YILDIZ TECHNICAL UNIVERSITY
FACULTY OF ELECTRICAL AND ELECTRONIC ENGINEERING
COMPUTER SCIENCE ENGINEERING DEPARTMENT

SENIOR PROJECT

GENETICALLY-OPTIMIZED RECURRENT NEURAL NETWORK MODEL FOR PARALLEL SYSTEM PERFORMANCE PREDICTION

Project Supervisor: Assist. Prof. Dr. Sırma YAVUZ

03011071 Ilir ÇOLLAKU

İstanbul, 2007

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF EQUATIONS

PREFACE

The aim of this project is to predict the perfomance of a parallel computer system using an Elman recurrent neural network model. Data about several attributes of computers - like CPU speed, problem dimension, arithmetic operation time, memory access, network connection and so on - are given the created network to train it in order to be able to predict future performance of the system.

The two methods of heuristic search, Genetic Algorithm and Simulated Annealing Algorithm are separately used to optimize the weights of the recurrent multi-layer neural network system.

I would like to express my thanks to the supervisor of this project, dear Mrs. Assist. Prof. Dr. Sırma Yavuz, for her help and support in all respects on every time.

ABSTRACT

This project is an implementation of Elman recurrent neural network model which is then optimized with Genetic Algorithm and Simulated Annealing Algorithm. These models are used to predict the arithmetic operation and communication performance of parallel systems using preceding data taken from them.

In opposition to multi-layer feed-forward networks, the output of a hidden unit on recurrent networks is sent back in order to be used as an input on the next step. Beside the input, hidden and output layer, a set of "context units" is added in the input layer here. There are connections from hidden layer to these context units with random weights or fixed with a value of one. At each time step, the input is propagated in a standard feed-forward fashion, and then a learning rule (usually back-propagation) is applied. The back connections result in the context units always maintaining a copy of the previous values of the hidden units (since they propagate over the connections before the learning rule is applied).

Since Elman network is basically trained with a standard back-propagation algorithm, there are trained the feed-forward connections only, and the feed-back connections are left as constant values. The right selection of these connection values is very important on training success of these networks, so in order to eliminate the limitations and make the training more effective, one of the best approaches is to use heuristic search algorithms which perceive the weights of the network as parameters.

So, first the Genetic Algorithm and then the Simulated Annealing Algorithm is used to train an Elman network. Finally the results are compared.

ÖZET

Bu projede temel genetik algoritma ve benzetilmiş tavlama algoritmaları ile optimize edilmiş basit geri dönüşümlü Elman ağı modeli gerçeklenmiştir. Gerçeklenen model gerçek veriler kullanılarak paralel sistemlerin aritmetik işlem ve haberleşme performansı tahmini için kullanılmıştır.

Çok katmanlı ileri beslemeli yapay sinir ağlarının aksine, geri dönüşümlü ağlarda, işlem elemanlarının çıktıları ağa belirli bir şekilde geri gönderilerek girdi olarak kullanılır. Girdi, ara katman ve çıktı elemanlarının yanı sıra bir de içerik elemanları vardır. İçerik elemanları, ara katman elemanlarının bir önceki aktivasyon değerlerini hatırlamak için kullanılırlar. Ağın bir t zamanındaki durumu, hem o andaki girdilere, hem de $t-1$ zamanındaki ara katman elemanlarının aktivasyon değerlerine bağlıdır. İleri doğru hesaplama yapıldıktan sonra oluşan ara katman elemanlarının aktivasyon değerleri, geriye doğru içerik elemanlarına gönderilir ve bir sonraki iterasyonda kullanılmak üzere saklanır.

Elman ağı temelde standart geriyayılım (back-propagation) öğrenme algoritması ile eğitilmektedir. Bu algoritmanın uygulanmasında, ağın sadece ileribesleme bağlantıları eğitilebilmekte, geribesleme bağlantıları ise, kullanıcının önceden deneme yanılma yoluyla belirlediği değerlerde sabit kalmaktadır. Bu ağlarda eğitme başarısı için, geribesleme bağlantı değerlerinin doğru seçilmesi oldukça önemlidir. Bu sınırlamaları ortadan kaldırarak ağın daha başarılı bir şekilde eğitilebilmesi için yapılan yaklaşımlardan birisi, ağdaki her bir ağırlık değerini birer parametre olarak algılayabilen dolayısıyla ileribesleme ya da geribesleme bağlantısı ayrımı yapmayan sezgisel algoritmaların eğitme amacıyla kullanılması olmuştur.

Bu projede, bu amaçla önce temel genetik algoritma kullanılmıştır. Daha sonra da etkili bir rasgele araştırma algoritması olan benzetilmiş tavlama algoritması gerçeklenerek sonuçlar kıyaslanmıştır.

1. INTRODUCTION

Before we move to the steps of the project let's see something about structures which are going to be used. The main structure here is an artificial neural network.

1.1. ARTIFICIAL NEURAL NETWORKS

As it was noticed, this project is a kind of implementation of neural network, so it is necessary to first take some knowledge about neural networks.

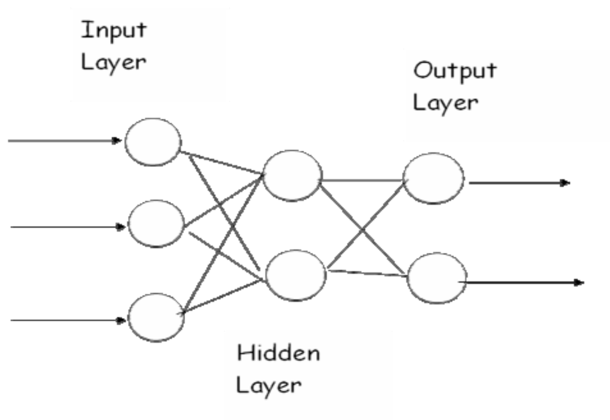


Figure 1.1 A simple neural network structure

1.1.1. What is a Neural Network?

An Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, process information. The key element of this paradigm is the novel structure of the information processing system. It is composed of a large number of highly interconnected processing elements (neurones) working in unison to solve specific problems. ANNs, like people, learn by example. An ANN is configured for a specific application, such as pattern recognition or data classification, through a learning process. Learning in biological systems involves adjustments to the synaptic connections that exist between the neurones. This is true of ANNs as well.

1.1.2. Historical Background

Neural network simulations appear to be a recent development. However, this field was established before the advent of computers, and has survived at least one major setback and several eras.

Many importand advances have been boosted by the use of inexpensive computer emulations. Following an initial period of enthusiasm, the field survived a period of frustration and disrepute. During this period when funding and professional support was minimal, important advances were made by relatively few reserchers. These pioneers were able to develop convincing technology which surpassed the limitations identified by Minsky and Papert. Minsky and Papert, published a book (in 1969) in which they summed up a general feeling of frustration (against neural networks) among researchers, and was thus accepted by most without further analysis. Currently, the neural network field enjoys a resurgence of interest and a corresponding increase in funding.

The first artificial neuron was produced in 1943 by the neurophysiologist Warren McCulloch and the logician Walter Pits. But the technology available at that time did not allow them to do too much.

1.1.3. Why Use Neural Networks?

Neural networks, with their remarkable ability to derive meaning from complicated or imprecise data, can be used to extract patterns and detect trends that are too complex to be noticed by either humans or other computer techniques. A trained neural network can be thought of as an "expert" in the category of information it has been given to analyze. This expert can then be used to provide projections given new situations of interest and answer "what if" questions. Other advantages include:

1. Adaptive learning: An ability to learn how to do tasks based on the data given for training or initial experience.
2. Self-Organisation: An ANN can create its own organisation or representation of the information it receives during learning time.
3. Real Time Operation: ANN computations may be carried out in parallel, and special hardware devices are being designed and manufactured which take advantage of this capability.
4. Fault Tolerance via Redundant Information Coding: Partial destruction of a network leads to the corresponding degradation of performance. However, some network capabilities may be retained even with major network damage.

1.1.4. Architecture of Neural Networks

There are several types of neural networks. The commonest type of artificial neural network consists of three groups (layers) of units: a layer of "input" units is connected to a layer of "hidden" units, which is connected to a layer of "output" units.

- The activity of the input units represents the raw information that is fed into the network.
- The activity of each hidden unit is determined by the activities of the input units and the weights on the connections between the input and the hidden units.
- The behaviour of the output units depends on the activity of the hidden units and the weights between the hidden and output units.

We also distinguish single-layer and multi-layer architectures. The single-layer organization, in which all units are connected to one another, constitutes the most general case and is of more potential computational power than hierarchically structured multi-layer organizations. In multi-layer networks, units are often numbered by layer, instead of following a global numbering

1.1.4.1 Feed-forward networks

Feed-forward ANNs allow signals to travel one way only; from input to output. There is no feedback (loops) i.e. the output of any layer does not affect that same layer. Feed-forward ANNs tend to be straight forward networks that associate inputs with outputs. They are extensively used in pattern recognition. This type of organization is also referred to as bottom-up or top-down.

1.1.4.2. Feedback networks

Feedback networks can have signals travelling in both directions by introducing loops in the network. Feedback networks are very powerful and can get extremely complicated. Feedback networks are dynamic; their 'state' is changing continuously until they reach an equilibrium point. They remain at the equilibrium point until the input changes and a new equilibrium needs to be found. Feedback architectures are also referred to as interactive or recurrent, although the latter term is often used to denote feedback connections in single-layer organisations.

1.1.5. Recurrent Neural Networks

Recurrent Neural Networks (RNN) have a closed loop in the network topology. They are developed to deal with the time varying or time-lagged patterns and are usable for the problems where the dynamics of the considered process is complex and the measured data is noisy. Specific groups of the units get the feedback signals from the previous time steps and these units are called *context* unit. The RNN can be either fully or partially connected. In a fully connected RNN all the hidden units are connected recurrently, whereas in a partially connected RNN the recurrent connections are omitted partially. Examples of recurrent neural networks are Hopfield networks, Regressive networks, Jordan-Elman networks, and Brain-State-In-A-Box (BSB) networks.

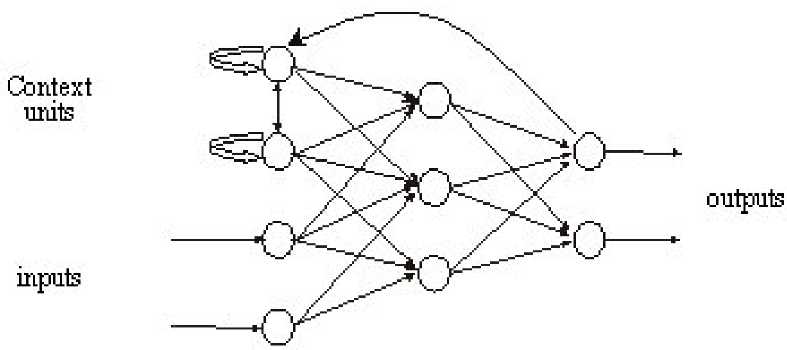


Figure 1.2 A recurrent neural network architecture

All types of recurrent neural networks are normally trained with the back-propagation learning rule by minimizing the error by the gradient descent method. Mostly they use some computational units which are called associative memories or context units, that can learn associations among dissimilar binary objects, where a set of binary inputs is fed to a matrix of resistors, producing a set of binary outputs. The outputs are '1' if the sum of the inputs is above a given threshold, otherwise it is zero. The weights (which are binary) are updated by using very simple rules based on *Hebbian* learning. These are very simple devices with one layer of linear units that maps N inputs (a point in N dimensional space) onto M outputs (a point in M dimensional space). However, they remember the past events.

1.1.5.1. Jordan-Elman Networks

Jordan and Elman networks combine the past values of the context unit with the present input (*x*) to obtain the present net output. The Jordan context unit acts as a so called *lowpass* filter, which creates an output that is the weighted (average) value of some of its most recent past outputs. The output (*y*) of the network is obtained by summing the past values multiplied by the scalar parameter τ^{n}. The input to the context unit is copied from the network layer, but the outputs of the context unit are incorporated in the net through their adaptive weights.

$$y(n) = \sum_{i=0}^{n} x(n)\,\tau^{n-i} \tag{1.1}$$

In these networks, the weighting over time is inflexible since we can only control the time constant (i.e. the exponential decay). Moreover, a small change in time is reflected as a large change in the weighting (due to the exponential relationship between the time constant and the amplitude). In general, we do not know how large the memory depth should be, so this makes the choice of τ problematic, without having a mechanism to adopt it.

In linear systems, the use of past input signals creates the moving average (MA) models. They can represent signals that have a spectrum with sharp valleys and broad peaks. The use of the past outputs creates what is known as the *autoregressive* (AR) models. These models can represent signals that have broad valleys and sharp spectral peaks. The Jordan net is a restricted case of a non-linear AR model, while the configuration with context units fed by the input layer is a restricted case of non-linear MA model. Elman's net does not have a counterpart in linear system theory. These two topologies have different processing power.

1.2. GENETIC ALGORITHM

A Genetic Algorithm (GA) is a heuristic search technique used in computing to find true or approximate solutions to optimization and search problems. Genetic algorithms are a particular class of evolutionary algorithms that use techniques inspired by evolutionary biology such as inheritance, mutation, selection, and cross-over (also called mating or recombination).

The genetic algorithm is a method for solving both constrained and unconstrained optimization problems that is based on natural selection, the process that drives biological evolution. The genetic algorithm repeatedly modifies a population of individual solutions. At each step, the genetic algorithm selects individuals at random from the current population to be parents and uses them produce the children for the next generation. Over successive generations, the population "evolves" toward an optimal solution. The genetic algorithm can solve a variety of optimization problems that are not well suited for standard optimization algorithms, including problems in which the objective function is discontinuous, nondifferentiable, stochastic, or highly nonlinear.

The genetic algorithm uses three main types of rules at each step to create the next generation from the current population:

- Selection rules select the individuals, called parents, that contribute to the population at the next generation.
- Cross-over rules combine two parents to form children for the next generation.
- Mutation rules apply random changes to individual parents to form children.

Popular and well-studied selection methods include roulette wheel selection and tournament selection. There can be several methods of cross-over. Three basic methods are uniform, one-point and two-point cross-over.

1.3. SIMULATED ANNEALING ALGORITHM

Simulated Annealing (SA) is a generic probabilistic meta-algorithm for the global optimization problem, namely locating a good approximation to the global optimum of a given function in a large search space. It was independently invented by S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi in 1983, and by V. Černý in 1985. It originated as a generalization of a Monte Carlo method for examining the equations of state and frozen states of n-body systems.

The name and inspiration come from annealing in metallurgy, a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects. The heat causes the atoms to become unstuck from their initial positions (a local minimum of the internal energy) and wander randomly through states of higher energy; the slow cooling gives them more chances of finding configurations with lower internal energy than the initial one.

By analogy with this physical process, each step of the SA algorithm replaces the current solution by a random "nearby" solution, chosen with a probability that depends on the difference between the corresponding function values and on a global parameter T (called the *temperature*), that is gradually decreased during the process. The dependency is such that the current solution changes almost randomly when T is large, but increasingly "downhill" as T goes to zero. The allowance for "uphill" moves saves the method from becoming stuck at local minima – which are the bane of greedier methods.

In the simulated annealing (SA) method, each point s of the search space is compared to a state of some physical system, and the function $E(s)$ to be minimized is interpreted as the internal energy of the system in that state. Therefore the goal is to bring the system, from an arbitrary *initial state*, to a state with the minimum possible energy. At each step, the SA heuristic considers some neighbor s' of the current state s, and probabilistically decides between moving the system to state s' or staying put in state s. The probabilities are chosen so that the system ultimately tends to move to states of lower energy. Typically this step is repeated until the system reaches a state which is good enough for the application, or until a given computation budget has been exhausted.

2. FEASIBILITY ANALYSIS

This application is planned to be completed on about 10 weeks (50 work-days) including here all the steps needed for software development. First 4 weeks are dedicated for research and preliminary studies about recurrent artificial neural networks and algorithms which have to be used to train the network, and also their earlier performances on this kind of implementation. On the following two weeks the system model structure has to be designed, the related diagrams have to be drawn, all these based on the scenario described earlier. Then the classes have to be created and the coding phase has to take the next three weeks. On the final week the results will be compared and other analysis have to be done.

2.1. Technical and Economical Feasibility

The program is planned to be written on Java. The minimum system requirements for the application are as follows:

Table 2.1 Hardware requirements

Equipment	Attribute	Cost
CPU	1.8 GHz	60 $
Motherboard	400 MHz FSB	40 $
RAM	512 MB	20 $
Monitor	15”	40 $
Video Card	64 MB	25 $
Hard Disk	40 GB	20 $

Since there will be only a single Java application, a computer with minimal system configuration will be enough to run it. The total cost for the hardware would be about 250 $, including here other necessary accessories like input devices (keyboard, mouse), PC case, etc.

As for software requirements, there will be enough for operating system to have a JVM (Java Virtual Machine) – JRE (Java Run-time Environment) and JDK (Java Development Kit) installed – whatever is it (Linux or Windows), of course, it is more logical to use the cheaper one.

Table 2.2 Software requirements

Type	**Name**	**Cost**
Operating System	Linux	FREE
Environment/Compiler program	JRE	FREE

The project has to be developed by one person working 4 hours daily with a cost 5$ per hour. The total labor cost for this project is expected to be: (10 weeks)*(5 days)*(4 hours)*(5 $) = 1000 $.

3. THE ELMAN RECURRENT NEURAL NETWORK STRUCTURE AND ITS' IMPLEMENTATION

As it was noticed earlier, the characteristic of an Elman network is the addition of copies of the hidden unit values to its' input layer which already contains the real inputs. In this way the number of neurons on the input layer is increased by the number of neurons on the hidden layer, which will act as input units at further steps of training in order to get better results. Before we pass to an implementation of the Elman net, let us first see its' structure and workflow.

3.1. The Elman Recurrent Neural Network Structure

A simple recurrent Elman net consists of three layers: input, hidden and output; where each neuron of a layer is connected to those of the subsequent one and vice versa, i.e. the connections between two layers, one with *m* units and the other with *n* units, can be represented by an *m*-to-*n* matrix.

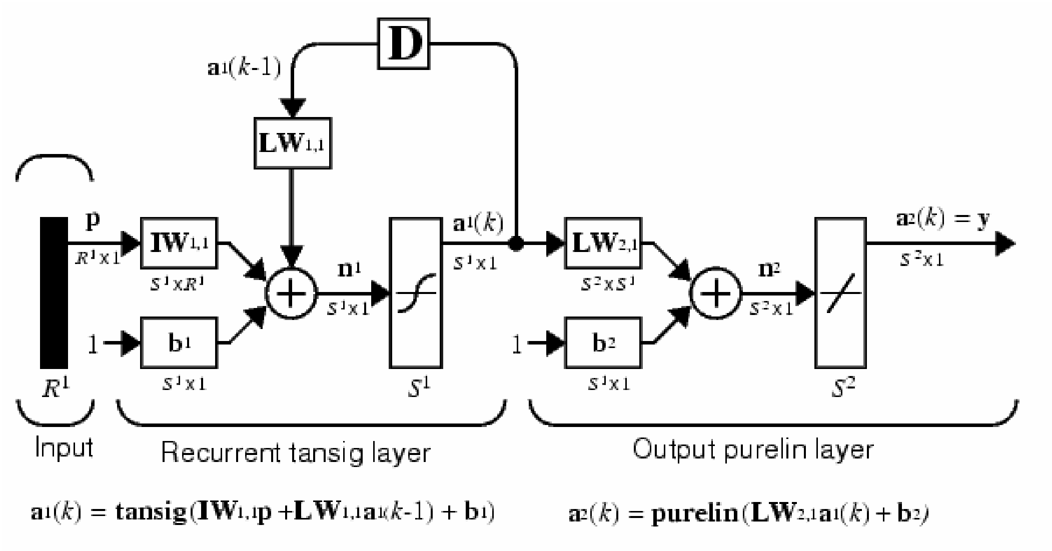


Figure 3.1 Elman recurrent neural network model structure

The Elman network has sigmoid or tangent hyperbolic neurons in its hidden (recurrent) layer, and linear neurons in its output layer. This combination is special in that two-layer networks with these transfer functions can approximate any function (with a finite number of discontinuities) with arbitrary accuracy. The only requirement is that the hidden layer must have enough neurons. More hidden neurons are needed as the function being fitted increases in complexity.

Note that the Elman network differs from conventional two-layer networks in that the first layer has a recurrent connection. The delay in this connection stores values from the previous time step, which can be used in the current time step. Thus, even if two Elman networks, with the same weights and biases, are given identical inputs at a given time step, their outputs can be different because of different feedback states. Because the network can store information for future reference, it is able to learn temporal patterns as well as spatial patterns. The Elman network can be trained to respond to, and to generate, both kinds of patterns.

The input values are given from the outside, so it is not necessary to keep them in a special data structure, therefore it will be enough to keep them on an array which will be used to train the network. The length of this array has to be equal to the number of input units plus the number of hidden units.

The output data are also given from the outside to train the network, while on the other hand the network adjustes the weights between layers and gives the predicted output values. Training the network can be done based on several patterns, so in order to understand and make the operations easier, it is appropriate to use two dimensional arrays to keep all these data.

As it is shown on the figure below, there are two matrices which will be used on training the net, *trainInputs[][]* and *trainOutputs[][]*. Throughout the training phase, the weights of synapses between input-hidden layer; and hidden-output layer are adjusted on each epoch. The weight values are also kept on matrices, *weightsIH[][]* and *weightsHO[][]*. Finally, there are two more structures, one for hidden unit values *hiddenVal[]* and the other for predicted output values *outPred[]*, all these calculated based on the error rate of one step earlier kept on *errThisPat[]*.

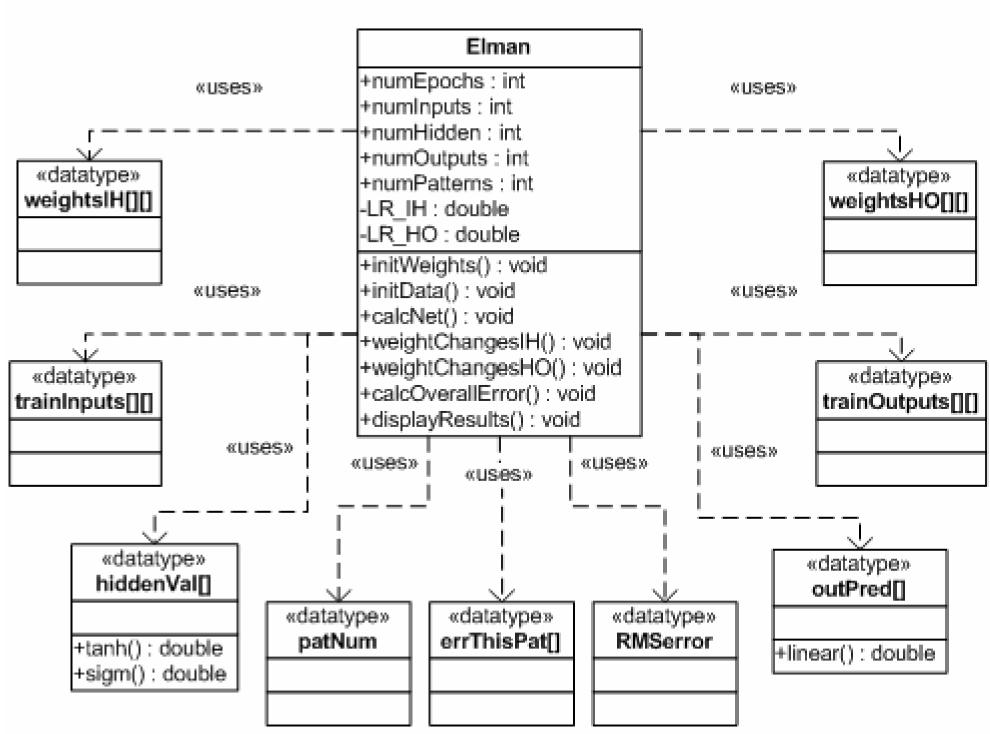


Figure 3.2 Elman recurrent neural network model class-diagram

The other data types used on the application are *numEpochs* (number of training epochs), *numInputs* (input neurons), *numHidden* (hidden neurons), *numOutputs* (outputs), *numPatterns* (input-output patterns), *LR_IH* (input-hidden synapse learn ratio), *LR_HO* (hidden-output synapse learn ratio) and *RMSerror* (root mean square error) or *PRCerror* (percentage error)

$$RMS = \sqrt{\frac{\sum_i errThisPat_i^2}{numPatterns}} \tag{3.1}$$

$$PRC = \frac{\sum_i (errThisPat_i * 100 / trainOutputs_{patNum,i})}{numPatterns} \tag{3.2}$$

3.2. The Backpropagation Training Algorithm

As it was discussed earlier, for multilayer networks the output of one layer becomes the input to following layer. The equations that describe this operation are

$$a^{m+1} = f^{m+1}(W^{m+1}a^m + b^{m+1}) \quad \text{for} \qquad m = 0,1,...,M-1 \tag{3.3}$$

where M is the number of layers in the network. The layers in the first layer receive external inputs:

$$a^0 = p = trainInputs_i \tag{3.4}$$

which provides the starting point for Equation 3.3. The outputs of the neurons in the last layer are considered the network outputs:

$$a = a^M = out\,\mathrm{Pr}\,ed_i \tag{3.5}$$

The backpropagation algorithm uses a mean square error as a performance index. The algorithm is provided with a set of examples of proper network behavior:

$$\{p_1, t_1\}, \{p_2, t_2\}, ..., \{p_Q, t_Q\} \tag{3.6}$$

or $\{trainInputs_{patNum,1}, trainOutputs_{patNum,1}\}, ..., \{trainInputs_{patNum,Q}, trainOutputs_{patNum,Q}\}$

where p is an input and t is a corresponding target output of the network. And the error which has to be minimized is the RMSerror of Equation 3.1, expressed here as:

$$F(x) = E[e^2] = E[(t-a)^2] \tag{3.7}$$

The steepest descent algorithm for the approximate RMSerror is

$$w_{i,j}^m(k+1) = w_{i,j}^m(k) - \alpha \frac{\partial F}{\partial w_{i,j}^m}, \tag{3.8}$$

$$b_i^m(k+1) = b_i^m(k) - \alpha \frac{\partial F}{\partial b_i^m} \tag{3.9}$$

where *w*’s are the weights of the synapses, *b*’s are the bias and α is the learning rate.

By applying the partial derivatives of functions using the chain rule, it becomes:

$$W^m(k+1) = W^m(k) - \alpha s^m (a^{m-1})^T, \tag{3.10}$$

$$b^m(k+1) = b^m(k) - \alpha s^m \tag{3.11}$$

where
$$s^m \equiv \frac{\partial F}{\partial n^m} \tag{3.12}$$

is a backpropagation sensitivity which will be used on adjusting the wights and n's are the output values of neurons. Then another application of the chain rule on the partial derivative of the error function gives us the Jacobian matrix which can be written as:

$$\frac{\partial n^{m+1}}{\partial n^m} = W^{m+1} F^m(n^m) \tag{3.13}$$

where
$$F^m(n^m) = \begin{bmatrix} f^m(n_1^m) & 0\ldots & 0 \\ 0 & f^m(n_2^m) & 0 \\ 0 & 0 & f^m(n_{s^m}^m) \end{bmatrix} \tag{3.14}$$

$$s^m = F^m(n^m)(W^{m+1})^T s^{m+1} \tag{3.15}$$

Now we can see where the backpropagation algorithm derives its name. The sensitivities are propagated backward through the network from the last layer to the first layer:

$$s^M \to s^{M-1} \to \ldots \to s^2 \to s^1. \tag{3.16}$$

Finally, the starting point s^M for the recurrence relation of Equation 3.15 is obtained at the final layer by taking a partial derivative of an error function on the last layer output, and is expressed as

$$s^M = -2F^M(n^M)(t-a) \tag{3.17}$$

3.3. The Elman Recurrent Neural Network Training Algorithm

To better understand the mathematical operations on the Elman net, it would be useful to take a glance at the flow diagrams designed in here.

The first one is the main program. After initializing the data and weight values, on the first step of training, the methods are called once for each pattern in order to assign the first values to the hidden units. Then these values are copied to the additional units of the input layer, and beginning from the second step of the training, this procedure is done on every step.

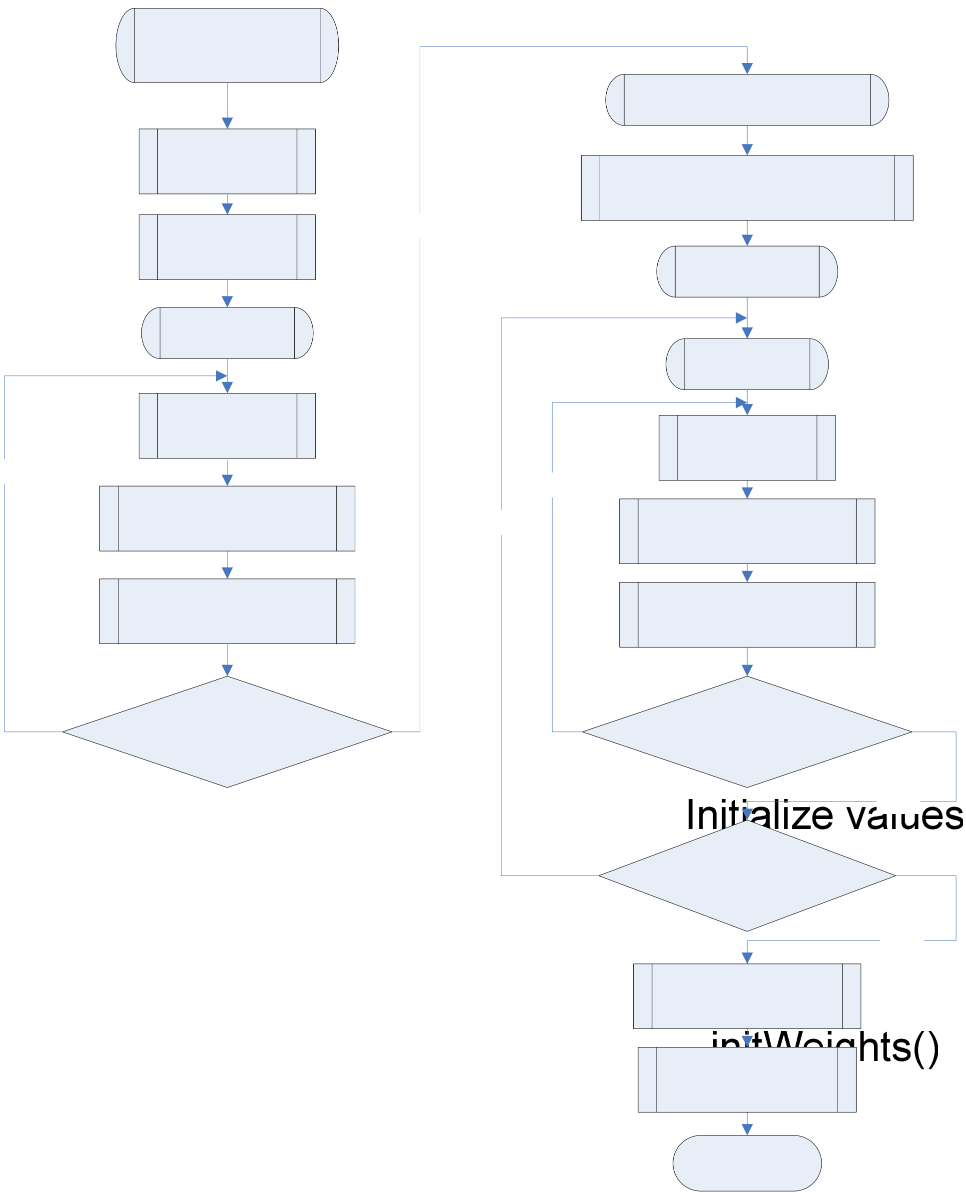


Figure 3.3 The main program flow-diagram

The *calcNet()* method is called on each training epoch, to calculate the values of hidden and output values of the network. The activation functions used for the hidden units are *tanh* (Tangent Hyperbolic) and/or *sigm* (Sigmoid).

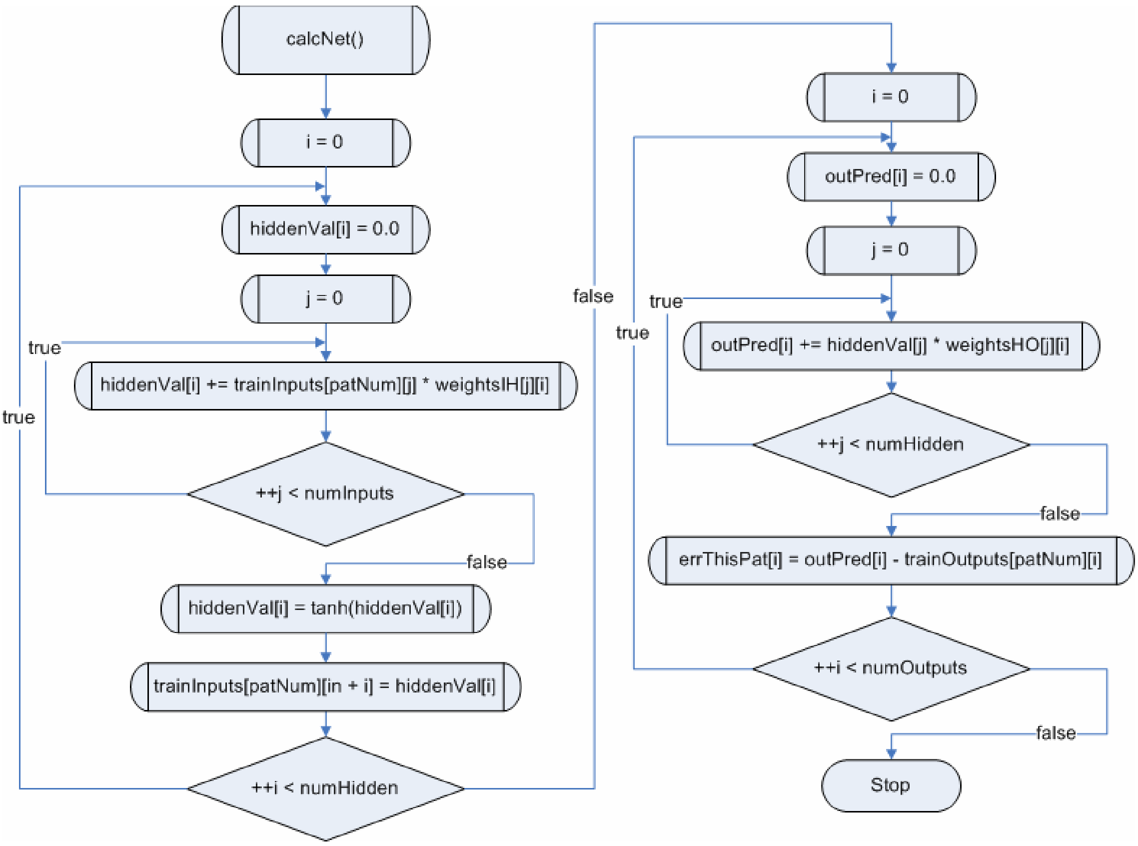


Figure 3.4 Flow-diagram for a method calcNet()

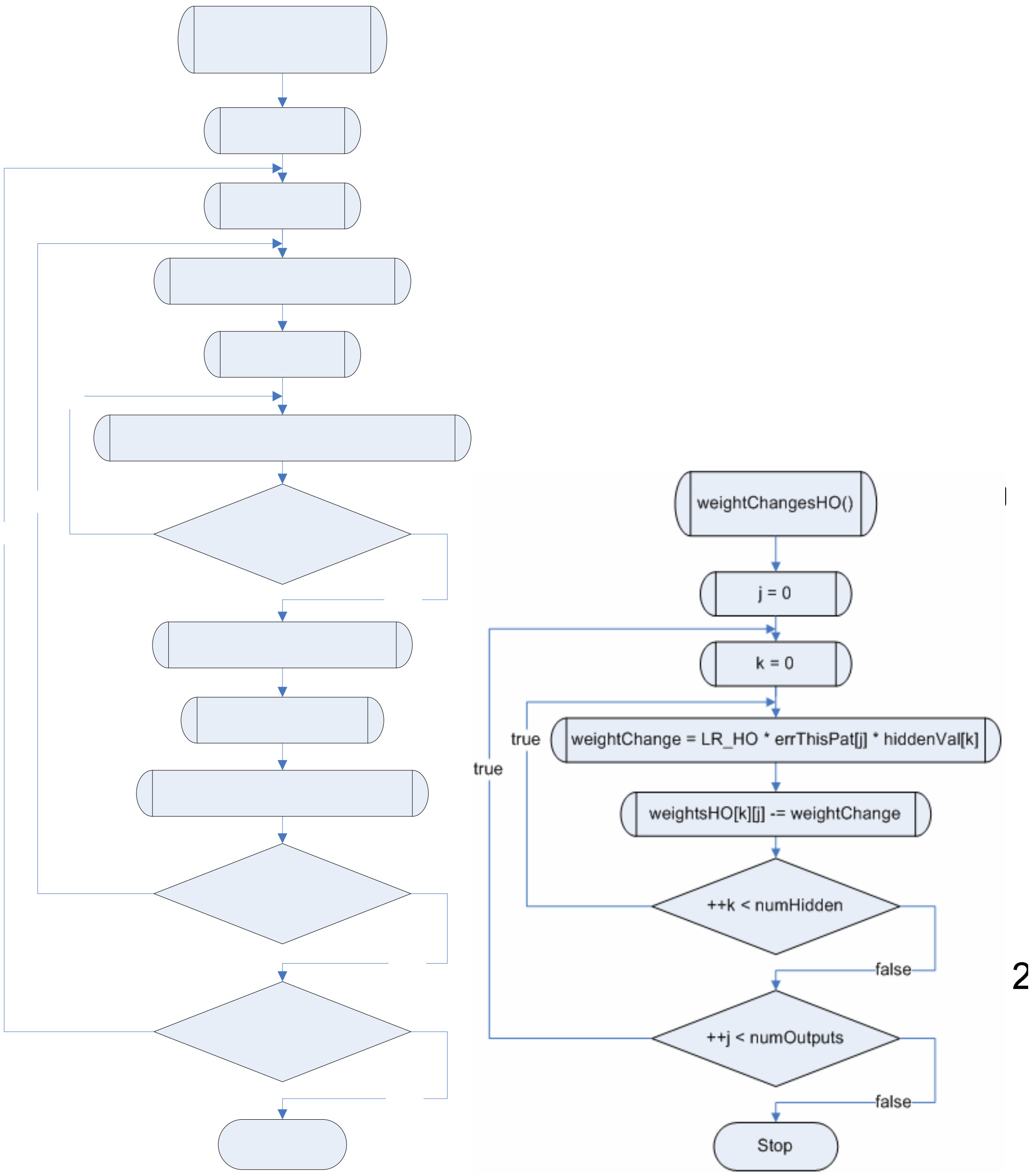
The tangent hyperbolic activation function is used most commonly when the output values are supposed to be bipolar (-1,1)

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.18}$$

The sigmoid activation function is used for the positive output values

$$sigm(x) = \frac{1}{1 + e^{-x}} \tag{3.19}$$

The weight-change methods also depend to the activation functions. As the hidden neurons are tanh and the output neurons are linear here, the flow diagrams of *weightChangesIH()* and *weightChangesHO()* look like on the figure below.



true

Figure 3.5 Methods used to calculate the weight changes on each epoch, weightChangesIH() and weightChangesHO()

x *= weightsHO[i][j] * errThisPat[j] * LR_IH

true

true

++j < numOutputs

false

3.4. Implementation of the Elman Recurrent Neural Network Model to the XOR Problem

As it was noticed earlier, a neural network can be taught to recognize different functions or patterns by adjusting the weights of the neuron connections. The goal here is to teach our neural network model to learn the XOR problem. The learning algorithm used on training the Elman net is ususally a standard back-propagation algorithm, which trains the feed-forward connections only. The initial weights of these connections can be generated randomly or given manually. The learning rates can also be given manually.

The model used in here is one with two input units (two input values of binary 0 and 1), four hidden neurons and one output. After the first epoch, the copies of hidden neurons are made to act as input units, so the number of input units is increased to seven. The number of patterns used in here is equal to the number of input combinations (0-0, 0-1, 1-0 and 1-1) of the XOR operation. The weights between input layer and hidden layer on the epoch t are calculated in relation with the activation function of the hidden neurons (sigm or tanh), the output values, error ratio on $t-1$ and input-hidden learning rate which is set to 0.5, while the hidden-output learning rate is supposed to be several times less (here it is set to 0.2). The initial values of the weights are assigned random values.
The net is trained with 200 epochs. Depending on the activation functions used on the hidden layer the results below were taken. Meanwhile, the activation function for the outputs was set to linear.

Table 3.1 Elman network model training results for the XOR problem

Activation function	RMSerror	PRCerror
Tangent hyperbolic	3.193677903391579E-11	2.1654660388069725E-8
Sigmoid	1.0017472443744082E-8	3.9129976506488164E-6

As it is shown on the table, the results for both activation functions, tanh and sigm, are acceptable. We will analyze these results in more details later in Section 6, Experimental Results.

4. USING GENETIC ALGORITHM TO OPTIMIZE THE ELMAN NETWORK

In Section 3.2 back-propagation was introduced. Back-propagation is a very effective means of training a neural network. However, there are some inherent flaws in the back-propagation training algorithm. One of the most fundamental flaws is the tendency for the back-propagation training algorithm to fall into a “local minima”. A local minimum is a false optimal weight matrix that prevents the back-propagation training algorithm from seeing the true solution.

In this section we will see how we can use genetic algorithm (GA) to supplement back-propagation and elude local minima by seeking a more optimal solution, if one does exist. The genetic algorithm theory was introduced in Section 1.2, and now we will see its structure and operations.

Genetic algorithm works by generating new individuals on the population created at the beginning. Every individual is a complete solution for the problem where the algorithm is used and is represented by a chromosome. Chromosomes are consisted of genes, which are, depending on the problem nature, the individual components of a solution. Determining a way to break a problem into related components (genes) is a very important part of the analysis of the problem that is to be used with a genetic algorithm. Here, on the neural networks, the set of all weights and bias is represented by a chromosome and each weight or bias value is a gene.

4.1. How Genetic Algorithms Work?

Now that we have seen the structure of a genetic algorithm, we will proceed to discuss how genetic algorithms actually work. A genetic algorithm begins by creating an initial population. This population consists of chromosomes that are given a random collection of genes. The steps involved in a genetic algorithm are as follows:

1. Create an initial population of chromosomes
2. Evaluate the fitness or "suitability" of each chromosome that makes up the population
3. Based on this fitness, select the chromosomes that will mate or those that have the "privilege" to mate
4. Cross-over or mate the selected chromosomes and produce offspring
5. Randomly mutate some of the genes of the chromosomes
6. Repeat steps three through five until a new population is created
7. The algorithm ends when the best solution has not changed for a preset number of generations

Genetic algorithms strive to determine the optimal solution to a problem by utilizing three genetic operators. These operators are selection, cross over, and mutation. GAs' search for the optimal solution until specific criteria is met causing termination. These results include providing good solutions as compared to one "optimal" solution for complex (such as "NP hard" or non-polynomial hard) problems. NP-hard defers to a problem which cannot be solved in polynomial time. Most problems solved with computers today are not NP-hard and can be solved in polynomial time. A P-problem or polynomial problem is a problem where the number of steps to complete the answer is bounded by a polynomial. A polynomial is a mathematical expression involving exponents and expressions. A NP-hard problem does not increase exponentially. An NP-hard problem often increases at a much greater rate, often described by the factorial operator (n!). One example of an NP-hard problem is the traveling salesman problem.

As it was noticed earlier, in a genetic algorithm, the population is comprised of organisms. Each of these organisms is composed of the single chromosome which represents one complete solution to the defined problem. On the initial population the genes of the chromosomes are usually initialized to random values based on the boundaries defined.

4.1.1. Calculating Fitness

Once the population is initialized, the fitness (suitability) for each organism has to be calculated. These is done by transforming genes of the chromosome to the weights and bias of the neural network, and calling the *CalcNet()* function defined earlier to calculate the outputs for each layer. Finally the *RMSerror* calculated here is a fitness value of the related chromosome.

Based to their fitness, the chromosomes inside the population are sorted beginning from that with a smallest fitness (*RMSerror* here, which will be minimized) which also represents the best solution to the neural network.

4.1.2. Mating

Usually the first few chromosomes (1/4 from the top of the population) are selected as most favored mating individes which have to mate with theirself or with the other quarter (these together form the group of mating chromosomes), while the other half of population is intented to die. This is called tournament selection.

The cross-over (mating) process is done by simply taking the two chromosomes which are going to mate and selecting two cut points. On this way both mating chromosomes are divided into three pieces. There would be created two new chromosomes (offspring) now, one taking its first and third part from the first parent and the second part from the second parent, and another taking the opposite parts.

This method of crossing-over can lead us to the problem of no new genetic material being produced, so to escape this probability we have to mutate the children when created.

4.1.3. Mutation

Mutation allows new genetic patterns to be introduced that were not already contained in the population. The main parameter used here is the *mutationRate* which is taken from the user. This parameter simply decides how many genes on the new created chromosome have to be changed/mutated. These genes are selected randomly and replaced with the random values.

It is practical to choose the mutation rate somewhere between 10% and 30%. If the high mutation rate is chosen, it will be performing nothing more than a random search.

5. USING SIMULATED ANNEALING ALGORITHM TO OPTIMIZE THE ELMAN NETWORK

There was introduced in Section 1.3 the simulated algorithm theory. Now we will examine this another technique to train and optimize our neural network model. Simulated annealing has become a popular method of neural network training.

5.1. The Simulated Annealing Algorithm Usage Areas

Simulated annealing can be used to find the minimum of an arbitrary equation that has a specified number of inputs. It will find the inputs to the equation that will produce a minimum value. In the case of a neural network, this equation is the error function of the neural network.

When simulated annealing was first introduced the algorithm was very popular for integrated circuit (IC) chip design. Most IC chips are composed internally of many logic gates. Simulated annealing is often used to find an IC chip design that has fewer logic gates than the original. This causes the chip to generate less heat and run faster.

The weight and bias matrix of a neural network makes for an excellent set of inputs for the simulated annealing algorithm to minimize for. Different sets of weights and bias are used for the neural network, until one is found that produces a sufficiently low return from the error function.

5.2. The Simulated Annealing Algorithm Structure

We will now examine the structure of the simulated annealing algorithm. There are several distinct steps that the simulated annealing process goes through as the temperature is decreased, and randomness is applied to the input values. Figure 5.1 shows this process as a flowchart.

There are two major processes that are occurring during the simulated annealing algorithm. First, for each temperature the simulated annealing algorithm runs through a number of cycles. This number of cycles is predetermined by the programmer. As the cycle runs the inputs are randomized. Only randomizations which produce a better suited set of inputs will be kept.

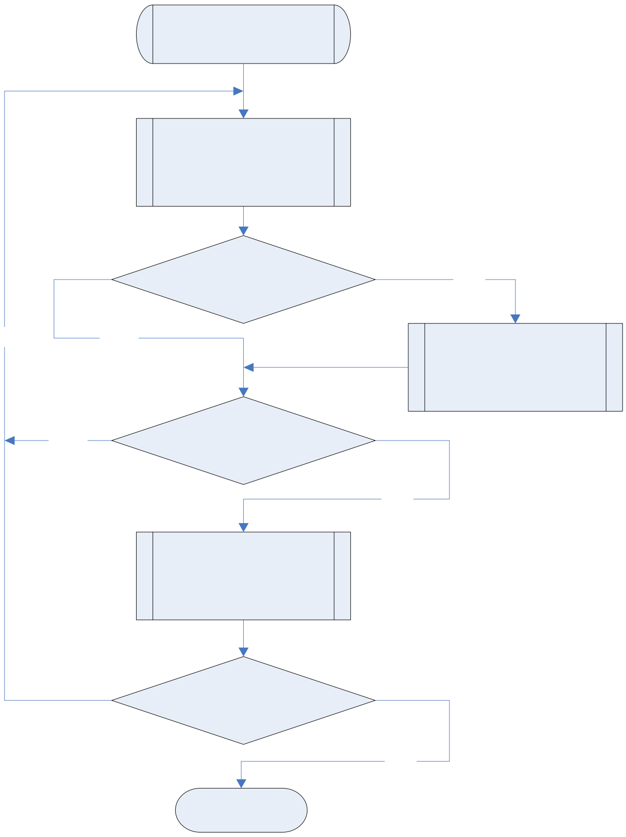


Figure 5.1 The simulated annealing process flow-chart

Once the specified number of training cycles has been completed, the temperature can be lowered. Once the temperature is lowered, it is determined of the temperature has reached the lowest allowed temperature. If the temperature is not lower than the lowest allowed temperature, then the temperature is lowered and another cycle of randomizations will take place. If the temperature is lower than the minimum temperature allowed, the simulated annealing algorithm is completed.

At the core of the simulated annealing algorithm is the randomization of the input values. This randomization is ultimately what causes simulated annealing to alter the input values that the algorithm is seeking to minimize. This randomization process must often be customized for different problems. In the next section we will examine how this randomization occurs.

To apply the simulated annealing algorithm to a neural network we simply treat the weights and bias of the neural network as the individual ions/atoms in the metal like were the genes of chromosomes on genetic algorithms. As the temperature falls, the weights of the neural network will achieve less excited states. As this process progresses the most optimal weight matrix is chosen, based on the error of the neural network.

A neural network's weight matrix can be thought of as a linear array of floating point numbers. Each weight is independent of the others. It does not matter if two weights contain the same value. The only major constraint is that there are ranges that all weights must fall within.

Because of this the process generally used to randomize the weight matrix of a neural network is relatively simple. Using the temperature, a random ratio is applied to all of the weights in the matrix. This ratio is calculated using the temperature and a random number. The higher the temperature, the more likely the ratio will cause a larger change in the weight matrix. A lower temperature will most likely produce a smaller ratio.

5.2.1. The Input Matrix Randomization

An important part of the simulated annealing process is how the inputs are randomized. This randomization process takes the previous values of the inputs and the current temperature as inputs. The input values are then randomized according to the temperature. A higher temperature will result in more randomization, while a lower temperature will result in less randomization.

There is no exact method defined by the simulated annealing algorithm for how to randomize the inputs. The exact nature by which this is done often depends on the nature of the problem being solved.

5.2.2. Temperature Reduction

There are several different methods that can be used for temperature reduction. The most common is to simply reduce the temperature by a fixed amount through each cycle.

Another method is to specify a beginning and ending temperature. This is the method that is used by the simulated annealing algorithm to train a neural network. To do this we must calculate a ratio at each step in the simulated annealing process. This is done by using an equation that guarantees that the step amount will cause the temperature to fall to the ending temperature in the number of cycles requested. The following equation shows how to logarithmically decrease the temperature between a beginning and ending temperature.

$$ratio = \ln \frac{\log_{10} \dfrac{stopTemperature}{startTemperature}}{cycles - 1} \tag{5.1}$$

Equation 5.1 calculates a ratio that should be multiplied against the current temperature. This will produce a change that will cause the temperature to reach the ending temperature in the specified number of cycles.

6. EXPERIMENTAL RESULTS

In this section we will first implement the XOR problem, and then we will train the network with performance data from a parallel system in order to be able to predict the future performance of the system. On the first step for both problems, the neural network (which will be an Elman or a simple backpropagation model) will be trained until the acceptable result is achieved. On the second step the genetic algorithm will be used to optimize the not-well trained neural network, and on the third step the simulated annealing algorithm will be used and the results will be compared. The optimization here means the escaping from the local minima, so it must be done before the network is trained for too much epochs.

6.1. The XOR Problem Results

The Elman network is trained with XOR patterns and then the simple backpropagation network is trained with the same patterns. It is concluded that at the same circumstances an Elman net is more successful than the simple backpropagation net because among the training phase it can remember values from the previous step. Both Elman and simple backpropagation networks have the same structure here, with 2 input units, 4 hidden units and 1 output unit, except an Elman net has weights between hidden layer and previous hidden layer. The results for both neural network structures are:

Table 6.1 XOR problem with Elman net

```
------------------------------------------------------------------------------------------
Network structure: Elman; Epochs: 100; Training patterns: 4
Function: sigm; Learning Rate IH: 0.5; Learning Rate HO: 0.5
------------------------------------------------------------------------------------------
pattern 1 : actual = 0.0 ; neural model = 2.599468829018736E-4
pattern 2 : actual = 1.0 ; neural model = 1.000208116342466
pattern 3 : actual = 1.0 ; neural model = 0.9998675564381211
pattern 4 : actual = 0.0 ; neural model = -1.7784932485009897E-4
==========================================================
RMS error: 2.000352927389775E-4     % error: 0.00117189049764016
------------------------------------------------------------------------------------------
```

Test patterns: 4

----

pattern 1 : actual = 0.0 ; neural model = 2.632502531191294E-4

pattern 2 : actual = 1.0 ; neural model = 1.0001606461371022

pattern 3 : actual = 1.0 ; neural model = 0.999854639828709

pattern 4 : actual = 0.0 ; neural model = -1.7796729846647485E-4

==

Test RMS: 1.9229522605510783E-4   Test %: 0.0011725907672351856

As it is seen, the train error and test error values aren’t exactly equal, so from here we can conclude that even if two Elman networks, with the same weights and biases, are given identical inputs at a given time step, their outputs can be different because of different feedback states. But this conclusion isn’t valid for the simple backpropagation network. These are the results for the gradient-descent backpropagation net.

Table 6.2 XOR problem with simple Back-propagation network

Network structure: Backpropagation; Epochs: 100; Training patterns: 4

Function: sigm; Learning Rate IH: 0.5; Learning Rate HO: 0.5

----

pattern 1 : actual = 0.0 ; neural model = -3.854064398872703E-4

pattern 2 : actual = 1.0 ; neural model = 0.999408116956466

pattern 3 : actual = 1.0 ; neural model = 0.9988844859983493

pattern 4 : actual = 0.0 ; neural model = -5.789860838435468E-4

==

RMS error: 7.20843262140387E-4   % error: 0.0027990260923235503

Test patterns: 4

----

pattern 1 : actual = 0.0 ; neural model = -3.854064398872703E-4

pattern 2 : actual = 1.0 ; neural model = 0.999408116956466

pattern 3 : actual = 1.0 ; neural model = 0.9988844859983493

pattern 4 : actual = 0.0 ; neural model = -5.789860838435468E-4

==

Test RMS: 7.20843262140387E-4   Test %: 0.0027990260923235503

Now that we have seen how the network is trained, let us implement the genetic algorithm to the seldom trained network and examine the results. The Elman net here will be first trained by 50 epochs.

Table 6.3 XOR problem with Elman net + Genetic Algorithm

```
-----------------------------------------------------------------------------------------------------
Network structure: Elman; Epochs: 50; Training patterns: 4
Function: sigm; Learning Rate IH: 0.5; Learning Rate HO: 0.5
-----------------------------------------------------------------------------------------------------
pattern 1 : actual = 0.0 ; neural model = 0.4440112286013108
pattern 2 : actual = 1.0 ; neural model = 1.688279895399476
pattern 3 : actual = 1.0 ; neural model = 1.4720734742078805
pattern 4 : actual = 0.0 ; neural model = 0.08200389453923485
==============================================================
 RMS error: 0.47446105986304593    % error: 1.5787544827621267
-----------------------------------------------------------------------------------------------------
Training with Genetic Algorithm; Patterns: 4; Generations: 100; Chromosomes: 20
Genes: 33; Mutation rate: 0.1 (Genes to mutate: 3); Tolerated error: 0.1
-----------------------------------------------------------------------------------------------------
Global fitness after generation 1: 0.25686650170233394
Global fitness after generation 2: 0.25294703496841
...
Global fitness after generation 5: 0.22297126430482608
Global fitness after generation 6: 0.14633793525620217
Global fitness after generation 7: 0.18866205891859752
Global fitness after generation 8: 0.18807751462143174
Global fitness after generation 9: 0.16009019739851177
Global fitness after generation 10: 0.1343656673941587
...
Global fitness after generation 13: 0.1380143798231657
Global fitness after generation 14: 0.12822024305302349
Global fitness after generation 15: 0.04801753916719025
Minimum fitness reached. Generation: 15, Chromosome: 13
-------------------------------------------------------------------------------------
pattern 1 : actual = 0.0 ; neural model = -0.032420528077555466
pattern 2 : actual = 1.0 ; neural model = 1.0224313215002092
pattern 3 : actual = 1.0 ; neural model = 0.9127374061101899
pattern 4 : actual = 0.0 ; neural model = 0.007329471552531719
```

Before we implement the Simulated Annealing Algorithm, we must first clear the weight and bias values of the network and then train it by 50 epochs.

Table 6.4 XOR problem with Elman net + Simulated Annealing Algorithm

```
-------------------------------------------------------------------------------------------
Network structure: Elman; Epochs: 50; Training patterns: 4
Function: sigm; Learning Rate IH: 0.5; Learning Rate HO: 0.5
-------------------------------------------------------------------------------------------
pattern 1 : actual = 0.0 ; neural model = 0.2434689178163445
pattern 2 : actual = 1.0 ; neural model = 1.4918457633155184
pattern 3 : actual = 1.0 ; neural model = 1.5363939761963827
pattern 4 : actual = 0.0 ; neural model = 0.34011675444619105
===========================================================
RMS error: 0.4196984849832267    % error: 1.6926550305454073
-------------------------------------------------------------------------------------------
Training with Simulated Annealing Algorithm; Patterns: 4; Cycles: 50
Iterations: 50; Beginning temperature: 10.0; Ending temperature: 1.0
-------------------------------------------------------------------------------------------
Cycle 1; Best error: 0.5033215310316713 on cycle 1, Iteration 1
Cycle 2; Best error: 0.5033215310316713 on cycle 1, Iteration 1
...
Cycle 6; Best error: 0.2475409481010761 on cycle 4, Iteration 3
Cycle 7; Best error: 0.2475409481010761 on cycle 4, Iteration 3
Cycle 8; Best error: 0.2475409481010761 on cycle 4, Iteration 3
...
Cycle 22; Best error: 0.21440644325369515 on cycle 21, Iteration 13
Cycle 23; Best error: 0.09272519823225393 on cycle 23, Iteration 2
...
Cycle 48; Best error: 0.09272519823225393 on cycle 23, Iteration 2
Cycle 49; Best error: 0.08857221337604591 on cycle 49, Iteration 7
Cycle 50; Best error: 0.08857221337604591 on cycle 49, Iteration 7
-------------------------------------------------------------------------------------------
pattern 1 : actual = 0.0 ; neural model = -0.05397223081416869
pattern 2 : actual = 1.0 ; neural model = 0.8519022714293203
pattern 3 : actual = 1.0 ; neural model = 0.9200968332601307
pattern 4 : actual = 0.0 ; neural model = 0.012234907740365009
```

The two heuristic search algorithms used here to optimize the neural network, can also be used to train the untrained network. When used on untrained networks, these algorithms help the network find later the right way to the best solution and sometimes can also produce acceptable results.

Now, let us see how the genetic algorithm trains the Elman network by 100 generations from the beginning and then with 50 epochs more the network achieves a good result:

Table 6.5 XOR problem with Genetic Algorithm + Elman net

```
----------------------------------------------------------------------------------------------------
Training with Genetic Algorithm; Patterns: 4; Generations: 100; Chromosomes: 20
Genes: 33; Mutation rate: 0.1 (Genes to mutate: 3); Tolerated error: 0.1
----------------------------------------------------------------------------------------------------
Global fitness of generation 1: 0.4826821018914213
Global fitness of generation 2: 0.4873319197088622
Global fitness of generation 3: 0.45935467520448736
Global fitness of generation 4: 0.4596012847348321
Global fitness of generation 5: 0.4596026834224938
...
Global fitness of generation 96: 0.3914234616416671
Global fitness of generation 97: 0.3902614319098676
Global fitness of generation 98: 0.39246451050253806
Global fitness of generation 99: 0.392282929488879
Global fitness of generation 100: 0.3913236101608219
Maximum number of generations reached.
=================================================================
Network structure: Elman; Epochs: 50; Training patterns: 4
Function: sigm; Learning Rate IH: 0.5; Learning Rate HO: 0.5
----------------------------------------------------------------------------------------------------
pattern 1 : actual = 0.0 ; neural model = -0.0034435765667126805
pattern 2 : actual = 1.0 ; neural model = 0.9927042190126174
pattern 3 : actual = 1.0 ; neural model = 0.9825246120897857
pattern 4 : actual = 0.0 ; neural model = -0.00849173222889954
=================================================================
RMS error: 0.010518856147966629     % error: 0.035468083102120754
----------------------------------------------------------------------------------------------------
```

This can also be done with Simulated Annealing algorithm. The weight and bias values of an untrained Elman neural network are simulated with annealing algorithm and then the network is trained by 50 epochs more.

Table 6.6 XOR problem with Simulated Annealing Algorithm + Elman net

----

Training with Simulated Annealing Algorithm; Patterns: 4; Cycles: 50

Iterations: 50; Beginning temperature: 10.0; Ending temperature: 1.0

Cycle 1; Best error: 0.6328501770268985 on cycle 1, Iteration 1

Cycle 2; Best error: 0.6328501770268985 on cycle 1, Iteration 1

...

Cycle 13; Best error: 0.50204778317415 on cycle 10, Iteration 1

Cycle 14; Best error: 0.50204778317415 on cycle 10, Iteration 1

...

Cycle 30; Best error: 0.35746206876490483 on cycle 30, Iteration 5

Cycle 31; Best error: 0.35746206876490483 on cycle 30, Iteration 5

...

Cycle 40; Best error: 0.19196602927699388 on cycle 36, Iteration 9

Cycle 41; Best error: 0.19196602927699388 on cycle 36, Iteration 9

...

Cycle 46; Best error: 0.17096161434240256 on cycle 46, Iteration 1

Cycle 47; Best error: 0.17096161434240256 on cycle 46, Iteration 1

Cycle 48; Best error: 0.17096161434240256 on cycle 46, Iteration 1

Cycle 49; Best error: 0.17096161434240256 on cycle 46, Iteration 1

Cycle 50; Best error: 0.17096161434240256 on cycle 46, Iteration 1

Network structure: Elman; Epochs: 50; Training patterns: 4

Function: sigm; Learning Rate IH: 0.5; Learning Rate HO: 0.5

----

pattern 1 : actual = 0.0 ; neural model = -0.004071633984435774

pattern 2 : actual = 1.0 ; neural model = 0.9957956685035073

pattern 3 : actual = 1.0 ; neural model = 0.9927381840795297

pattern 4 : actual = 0.0 ; neural model = -0.0016696276743943805

===

RMS error: 0.004737516059854354 % error: 0.016959096741839694

6.2. Parallel System Performance Prediction Results

In previous section there was examined an example of training the neural network with XOR patterns, and now let us see how the neural network is trained with more complex data, such as those of a parallel system performance. Here the network structure is a bit wider than the previous one, consisting of 6 neurons on the input layer, 5 neurons on the hidden layer, and a linear output neuron.

The Elman network is trained by 2000 epochs with 80 patterns of performance data taken from a parallel system of processors, and then the network is tested with 10 other patterns from the same data set.

Table 6.7 Parallel System Performance Prediction problem with Elman net

----

Network structure: Elman; Epochs: 2000; Training patterns: 80

Function: sigm; Learning Rate IH: 0.5; Learning Rate HO: 0.5

----

pattern 1 : actual = 0.002738336 ; neural model = 0.00277204424769828
pattern 2 : actual = 0.012065316 ; neural model = 0.011524138794547278
pattern 3 : actual = 0.052908872 ; neural model = 0.04727163042814719
pattern 4 : actual = 0.23068656 ; neural model = 0.22537438826730471
pattern 5 : actual = 0.999999109 ; neural model = 0.9953861921120465
pattern 6 : actual = 0.002739227 ; neural model = -0.003625268320922803
pattern 7 : actual = 0.012066207 ; neural model = 0.006630704583631242
pattern 8 : actual = 0.052909763 ; neural model = 0.04891603860573934
pattern 9 : actual = 0.230687451 ; neural model = 0.22381034998543714
pattern 10 : actual = 1.0 ; neural model = 0.9920456272772359
pattern 11 : actual = 0.001385383 ; neural model = -0.007049288166228851
pattern 12 : actual = 0.006088417 ; neural model = -0.00253606970100384
pattern 13 : actual = 0.026658913 ; neural model = 0.019546974249205484
pattern 14 : actual = 0.116122525 ; neural model = 0.10886907958674219
pattern 15 : actual = 0.503035302 ; neural model = 0.500650614826216
pattern 16 : actual = 0.001386274 ; neural model = -3.4944146817639243E-4
pattern 17 : actual = 0.006089308 ; neural model = 0.0026308592085216853
pattern 18 : actual = 0.026659804 ; neural model = 0.024252291571972617
pattern 19 : actual = 0.116123416 ; neural model = 0.1129556786884518
pattern 20 : actual = 0.503036193 ; neural model = 0.49891589367858247

pattern 21 : actual = 0.001708906 ; neural model = -0.003095717054718339

pattern 22 : actual = 0.003099968 ; neural model = -0.0032525513932691874

pattern 23 : actual = 0.013533934 ; neural model = 0.007401902799262228

pattern 24 : actual = 0.058840507 ; neural model = 0.05391848991831849

pattern 25 : actual = 0.254553398 ; neural model = 0.2498458145336213

pattern 26 : actual = 0.001709797 ; neural model = 2.233351786848914E-4

pattern 27 : actual = 0.003100859 ; neural model = 0.0017257495718985272

pattern 28 : actual = 0.013534825 ; neural model = 0.012184559226357083

pattern 29 : actual = 0.058841398 ; neural model = 0.05789946535181645

pattern 30 : actual = 0.254554289 ; neural model = 0.25071618226675163

pattern 31 : actual = 0.001270668 ; neural model = -0.0018866194958042648

pattern 32 : actual = 0.001605744 ; neural model = -0.0010459459031996188

pattern 33 : actual = 0.006971445 ; neural model = 0.0036407797796043084

pattern 34 : actual = 0.030199498 ; neural model = 0.026546397551261203

pattern 35 : actual = 0.130312446 ; neural model = 0.12671927539936112

pattern 36 : actual = 0.001071559 ; neural model = 5.404329548671649E-4

pattern 37 : actual = 0.003606635 ; neural model = 0.0025013921853435095

pattern 38 : actual = 0.010972336 ; neural model = 0.0071879756488383295

pattern 39 : actual = 0.030200389 ; neural model = 0.030438809615573326

pattern 40 : actual = 0.130313337 ; neural model = 0.12885997985309516

pattern 41 : actual = 7.01548E-4 ; neural model = -0.0017022057945786928

pattern 42 : actual = 0.001058631 ; neural model = -0.0015893890100681096

pattern 43 : actual = 0.0036902 ; neural model = 2.6612393318847793E-4

pattern 44 : actual = 0.015878994 ; neural model = 0.01031811955921813

pattern 45 : actual = 0.06819197 ; neural model = 0.06594951146040262

pattern 46 : actual = 0.001002439 ; neural model = 8.067465141914365E-5

pattern 47 : actual = 0.001259522 ; neural model = 0.0015029673761598472

pattern 48 : actual = 0.003691091 ; neural model = 0.005154691489007823

pattern 49 : actual = 0.015879885 ; neural model = 0.01667426042219866

pattern 50 : actual = 0.068192862 ; neural model = 0.06869015233254216

pattern 51 : actual = 8.16989E-4 ; neural model = -0.002108146389379617

pattern 52 : actual = 0.001285075 ; neural model = -0.0019981516512153075

pattern 53 : actual = 0.005049577 ; neural model = -0.002734897358704802

pattern 54 : actual = 0.011718741 ; neural model = 0.00592478148170561

pattern 55 : actual = 0.037131733 ; neural model = 0.03569537589490224

pattern 56 : actual = 1.1788E-4 ; neural model = -9.891996710531537E-5

pattern 57 : actual = 4.85966E-4 ; neural model = 0.0011875171760681313

pattern 58 : actual = 0.002050468 ; neural model = 0.0014797550534442205

pattern 59 : actual = 0.008719632 ; neural model = 0.00964618117668914

pattern 60 : actual = 0.037132624 ; neural model = 0.038650967833047556

pattern 61 : actual = 1.37614E-4 ; neural model = -0.0024186611014964665

pattern 62 : actual = 2.98297E-4 ; neural model = -0.0021309132585769497

pattern 63 : actual = 0.001229266 ; neural model = -0.0036046605203404747

pattern 64 : actual = 0.005138615 ; neural model = 0.002273626409057372

pattern 65 : actual = 0.021601614 ; neural model = 0.02034348509528794

pattern 66 : actual = 1.37614E-4 ; neural model = -3.9600974122688815E-4

pattern 67 : actual = 2.99188E-4 ; neural model = 8.801303816978745E-4

pattern 68 : actual = 0.001230157 ; neural model = 3.832967481787186E-4

pattern 69 : actual = 0.005139506 ; neural model = 0.005810038837735898

pattern 70 : actual = 0.021602505 ; neural model = 0.02327005219011624

pattern 71 : actual = 1.07614E-4 ; neural model = -0.003194428027480156

pattern 72 : actual = 2.04908E-4 ; neural model = -0.002715506195420603

pattern 73 : actual = 8.19111E-4 ; neural model = -0.004415578322721214

pattern 74 : actual = 0.005138615 ; neural model = 0.0017044244220998372

pattern 75 : actual = 0.021601614 ; neural model = 0.019465128421863986

pattern 76 : actual = 1.17614E-4 ; neural model = -0.001107557760069544

pattern 77 : actual = 2.05799E-4 ; neural model = 1.419997139918694E-4

pattern 78 : actual = 8.20002E-4 ; neural model = -8.437028850255546E-4

pattern 79 : actual = 0.005139506 ; neural model = 0.004845960446469233

pattern 80 : actual = 0.021602505 ; neural model = 0.022034432093345102

==

RMS error: 0.00374755048253437 % error: 0.022699615267758853

Test patterns: 10

----

pattern 1 : actual = 7.01548E-4 ; neural model = 3.4766939893793314E-4

pattern 2 : actual = 0.001058631 ; neural model = 8.92269155465808E-4

pattern 3 : actual = 0.0036902 ; neural model = 6.16221419962204E-4

pattern 4 : actual = 0.015878994 ; neural model = 0.01058843016263733

pattern 5 : actual = 0.06819197 ; neural model = 0.06586655687885778

pattern 6 : actual = 0.001002439 ; neural model = 7.148958534264338E-5

pattern 7 : actual = 0.001259522 ; neural model = 0.0015068539364411215

pattern 8 : actual = 0.003691091 ; neural model = 0.005154526882981458

pattern 9 : actual = 0.015879885 ; neural model = 0.01667429604820747

pattern 10 : actual = 0.068192862 ; neural model = 0.06869012353877058

==

Test RMS: 0.002166741908345674 Test %: 0.012987304070386155

From here we can see that the test error value can sometimes be smaller than train error, and this is because the test data happen to be more suited to the trained network and the Elman net doesn't always produce the same results for the same data.

When the simple backpropagation network of a same structure is trained with the same data by the same number of epochs, it is seen that the Elman network, as was in the XOR example, is again a little more successful than the simple backpropagation gradient-descent training network.

Table 6.8 Parallel System Performance Prediction with Back-propagation network

----

Network structure: Backpropagation; Epochs: 2000; Training patterns: 80

Function: sigm; Learning Rate IH: 0.5; Learning Rate HO: 0.5

----

pattern 1 : actual = 0.002738336 ; neural model = -0.005886584070009859
pattern 2 : actual = 0.012065316 ; neural model = 0.0035061943593635014
pattern 3 : actual = 0.052908872 ; neural model = 0.04366514309816416
pattern 4 : actual = 0.23068656 ; neural model = 0.22126439155012367
pattern 5 : actual = 0.999999109 ; neural model = 0.991946433734839
pattern 6 : actual = 0.002739227 ; neural model = -0.001711425700886282
pattern 7 : actual = 0.012066207 ; neural model = 0.007577554504398676
pattern 8 : actual = 0.052909763 ; neural model = 0.04730392144787088
pattern 9 : actual = 0.230687451 ; neural model = 0.22307716150788903
pattern 10 : actual = 1.0 ; neural model = 0.9881663936212072
pattern 11 : actual = 0.001385383 ; neural model = -0.00710268208407383
pattern 12 : actual = 0.006088417 ; neural model = -0.00211744918181922
pattern 13 : actual = 0.026658913 ; neural model = 0.018484354229064914
pattern 14 : actual = 0.116122525 ; neural model = 0.10658422136917911
pattern 15 : actual = 0.503035302 ; neural model = 0.49867181414818307
pattern 16 : actual = 0.001386274 ; neural model = -0.002924121075612851
pattern 17 : actual = 0.006089308 ; neural model = 0.0020038872562044285
pattern 18 : actual = 0.026659804 ; neural model = 0.022378230217501738
pattern 19 : actual = 0.116123416 ; neural model = 0.10953383467860334
pattern 20 : actual = 0.503036193 ; neural model = 0.497839102476485
pattern 21 : actual = 0.001708906 ; neural model = -0.007604390781393633
pattern 22 : actual = 0.003099968 ; neural model = -0.004821555272663147
pattern 23 : actual = 0.013533934 ; neural model = 0.006095079642868817
pattern 24 : actual = 0.058840507 ; neural model = 0.051076958280250606

pattern 25 : actual = 0.254553398 ; neural model = 0.24381419106952373

pattern 26 : actual = 0.001709797 ; neural model = -0.0034394167399854902

pattern 27 : actual = 0.003100859 ; neural model = -6.905086099410207E-4

pattern 28 : actual = 0.013534825 ; neural model = 0.010101219970645947

pattern 29 : actual = 0.058841398 ; neural model = 0.05458630922280633

pattern 30 : actual = 0.254554289 ; neural model = 0.2452923820991767

pattern 31 : actual = 0.001270668 ; neural model = -0.007420923585031991

pattern 32 : actual = 0.001605744 ; neural model = -0.004619346997690732

pattern 33 : actual = 0.006971445 ; neural model = 0.0019062864582325423

pattern 34 : actual = 0.030199498 ; neural model = 0.023963730960632756

pattern 35 : actual = 0.130312446 ; neural model = 0.12172816796384839

pattern 36 : actual = 0.001071559 ; neural model = -0.003206218396296423

pattern 37 : actual = 0.003606635 ; neural model = -9.745655574004974E-4

pattern 38 : actual = 0.010972336 ; neural model = 0.0050309502589370725

pattern 39 : actual = 0.030200389 ; neural model = 0.02772278999415606

pattern 40 : actual = 0.130313337 ; neural model = 0.12441105479864095

pattern 41 : actual = 7.01548E-4 ; neural model = -0.006997882545439982

pattern 42 : actual = 0.001058631 ; neural model = -0.005669258720993775

pattern 43 : actual = 0.0036902 ; neural model = -0.0011843114988152603

pattern 44 : actual = 0.015878994 ; neural model = 0.008614572994199543

pattern 45 : actual = 0.06819197 ; neural model = 0.06284836442280328

pattern 46 : actual = 0.001002439 ; neural model = -0.003219894880276686

pattern 47 : actual = 0.001259522 ; neural model = -0.0019978203061564725

pattern 48 : actual = 0.003691091 ; neural model = 0.0032261634674828343

pattern 49 : actual = 0.015879885 ; neural model = 0.01464331789149903

pattern 50 : actual = 0.068192862 ; neural model = 0.0660780572047438

pattern 51 : actual = 8.16989E-4 ; neural model = -0.006313923529575272

pattern 52 : actual = 0.001285075 ; neural model = -0.0055511245459658465

pattern 53 : actual = 0.005049577 ; neural model = -0.0031945842171891004

pattern 54 : actual = 0.011718741 ; neural model = 0.004788626995221645

pattern 55 : actual = 0.037131733 ; neural model = 0.03420510831736839

pattern 56 : actual = 1.1788E-4 ; neural model = -0.002427314057917296

pattern 57 : actual = 4.85966E-4 ; neural model = -0.0016780798293482557

pattern 58 : actual = 0.002050468 ; neural model = 6.440172420344448E-4

pattern 59 : actual = 0.008719632 ; neural model = 0.008523849480923107

pattern 60 : actual = 0.037132624 ; neural model = 0.037589693361994025

pattern 61 : actual = 1.37614E-4 ; neural model = -0.00459721024069859

pattern 62 : actual = 2.98297E-4 ; neural model = -0.004079948794948884

pattern 63 : actual = 0.001229266 ; neural model = -0.0024946009235095046

pattern 64 : actual = 0.005138615 ; neural model = 0.002552206999854700З

pattern 65 : actual = 0.021601614 ; neural model = 0.02024854135787907

pattern 66 : actual = 1.37614E-4 ; neural model = -0.0010345676972670637

pattern 67 : actual = 2.99188E-4 ; neural model = -5.273104960222819E-4

pattern 68 : actual = 0.001230157 ; neural model = 0.001031333505737475

pattern 69 : actual = 0.005139506 ; neural model = 0.006005696850666431

pattern 70 : actual = 0.021602505 ; neural model = 0.023478434189884823

pattern 71 : actual = 1.07614E-4 ; neural model = -0.0022212093179988512

pattern 72 : actual = 2.04908E-4 ; neural model = -0.0019318013257609845

pattern 73 : actual = 8.19111E-4 ; neural model = -0.0010248966300656637

pattern 74 : actual = 0.005138615 ; neural model = 0.003574744580432887

pattern 75 : actual = 0.021601614 ; neural model = 0.0193669668922265

pattern 76 : actual = 1.17614E-4 ; neural model = 6.943878106832613E-4

pattern 77 : actual = 2.05799E-4 ; neural model = 9.947324654110412E-4

pattern 78 : actual = 8.20002E-4 ; neural model = 0.0018777033971885126

pattern 79 : actual = 0.005139506 ; neural model = 0.006409559486855676

pattern 80 : actual = 0.021602505 ; neural model = 0.022000072682766936

==

RMS error: 0.005674751727860728 % error: 0.036076055375548415

Test patterns: 10

----

pattern 1 : actual = 7.01548E-4 ; neural model = -0.006997882545439982

pattern 2 : actual = 0.001058631 ; neural model = -0.005669258720993775

pattern 3 : actual = 0.0036902 ; neural model = -0.0011843114988152603

pattern 4 : actual = 0.015878994 ; neural model = 0.008614572994199543

pattern 5 : actual = 0.06819197 ; neural model = 0.06284836442280328

pattern 6 : actual = 0.001002439 ; neural model = -0.003219894880276686

pattern 7 : actual = 0.001259522 ; neural model = -0.0019978203061564725

pattern 8 : actual = 0.003691091 ; neural model = 0.0032261634674828343

pattern 9 : actual = 0.015879885 ; neural model = 0.01464331789149903

pattern 10 : actual = 0.068192862 ; neural model = 0.0660780572047438

==

Test RMS: 0.004942555651993406 Test %: 0.03862040736978732

Now let us see how the Genetic Algorithm would optimize the not-well trained Elman network. The network is trained by 100 epochs and then the 100 generation Genetic Algorithm is applied.

Table 6.9 Parallel System Performance Prediction with Elman + Genetic Alg.

Network structure: Elman; Epochs: 100; Training patterns: 80

Function: sigm; Learning Rate IH: 0.5; Learning Rate HO: 0.5

----

pattern 1 : actual = 0.002738336 ; neural model = 0.01097427761244324

pattern 2 : actual = 0.012065316 ; neural model = 0.016395331204614727

pattern 3 : actual = 0.052908872 ; neural model = 0.04961595795687762

pattern 4 : actual = 0.23068656 ; neural model = 0.21229566677552453

pattern 5 : actual = 0.999999109 ; neural model = 0.9573131817951781

pattern 6 : actual = 0.002739227 ; neural model = -0.027345329110038008

pattern 7 : actual = 0.012066207 ; neural model = -0.007163698385088846

pattern 8 : actual = 0.052909763 ; neural model = 0.05855479533091229

pattern 9 : actual = 0.230687451 ; neural model = 0.2072626308912922

pattern 10 : actual = 1.0 ; neural model = 0.9505230677851257

pattern 11 : actual = 0.001385383 ; neural model = -0.021056765399887545

pattern 12 : actual = 0.006088417 ; neural model = -0.007945218408852595

pattern 13 : actual = 0.026658913 ; neural model = 0.040279771194711766

pattern 14 : actual = 0.116122525 ; neural model = 0.10589507829697348

pattern 15 : actual = 0.503035302 ; neural model = 0.5018418210666631

pattern 16 : actual = 0.001386274 ; neural model = -0.012286314717175839

pattern 17 : actual = 0.006089308 ; neural model = -0.0015645084725792735

pattern 18 : actual = 0.026659804 ; neural model = 0.032041561214299574

pattern 19 : actual = 0.116123416 ; neural model = 0.10304328761126463

pattern 20 : actual = 0.503036193 ; neural model = 0.49756254057281485

pattern 21 : actual = 0.001708906 ; neural model = -0.005656879053675956

pattern 22 : actual = 0.003099968 ; neural model = -3.456146675473448E-4

pattern 23 : actual = 0.013533934 ; neural model = 0.02428121169955011

pattern 24 : actual = 0.058840507 ; neural model = 0.05790537890289177

pattern 25 : actual = 0.254553398 ; neural model = 0.2445462810184515

pattern 26 : actual = 0.001709797 ; neural model = -0.0046089388244122675

pattern 27 : actual = 0.003100859 ; neural model = 0.0013906391766689286

pattern 28 : actual = 0.013534825 ; neural model = 0.018643222011781868

pattern 29 : actual = 0.058841398 ; neural model = 0.05497567009110388

pattern 30 : actual = 0.254554289 ; neural model = 0.2408537169446317

pattern 31 : actual = 0.001270668 ; neural model = 0.0025248831317562503

pattern 32 : actual = 0.001605744 ; neural model = 0.004568785075211856

pattern 33 : actual = 0.006971445 ; neural model = 0.017219956050109286

pattern 34 : actual = 0.030199498 ; neural model = 0.034685623216787365

pattern 35 : actual = 0.130312446 ; neural model = 0.12709201466283

pattern 36 : actual = 0.001071559 ; neural model = -0.001180716456751385

pattern 37 : actual = 0.003606635 ; neural model = 0.003099761612857743

pattern 38 : actual = 0.010972336 ; neural model = 0.012591083982283868

pattern 39 : actual = 0.030200389 ; neural model = 0.0317334756505222

pattern 40 : actual = 0.130313337 ; neural model = 0.12372410642275605

pattern 41 : actual = 7.01548E-4 ; neural model = 0.005637607151884494

pattern 42 : actual = 0.001058631 ; neural model = 0.0049919684940429865

pattern 43 : actual = 0.0036902 ; neural model = 0.012182133609899837

pattern 44 : actual = 0.015878994 ; neural model = 0.02094606345391642

pattern 45 : actual = 0.06819197 ; neural model = 0.07260966294408516

pattern 46 : actual = 0.001002439 ; neural model = -5.19191530762797E-4

pattern 47 : actual = 0.001259522 ; neural model = 0.0023021105574005385

pattern 48 : actual = 0.003691091 ; neural model = 0.009493565234419854

pattern 49 : actual = 0.015879885 ; neural model = 0.019841234559771437

pattern 50 : actual = 0.068192862 ; neural model = 0.0693124616286947

pattern 51 : actual = 8.16989E-4 ; neural model = 0.005778920542147237

pattern 52 : actual = 0.001285075 ; neural model = 0.003992244205565265

pattern 53 : actual = 0.005049577 ; neural model = 0.007713831577918495

pattern 54 : actual = 0.011718741 ; neural model = 0.015925990524362332

pattern 55 : actual = 0.037131733 ; neural model = 0.045452109660255646

pattern 56 : actual = 1.1788E-4 ; neural model = -0.0012413337811281733

pattern 57 : actual = 4.85966E-4 ; neural model = 0.00102719541414531

pattern 58 : actual = 0.002050468 ; neural model = 0.004984260473663904

pattern 59 : actual = 0.008719632 ; neural model = 0.0128975341917405

pattern 60 : actual = 0.037132624 ; neural model = 0.04231478350419765

pattern 61 : actual = 1.37614E-4 ; neural model = 0.003880625742267385

pattern 62 : actual = 2.98297E-4 ; neural model = 0.0015674110135979191

pattern 63 : actual = 0.001229266 ; neural model = 0.003937162545476214

pattern 64 : actual = 0.005138615 ; neural model = 0.010278293328625898

pattern 65 : actual = 0.021601614 ; neural model = 0.029946246440317237

pattern 66 : actual = 1.37614E-4 ; neural model = -0.00367364252898017

pattern 67 : actual = 2.99188E-4 ; neural model = -0.001700388533394992

pattern 68 : actual = 0.001230157 ; neural model = 0.0013333828725689556

pattern 69 : actual = 0.005139506 ; neural model = 0.007222754618641303

pattern 70 : actual = 0.021602505 ; neural model = 0.026830573357460752

pattern 71 : actual = 1.07614E-4 ; neural model = -8.861133220348649E-4

pattern 72 : actual = 2.04908E-4 ; neural model = -0.0036291031509280702

pattern 73 : actual = 8.19111E-4 ; neural model = -0.0019798079075599717

pattern 74 : actual = 0.005138615 ; neural model = 0.004642933250214976

pattern 75 : actual = 0.021601614 ; neural model = 0.023848501783967024

pattern 76 : actual = 1.17614E-4 ; neural model = -0.009083994891525116

pattern 77 : actual = 2.05799E-4 ; neural model = -0.007214553093994713

pattern 78 : actual = 8.20002E-4 ; neural model = -0.004515957141234039

pattern 79 : actual = 0.005139506 ; neural model = 0.0015198776056115082

pattern 80 : actual = 0.021602505 ; neural model = 0.020670861634148163

==

RMS error: 0.010928789681105819 % error: 0.04665629811528724

Training with Genetic Algorithm; Patterns: 80; Generations: 100; Chromosomes: 20

Genes: 66; Mutation rate: 0.1 (Genes to mutate: 6); Tolerated error: 0.0010

Global fitness after generation 1: 0.010929074657596634

Global fitness after generation 2: 0.010929074657596634

Global fitness after generation 3: 0.010929074657596634

...

Global fitness after generation 40: 0.009571359018784992

Global fitness after generation 41: 0.00958200633569838

Global fitness after generation 42: 0.009572628945528298

...

Global fitness after generation 98: 0.009486907684329074

Global fitness after generation 99: 0.009486907684329074

Global fitness after generation 100: 0.009486907684329074

Maximum number of generations reached.

----

pattern 1 : actual = 0.002738336 ; neural model = 0.0021058653562872065

pattern 2 : actual = 0.012065316 ; neural model = 0.008806576457819404

pattern 3 : actual = 0.052908872 ; neural model = 0.058491225157803195

pattern 4 : actual = 0.23068656 ; neural model = 0.22540361359817457

pattern 5 : actual = 0.999999109 ; neural model = 1.0136136966123404

pattern 6 : actual = 0.002739227 ; neural model = -0.006913378324405028

pattern 7 : actual = 0.012066207 ; neural model = -0.019153826278710318

pattern 8 : actual = 0.052909763 ; neural model = 0.05528404836982442

pattern 9 : actual = 0.230687451 ; neural model = 0.22361188149403394

pattern 10 : actual = 1.0 ; neural model = 1.0064635093993597

pattern 11 : actual = 0.001385383 ; neural model = -3.99343687953857E-4

pattern 12 : actual = 0.006088417 ; neural model = -0.019060206721791983

pattern 13 : actual = 0.026658913 ; neural model = 0.0359005308016615

pattern 14 : actual = 0.116122525 ; neural model = 0.11785262205259245

pattern 15 : actual = 0.503035302 ; neural model = 0.529022232280234

pattern 16 : actual = 0.001386274 ; neural model = -9.360372465117006E-5

pattern 17 : actual = 0.006089308 ; neural model = -0.0060281120169857205

pattern 18 : actual = 0.026659804 ; neural model = 0.031968718968370224

pattern 19 : actual = 0.116123416 ; neural model = 0.11315033886859488

pattern 20 : actual = 0.503036193 ; neural model = 0.5239472780727328

pattern 21 : actual = 0.001708906 ; neural model = 0.0065468315018667456

pattern 22 : actual = 0.003099968 ; neural model = -0.0037463716833928684

pattern 23 : actual = 0.013533934 ; neural model = 0.02345583377510929

pattern 24 : actual = 0.058840507 ; neural model = 0.06574639827653078

pattern 25 : actual = 0.254553398 ; neural model = 0.2592341181411501

pattern 26 : actual = 0.001709797 ; neural model = 0.0028323027418151736

pattern 27 : actual = 0.003100859 ; neural model = 3.4570165489838933E-4

pattern 28 : actual = 0.013534825 ; neural model = 0.020264728471114213

pattern 29 : actual = 0.058841398 ; neural model = 0.06178769264948697

pattern 30 : actual = 0.254554289 ; neural model = 0.255135421774711

pattern 31 : actual = 0.001270668 ; neural model = 0.009482120671071126

pattern 32 : actual = 0.001605744 ; neural model = 0.004445016431127546

pattern 33 : actual = 0.006971445 ; neural model = 0.018117153895544746

pattern 34 : actual = 0.030199498 ; neural model = 0.0399639070484174

pattern 35 : actual = 0.130312446 ; neural model = 0.13590883871381826

pattern 36 : actual = 0.001071559 ; neural model = 0.003716279467905703

pattern 37 : actual = 0.003606635 ; neural model = 0.0033105153231247075

pattern 38 : actual = 0.010972336 ; neural model = 0.014563618602681072

pattern 39 : actual = 0.030200389 ; neural model = 0.036578372151069016

pattern 40 : actual = 0.130313337 ; neural model = 0.1323705995545858

pattern 41 : actual = 7.01548E-4 ; neural model = 0.009357985646381128

pattern 42 : actual = 0.001058631 ; neural model = 0.005524735991318586

pattern 43 : actual = 0.0036902 ; neural model = 0.012978113112859513

pattern 44 : actual = 0.015878994 ; neural model = 0.024161321852318396

pattern 45 : actual = 0.06819197 ; neural model = 0.0778274172555013

pattern 46 : actual = 0.001002439 ; neural model = 0.0024455920081549176

pattern 47 : actual = 0.001259522 ; neural model = 0.0023491774349014283

pattern 48 : actual = 0.003691091 ; neural model = 0.010788140594743367

pattern 49 : actual = 0.015879885 ; neural model = 0.022950420129865257

pattern 50 : actual = 0.068192862 ; neural model = 0.0745058385779766

pattern 51 : actual = 8.16989E-4 ; neural model = 0.006323616296718182

pattern 52 : actual = 0.001285075 ; neural model = 0.003084348461348363

pattern 53 : actual = 0.005049577 ; neural model = 0.0068039523649297

pattern 54 : actual = 0.011718741 ; neural model = 0.016529312596411327

pattern 55 : actual = 0.037131733 ; neural model = 0.04734554751380199

pattern 56 : actual = 1.1788E-4 ; neural model = -7.435476199098012E-4

pattern 57 : actual = 4.85966E-4 ; neural model = -5.228600051473409E-4

pattern 58 : actual = 0.002050468 ; neural model = 0.004419708996078131

pattern 59 : actual = 0.008719632 ; neural model = 0.013562590223169302

pattern 60 : actual = 0.037132624 ; neural model = 0.0442458927444635

pattern 61 : actual = 1.37614E-4 ; neural model = -2.0439885017492498E-4

pattern 62 : actual = 2.98297E-4 ; neural model = -0.0033055201357683472

pattern 63 : actual = 0.001229266 ; neural model = -8.329650378468556E-4

pattern 64 : actual = 0.005138615 ; neural model = 0.006612885222672671

pattern 65 : actual = 0.021601614 ; neural model = 0.02710413201407086

pattern 66 : actual = 1.37614E-4 ; neural model = -0.007198075843318219

pattern 67 : actual = 2.99188E-4 ; neural model = -0.006794198787681394

pattern 68 : actual = 0.001230157 ; neural model = -0.002987425583986736

pattern 69 : actual = 0.005139506 ; neural model = 0.0038006907909196586

pattern 70 : actual = 0.021602505 ; neural model = 0.024158058338172084

pattern 71 : actual = 1.07614E-4 ; neural model = -0.012599550743243426

pattern 72 : actual = 2.04908E-4 ; neural model = -0.01600407285404276

pattern 73 : actual = 8.19111E-4 ; neural model = -0.013811897541195217

pattern 74 : actual = 0.005138615 ; neural model = -0.006221983797955244

pattern 75 : actual = 0.021601614 ; neural model = 0.013582178899218478

pattern 76 : actual = 1.17614E-4 ; neural model = -0.019289408395464225

pattern 77 : actual = 2.05799E-4 ; neural model = -0.01885173615637248

pattern 78 : actual = 8.20002E-4 ; neural model = -0.015586422002336564

pattern 79 : actual = 0.005139506 ; neural model = -0.008818553255016903

pattern 80 : actual = 0.021602505 ; neural model = 0.010795906466367633

----

fitnessThisChrom[16] = 0.009486907684329074

===

After the Genetic Algorithm is applied, the error is decreased by a small ratio, in comparison with that in the beginning, the new error is 0.009486907684329074.

In the same way we also apply the Simulated Annealing Algorithm. Before this, the weights are initialized and the network is trained by 50 epochs. Then the Simulated Annealing with 100 cycles is applied.

Table 6.10 Parallel System Performance Prediction with Elman + Sim. Annealing

----

Network structure: Elman; Epochs: 50; Training patterns: 80

Function: sigm; Learning Rate IH: 0.5; Learning Rate HO: 0.5

----

pattern 1 : actual = 0.002738336 ; neural model = -0.0142259632566209

pattern 2 : actual = 0.012065316 ; neural model = -0.009996462835396347

pattern 3 : actual = 0.052908872 ; neural model = 0.031508545743523464

pattern 4 : actual = 0.23068656 ; neural model = 0.19042259987182286

pattern 5 : actual = 0.999999109 ; neural model = 0.9566241824725319

pattern 6 : actual = 0.002739227 ; neural model = -0.03237570711129231

pattern 7 : actual = 0.012066207 ; neural model = 0.03490887330590589

pattern 8 : actual = 0.052909763 ; neural model = 0.03959519439772363

pattern 9 : actual = 0.230687451 ; neural model = 0.18747326287530686

pattern 10 : actual = 1.0 ; neural model = 0.9522148038824543

pattern 11 : actual = 0.001385383 ; neural model = -0.04292529159976727

pattern 12 : actual = 0.006088417 ; neural model = 0.02027591014031324

pattern 13 : actual = 0.026658913 ; neural model = 0.013765040347405277

pattern 14 : actual = 0.116122525 ; neural model = 0.08356945056659193

pattern 15 : actual = 0.503035302 ; neural model = 0.475197788376

pattern 16 : actual = 0.001386274 ; neural model = -0.018562335126605684

pattern 17 : actual = 0.006089308 ; neural model = 0.012425812906613487

pattern 18 : actual = 0.026659804 ; neural model = 0.017310328198845537

pattern 19 : actual = 0.116123416 ; neural model = 0.09042443099067476

pattern 20 : actual = 0.503036193 ; neural model = 0.47174628795324164

pattern 21 : actual = 0.001708906 ; neural model = -0.027824144080112218

pattern 22 : actual = 0.003099968 ; neural model = -0.0010577937913152091

pattern 23 : actual = 0.013533934 ; neural model = 0.0016739902031819298

pattern 24 : actual = 0.058840507 ; neural model = 0.03791470631133931

pattern 25 : actual = 0.254553398 ; neural model = 0.227106655405022

pattern 26 : actual = 0.001709797 ; neural model = -0.011943627302961668

pattern 27 : actual = 0.003100859 ; neural model = 0.0021984554820788094

pattern 28 : actual = 0.013534825 ; neural model = 0.006016895607025757

pattern 29 : actual = 0.058841398 ; neural model = 0.044586851944251826

pattern 30 : actual = 0.254554289 ; neural model = 0.22799379900913064

pattern 31 : actual = 0.001270668 ; neural model = -0.02047471154228947

pattern 32 : actual = 0.001605744 ; neural model = -0.009453710877048854

pattern 33 : actual = 0.006971445 ; neural model = -0.0031611025404640336

pattern 34 : actual = 0.030199498 ; neural model = 0.01563370803310027

pattern 35 : actual = 0.130312446 ; neural model = 0.11263554795021546

pattern 36 : actual = 0.001071559 ; neural model = -0.00912634676551774

pattern 37 : actual = 0.003606635 ; neural model = -0.0012499731430593575

pattern 38 : actual = 0.010972336 ; neural model = 0.0010038445230961257

pattern 39 : actual = 0.030200389 ; neural model = 0.022160969602024072

pattern 40 : actual = 0.130313337 ; neural model = 0.11619518937720874

pattern 41 : actual = 7.01548E-4 ; neural model = -0.017737892907932706

pattern 42 : actual = 0.001058631 ; neural model = -0.014022355235449036

pattern 43 : actual = 0.0036902 ; neural model = -0.007039272504585664

pattern 44 : actual = 0.015878994 ; neural model = 0.002350567053314767

pattern 45 : actual = 0.06819197 ; neural model = 0.058284157481305665

pattern 46 : actual = 0.001002439 ; neural model = -0.008544813263486023

pattern 47 : actual = 0.001259522 ; neural model = -0.0039041602562067956

pattern 48 : actual = 0.003691091 ; neural model = -0.0013100036167943419

pattern 49 : actual = 0.015879885 ; neural model = 0.010844312289911617

pattern 50 : actual = 0.068192862 ; neural model = 0.06319541029591491

pattern 51 : actual = 8.16989E-4 ; neural model = -0.017157504704765597

pattern 52 : actual = 0.001285075 ; neural model = -0.01629203309197333

pattern 53 : actual = 0.005049577 ; neural model = -0.01041579685700067

pattern 54 : actual = 0.011718741 ; neural model = -0.0017032131065706224

pattern 55 : actual = 0.037131733 ; neural model = 0.03131981243801768

pattern 56 : actual = 1.1788E-4 ; neural model = -0.00848634702170542

pattern 57 : actual = 4.85966E-4 ; neural model = -0.005323731015413363

pattern 58 : actual = 0.002050468 ; neural model = -0.0047912821950190365

pattern 59 : actual = 0.008719632 ; neural model = 0.004832172450128097

pattern 60 : actual = 0.037132624 ; neural model = 0.036959542133577106

pattern 61 : actual = 1.37614E-4 ; neural model = -0.01745996055526211

pattern 62 : actual = 2.98297E-4 ; neural model = -0.017600909780540658

pattern 63 : actual = 0.001229266 ; neural model = -0.012359196044302218

pattern 64 : actual = 0.005138615 ; neural model = -0.005594920910549389

pattern 65 : actual = 0.021601614 ; neural model = 0.01705223098082065

pattern 66 : actual = 1.37614E-4 ; neural model = -0.00917267901291477

pattern 67 : actual = 2.99188E-4 ; neural model = -0.006734815197294658

pattern 68 : actual = 0.001230157 ; neural model = -0.006509573794062695

pattern 69 : actual = 0.005139506 ; neural model = 9.894527775063389E-4

pattern 70 : actual = 0.021602505 ; neural model = 0.023041330165767593

pattern 71 : actual = 1.07614E-4 ; neural model = -0.01873103084947675

pattern 72 : actual = 2.04908E-4 ; neural model = -0.019036234678956265

pattern 73 : actual = 8.19111E-4 ; neural model = -0.014832784874481753

pattern 74 : actual = 0.005138615 ; neural model = -0.007741747312820624

pattern 75 : actual = 0.021601614 ; neural model = 0.014115898331663379

pattern 76 : actual = 1.17614E-4 ; neural model = -0.010881328370428878

pattern 77 : actual = 2.05799E-4 ; neural model = -0.008675969618081925

pattern 78 : actual = 8.20002E-4 ; neural model = -0.008666428609181875

pattern 79 : actual = 0.005139506 ; neural model = -0.0011413646885685969

pattern 80 : actual = 0.021602505 ; neural model = 0.020132183109953433

==

RMS error: 0.018485494723617355 % error: 0.10770828939013169

Training with Simulated Annealing Algorithm; Patterns: 80; Cycles: 100

Iterations: 50; Beginning temperature: 20.0; Ending temperature: 1.0

Cycle 1; Best error: 0.018485765842688046 on cycle 1, Iteration 1

...

Cycle 27; Best error: 0.018485765842688046 on cycle 1, Iteration 1

...

Cycle 53; Best error: 0.018485765842688046 on cycle 1, Iteration 1

...

Cycle 72; Best error: 0.018485765842688046 on cycle 1, Iteration 1

Cycle 73; Best error: 0.017049643022346014 on cycle 73, Iteration 1

...

Cycle 90; Best error: 0.017049643022346014 on cycle 73, Iteration 1

Cycle 91; Best error: 0.013566278463712026 on cycle 91, Iteration 2

...

Cycle 100; Best error: 0.013566278463712026 on cycle 91, Iteration 2

----

pattern 1 : actual = 0.002738336 ; neural model = -0.013180673481784505

pattern 2 : actual = 0.012065316 ; neural model = -0.007556054371764015

pattern 3 : actual = 0.052908872 ; neural model = 0.031779017408392424

pattern 4 : actual = 0.23068656 ; neural model = 0.1937964825621179

pattern 5 : actual = 0.999999109 ; neural model = 0.9804529350204017

pattern 6 : actual = 0.002739227 ; neural model = -0.016453944799817988

pattern 7 : actual = 0.012066207 ; neural model = 0.035183945366280917

pattern 8 : actual = 0.052909763 ; neural model = 0.04631779348128959

pattern 9 : actual = 0.230687451 ; neural model = 0.19547171239325167

pattern 10 : actual = 1.0 ; neural model = 0.977874519097676

pattern 11 : actual = 0.001385383 ; neural model = -0.02771911985878464

pattern 12 : actual = 0.006088417 ; neural model = 0.018783176193003542

pattern 13 : actual = 0.026658913 ; neural model = 0.017588397952310098

pattern 14 : actual = 0.116122525 ; neural model = 0.09016801136398211

pattern 15 : actual = 0.503035302 ; neural model = 0.49981192944877906

pattern 16 : actual = 0.001386274 ; neural model = -0.009434997808097922

pattern 17 : actual = 0.006089308 ; neural model = 0.015971093578441564

pattern 18 : actual = 0.026659804 ; neural model = 0.023820213999764445

pattern 19 : actual = 0.116123416 ; neural model = 0.09915128397864709

pattern 20 : actual = 0.503036193 ; neural model = 0.49790931890013346

pattern 21 : actual = 0.001708906 ; neural model = -0.019325188223636353

pattern 22 : actual = 0.003099968 ; neural model = 9.211106395284885E-5

pattern 23 : actual = 0.013533934 ; neural model = 0.004717632265496041

pattern 24 : actual = 0.058840507 ; neural model = 0.04492022771319609

pattern 25 : actual = 0.254553398 ; neural model = 0.24568646313307585

pattern 26 : actual = 0.001709797 ; neural model = -0.005896241767840843

pattern 27 : actual = 0.003100859 ; neural model = 0.006561658283196631

pattern 28 : actual = 0.013534825 ; neural model = 0.012639603144900535

pattern 29 : actual = 0.058841398 ; neural model = 0.054497176966045446

pattern 30 : actual = 0.254554289 ; neural model = 0.24861556686620612

pattern 31 : actual = 0.001270668 ; neural model = -0.014926718872179834

pattern 32 : actual = 0.001605744 ; neural model = -0.007874172328502349

pattern 33 : actual = 0.006971445 ; neural model = -3.74841307162338E-4

pattern 34 : actual = 0.030199498 ; neural model = 0.02323181161405763

pattern 35 : actual = 0.130312446 ; neural model = 0.130658043407328

pattern 36 : actual = 0.001071559 ; neural model = -0.004324300994241026

pattern 37 : actual = 0.003606635 ; neural model = 0.00339190976907755

pattern 38 : actual = 0.010972336 ; neural model = 0.007884990048707935

pattern 39 : actual = 0.030200389 ; neural model = 0.03304425561426269

pattern 40 : actual = 0.130313337 ; neural model = 0.13667150648191645

pattern 41 : actual = 7.01548E-4 ; neural model = -0.01316567680344613

pattern 42 : actual = 0.001058631 ; neural model = -0.012102673696194888

pattern 43 : actual = 0.0036902 ; neural model = -0.00389095991449781

pattern 44 : actual = 0.015878994 ; neural model = 0.01076206570654703

pattern 45 : actual = 0.06819197 ; neural model = 0.07717196821840189

pattern 46 : actual = 0.001002439 ; neural model = -0.0039777474436698546

pattern 47 : actual = 0.001259522 ; neural model = 0.0012428615281005473

pattern 48 : actual = 0.003691091 ; neural model = 0.005981517982007439

pattern 49 : actual = 0.015879885 ; neural model = 0.02260523943565043

pattern 50 : actual = 0.068192862 ; neural model = 0.08483519358740016

pattern 51 : actual = 8.16989E-4 ; neural model = -0.01239042898487494

pattern 52 : actual = 0.001285075 ; neural model = -0.013906431989665874

pattern 53 : actual = 0.005049577 ; neural model = -0.006393634949515031

pattern 54 : actual = 0.011718741 ; neural model = 0.007450261365611888

pattern 55 : actual = 0.037131733 ; neural model = 0.05159850729356097

pattern 56 : actual = 1.1788E-4 ; neural model = -0.003479523503853277

pattern 57 : actual = 4.85966E-4 ; neural model = 6.361266738759142E-4

pattern 58 : actual = 0.002050468 ; neural model = 0.0035059656843797193

pattern 59 : actual = 0.008719632 ; neural model = 0.017640451396413653

pattern 60 : actual = 0.037132624 ; neural model = 0.060142127069110096

pattern 61 : actual = 1.37614E-4 ; neural model = -0.011480484961256865

pattern 62 : actual = 2.98297E-4 ; neural model = -0.014434140064593898

pattern 63 : actual = 0.001229266 ; neural model = -0.006904644202805982

pattern 64 : actual = 0.005138615 ; neural model = 0.005088170340731696

pattern 65 : actual = 0.021601614 ; neural model = 0.03928721616273595

pattern 66 : actual = 1.37614E-4 ; neural model = -0.002927754883369299

pattern 67 : actual = 2.99188E-4 ; neural model = 7.070721078592801E-4

pattern 68 : actual = 0.001230157 ; neural model = 0.0032906559634083288

pattern 69 : actual = 0.005139506 ; neural model = 0.015506160047530415

pattern 70 : actual = 0.021602505 ; neural model = 0.048416974004861335

pattern 71 : actual = 1.07614E-4 ; neural model = -0.010358725219779374

pattern 72 : actual = 2.04908E-4 ; neural model = -0.014710343857362224

pattern 73 : actual = 8.19111E-4 ; neural model = -0.0070193132137268965

pattern 74 : actual = 0.005138615 ; neural model = 0.005380962082508517

pattern 75 : actual = 0.021601614 ; neural model = 0.03946605989309554

pattern 76 : actual = 1.17614E-4 ; neural model = -0.0023315651999789855

pattern 77 : actual = 2.05799E-4 ; neural model = 0.0012657880757630247

pattern 78 : actual = 8.20002E-4 ; neural model = 0.003623992946017701

pattern 79 : actual = 0.005139506 ; neural model = 0.016125862334720104

pattern 80 : actual = 0.021602505 ; neural model = 0.04899228170086184

This is how the Elman network is optimized using the Simulated Annealing Algorithm. While the error after 50 epochs of training the Elman net was 0.018485494723617355, after implementing the Simulated Annealing it is decreased to 0.013566278463712026.

CONCLUSION

This project is an implementation of Elman recurrent neural network model with backpropagation which is trained to recognize solutions to problems of different nature.

While backpropagation algorithm uses gradient-descent training method to train the neural network model, the Elman recurrent neural network model has a special architecture which allows taking feedback from previous step of training. This is a copy of hidden layer neurons acting as input neurons. This characteristic makes possible to better learn and recognize patterns. Because of this characteristic the Elman net is known as “neural network with memory”. It is clear that Elman nets are more successful than simple backpropagation neural network models.

Another important point of this project was optimizing the Elman network with two popular heuristic search algorithms, Genetic Algorithm and Simulated Annealing Algorithm. These two algorithms are used to find the adequate combinations of weights and biases of the network which constitute complete solutions to the problem.

In genetic algorithm, the weights and biases are taken as a single chromosome. Then the genetic algorithm proceeds to splice the genes of this chromosome with other suitable chromosomes. Through subsequent generations the suitability of the neural network is increased as less fit chromosomes are replaced with better suited ones. This process continues until no improvements have occurred for a specified number of generations. The genetic algorithm generally takes up a great deal of memory and executes much slower than simulated annealing algorithm. Because of this, simulated annealing has become a popular method of neural network training.

Simulated annealing algorithm begins by “randomizing” the weight values taking into consideration the current “temperature” and the suitability of the current weight matrix. The temperature is decreased and the weight matrix ideally converges on an ideal solution. This process continues until the temperature reaches zero or no improvements have occurred for a specified number of cycles. The simulated annealing algorithm executes relatively quickly.

These two algorithms can also be used independently to completely train the neural network to solve problems, but it may not be very successful. Another way to use these two algorithms in neural networks is to help neural networks escaping local minima by training them with genetic or simulated annealing algorithm at the beginning, like it is done on the XOR problem. This makes the network able to correct its' weight matrix faster and achieve better results with few training epochs.

Of course the process of simulated annealing and genetic algorithms may produce a less suitable weight matrix than what was started with. This can happen when a simulated annealing or a genetic algorithm is used against an already well trained network. This lack of improvement is not always a bad thing, as the weight matrix may have moved beyond the local minimum. Further back-propagation training may allow the neural network to converge on a better solution. However, it is still always best to remember the previous local minimum incase a better solution simply cannot be found.

Finally, although using genetic and simulated algorithms sometimes may produce better solutions, Elman network model remains to be the appropriate neural network structure to recognize both temporal and spatial patterns of different nature problems.

BIBLIOGRAPHY

[1] Introduction to Neural Networks with Java, Jeff Heaton

[2] Neural Network Design, Martin T. Hagan, Howard B. Demuth, Mark Beale

[3] Finding Structure in Time, Jeffrey L. Elman, University of California, San Diego – *Cognitive Science*, *14, 179-211* (1990)

[4] Application of Genetic Algorithms and Neural Networks to the Solution of Inverse Heat Conduction Problems, A Tutorial, Keith A. Woodbury, Mechanical Engineering Department, University of Alabama

[5] Introduction to Neural Networks, Nici Schraudolph and Fred Cummins, Istituto Dalle Molle Di Studi sull'Intelligenza Artificiale Lugano, CH

[6] Neural Networks, by Christos Stergiou and Dimitrios Siganos

[7] Using Neural Networks and Genetic Algorithms to Predict Stock Market Returns, by Efstathios Kalyvas, Department of Computer Science, University of Manchester, 2001

[8] Genetic Algorithm, Simulated Annealing Algorithm – Wikipedia, the free encyclopedia

CURRICULUM VITAE

Name and Surname	: Ilir ÇOLLAKU
Date of Birth	: 29.06.1983
Place of Birth	: Prizren - KOSOVA
High School	: “Gjon Buzuku” Gymnasium, Department of Applied Science, Prizren (1998-2002)
Places of Internship	: 1. ProCredit Bank - Kosova, Head Office IT Department, Prishtina (5 weeks) 2. PRONET - IT Consulting, Engineering & Telecommunications, Software Development Department, Prishtina (8 weeks)