach that used indirect encoding methods for designing network architectures. The results revealed the efficiency of the proposed MLEANN–CA in obtaining an efficient design of feed-forward network architecture that is smaller, faster and with better generalization performance.

CHAPTER SIX

CONCLUSIONS AND FUTURE WORKS

This chapter deals with the conclusions and future works in our research. In section 6.1 we start by presenting the main conclusions and in section 6.2 we provide some recommended suggestions on future works.

6.1 Main Conclusions

In this thesis, we had proposed and formulated; MLEANN-CA: an adaptive computational framework based on evolutionary computation and local search procedures for the automatic design of optimal artificial neural networks using direct and indirect encoding methods. In this framework, the evolutionary cellular configurations (indirect encoding methods) were used for designing small feed-forward neural network architectures. Then all the generated architectures were trained and evolved separately using the meta-learning algorithm with the direct evolutionary approach, where four different learning algorithms (BP, SCG, QNA, LM) were used separately for training the neural networks in parallel mode. We tested and explored, experimentally, the performance of the MLEANN-CA, MLEANN, and ANNs using NeuroSolutions and NeuroGenetic Optimizer toolboxes, and two famous chaotic time series. We also explored and evaluated the performance of different neural network learning algorithms for the two chaotic time series when the architecture was changed. We compared the performance of the MLEANN-CA approach with the previous MLEANN that used only the direct codifications in optimizing network architectures and with the conventional design of ANNs. Empirical results illustrated the importance, scalability, and the efficacy of this MLEANN-CA approach in obtaining an efficient design of feed-forward network architecture that was smaller, faster and with better generalization performance.

The three main experiments that were performed in this thesis, using NeuroSolutions / NeuroGenetic Optimizer toolboxes and two chaotic time series, are summarized below:

1. In the first experiment (Artificial neural networks): the performance of the conventional design of ANNs was tested for the two time series. We found that the training speed and generalization performance (i.e. test set RMSE) of an ANN was totally dependant on the learning algorithms, architectures - hidden neurons, transfer functions, initial weights, and the type of the data sets used. The main drawback in this experiment was that for the two time series with different number of hidden neurons, the values of the RMSE were very large in comparison with the results in the other two experiments. Also, there was no optimization and reduction in the network architectures.

2. In the second experiment (MLEANN): the performance of the MLEANN framework was explored. This experiment showed that the EANN design performs more efficiently than conventional ANN design for the two time series since the RMSE values and the numbers of hidden neurons were clearly reduced. So, using the

MLEANN framework improved the learning process and obtained efficient design of network architectures with faster convergence. These results were similar and very close to what was found by Ajith in his experiment with MLEANN framework (Abraham, 2002), (Abraham, 2004). The main drawback in this experiment was in using the direct encoding methods for evolving and optimizing network architectures which required much larger chromosomes especially for ANNs with complex architectures. This ended in a too huge space search and thus had longer time in the training and evolving processes (scalability problem), besides the difficulty in implementing crossover operation.

3. In the third experiment (MLEANN-CA): the performance of the MLEANN-CA framework was investigated. In this experiment we used the evolutionary cellular configurations for designing and optimizing network architectures, instead of using the direct encoding methods as in Ajith's work (Abraham, 2004), and then we applied the meta-learning algorithm for training and evolving these new architectures. This work was different to what was done by Gutteriez (Gutierrez, Isasi, Molina, Sanchis, & Galvan, 2001) since he used the evolutionary cellular configurations for designing feed-forward neural networks architectures but he did not apply the idea of meta-learning algorithm for training the new generated architectures. The experiment results revealed the efficiency of the proposed MLEANN-CA in obtaining an efficient design of feed-forward network architecture that was smaller, faster and with better generalization performance (small values of RMSE).

In general, the following points summarize the main conclusions and notes in this thesis:

1. Selecting the architecture of a network (number of layers, hidden neurons, activation functions, and connection weights) and the correct learning algorithm with its correct parameters is a tedious task for designing an optimal ANN. Moreover, the optimal design of network architecture often becomes a necessity for critical applications and hardware implementations.

2. Evolutionary computation techniques are good approaches for automatically generate appropriate neural network architectures. However the codification of the network is a crucial point in the success of the method. Direct codifications become inefficient from a practical point of view. They don't allow scalability, so to represent large network architectures; very large structures of chromosomes are required which need long time during their operations. Moreover, implementation of crossover is often difficult due to production of non-functional offspring's. To solve these problems an indirect constructive encoding method is used although the search of architectures is restricted to layers. Indirect encoding method, based on evolutionary cellular configurations, is driven to reduce the search space in such a way that similar solutions are eliminated and represented by the only one representative. In this case, the codification makes the method able to find appropriate architectures, which are smaller, and faster.

3. In the meta-learning algorithm, all the generated architectures of the initial population are trained and evolved separately by four different learning algorithms in a parallel environment. Therefore, meta-learning improves the performance, efficiency,

accuracy, and scalability. Meta-learning is also generic, meaning that it is algorithm independent, hence it can benefit from fast and efficient learning algorithms.

4. Different learning algorithms have their staunch proponents, who can always construct instances in which their algorithm performs better than most others. This study reveals the difficulty to generalize which is the best local search algorithm that would work for all the problems. As example, for smaller networks with few numbers of hidden neurons, LM algorithm gave the best results, while SCG algorithm produced the best results with large number of hidden neurons for specific problems.

6.2 Future Works

Like most researches in artificial neural networks and evolutionary computations, this thesis is widely open for improvement. As a future work we recommend the followings:

1. Use the proposed MLEANN-CA approach for optimizing recurrent neural networks, morphological neural networks, and other connectionist networks instead of feed-forward neural networks.

2. Study the influence of the rules in the cellular automata evolution and the capability of other rules to generate a complete space of NN architectures.

3. Use other different time series, as waste water flow prediction (Kasabov, 1996)**,** in training and evolving the neural networks and investigate its effect and performance.

4. Use other different learning algorithms for training neural networks and investigate their effect and performance.

5. Use the MLEANN-CA approach in variety of applications including:

    - **Financial, Insurance, and Securities:** Real estate appraisal, loan advisor, credit line use analysis, corporate financial analysis, currency price prediction, Policy application evaluation, product optimization, market analysis, automatic bond rating, stock trading advisory systems.

    - **Manufacturing**: Manufacturing process control, product design and analysis, process and machine diagnosis, real-time particle identification, visual quality inspection systems, beer testing, paper quality prediction, computer chip quality analysis, analysis of grinding operations, chemical product design analysis, machine maintenance analysis, dynamic modeling of chemical process system, selecting flexible manufacturing systems (FMS), from a group of candidate-FMSs, under disparate level-of-satisfaction of decision maker (Bhattacharya, Abraham, Grosan, & Vasant, 2006).

- **Medical:** Breast cancer cell analysis, EEG and ECG analysis, prosthesis design, optimization of transplant times, hospital expense reduction, hospital quality improvement, emergency room test advisement.
- **Electronics and Telecommunications**: Code sequence prediction, integrated circuit chip layout, chip failure analysis, machine vision, voice synthesis, nonlinear modeling, image and data compression, automated information services, real-time translation of spoken language.

- **Defense**: Target tracking, facial recognition, radar and image signal processing including data compression, feature extraction and noise suppression, signal/image identification.

REFERENCES

Abraham, A., Nath, B., (2000): "Optimal Design of Neural Nets Using Hybrid Algorithms". In Proceedings of the Sixth Pacific Rim International Conference on Artificial Intelligence (PRICAI 2000), Melbourne, pp. 510-520.

Abraham, A., Nath, B., (2001): "ALEC—An Adaptive Learning Framework for Optimizing Artificial Neural Networks". In: N.A. Vassil, et al., (Eds.), Computational Science, Springer, Germany, San Francisco, USA, pp. 171–180

Abraham, A., (2002): "Optimization of Evolutionary Neural Networks Using Hybrid Learning Algorithms". IEEE 2002 Joint International Conference on Neural Networks, Vol. 3, IEEE Press, New York, pp.2797–2802

Abraham, A., (2004): "Meta-Learning Evolutionary Artificial Neural Networks". Neurocomputing Journal, Elsevier Science, Netherlands, Vol. 56c, pp. 1-38.

Abu Salah, A. Al-Salqan, Y. (2006-A): "Meta-learning Evolutionary Artificial Neural Networks: By Means of Cellular Automata". In: Proceedings of IEEE International Conference on Computational Intelligence for Modelling, Control and Automation (CIMCA2005), November 2005, Vienna-Austria, IEEE press, USA.

Abu Salah, A. Al-Salqan, Y. (2006-B): "Meta-Learning Evolutionary Artificial Neural Networks Using Cellular Configurations: Experimental Works", IEEE International Conference on Intelligent Computing, China, Lecture Notes in Computer Science, Vol. 4113, Springer, Verlag Berlin Heidelberg, pp. 178-193.

Andersen, H., Tsoi, A., (1993): "A Constructive Algorithm for the Training of a Multilayer Perceptron Based on the Genetic Algorithm". Complex System. vol. 7, no. 4, pp. 249–268.

Battiti, R. (1992): "First and second-order methods for learning: between steepest descent and Newton's method". Neural Computation 4, pp. 141-166.

Baxter, J., (1992): "The Evolution of learning Algorithms for Artificial Neural Networks". In Complex Systems, D. Green and T. Bossomaier, Eds. Amsterdam. The Netherlands: IOS, pp. 313–326.

Belew, R.., McInerney, J., Schraudolph, N., (1991): Evolving networks: Using genetic algorithm with connectionist learning. Technical Report no. CS90-174 (revised). Computer Science Engineering Department (C-014), Univ. of California, San Diego,

Bengio, S., Bengio, Y., Cloutier, J., Gecsei, J., (1992): "On the Optimization of a Synaptic Learning Rule". In Preprints Conf. Optimality in Artificial and Biological Neural Networks, University of Texas, Dallas.

Bhattacharya, A., Abraham, A., Grosan, C., Vasant, P., (2006): "Meta-Learning Evolutionary Artificial Neural Network for Selecting Flexible Manufacturing Systems under Disparate Level-of-Satisfaction of Decision Maker". IEEE International Symposium on Neural Networks, China, Lecture Notes in Computer Science, Springer Verlag.

Binos, T., (2003): Evolving Neural Network Architecture and Weights Using An Evolutionary Algorithm, Department Of Computer Science, RMIT.

Blickle, T., Thiele, L., (1995): A comparison of selection schemes used in genetic algorithms. (Technical Report, no.11). Computer Engineering and Communication Network Lab (TIK), Swiss Federal Institute of technology, Zurich, Switzerland.

Box, G.E.P., Jenkins, G.M., (1970): Time Series Analysis, Forecasting and Control. Holden Day, San Francisco, 1970.

Branke, J., (1995): "Evolutionary Algorithms for Neural Network Design and Training", In Jarmo Talander, ed.; Proceedings of the 1st Nordic Workshop Genetic Algorithms Applications. Vaasa, Finland.

Braun, H., Weisbrod, J. (1993): "Evolving neural feedforward networks". In Albrecht, R. F., Reeves, C. R., editors, Proceedings of ANNGA93, International Conference on Artificial Neural Networks and Genetic Algorithms, Springer-Verlag, Innsbruck, pages 25–32.

Burney, S.M., Jilani, T.A., Ardil, C. (2004): "A Comparison of First and Second Order Training Algorithms for Artificial Neural Networks". International journal of computational intelligence, volume 1, no. 3 issn:1304-4508.

Caudell, T.P., Dolan, C.P. (1989): "Parametric Connectivity: Training of Constrained Networks using Genetic Algorithms", Proc. of the third International Conference on Genetic Algorithms and their applications, Morgan Kaufman, pp.370-374.

Chalmers, D., (1990): "The Evolution of learning: An Experiment in genetic Connectionism". In Proc. of Connectionist Models Summer School. San Mateo, CA: Morgan Kaufmann, pp. 81–90.

Chan, P., Stolfo, S., (1993): Toward Parallel and Distributed Learning by Meta-Learning. Department of Computer Science, Columbia University, New York.

Chval, J., (2002): Evolving Artificial Neural Networks by Means of Evolutionary Algorithms with L-systems based Encoding. Research Report. Faculty of Electrical Engineering, Technical University in Prague.

Fogel, D. B., Wasson, E. C., Boughton, E. M., (1995): "Evolving neural networks for detecting breast cancer," *Cancer Lett.*, vol. 96, no. 1, pp. 49–53.

Gardner, M., (1970): "The fantastic combinations of John Conway's new solitaire game life", Scientific American, vol. 223, no. 4, pp. 120-123

Gruau, F., Whitley, L.D., (1993): "Adding Learning to the Cellular Development of Neural Networks: Evolution and the Baldwin Effect". Evolutionary Computation, vol. 1, no. 3, pp. 213-233

Gruau F., Whitley, L.D, Pyeatt, L., (1995): "Cellular Encoding Applied to Neurocontrol". In Proceedings of the Sixth International Conference on Genetic Algorithms.

Guo, Z., Uhrig, R. E., (1992): "Using genetic algorithms to select inputs for neural networks," In *Proc.* Int. Workshop Combinations of Genetic Algorithms and Neural Networks (COGANN-92), D. Whitley and J. D. Schaffer, Eds. Los Alamitos, CA: IEEE Computer Soc., pp. 223–234.

Gutierrez. G., Isasi, P., Molina, J.M., Sanchis, A., Galvan, I.M. (2001): "Evolutionary cellular configurations for designing feedforward neural network architectures". Lecture Notes in Computer Science, Vol. 2084, Springer, Germany, pp. 514–521.

Gutowitz, H., (1991): "Cellular Automata: Theory and Experiment". MIT Press. Special issue of Physica D, Volume 45, Nos. 1/3.

Harp, S., Samad T., Guha A. (1989): "Towards the Genetic Synthesis of Neural Networks". Proceedings of the Third International Conference on Genetic Algorithms and their applications, San Mateo, CA, USA, pp 360-369.

Harp, S.A., Samad, T., Guha, A., (1990): "Designing application specific using genetic algorithms", In Advances in Neural Information Processing Systems 2, Morgan Kaufmann, San Mateo, CA, pp. 447-454.

Haupt, L., (December, 1997): "Introduction to optimization". In Practical Genetic Algorithms. John Wiley and Sons, pages 1-24.

Hendtlass, T., Podlena, J., (1995): "Evolving complex neural networks that age". IEEE International Conference on Evolutionary Computation, 2:590-595.

Hestenes, M. R., Stifle, E. (1952): "Methods of conjugate gradients for solving linear systems". Journal of Research of the National Bureau of Standards-49, pp. 409–436.

Hinton, G. E. (September, 1989): "Connectionist learning procedures". Artificial Intelligence, vol.40, pp. 185-234.

Hunt, S.D., Deller, J. R., (1995): "Selective training of feedforward artificial neural networks using matrix perturbation theory". Neural networks, vol. 8, no. 6, pp 931-944.

Husken, M., Gayko, J.E., Sendoff, B. (2000): "Optimization for Problem Classes-Neural Networks that Learn to Learn". IEEE Symposium of Evolutionary computation and Neural networks (ECNN-2000), pages 98-109, IEEE Press 2000.

Hussain, J.E,. Browse, R.A., (1998): “Attribute Grammars for Genetic Representations of Neural Networks and Syntactic Constrains of Genetic Programming”. In AIVIGI’98, Workshop on Evolutionary Computation.

Jacob, C., Rehder, J., (1993): "Evolution of neural networks architectures by a hierarchical grammar-based genetic system". ANNGA'93, Proc. of the International Conference on Artificial Neural Networks & Genetic Algorithms, Innsbruck, pp.72 - 79.

Jain, A. K., Mao, J., Mohiuddin, K. (March, 1996): “Artificial neural networks: A tutorial”. IEEE Computer special issue on Neural Computing, pp. 31-44.

Kasabov, N., (1996): Foundations of Neural Networks, Fuzzy Systems and Knowledge Engineering. The MIT Press.

Kitano, H., (1990): “Designing Neural Networks using Genetic Algorithms with Graph Generation System”, Complex System. Vol. 4, no. 4, pp. 461–476

Korning, P. G. (1995): “Training neural networks by means of genetic algorithms working on very long chromosomes”. *Int. J. Neural System*, vol. 6, no. 3, pp. 299–316.

Koza, J., Rice, J., (1991): “Genetic generation of both the weights and architecture for a neural network”. In Proceedings of IEEE International Joint Conference of Neural Networks (IJCNN’91 Seattle), vol. 2, pp. 397–404.

Leung, F., Lam, H. K., Ling, S. H., (January, 2003): “Tuning of the Structure and Parameters of a Neural Network Using an Improved Genetic Algorithm”. IEEE transaction on Neural networks, Vol. 14 No 1.

Lippmann, R. P. (1987): “An introduction to computing with neural nets”. IEEE ASSP Magazine, pp. 4-22.

Liu, Y., Yao, X., (1998): “Toward Designing Neural Network Ensembles by Evolution”. In Parallel Problem Solving from Nature (PPSN) V, Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag, vol. 1498, pp. 623–632.

Luke, S., Spector, L., (1996): “Evolving Graphs and Networks with Edge Encoding: Preliminary Report”. Genetic Programming conference (GP96), Stanford

Mackey, M.C., Glass, L., (1977): Oscillation and Chaos in Physiological Control Systems. Science 197. pp. 287–289.

Magoulas, G., Plagianakos, V., Vrahatis, M., (2001): “Hybrid methods using evolutionary algorithms for on-line training”. In Proc. of the IEEE International Joint Conference on Neural Networks (IJCNN'2001), Washington.

Merz, C.J. (1996): “Dynamical selection of learning algorithms”. In: edited by F. Filnstone and B. Bunny. Artificial Intelligence and Statistics. New York: Springer-Verlag.

Molar, F., (1997): “Efficient Training of feedforward Neural Networks”. Computer Science Department, Aarhus University.

Molina, J.M., Galván, I., Isasi, P., Sanchis, A., (2000-A): “Evolution of Context-free Grammars for Designing Optimal Neural Networks Architectures”. GECCO 2000, Workshop on Evolutionary Computation in development of ANN. USA.

Molina, J.M., Galvan, I., Isasi, P., Sanchis, A. (2000-B): “Grammars and cellular automata for evolving neural network architectures”. IEEE International Conference on Systems, Man, and Cybernetics, 4:2497-2502.

Moller, A. F. (1993): “A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning. Neural Networks”. Vol. 6, no.4, pp. 525-533.

Nehaniv, C.L., (2002): “Self-Reproduction in Asynchronous Cellular Automata”. Adaptive Systems Research Group, Faculty of Engineering & Information Sciences, University of Hertfordshire, United Kingdom.

Pearlmutter, B.A., (1994): “Fast exact multiplication by the Hessian”. Neural Computation. 6(1), pp.147–160.

Prodromidis, A., Stolfo, S., (1998): “Pruning meta-classifiers in a distributed data mining system”. In Proc of the KDD’98 workshop in Distributed Data Mining.

Prodromidis, A., (1999): “Management of Intelligent Learning Agents in Distributed Data Mining Systems”. Department of Computer Science, Columbia University, New York, NY.

Rumelhart, E., Hinton, G. E., Williams, R. J. (1986): “Learning internal representations by error propagation”. Parallel distributed processing. Vol. I, MIT Press, Cambridge, MA, pp. 318-362.

Salomon, R., (1998): “Evolutionary algorithms and gradient search: Similarities and differences”. IEEE Transactions on Evolutionary Computation, vol. 2, no. 2, pp. 45-55.

Schiffmann, W., Joost, M., Werner, R., (1993): “Comparison of optimized backpropagation algorithms”. Proceedings f the European Symposium on Artificial Neural Networks, Brussels, Belgium, pp. 97–104.

Schraudolph, N. N. (1999): “Local gain adaptation in stochastic-gradient descent”. In Proceedings of the Ninth International Conference on Artificial Neural Networks, Edinburgh, Scotland, IEE, London, pp. 569–574.

Schraudolph, N.N., Grapple, T. (January, 2003): “Combining Conjugate Direction Methods with Stochastic Approximation of Gradients”. Proceedings of the Ninth International Workshop on Artificial Intelligence and Statistics.

Seppo, P., Vagan, T., Alexey, T., (1999-A): "A Dynamic Integration Algorithm for an Ensemble of Classifiers". University of Finland, Finland.

Seppo, P., Vagan, T., Alexey, T., (1999-B): "Dynamic Integration of Data Mining Methods Using Selection in a Knowledge Discovery Management System". University of Finland, Finland.

Sexton, R. S., Dorsey, R. E., Johnson, J. D. (1998): "Toward global optimization of neural networks: A comparison of the genetic algorithm and backpropagation". Decision Support System., vol. 22, no. 2, pp. 171–185.

Topchy, A., Lebedko, O., Miagkikh, V., (1995): "Fast learning in multilayered neural networks by means of hybrid evolutionary and gradient algorithms". Research Institute for Multiprocessor Computer Systems Chekhov, Taganrog, RUSSIA.

Whitley, D. (1994): "A genetic algorithm tutorial". Statistics and Computing, volume 4, pp. 65-85.

Wolfram, S. (1994): "Cellular automata and complexity". Addison-Wesley.

Wong, M.H., Chung, T.S., Wong, Y.K., (1998): "Application of evolving neural network to unit commitment". Proceedings of EMPD '98 International Conference on Energy Management and Power Delivery, 1:154 -159.

Yao, X., (1993): "A review of Evolutionary Artificial Neural Networks", International Journal of Intelligent Systems, vol. 8, pp. 539-567.

Yao, X. (1999): "Evolving artificial neural networks". Proceedings of the IEEE, vol. 87, No. 9, pp.1423-1447.

Yao, X., Liu, Y., (1996): "Ensemble Structure of Evolutionary Artificial Neural Networks". In Proc. IEEE International Conference on Evolutionary Computation (ICEC'96), Nagoya, Japan, pp. 659–664.

Yao, X., Liu, Y., (1997): "A New Evolutionary System for Evolving Artificial Neural Networks", IEEE Trans. Neural Networks, vol. 8, no. 3, pp. 694–713.

Yao, X., Liu, Y., (1998): "Towards designing artificial neural networks by evolution". Applied Mathematics and Computation. vol. 91, no. 1, pp.83–90.

Yoon, B., Holmes, D. J., Langholz, G. (1994): "Efficient genetic algorithms for training layered feedforward neural networks". *Inform. Sci.*, vol. 76, nos. 1–2, pp. 67–85

Zhou, G., Si, J., (1998): "Advanced neural-network training algorithm with reduced complexity based on Jacobian deficiency". IEEE Trans Neural Networks. Vol. 9, no. 3, pp. 448-453.

APPENDIX A

TRAINING AND EVOLVING ANNS USING NEUROSOLUTIONS AND NEUROGENETIC OPTIMIZER TOOLBOXES

In this appendix we introduce short description about the NeuroSolutions and NeuroGenetic Optimizer toolboxes we used in our experiments (chapter 5). In addition, we present the important screens and windows that are shown during the experiments simulations using these toolboxes.

A.1 NeuroGenetic Optimizer (version 2.1)

NeuroGenetic Optimizer (NGO) automates much of the neural network design and development chores we used to do, probably by hand using trial and error. Some of these tedious tasks include testing/training data set selection, determining which input variables to use and neural network type selection and architectural design. The NGO uses Genetic Algorithms to evolve neural network structures and select suitable input variables. This evolving, learning, adapting Artificial Life capability is a powerful problem solving paradigm. We can use these techniques to solve any number of real world challenges.

The NGO, like most other leading neural network tools, is being used in a wide variety of applications, including financial predictions, medical diagnosis, market classification, modeling manufacturing processes and resulting product quality, classification of biological organisms, job cost estimating, fraud detection and many others. The NGO is a general purpose, robust, practical tool to naturally genetically engineer neural networks. This system emerged from the need to easily and quickly discover the best data elements and neural network architectures to build effective neural network applications. Previously, many hours of human effort were spent attempting to find the best networks manually. It was clear that an effective automation tool was needed to off-load these hours of effort onto computers and hence the NGO was born.

During a run, the NGO provides us the ability to view the status of what is happening, view the evolving population, see the configurations and statistics of the Top 10 networks found so far, observe learning curves and watch the neural outputs match our desired data for the network being trained and view and/or print reports on the specifics of the system setup and the resulting top networks. The NGO was developed by BioComp Systems, Inc.

A.1.1 NGO Screens:

The following screens show how to make training or evolving (optimizing) for ANNs using NGO toolbox.

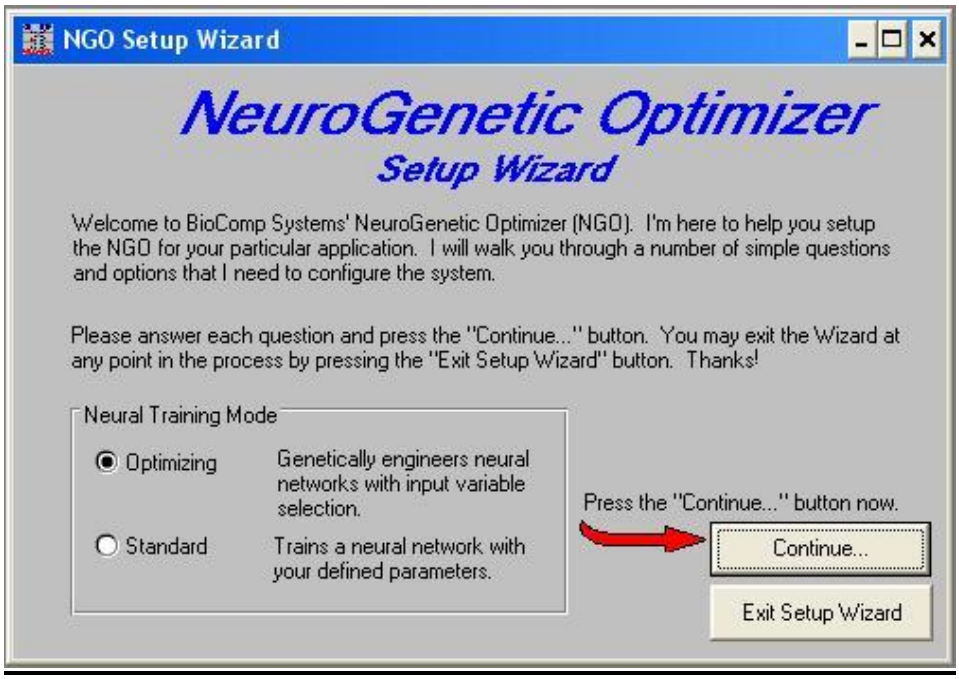


Figure A.1.1: Neural network training mode: optimizing, or standard training

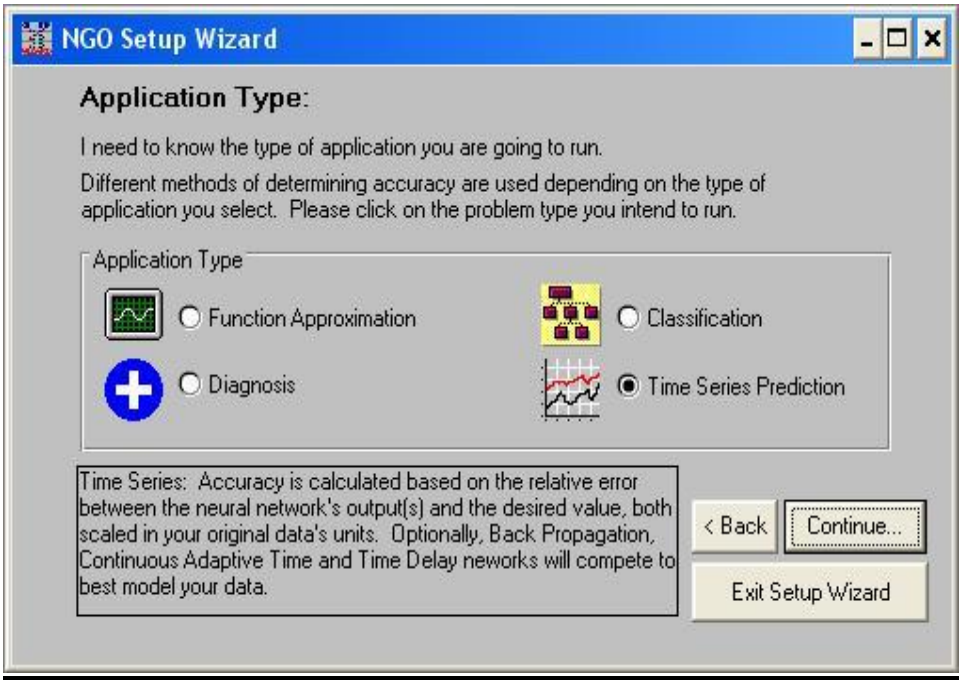


Figure A.1.2: Application type: time series prediction, classification, diagnosis, etc

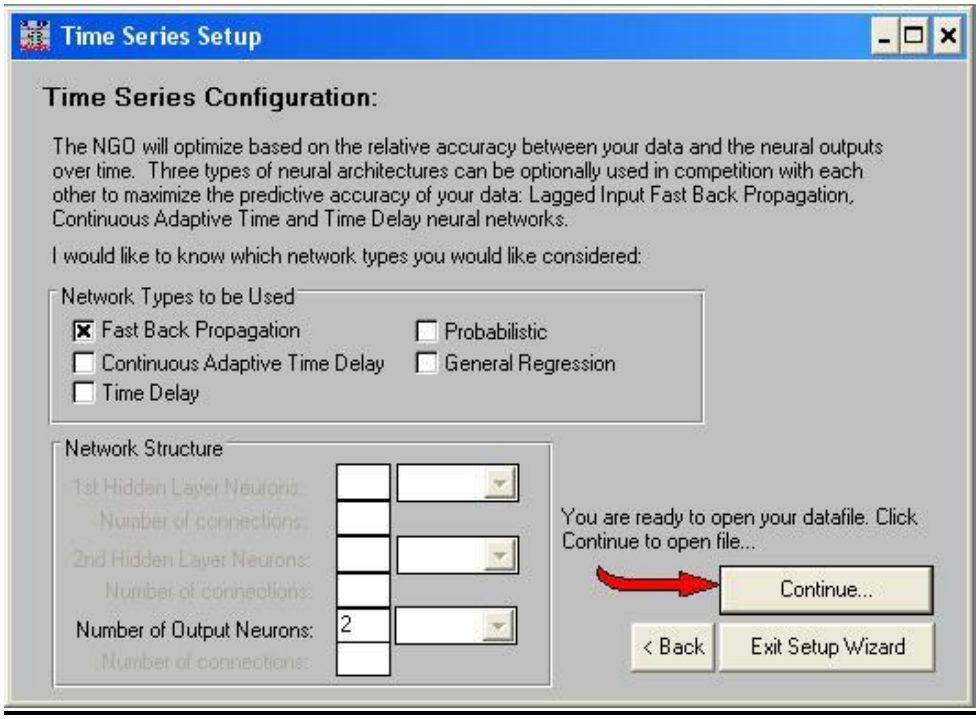


Figure A.1.3: Time series configuration: optimizing mode

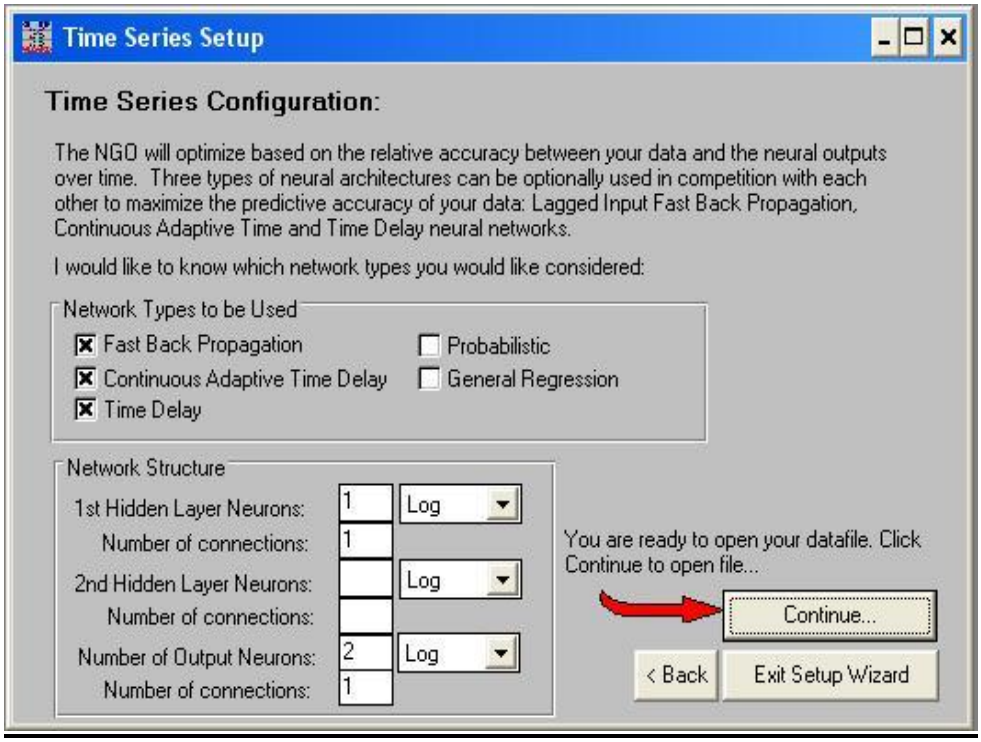


Figure A.1.4: Time series configuration: standard training mode

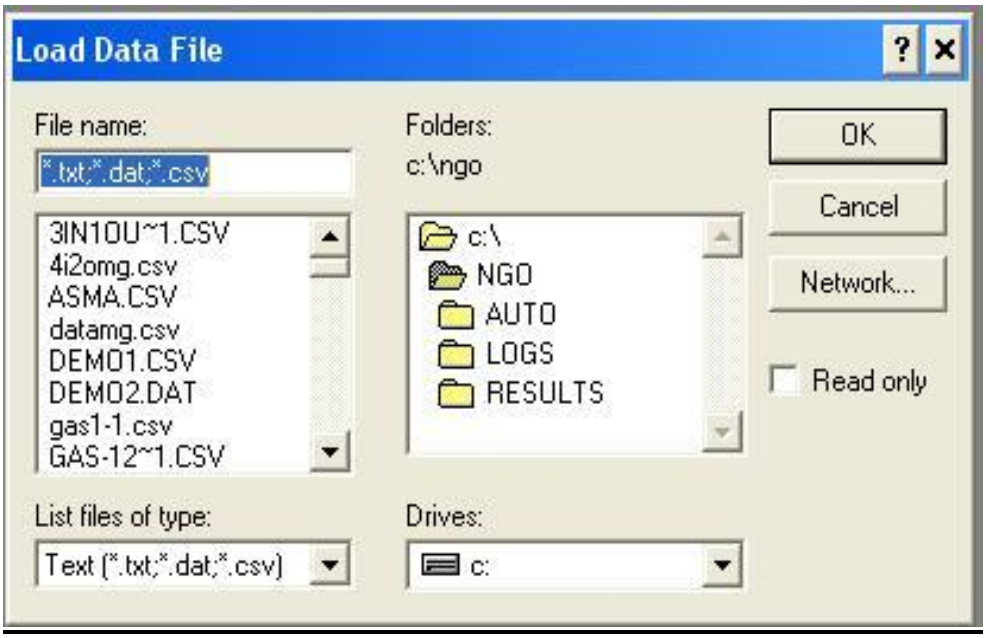
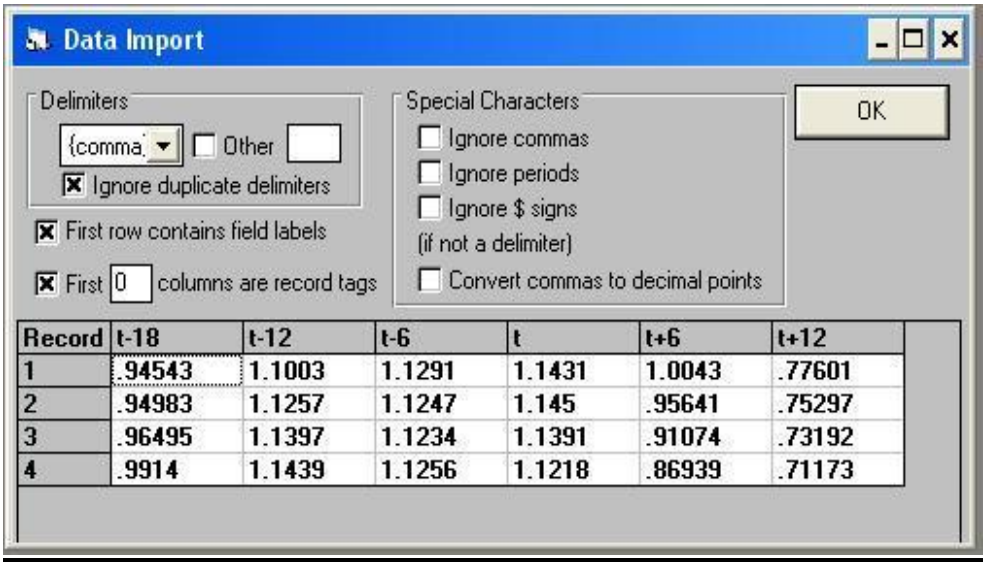


Figure A.1.5: Load data file: time series problem (Mackey-glass or Gas furnace)



Record	t-18	t-12	t-6	t	t+6	t+12
1	.94543	1.1003	1.1291	1.1431	1.0043	.77601
2	.94983	1.1257	1.1247	1.145	.95641	.75297
3	.96495	1.1397	1.1234	1.1391	.91074	.73192
4	.9914	1.1439	1.1256	1.1218	.86939	.71173

Figure A.1.6: Data import: includes network inputs & outputs (4 inputs, 2 outputs)

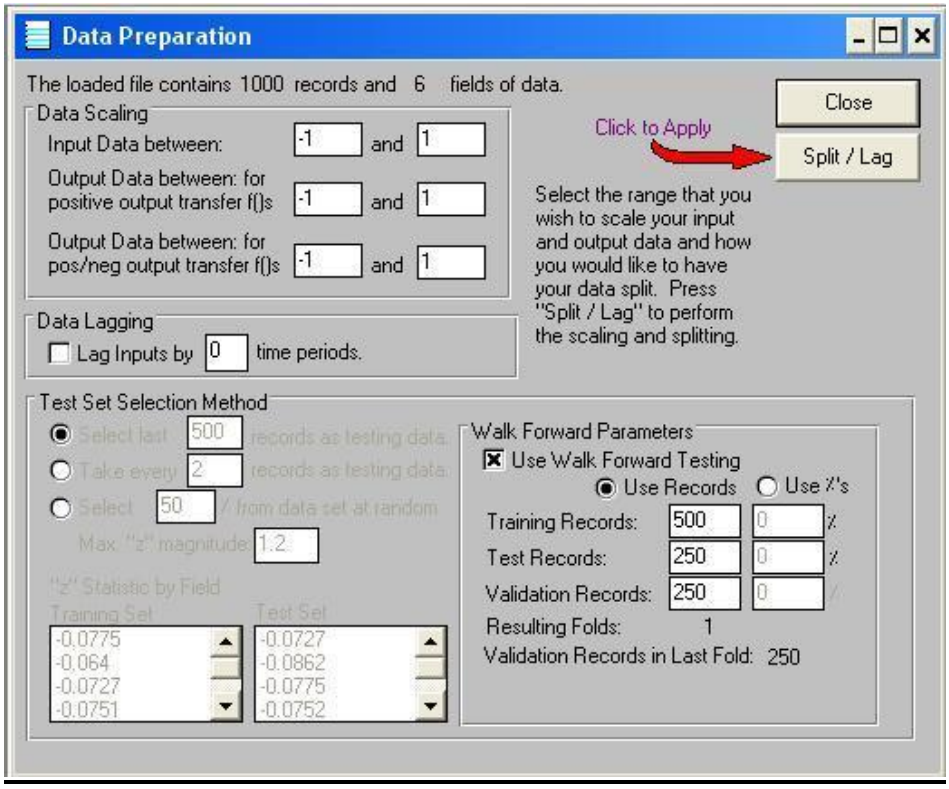


Figure A.1.7: Data preparation: scaling and splitting

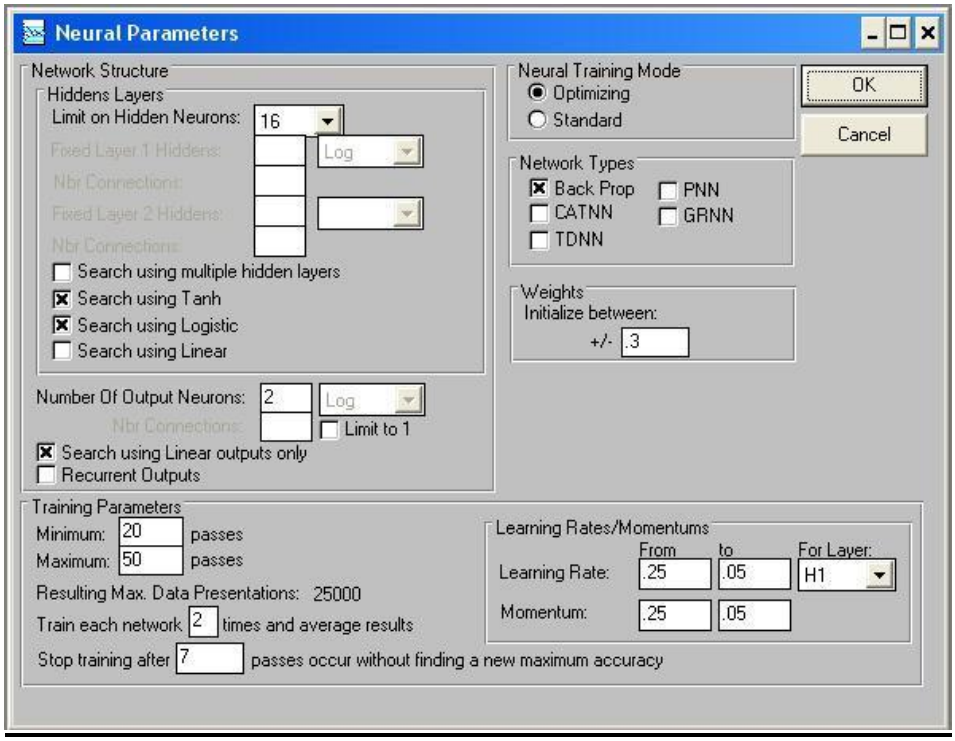


Figure A.1.8: Neural network parameters: hidden nodes, transfer function, initial weight, etc

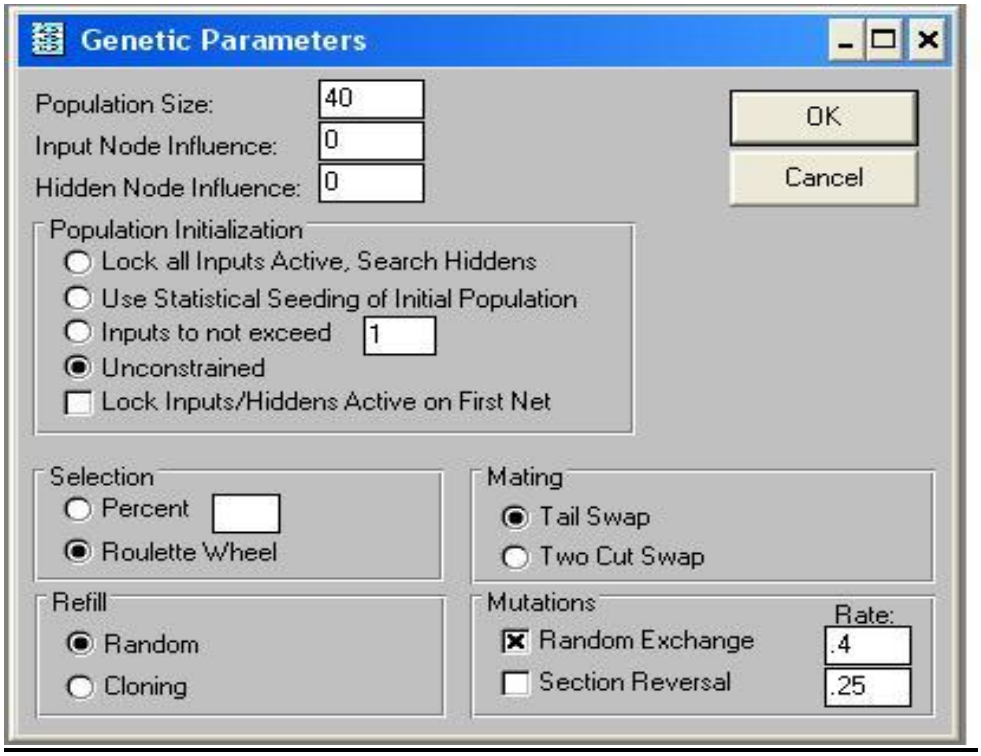


Figure A.1.9: Genetic algorithm parameters: population size, selection, mutation, etc

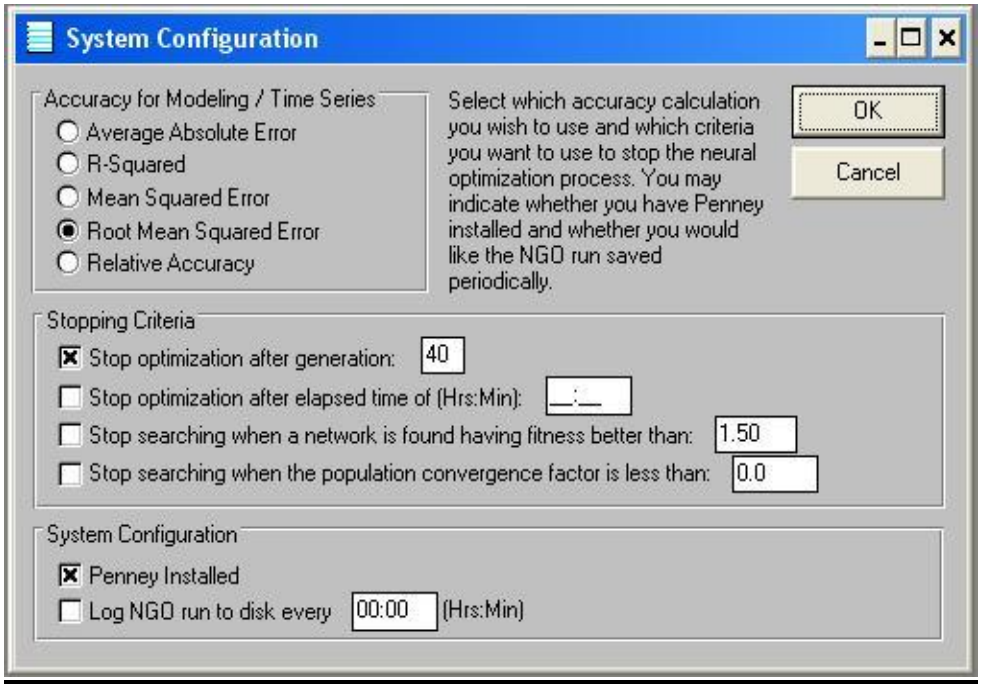


Figure A.1.10: System configuration: type of error, stopping criteria and maximum generation, etc

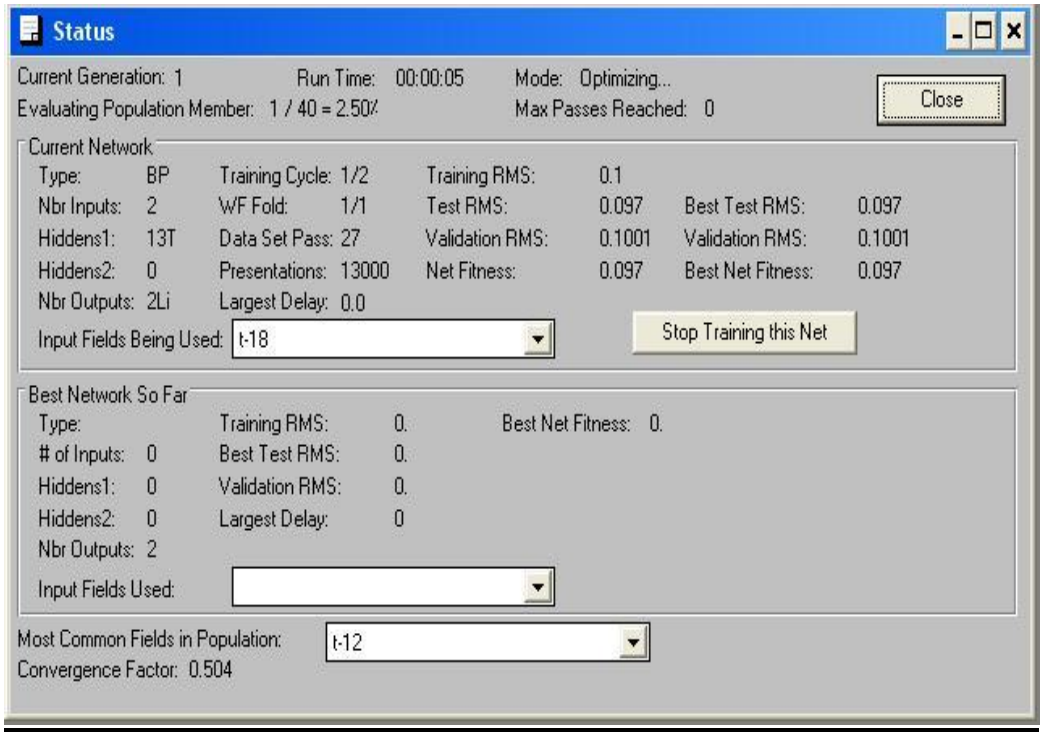


Figure A.1.11: Status of what happing during training and optimizing neural networks

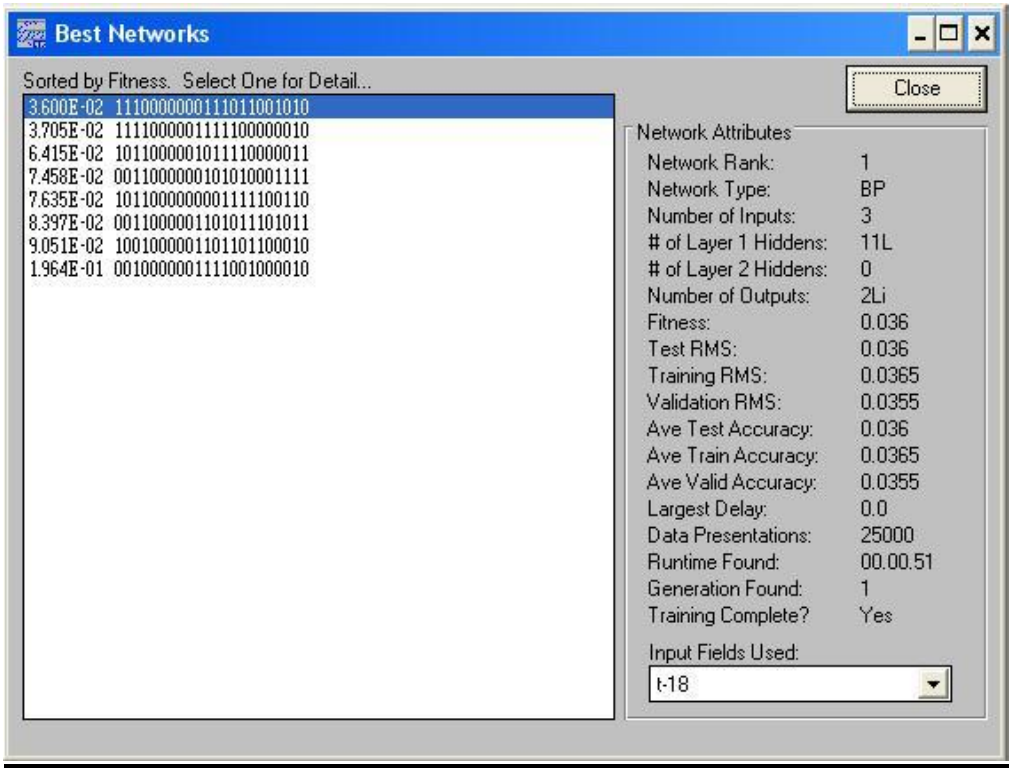


Figure A.1.12: Configurations and status of top 10 networks

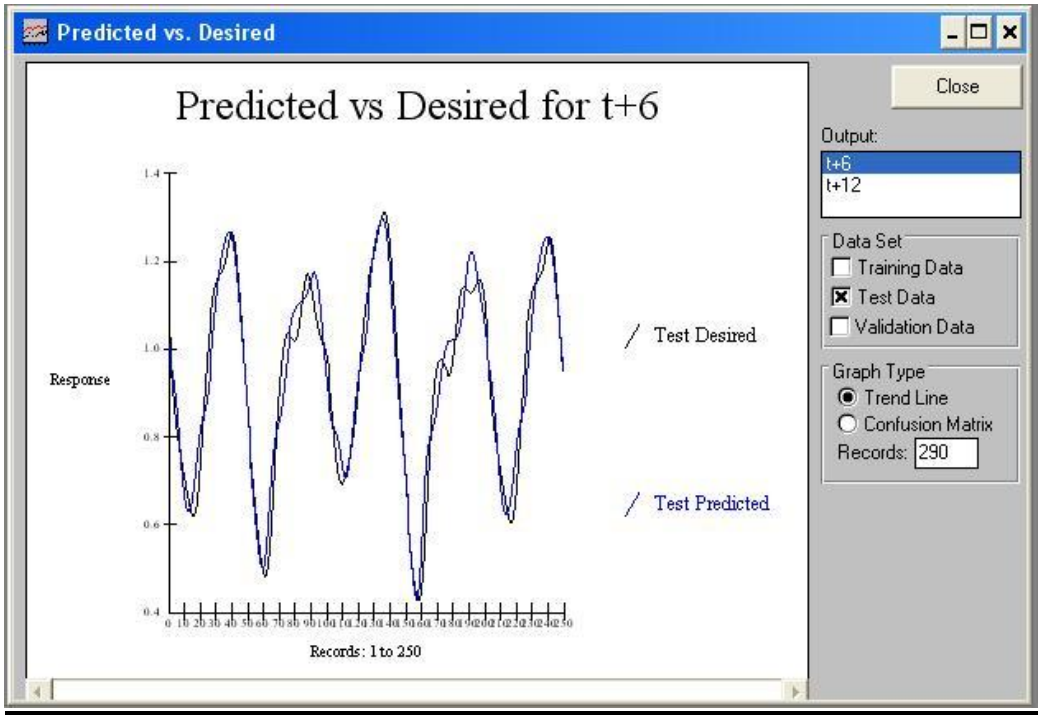


Figure A.1.13: Neural network output: desired and predicted

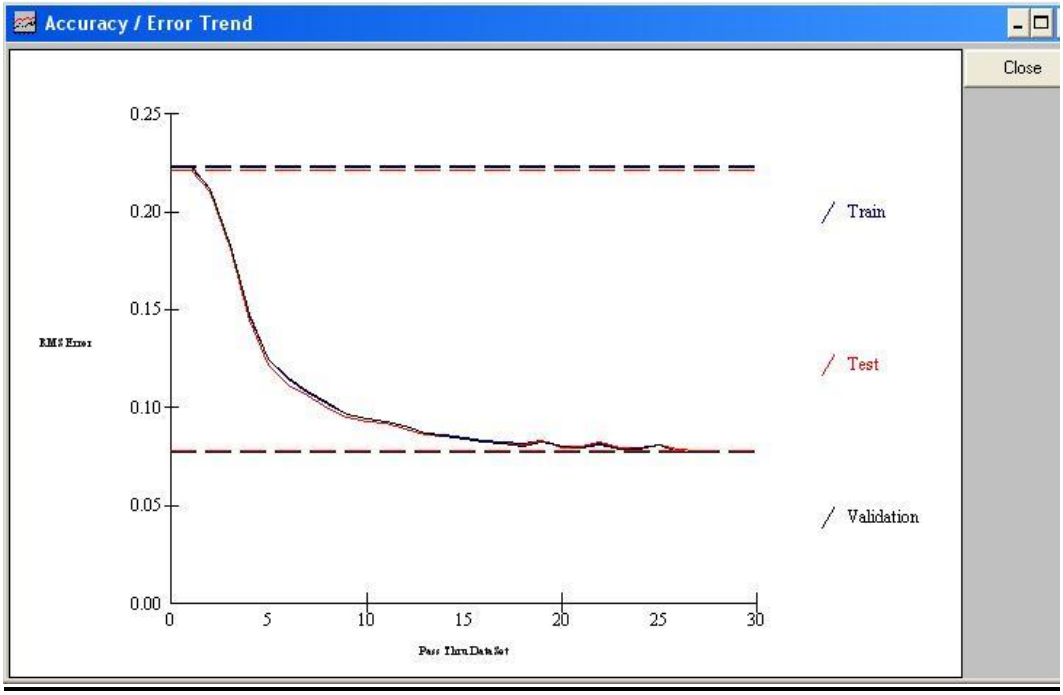


Figure A.1.14: Learning curves: accuracy / error trend

A.2 NeuroSolutions (version 5.01)

For many years neural networks have been successfully applied to various data prediction, data classification and data mining problems in research, business and industrial environments. NeuroSolutions is a highly graphical neural network simulation tool for Windows 98/2000/XP. This leading-edge software combines a modular, icon-based network design interface with an implementation of advanced learning procedures (such as conjugate gradients, Levenberg Marquard, and backpropagation) and genetic optimization giving us the power and flexibility needed to design the neural network that produces the best solution for our specific problem.

Some other notable features include C++ source code generation, customized components through DLLs, neuro-fuzzy architectures, and programmatic control from Visual Basic using OLE Automation. NeuroSolutions includes Genetic Optimization which allows us to optimize virtually any parameter in a neural network to produce the lowest error. For example, the number of hidden units, the learning rates, and the input selection can all be optimized to improve the network performance. Individual weights used in the neural network can even be updated through Genetic Optimization as an alternative to traditional training methods.

NeuroSolutions includes number of important wizards and the NeuralBuilder is one of them, which I used in my experiments simulations. The NeuralBuilder is a sophisticated neural network builder that sends commands to NeuroSolutions to automatically construct a fully-functional neural network. The object-oriented simulation environment of NeuroSolutions gives the user an unprecedented flexibility to construct neural network simulations. However, flexibility and power require a substantial amount of knowledge about neural networks. The NeuralBuilder aids the user by encapsulating the network building rules and reducing the user decisions down to an easy, step-by-step procedure.

Much of the construction effort necessary to build neural networks with NeuroSolutions becomes transparent to the user. There is a wide range of conventional neural network architectures (models) to choose from. Some of these models / architectures include: Multilayer Perceptron, Generalized Feedforward, Modular, Probabilistic Neural Network (PNN), Self-Organizing Map (SOM), etc. When an architecture is selected, the user is lead through a series of panels containing the configuration parameters for the model such as: the number of hidden layers, the number of processing elements, the learning algorithm, and the transfer function. We can also use the genetic algorithm to optimize any parameter. After completing all the panels, the utility makes calls to NeuroSolutions to automatically construct the network according to the specifications. NeuroSolutions is developed by NeuroDimension Incorporated.

A.2.1 NeuroSolutions Screens:

The following screens show how to make training or evolving (optimizing) for ANNs using NeuroSolutions toolbox.

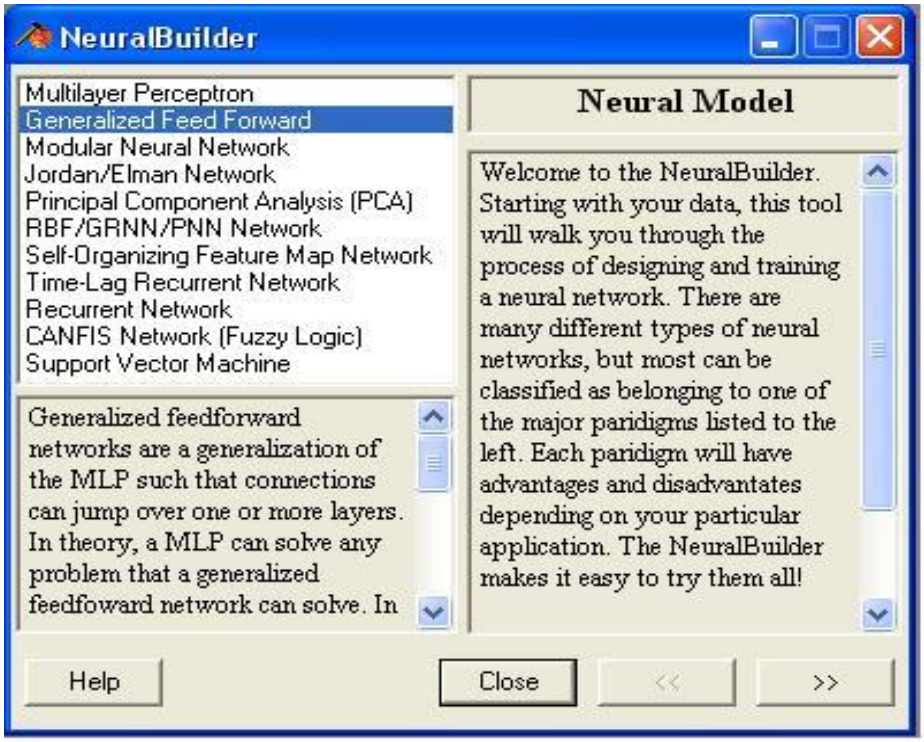


Figure A.2.1: Selecting the network architecture we want to build

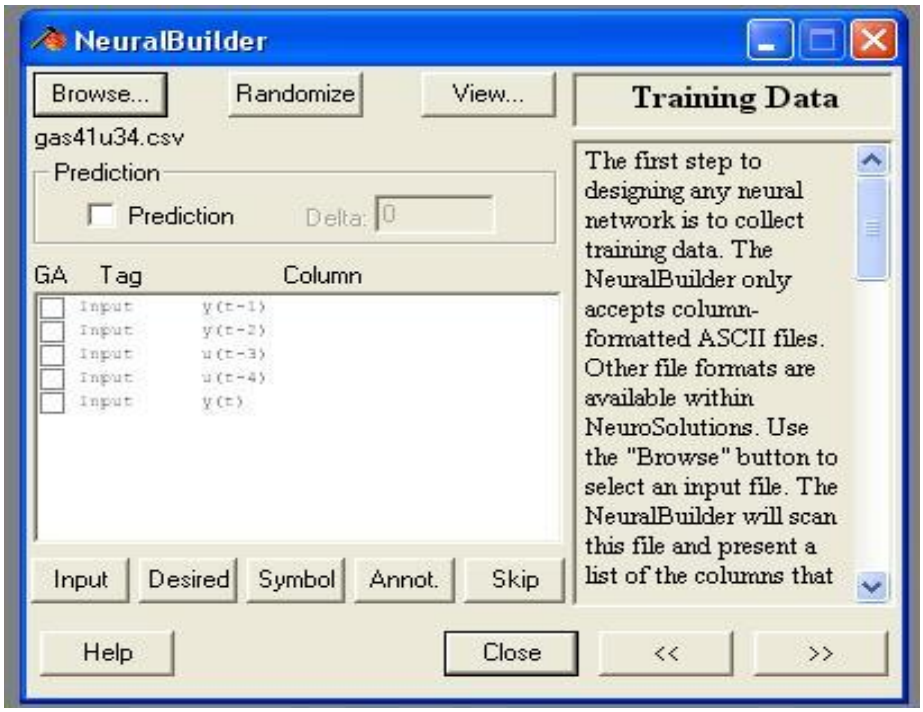


Figure A.2.2: Importing data: training data and the desired response

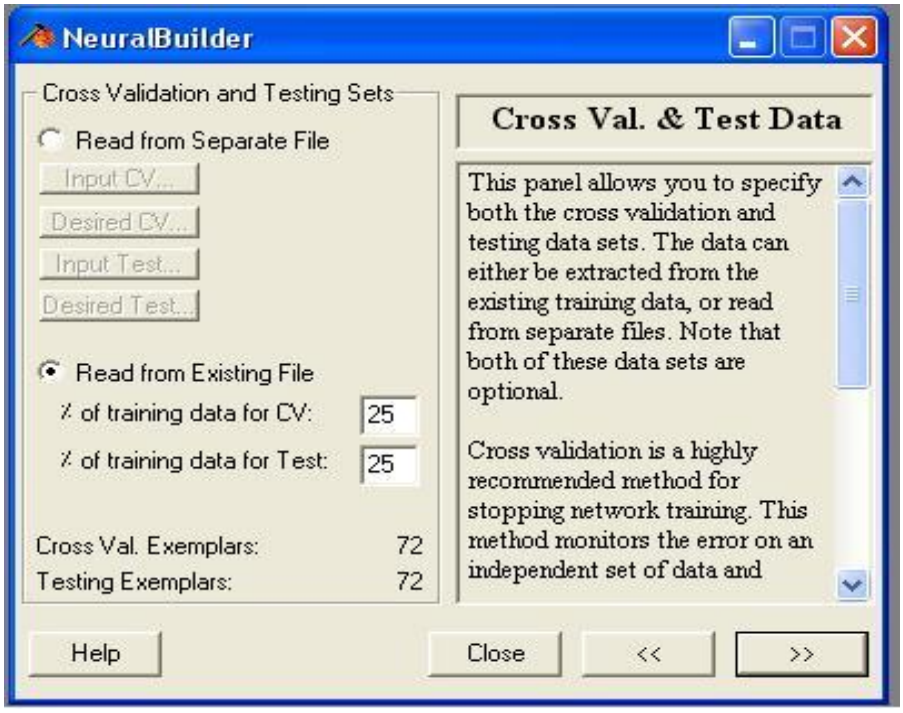


Figure A.2.3: Splitting data: specify data for testing and validation

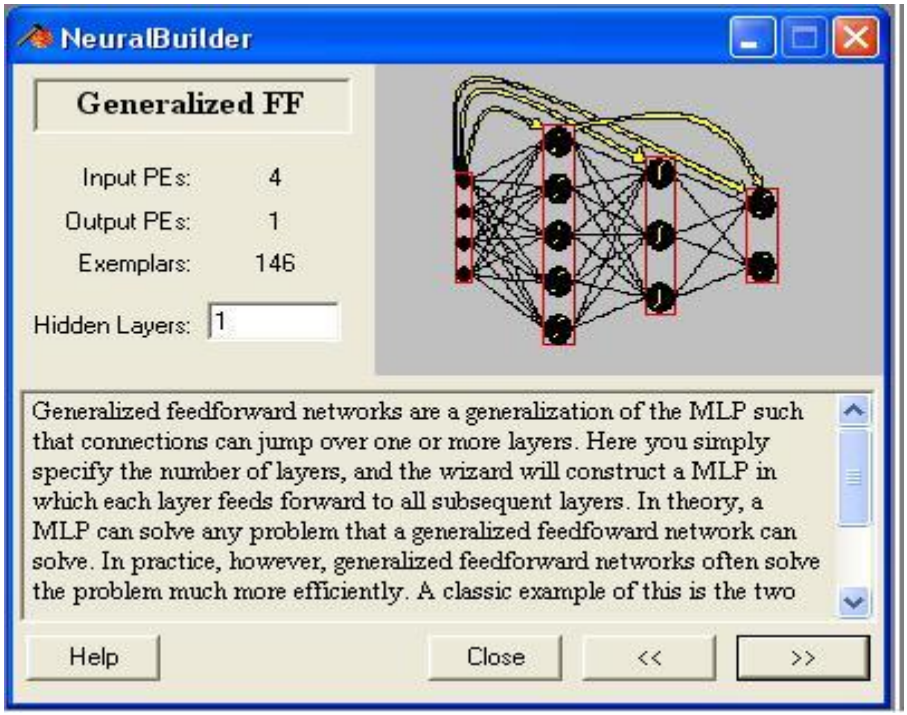


Figure A.2.4: Specifying the number of hidden layers in the network

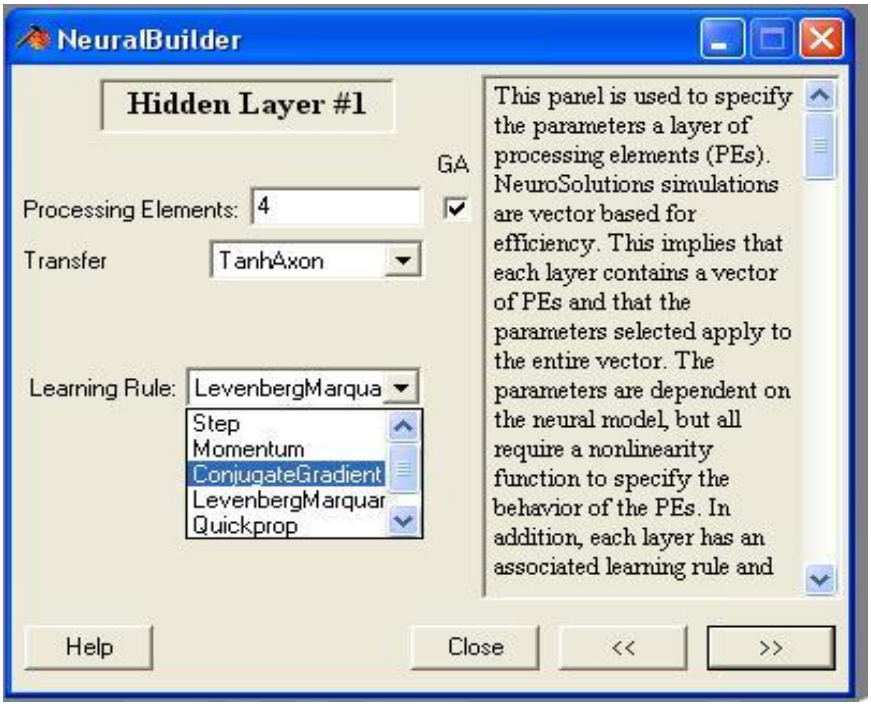


Figure A.2.5: Specifying the number of nodes in the hidden layer, transfer function, learning algorithm, and selecting GA for optimization

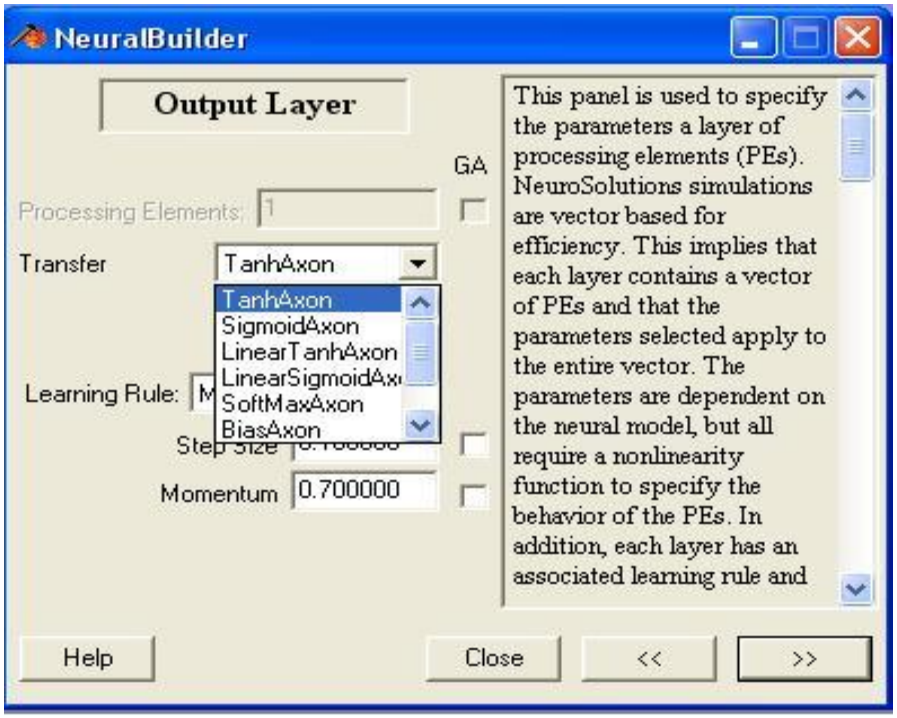


Figure A.2.6: Specifying the transfer function and the learning rule in the output layer

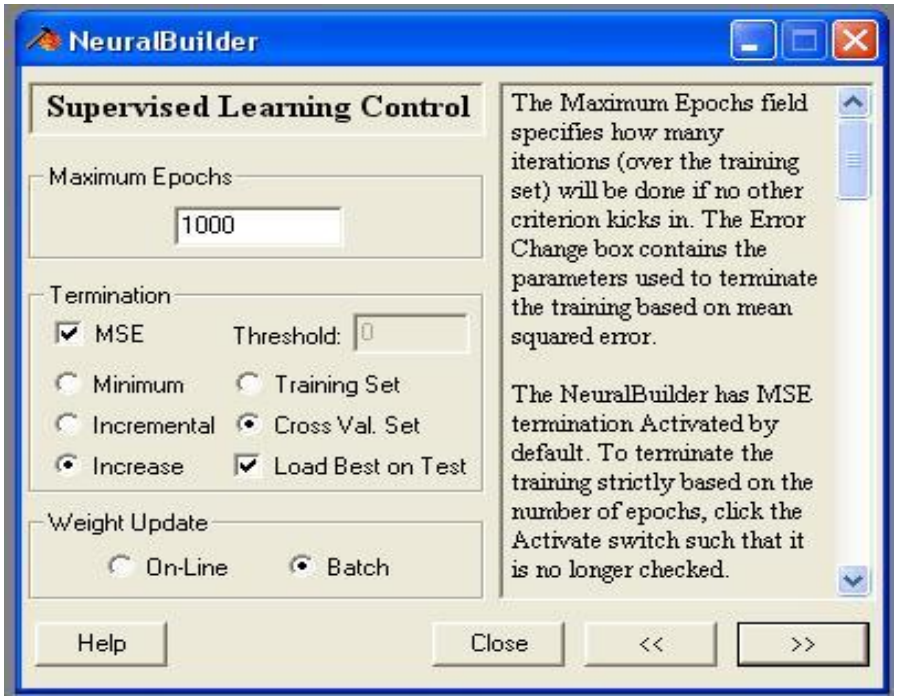


Figure A.2.7: Specifying the maximum epochs, termination, and MSE

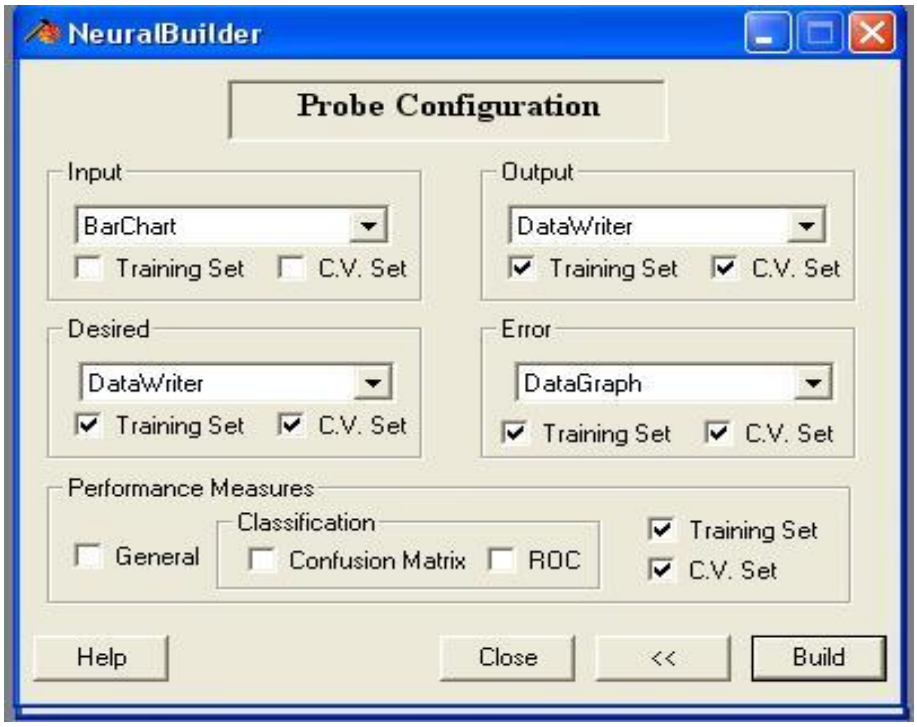


Figure A.2.8: Probe configuration panel: visualizing the input, output, desired, and error

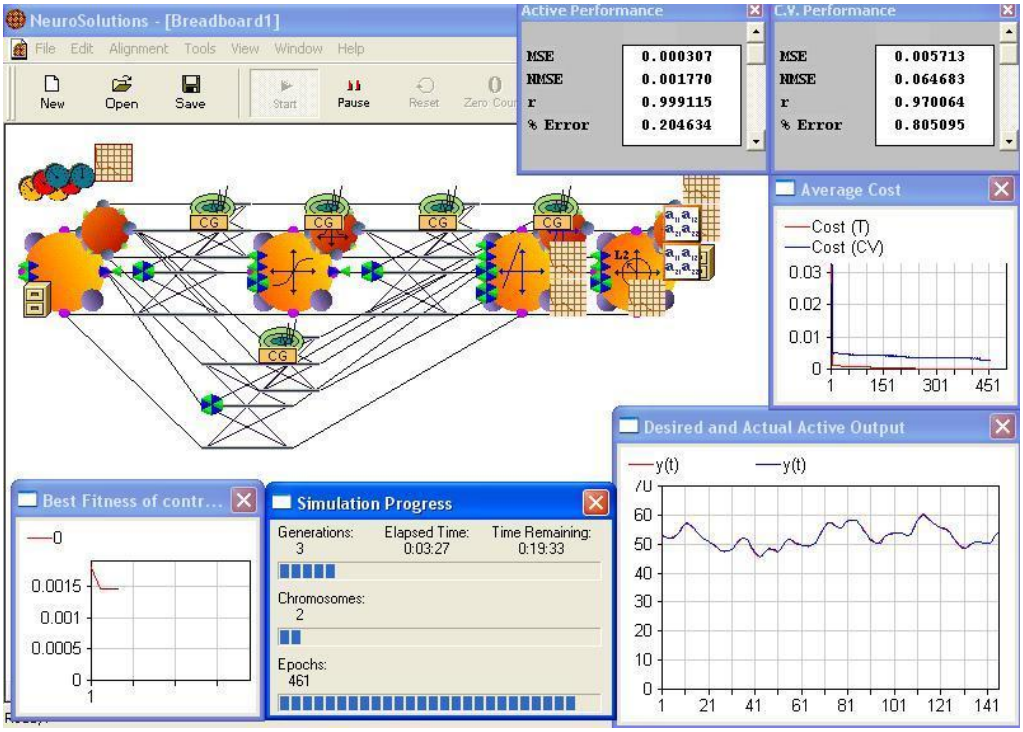


Figure A.2.9: Breadboard including generalized feed-forward network architecture and its screens while evolving and optimizing process

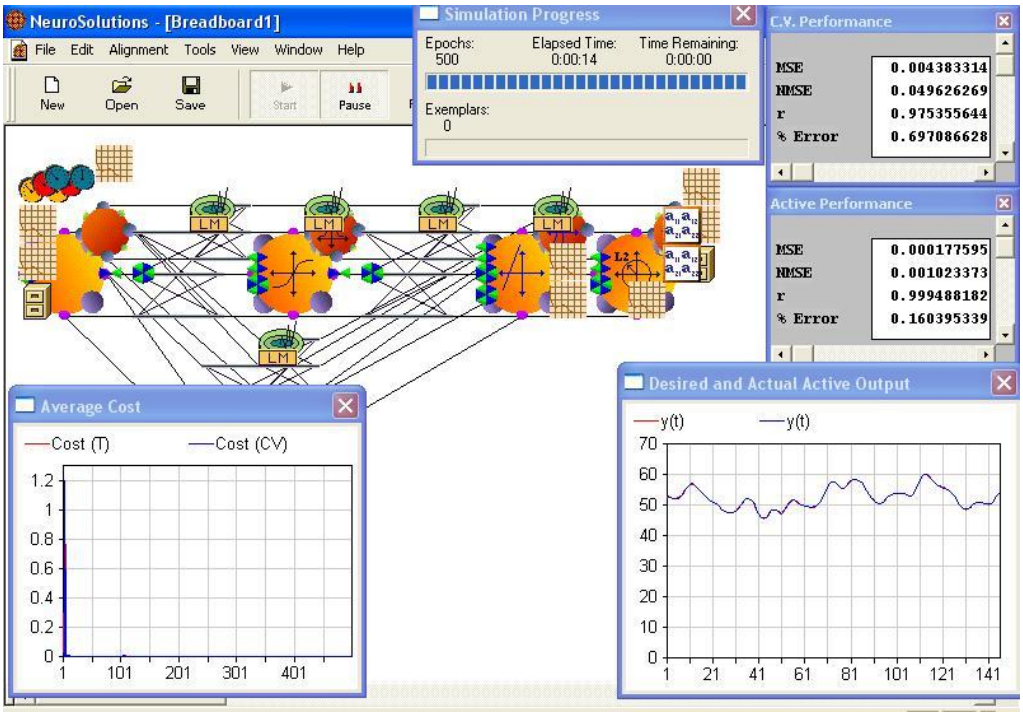


Figure A.2.10: Breadboard including generalized feed-forward network architecture and its screens while standard training process

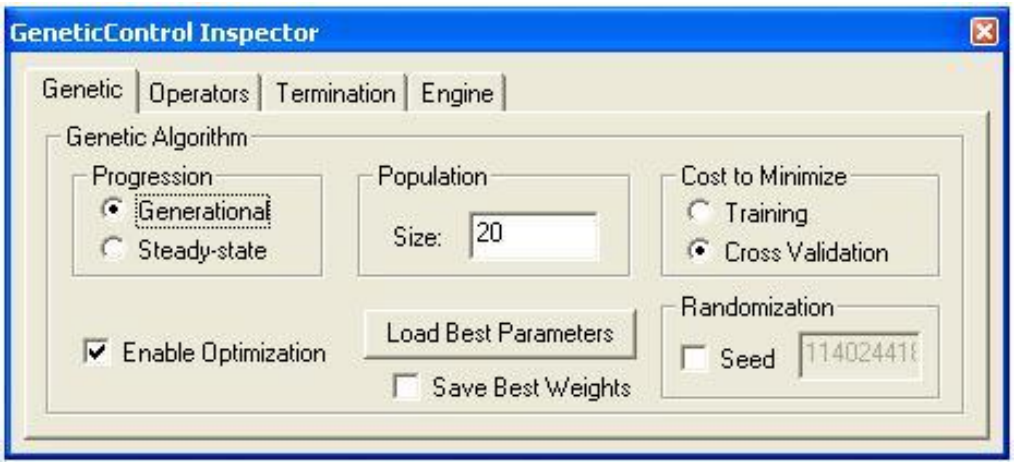


Figure A.2.11: Genetic Algorithm parameter: population size

Figure A.2.12: Genetic Algorithm operators: selection, crossover, mutation

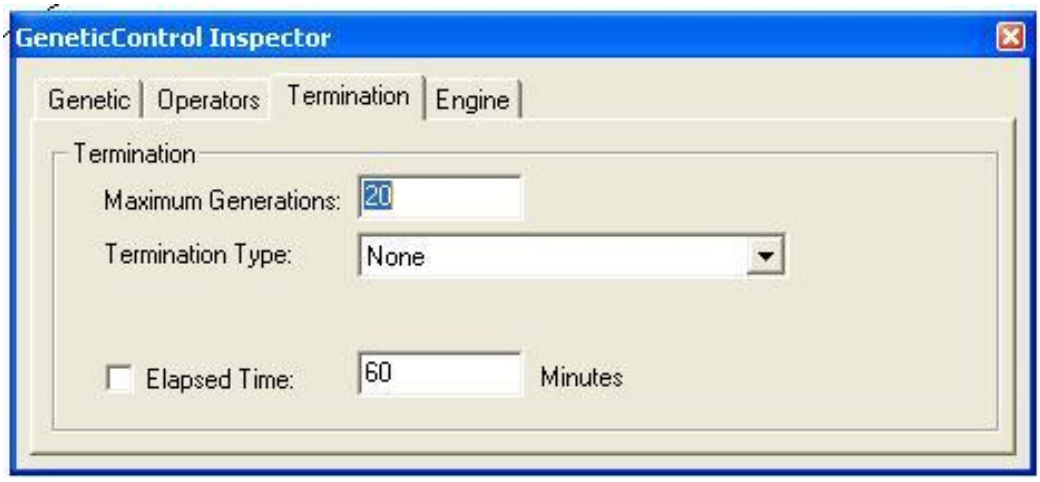


Figure A.2.13: Genetic Algorithm parameters: termination type and maximum generation

ملخّص

التعليم والتدريب المتعدد للشبكات العصبيّة الصناعيّة المتطورة: استخدام المحاكيات والنظم الخلويّة في تصميم هيكلية الشبكات

في هذه الرّسالة، قمت باقتراح نموذج للتعليم والتدريب المتعدد للشبكات العصبيّة الصناعيّة المتطورة بواسطة المحاكيات الخلويّة (MLEANN-CA). هذا النموذج عبارة عن بناء محوسب ومهيئ يعتمد على عملية تعليم متطورة و إجراءات بحث محلية لعمل تّصميم أوتوماتيكيّ لافضل الشّبكات العصبيّة الصّناعيّة باستخدام طّرق الترميز المباشرة و الغير مباشرة . في هذا البناء والنموذج المقترح، تم استخدام المحاكيات والنظم الخلويّة المتطورة (طريقة ترميز غير مباشرة) لعمل تصميم صغير الحجم لتركيبة وهيكلية الشبكات العصبية (الامامية التغذية). ثمّ بعد ذلك تم عمل تدريب و تطوير لهذه التراكيب الجديدة باستخدام خوارزمية التعليم المتعدد والمتكرر حيث تم تطبيق واستخدام اربع خوارزميات مختلفة بشكل متوازي ومنفصل وبالاستعانة بطريقة الترميز المباشرة المتطورة. ولقد تم تحديد هيكلية الشبكة العصبية الاساسية، الوسيلة المنشطة، اوزان الروابط بين عناصر الشبكة، و خوارزمية التعليم حسب طبيعة المشكلة . في هذه الدراسة، قمت بفحص وتقييم اداء النموذج المقترح من خلال تجارب عديدة استخدمت فيها اثنتين من اشهر المتسلسلات الزمنية الفوضوية و اثنتين من أدوات الفحص: NeuroSolutions و NeuroGenetic Optimizer. اضافة الى ذلك، قمت بعمل فحص واستكشاف لأداء عدد من خوارزميات التعليم الخاصة بالشبكات العصبية وذلك اثناء تغيير شكل وحجم هيكلية الشبكة وتطبيق المتسلسلات الزمنية عليها. واخيرا"، قمت بمقارنة النت ائج الخاصة بالنموذج المقترح مع نموذج سابق استخدم فيه طريقة الترميز المباشرة فقط ومع التصميم التقليدي للشبكات العصبية الصناعية. و قد اظهرت النتائج مدى كفاءة و فاعلية واتزان النموذج الذي اقترحناه في الحصول على تصاميم فعالة لشبكات عصبية صناعية صغيرة الحجم، سريعة التعلم، وباداء افضل واشمل.