Software Reliability Modeling Using Neural Network Technique

[1]Dr.Holkar S.R., [2]Dr. Manu Pratap Singh

[1]Department of Computer Science
Mahatma Gandhi Mahavidyalaya
Ahmedpur Dist.Latur
Maharashtra, India

[2]Professor & Head
Institute of Engineering & Technology
Dr. B. R. Ambedkar University, Agra
Uttar Pradesh, India

***Abstract-* Software reliability is defined as the probability of failure-free software operation for a specified period of time in a specified environment. Software reliability modeling has gained a lot of importance in the recent years. Criticality of software in many of the present day applications has led to a tremendous increase in the amount of work being carried out in this area. The use of intelligent neural network and hybrid techniques in place of the traditional statistical techniques has shown a remarkable improvement in the development of prediction models for software reliability in the recent years. Among the intelligent and the statistical techniques it is not easy to identify the best one since their performance varies with the change in data.**

In this paper, we propose an artificial neural network-based approach for developing the model for software reliability estimation. We first explain the neural networks from the mathematical viewpoints of software reliability modeling. That is, we will show how to apply neural network to develop a model for the prediction of software reliability. The implementation of proposed model is shown with real software failure data sets. From simulation results, we can see that the proposed model significantly outperforms the traditional software reliability models.

***Keywords:* Software Reliability model, estimation of software reliability, Artificial Neural network, Reliability Prediction.**

1. Introduction

Current scenario of working in modern society is heavily depending upon the computation. Here the world computation caries the applications those runs on the computers i.e. software. The natural demand of the work and dependence on the performance of software it is required that the software must be reliable. Hence under these conditions the software reliability is an important aspect of the efficient and effective working so that the research of the computer software reliability becomes more and more essential. Thus, the prediction of software reliability and its estimation is an important process with the software development. The software reliability is defined as the probability that the software will operate without a failure under a given environmental condition during a specified period of time [1]. Since 1970, many software reliability growth models (SRGMs) [2-4] have been proposed. Most published reliability analysis methods are based on parametric and non-parametric statistical models of time-to-failure data and its associated metrics. The underlying assumption of these methods is that a coherent, statistical model of system failure time can be developed that will prove stable enough to accurately predict a system's behavior over its lifetime. However, given the increasing complexity of the component dependencies and failure behaviors of today's real-time safety-critical systems, the statistical models may not be feasible to build or computationally tractable. This has led to an increasing interest in more flexible modeling frameworks for reliability analysis. In general, there are two major types of software reliability models: the deterministic and the probabilistic. The deterministic one is employed to study the number of distinct operators and operands in the program. The probabilistic one represents the failure occurrences and the fault removals as probabilistic events. The probabilistic models can be further classified into different classes, such as error seeding, failure rate, and non-homogeneous Poisson process (NHPP). Among these classes, the NHPP models are the most popular ones. The reason is the NHPP model has ability to describe the software failure phenomenon. The first NHPP model, which strongly influences the development of many other models, was proposed by Goel and Okumoto [6]. Later, Ohba [7] presented a NHPP model with S-shaped mean value function. Yamada and Osaki [8, 9] also made further progress in various S-Shaped NHPP models. Many software reliability models have been developed from past three decades. They are developed through either an analytical or data-driven approach. Analytical software reliability growth models (SRGMs) represented by Non-Homogeneous Poisson Process (NHPP), are stochastic models focusing on software failure process. Data-driven models are developed from historical software fault-related data, following the approach of regression or time series analysis. Although these NHPP models are widely used, they impose certain restrictions or a priori assumptions about the nature of software faults and the stochastic behavior of software failure process. To overcome this problem, several alternative solutions are introduced. One possible solution is to employ the neural network, since it can build a model adaptively from the given data set of failure processes. Many researchers [10-19] have been successfully adapted neural networks to software reliability issues. Motivated by these successful cases, we employ the neural network to solve the problems for software reliability assessment.

ANN (Artificial Neural Network) software reliability models have recently aroused more research interest [20 -23]. Traditionally, both kinds of models only consider single fault detection process (FDP) and data for analysis are only from FDP. However, while data from both FDP&FCP (fault correction process) are available, NHPP and ANN models can be extended into paired NHPP models and combined ANN models, providing more accurate predictions. Generally speaking, data-driven approach is much less restrictive in assumptions compared to analytical approach. Generally, accurate predictions cannot be obtained in the early phase of testing through both approaches, as there are not enough data for parameter estimation or learning at this stage. However, early software reliability prediction is useful for timely software development process control. Smidts et al. [24] tried to develop early prediction with Bayes framework with subjective or/and objective data from older projects. Xie et al. [25] proposed a more practical approach to develop early reliability prediction based on NHPP models. In that paper, NHPP models were adjusted to incorporate failure history information from a similar project by assuming the same failure rate. This approach has also been extended to paired NHPP models, taking both the testing and debugging environments as the same [26]. In addition, historical fault-related data reuse is a practical approach for modern mature software manufacturers, as they have plenty of reusable information from previous releases or similar projects stored in their database.
In this paper, we propose an artificial neural network-based approach for developing the model for software reliability estimation. We first explain the neural networks from the mathematical viewpoints of software reliability modeling. That is, we will show how to apply neural network to develop a model for the prediction of software reliability. The implementation of proposed model is shown with real software failure data sets. From simulation results, we can see that the proposed model significantly outperforms the traditional software reliability models.

2. Software reliability and Modeling with Neural Network:

There are various models for the estimation of prediction for software reliability but most of the software reliability models involve certain restrictions or assumptions. Therefore to select an appropriate model according to the characteristics of the software projects is challenging. In order to locate the suitable model, two approaches are adapted. The first one is to design a guideline, which could suggest fitting models for software projects. The other is to select the one with the highest confidence after various assessments. In the last few years many research studies has been carried out in the area of software reliability modeling. They included the application of neural networks, fuzzy logic models; Genetic algorithms (GA) based neural networks, recurrent neural networks, Bayesian neural networks, and support vector machine (SVM) based techniques, to name a few. For example, Karunaithi et al. [10] applied some kinds of neural network architecture to estimate the software reliability and used the execution time as input, cumulative the number of detected faults as desired output, and encoded the input and output into the binary bit string. The results showed that the neural network approach was good at identifying defect-prone modules software failures. Khoshgoftaar et al. [13-14] ever used the neural network as a tool for predicting the number of faults in programs. They introduced an approach for static reliability modeling and concluded that the neural networks produce models with better quality of fit and predictive quality. In addition, Cai et.al [18] examined the effectiveness of the neural network approach in handling dynamic software reliability data overall and present several new findings. They found that the neural network approach is more appropriate for handling datasets with `smooth' trends than for handling datasets with large fluctuations and the training results are much better than the prediction results in general. Sitte [14] made a comparative study of neural networks and parametric-recalibration models in software reliability prediction and found neural networks to be much simpler to use and also to be better predictors. Also, through empirical results it was shown that the neural network models are better trend predictors. Ho et al [19] performed a comprehensive study of connectionist models and their applicability to software reliability prediction and found them to be better and more flexible than the traditional models. A comparative study was performed between their proposed modified Elman recurrent neural network, with the more popular feedforward neural network, the Jordan recurrent model, and some traditional software reliability growth models. Numerical results show that the proposed network architecture performed better than the other models in terms of predictions. Despite of the recent advancements in the software reliability growth models, it was observed that different models have different predictive capabilities and also no single model is suitable under all circumstances. Tian and Noore [27] proposed an on-line adaptive software reliability prediction model using evolutionary connectionist approach based on multiple-delayed-input single-output architecture. The proposed approach, as shown by their results, had a better performance with respect to next-step predictability compared to existing neural network model for failure time prediction. Tian and Noore [22] proposed an evolutionary neural network modeling approach for software cumulative failure time prediction. Their results were found to be better than the existing neural network models. It was also shown that the neural network architecture has a great impact on the performance of the network. According to Bai et al [28] Bayesian networks show a strong ability to adapt in problems involving complex variant factors. They developed a software prediction model based on Markov Bayesian networks, and a method to solve the network model was proposed. Reformat [29] proposed an approach leading to a multi-technique knowledge extraction and development of a comprehensive meta-model prediction system in the area of corrective maintenance of software. The system was based on evidence theory and a number of fuzzy-based models. In addition they carried out a detailed case study for estimating the number of defects in a medical imaging system using the proposed approach. Pai and Hong [30] have applied support vector machine (SVM) for forecasting software reliability in which simulated annealing (SA) algorithm was used to select the parameters of the SVM model. The experimental results show that the proposed model gave better predictions than the other compared methods. Su and Huang [31] showed how to apply neural networks to predict software reliability. Further they made use of the neural network approach to build a dynamic weighted combinational model (DWCM) and experimental results show that the proposed model gave significantly better predictions. Also recently, neural networks were applied for predicting faults in object-oriented software [32]. The study showed neural network models to be performing much better than the statistical methods. Application of intelligent techniques in place of the statistical techniques has increased by leaps and bounds in the recent years. Application of Neural network techniques in software reliability engineering has come up recently [33]. Despite the recent advancements in the software reliability growth models, it was observed that different models have different predictive capabilities and also no single model is suitable under all circumstances.

Modeling of Neural Network

A simple model of the neuron that shows inputs from other neurons and a corresponding output is depicted in figure 1. As can be seen in the figure, three neurons feed the single neuron, with one output emanating from the single neuron.

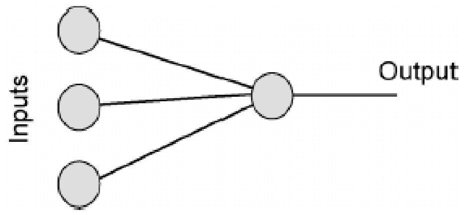

Figure 1: Artificial neuron with inputs and a single output.

Mathematically, the inputs and the corresponding weights are vectors which can be represented as $(i_1, i_2 ... i_n)$ and $(w_1, w_2 ... w_n)$. The total input signal is the dot, or inner, product of these two vectors. This simplistic summation function is found by multiplying each component of the i vector by the corresponding component of the w vector and then adding up all the products. $input_1 = i_1 * w_1$, $input_2 = i_2 * w_2$, etc., are added as $input_1 + input_2 + ... + input_n$. The result is a single number, not a multi-element vector. Geometrically, the inner product of two vectors can be considered a measure of their similarity. If the vectors point is in the same direction then the inner product will maximum. If the vectors point in opposite direction (180 degrees out of phase), their inner product is minimum. The summation function can be more complex than just the simple input and weight sum of products. The input and weighting coefficients can be combined in many different ways before passing on to the transfer function. In addition to a simple product summing, the summation function can select the minimum, maximum, majority, product, or several normalizing algorithms. The specific algorithm for combining neural inputs is determined by the chosen network architecture and paradigm.

The connection weight matrix **W** = [w_{ij}], where w_{ij} denotes the connection weight from node i to node j, is used to describe the network architecture. When $w_{ij} = 0$, there is no connection from node i to node j. By setting the connection weights between nodes as zero, one can realize different network topologies. Basically, all artificial neural networks have a similar structure or topology as shown in figure 2. In that, structure some of the neurons interfaces to the real world to receive its inputs. Other neurons provide the real world with the network's outputs. This output might be the particular character that the network thinks that it has scanned or the particular image it thinks is being viewed. All the rest of the neurons are hidden from view.

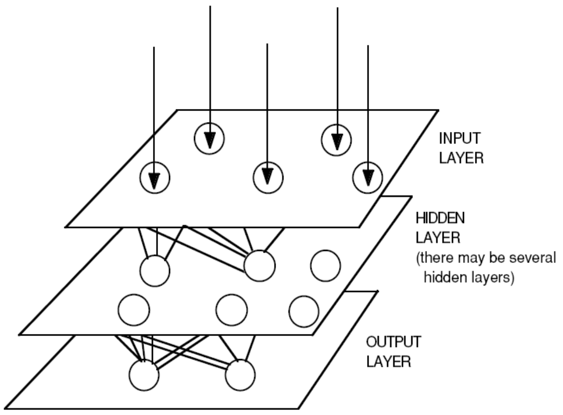

Figure 2: A simple neural network diagram

According to the architecture, neural networks can be grossly classified into feedforward neural networks (FNNs), recurrent neural networks (RNNs), and their combinations. Some popular network topologies include fully connected layered FNNs, RNNs, lattice networks, layered FNNs with lateral connections. The nonzero elements of **W** can be adapted by a learning algorithm. In an FNN, the connections between neurons are in a feedforward manner. The network is usually arranged in the form of layers. In layered FNNs, there is no connection between the neurons within each layer, and no feedback between layers. A fully connected layered FNN is a network such that every node in any layer is connected to every node in its adjacent forward layer. When some of the connections are missing, it becomes a partially connected layered FNN. FNNs exhibit no dynamic properties and the networks are simply a nonlinear mapping. The popular MLP and RBFN are fully connected layered FNNs. In an RNN, there is at least one feedback connection that corresponds to an integration operation or unit delay. Thus, an RNN actually represents a nonlinear dynamic system.

Most applications require networks that contain at least the three normal types of layers - input, hidden, and output. The layer of input neurons receives the data either from input files or directly from electronic sensors in real-time applications. The output layer sends information directly to the outside world, to a secondary computer process, or to other devices such as a mechanical control system. Between these two layers there can be many hidden layers. These internal layers contain many of the neurons in various interconnected structures. The inputs and outputs of each of these hidden neurons simply go to other neurons. In most networks, each neuron in a hidden layer receives the signals from all of the neurons in a layer above it, typically an input layer. After a neuron performs its function it passes its output to all of the neurons in the layer below it, providing a feed forward path to the output. The way that the neurons are connected to each other has a significant impact on the operation of the network.

After finalizing the architecture of the neural network for a given application, the training or learning is required for getting the desired output from the network. Training or learning of a neural network is an optimization process that produces an output that is as close as possible to the desired output by adjusting network parameters. This kind of parameter estimation is also called learning or *training algorithm*. Neural networks are usually trained by epoch. An epoch is a complete run when all the training examples are presented to the network and are processed using the learning algorithm only once. After learning, a neural network represents a complex relationship, and possesses the ability for generalization. When a new input is presented to the trained neural network, a reasonable output is produced. Learning methods are conventionally divided into supervised, unsupervised, reinforcement, and evolutionary learning. Supervised learning is widely used in pattern recognition, approximation, control, modeling and identification, signal processing, and optimization. Reinforcement learning is usually used in control. Unsupervised learning schemes are mainly used for pattern recognition, clustering, vector quantization, signal coding, and data analysis. Evolutionary computation is a class of optimization techniques, which can be used to search for the global minima/maxima of an objective function. Evolutionary learning is used for adjusting neural network architecture and parameters using an evolutionary algorithm (EA), and can also be used to optimize the control parameters in a supervised or unsupervised learning algorithm.

After finalizing the architecture of the neural network for a given application, the training or learning is required for getting the desired output from the network. Training or learning of a neural network is an optimization process that produces an output that is as close as possible to the desired output by adjusting network parameters. This kind of parameter estimation is also called learning or *training algorithm*. Neural networks are usually trained by epoch. An epoch is a complete run when all the training examples are presented to the network and are processed using the learning algorithm only once. After learning, a neural network represents a complex relationship, and possesses the ability for generalization. When a new input is presented to the trained neural network, a reasonable output is produced. Learning methods are conventionally divided into supervised, unsupervised, reinforcement, and evolutionary learning. Supervised learning is widely used in pattern recognition, approximation, control, modeling and identification, signal processing, and optimization. Reinforcement learning is usually used in control. Unsupervised learning schemes are mainly used for pattern recognition, clustering, vector quantization, signal coding, and data analysis. Evolutionary computation is a class of optimization techniques, which can be used to search for the global minima/maxima of an objective function. Evolutionary learning is used for adjusting neural network architecture and parameters using an evolutionary algorithm (EA), and can also be used to optimize the control parameters in a supervised or unsupervised learning algorithm.

Supervised learning is based on a direct comparison between the actual network output and the desired output. Network parameters (weights) are adjusted by a combination of the training pattern set and the corresponding errors between the desired output and the actual network response. The errors first calculated then propagated back through the system, causing the system to adjust the weights, which evolve the learning process. The pattern set, which enables the learning, is called the "training set." During the learning of a network the same set of data is processed many times as the connection weights are ever refined. So supervised learning can be defined as a closed-loop feedback system, where the error is the feedback signal. The trained network is used to emulate the system. To control a learning process, a criterion is needed to decide the time for terminating the process. For supervised learning, an error measure, which shows the difference between the network output and the output from the training samples, is used to guide the learning process. The error measure is usually defined by the mean squared error and calculated by the error function:

$$E = \frac{1}{N}\sum_{p=1}^{N}\left|z_p - \hat{z}_p\right|^2 \qquad (2.1)$$

Where the N is the total number of patterns pair from a sample training set, z_p is the actual output and $\hat{z}_p$ is the output calculated by the network for p^{th} pair of sample of training set. This function is also known as the objective function to optimize the network. The error E is calculated a new after each epoch. This process of network training is terminated when E is sufficiently small or a failure criterion is met. To minimize the error up to the non significant value, a gradient-descent procedure is usually applied. The LMS [34] and back propagation algorithms [35] are two early, but most popular, supervised learning algorithms. Both of them are derived using a gradient-descent procedure. When finally, the system has been correctly learned, and no further learning is needed, the weights can, if desired, be "frozen." In some systems, this finalized network is then turned into hardware so that it can be fast. Other systems don't lock themselves in but continue to learn while in production use.

Unsupervised learning involves no target values. It tries to auto associate information from the inputs to decide what features it will use to group the input data. Unsupervised learning is solely based on the correlations among the input data, and is used to find the significant patterns or features in the input data without any supervision. A criterion is needed to terminate the learning process. Without a termination criterion, a continuous learning process continues even when a pattern, which does not belong to the training patterns set, is presented to the network. The network is adapted according to a constantly changing environment. Hebbian learning [36], competitive learning [37], and Kohonen's SOM [38, 39] are the three mostly used unsupervised learning approaches. In general the unsupervised learning is slow to settle into stable conditions. In Hebbian learning [36], learning is a purely local phenomenon, involving only two neurons and a synapse. The synaptic weight change is proportional to the correlation between the pre and postsynaptic signals. The *C*-means algorithm is a popular competitive learning-based clustering method [40]. By using the correlation of the input vectors, the learning rule changes the network weights to group the input vectors into clusters. The Boltzmann machine [41] uses a kind of stochastic training technique known as SA [42], which can been treated as a special type of unsupervised learning based on the inherent property of a physical system. Tuevo Kohonen, an electrical engineer at the Helsinki University of Technology developed a self-organizing network [43], sometimes called an auto-association that learns without the benefit of knowing the right answer. It is an unusual looking network in that it contains one single layer with many connections. The weights for those connections have to be initialized and the inputs have to be normalized. The neurons are set up to compete in a winner-take-all fashion. The other most common algorithm of unsupervised learning is the Hopfield neural network model [44, 45] of associative memory. Hopfield network is fully interconnected network with symmetric weights, no self-feedback and asynchronous update of the state of processing elements.

Reinforcement learning [46] is a special case of supervised learning, where the exact desired output is unknown. It is based only on the information as to whether or not the actual output is close to the estimate. Explicit computation of derivatives is not required. This, however, presents a slower learning process. Reinforcement learning is a learning procedure that *rewards* the neural network for its *good* output result and *punishes* it for the *bad* output result. It is used in the case when the correct output for an input pattern is not available and there is need for developing a certain output. The evaluation of an output as *good* or *bad* depends on the specific problem and the environment. For a control system, if the controller still works properly after an input, the output is judged as *good*; otherwise, it is considered as *bad*. The evaluation of the output is binary, and is called *external reinforcement*. Thus, reinforcement learning is a kind of supervised learning with the external reinforcement as the error signal. Reinforcement learning can learn the system structure by trial-and-error, and is suitable for online learning [47-48].

Evolutionary learning approach is attractive since it can handle the global search problem better on a vast, complex, multimodal, and no differentiable surface. It is not dependent on the gradient information of the error (or fitness) function, and thus is particularly appealing when this information is unavailable or very costly to obtain or estimate. Evolutionary Algorithms can be used to search for the optimal control parameters in supervised as well as unsupervised learning by optimizing their respective objective functions. It can also be used as an independent training method for network parameters by optimizing the error function. Evolutionary Algorithms are widely used for training neural networks and tuning fuzzy systems, and are generally much less sensitive to the initial conditions. They always search for a globally optimal solution, while supervised and unsupervised learning algorithms can only find a local optimum in a neighborhood of the initial solution [49].

The Backpropagation Learning Algorithm

The backpropagation (BP) learning algorithm is currently the most popular supervised learning rule for performing pattern classification tasks [50]. It is not only used to train feed forward neural networks such as the multilayer perceptron, it has also been adapted to recurring neural networks. The BP algorithm is a generalization of the delta rule, known as the least mean square *algorithm*. Thus, it is also called the *generalized delta rule*. The BP overcomes the limitations of the perceptron learning enumerated by Minsky and Papert [51].Due to the BP algorithm, the MLP can be extended to many layers. The BP algorithm propagates backward the error between the desired signal and the network output through the network. After providing an input pattern, the output of the network is then compared with a given target pattern and the error of each output unit calculated. This error signal is propagated backward, and a closed-loop control system is thus established. The weights can be adjusted by a gradient-descent-based algorithm. In order to implement the BP algorithm, a continuous, nonlinear, monotonically increasing, differentiable activation function is required. The two most-used activation functions are the logistic function and the hyperbolic tangent function, and both are sigmoid functions.

We want to train a multi-layer feed forward network by gradient descent to approximate an unknown function, based on some training data consisting of pairs $(x\ ,z\ )\in S$. The vector $\boldsymbol{x}$ represents a pattern of input to the network, and the vector $\boldsymbol{z}$ the corresponding desired output from the training set $\boldsymbol{S}$. The objective function for optimization is defined as the error MSE can be calculated by equation (2.1).

All the network parameters $W^{(m-1)}$ and θ^m , $m = 2\ \cdot\ \cdot\ \cdot M$, can be combined and represented by the matrix $W = \lfloor w_{ij} \rfloor$. The error function E can be minimized by applying the gradient-descent procedure as:

$$\Delta\ W = -\eta \frac{\partial E}{\partial W} \tag{2.2}$$

where η is a learning rate or step size, provided that it is a sufficiently small positive number.

Applying the chain rule, the equation (2.2) can express as

$$\frac{\partial E}{\partial w_{ij}^{(m)}} = \frac{\partial E}{\partial u_j^{(m+1)}} \frac{\partial u_j^{(m+1)}}{\partial w_{ij}^{(m)}} \tag{2.3}$$

while
$$\frac{\partial u_j^{(m+1)}}{\partial w_{ij}^{(m)}} = \frac{\partial}{\partial w_{ij}^{(m)}} \left(\sum w_j^{(m)} o^{(m)} + \theta_j^{(m+1)}\right) = o_i^{(m)} \tag{2.4}$$

and
$$\frac{\partial E}{\partial u_j^{(m+1)}} = \frac{\partial E}{\partial o_j^{(m+1)}} \frac{\partial o_j^{(m+1)}}{\partial u_j^{(m+i)}} = \frac{\partial E}{\partial o_j^{(m+1)}} \phi_j^{(m+1)}\left(u_j^{(m+1)}\right) \tag{2.5}$$

For the output unit $m=M-1$

$$\frac{\partial E}{\partial o_j^{(m+1)}} = e_j \tag{2.6}$$

For the hidden units, $m = 1,2,3\ldots\ldots\ldots,M-2$,

$$\frac{\partial E}{\partial o_j^{(m+1)}} = \sum_{\omega=2}^{j_{m+2}} \frac{\partial E}{\partial u_\omega^{m+2}} \omega_{j\omega}^{m+1} \tag{2.7}$$

Define the delta function by

$$\delta_j^{(m)} = \frac{\partial E}{\partial u_p^{(m)}} \tag{2.8}$$

for $m = m = 2,3\dots\dots\dots,M$. By substituting (1.78), (1.79), and (1.80) into (1.83), we finally obtain the following.
For the output units, $m = M - 1$,

$$\delta_j^{(M)} = -e_j \phi_j^{(M)}\left(u_j^{(M)}\right) \tag{2.9}$$

For hidden units, $m = 1,\dots\dots.,M - 2$,

$$\delta_j^{(M)} = -e_j \phi_j^{(M)}\left(u_j^{(M)}\right) \sum_{\omega=1}^{J_{m+2}} \delta_\omega^{(m+2)} \omega_\omega^{m+1} \tag{2.10}$$

Equations (1.84) and (1.85) provide a recursive method to solve $\delta_j^{(m+1)}$ for the whole network. Thus, **W** can be adjusted by

$$\frac{\partial E}{\partial \omega_{ij}^{(m)}} = -\delta_j^{(m+1)} o_i^{(m)} \tag{2.11}$$

For the activation functions, we have the following relations:
For the logistic function

$$\phi(u) = \beta\phi(u)[1 - \phi(u)] \tag{2.12}$$

For the *tanh* function

$$\phi(u) = \beta\left[1 - \phi^2(u)\right] \tag{2.13}$$

The update for the biases can be in two ways. The biases in the (m+1)th layer $\theta^{(m+1)}$ can be expressed as the expansion of the weight $W^{(m)}$, that is, $\theta^{(m+1)} = \left(\omega_{0,1}^{(m)},\dots\dots\dots\dots\dots\omega_{0,J_{m+1}}^{(m)}\right)$. Accordingly, the output **o**(m) is expanded into $o^{(m)} = \left(1, o_1^{(m)},\dots\dots\dots\dots, o_{J_m}^{(m)}\right)$. Another way is to use a gradient-descent method with regard to $\boldsymbol{\theta}^{(m)}$, by following the above procedure. Since the biases can be treated as special weights, these are usually omitted in practical applications. The algorithm is convergent in the mean if $0 < \eta < \frac{2}{\lambda_{\max}}$, where $\lambda_{\max}$ is the largest eigenvalue of the autocorrelation of the vector **x**, denoted as **C**. When η is too small, the possibility of getting stuck at a local minimum of the error function is increased. In contrast, the possibility of falling into oscillatory traps is high when η is too large. By statistically preprocessing the input patterns, namely, de correlating the input patterns, the excessively large eigenvalues of **C** can be avoided and thus, increasing η can effectively speed up the convergence. PCA preconditioning speeds up the BP in most cases, except when the pattern set consists of sparse vectors. In practice, η is usually chosen to be $0 < \eta < 1$ so that successive weight changes do not overshoot the minimum of the error surface. The BP algorithm can be improved by adding a momentum term:

$$\Delta W(t) = -\eta \frac{\partial E}{\partial W} + \alpha \Delta W(t-1) \tag{2.14}$$

where α is the momentum factor, usually $0 < \alpha \le 1$. The typical value for α is 0.9. This method is usually called the *BP with momentum (BPM)* algorithm.
The BP algorithm is a supervised gradient-descent technique, wherein the MSE between the actual output of the network and the desired output is minimized. It is prone to local minima in the cost function. The performance can be improved and the occurrence of local minima reduced by allowing extra hidden units, lowering the gain term, and by training with different initial random weights.

3. Software Reliability Modeling with Artificial Neural network

It has been seen from the previous section that the error function i.e. mean square error (MSE) works as the objective function for the convergence of neural network. This objective function can be considered as compound functions. In other words, if we can derive a form of compound functions from the conventional software reliability models, we can build a neural-network-based model for software reliability. Therefore to accomplish this we consider the logistic growth curve model [3]. This model simply fits the mean value function with a form of the logistic function. Its mean value function is given by:

$$m(t) = \frac{a}{1 + ke^{-bt}}, a > 0, b > 0, k > 0 \tag{3.1}$$

We can derive a form of compound functions from its mean value function by replacing k with e^{-c} as:

$$m(t) = \frac{a}{1 + ke^{-bt}} = \frac{a}{1 + e^{-c}e^{-bc}} = \frac{a}{1 + e^{-(bt+c)}} \tag{3.2}$$

Assume that: $g(x) = bx + c,\ f(x) = \frac{1}{1+e^{-x}}, m(x) = ax$ (3.3)

Therefore, we can get:

$$m(f(g(x))) = m(f(bx+c)) = m(\frac{1}{1+e^{-(bt+c)}}) = \frac{a}{1+e^{-(bt+c)}} \quad (3.4)$$

This means that the mean value function of logistic growth curve model is composed of $g(x)$, $f(x)$ and $m(x)$. Subsequently, we derive the compound functions from the viewpoints of neural network. Consider the basic feedforward neural network as shown in figure 3. This network has only one neuron in each layer as w_{11}^1 and w_{11}^0 is the weight and b_1, b_0 is the bias. When the input $x(t)$, at time t is fed to the input layer, we have the following expressions for the hidden layer and output layer respectively:

$$g = w_{11}^1 t + b_1; h(t) = f(g)$$
$$y = w_{11}^0 h(t) + b_0; O(t) = f(y) \quad (3.5)$$

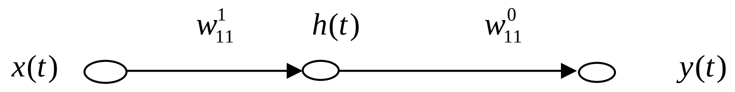

Fig 3: Feed-Forward neural network architecture with single neuron in each layer

Now we consider the output signal function from the neuron as:

$$f(x) = \frac{1}{1+e^{-x}}; g(x) = x \quad (3.6)$$

Now after removing the bias term from the output layer we have:

$$O(t) = w_{11}^o h(t) = w_{11}^0 f(h(t)) = \frac{w_{11}^0}{1+e^{-(w_{11}^1 x + b1)}} \quad (3.7)$$

Therefore from the equation (2.21), we can see that it models the neural network as the logistic growth curve model. Hence with the same process we can derive the neural network into many other existing models. Let us consider a neural network with an output signal function as $1-e^{-x}$ in the hidden layer, $w_{11}^1 = b, w_{11}^0 = a$ and there is no bias in both hidden layer and output layer. Thus we have:

$$h(t) = 1 - e^{-bt} and O(t) = a(1-e^{-bt}) \quad (3.8)$$

Hence it is the Goel-Okumoto model [5] and this model has strongly influenced the development of other models. There is another example for confirming that the neural network can apply for software modeling. Now, if we consider a neural network with output signal function as $1-(1+x)e^{-x}$ in the hidden layer, $w_{11}^1 = b, w_{11}^0 = a$ and output layer than we have:

$$h(t) = 1-(1+bt)e^{-bt}; and, O(t) = a(1-(1+bt)e^{-bt}) \quad (3.9)$$

This can see that the equation 2.23 exhibits the Yamada Delay S-shaped model [7, 8]. The model describes the fault detection process as a learning process in which testing members become familiar with the test environment or testing tools. Thus, their testing skills gradually improved. We have mentioned that selecting a particular model is very important for the estimation of software reliability. But sometimes, software projects can not fit the assumptions of a unique model. To overcome this problem, Lyu and Allen [52] have proposed a solution by combining the results of different software reliability models. This approach inspired us to use the neural-network-based approach to combine the models. Thus, we consider an application of proposed approach to each combinational model. We implement the neural network with single input single output but more than one neuron in the hidden layer. In this approach we consider the number of neurons in the hidden layer by the number of models which assumptions are partially suitable to the software project. We use different output signal functions in the hidden layer at the same time to achieve combinational models. Hence in order to implement the combinational model we consider the combination of GO model, the Delay-S-shaped, and the logistic growth curve model. Now we consider the $1-e^{-x}, 1-(1+x)e^{-x}, \frac{1}{1+e^{-x}}, and, \frac{e^x - e^{-x}}{e^x + e^{-x}}$ output signal functions in the hidden layer. The output of the network is defined as:

$$O(t) = w_{21}(1-e^{-w_{12}t}) + w_{22}(1-(1+w_{12}t)e^{-w_{12}t}) + \frac{w_{23}}{1+e^{-w_{12}}} + w_{24}(\frac{e^{w_{12}t} - e^{-w_{12}t}}{e^{w_{12}t} + e^{-w_{12}t}}) \quad (3.10)$$

Thus, this combinational model can adapt the characteristic of the given data set. Hence, this model itself can be considered as a general model for all software projects but this method differs from those proposed by Lyu et al., since their model only combined the results from various models based on assigned weights. This proposed approach automatically determines the weight of each model based on the characteristic of the given data set.

4. Experiment and implementation detail

Here in this section we implement the proposed method and prepare the simulation for it. In this process the implementation of the proposed neural-network-based models is described. The following steps are required to apply our approach to model the prediction for software reliability.

1. Analyze the assumptions of the software project and if there is a suitable model, select the model. If there is no suitable model, select those models which partially meet the assumptions of project.
2. Construct the neural network of selected models by designing the activation functions and bias.
3. Given n fault-detection time interval data $x_1, x_2, \dots\dots\dots, x_n$, we accumulate the execution time and divide it into 100 time units $t_1, t_2, \dots\dots\dots, t_{100}$ and then calculate the number of failures, $T_1, T_2, \dots\dots\dots, T_{100}$.
4. Feed the input output pattern pairs of $\{t_i, T_i\}$ to the network to train the network by the back-propagation algorithm.
5. When the network trained, feed the future testing time to the network, and the network output is represented as the predicated number of faults in the future.

Therefore with these five steps, we can use the neural-network-based models to predict the reliability of the software from the prediction of total faults to be detected in the future. Hence to validate the performances of our proposed approach, we use the following real software failure data set as shown in table 1.

Time (t)	**Observed failure in software (T)**	**Time (t)**	**Observed failure in software (T)**
0	5.7683	34	10.6301
1	9.5743	35	8.333
2	9.105	36	11.315
3	7.9655	37	9.4871
4	8.6482	38	8.1391
5	9.9887	39	8.6713
6	10.1962	40	6.4615
7	11.6399	41	6.4615
8	11.6275	42	7.6955
9	6.4922	43	4.7005
10	7.901	44	10.0024
11	10.2679	45	11.0129
12	7.6839	46	10.8621
13	8.8905	47	9.4372
14	9.2933	48	6.6644
15	8.3499	49	9.2294
16	9.0431	50	8.9671
17	9.6027	51	10.3534
18	9.3736	52	10.0998
19	8.5869	53	12.6078
20	8.7877	54	7.1546
21	8.7794	55	10.0033
21	8.0469	56	9.8601
22	8.0469	57	7.8675
23	10.8459	58	10.5757
24	8.7416	59	10.9294
25	7.5443	60	10.6604
26	8.5941	61	12.4972
27	11.0399	62	11.3745
28	10.1196	63	11.9158
29	10.1786	64	9.575
30	5.8944	65	10.4504
31	9.546	66	10.5866
32	9.6197	67	12.7201
33	10.3852	68	12.5982

Table 1: Data of Software failure in the given time

Now first we analyze this data set by using Laplace trend test [53] because software reliability studies are usually based on the application of growth models to obtain various measures. Hence in this model, let us consider the time interval [0, t] be divided into k units of time, and let $n(i)$ be the number of faults observed during time unit i. The Laplace factor, $u(k)$, is given by:

$$u(k) = \frac{\sum_{i=1}^{k}(i-1)n(i) - \frac{(K-1)}{2}\sum_{i=1}^{k}n(i)}{\sqrt{\frac{k^2-1}{12}\sum_{i=1}^{k}n(i)}} \tag{4.1}$$

Since the negative value of $u(k)$ indicates decreasing failure intensity, and thus reliability growth, while positive value indicates increasing failure intensity, and reliability decrease. Hence figure 4 show the Laplace trend test for the given data set. Thus we can find that the reliability of the system is growth. If the trend of the data set is shown as reliability growth, then this application of data set is suitable to use exponential type of model, for example, the GO model. If the trend of the data set is shown as reliability decay followed by growth, then the data set is suitable to use the S-Shaped model, for example, the Delay S-Shaped model or the Inflection S-Shaped model.

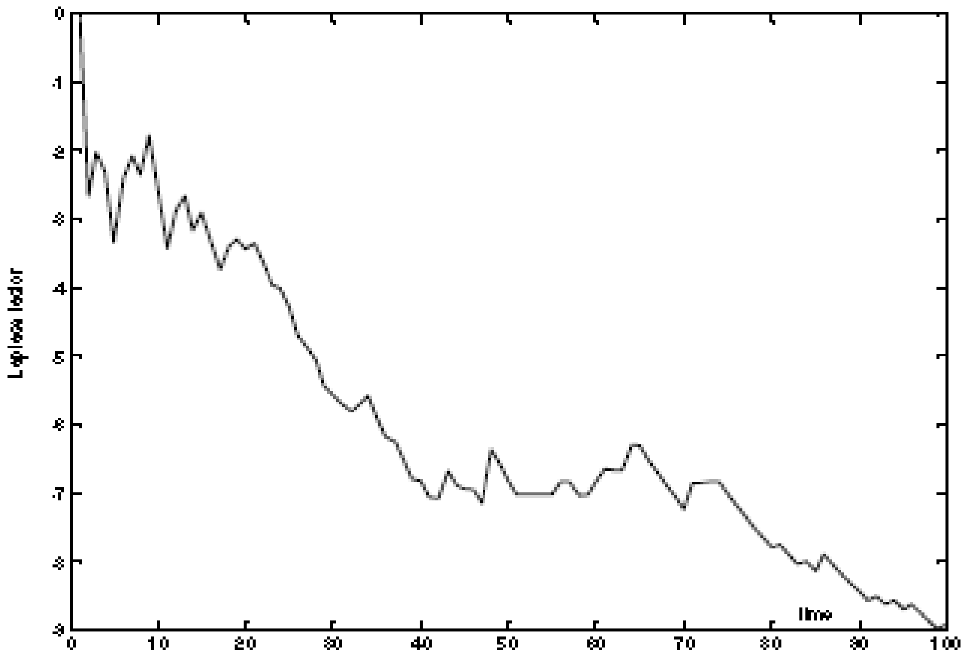

Figure 4: Laplace trend for the given data set

Here in this simulation design we consider the three criteria for the evaluation. These criteria's are Root mean Square error (RMSE), Average Error (AE) and Average Bias (AB). Beside this we also consider the differences between predicted and observed failure behavior. The purpose of this difference is to determine whether the predictions are on average close to the true distributions or not. A common way to test the significant difference between predictions and actual values is Kolmogorov distance (KD). Thus we consider the KD to determine that whether the predictions are on average close to the true distributions or not.

5. Experimental Results

It has been discussed in simulation design that we consider two models of neural networks. The first model is NN-1 which is neural network model with one neuron in hidden layer and the NN-3 means combinational model with three neurons in the hidden layer. Here our objective is to analyze the performances of our proposed neural-network-based models with three selected software reliability growth models i.e. the Goel-Okumoto model (GO), the Yamada Delay S-Shaped model (DS), and the Inflection S-Shaped model (IS). We use the least square estimation to solve the parameters of above model. Here we find that the NN models give the better fit than others on the basis of root mean square error (RMSE) criteria as shown in Table 2. It can observe from the table that the NN-2 model has significantly better performance for RMSE in comparison. The neural networks have been trained with different sizes of the training set as shown in table 1, varied from 30% of the total testing time to 100% testing time. After training, we feed the t_{100} to predict the number of faults, $M_{10}(t_{100})$. Then, we calculate the Prediction error (PR) between the actual number of faults and the predicated faults. Following the similar procedures, we can calculate the R_{20}, ..., R_{100}. Table 3 gives the relative error in fault predictions for each model. This result shows that the endpoint predicted value of the neural networks models can be much closed to the actual value. That implies the neural-network-based models have better fault predictions than others. This comparison can also observe from the graph of figure 5

	Neural Network with Single Neuron (NN-1)	**Neural Network as Combinational Model (NN-2)**	**GO Model**	**DS Model**	**IS Model**
RMSE	3.47	1.17	5.74	11.27	3.98
AE	17.68	14.81	16.29	23.90	8.70

AB	-17.68	-14.81	-16.29	-23.90	-8.70
KD	0.118	0.083	0.091	0.263	0.85

Table 2: Comparisons of RMSE, AE, AB and KD

R	Neural Network with Single Neuron (NN-1)	Neural Network as Combinational Model (NN-2)	GO Model	DS Model	IS Model
R_{20}	-0.388	-0.386	-0.296	-0.416	-0.133
R_{30}	-0.382	-0.372	-0.285	-0.406	-0.127
R_{40}	-0.319	-0.302	-0.274	-0.350	-0.155
R_{50}	-0.287	-0.185	-0.242	-0.309	-0.145
R_{60}	-0.228	-0.165	-0.194	-0.263	-0.116
R_{70}	-0.065	-0.084	-0.131	-0.208	-0.071
R_{80}	-0.084	-0.033	-0.083	-0.159	-0.039
R_{90}	-0.033	-0.027	-0.056	-0.122	-0.025
R_{100}	-0.017	-0.014	-0.039	-0.094	-0.018

Table 3: Relative error in Fault Predictions

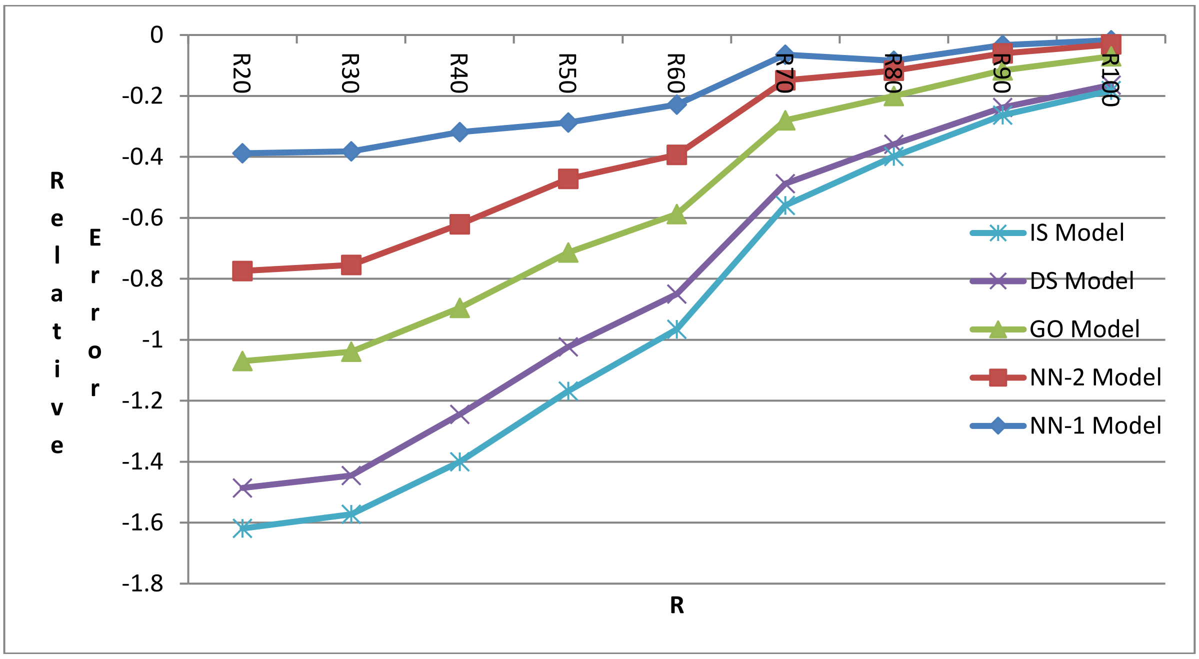

Figure 5: Comparison Chart of relative error for the different models

6. Conclusion

In this proposed approach we applied neural network modeling for the prediction of reliability in any given software. It has been shown that the neural network modeling is applied to represent the existing software growth models in detail. Further the neural-network-based modeling is used to achieve a dynamic weighted combinational model. To validate proposed models, we use a numerical example of real software failure data set. We have compared the performances of the neural network models with some conventional software growth models from point of view of goodness of fit and prediction ability for the faults. The relative error in the prediction is also calculated on the basis of root mean square error. It has been confirmed from the experimental results that the neural network models are workable and have more accuracy with both goodness of fit and the prediction ability of faults compared to existing conventional models.

REFERENCES:

1. M. R. Lyu, “Handbook of Software Reliability Engineering”, McGraw-Hill, 1996.
2. J. D. Musa, A. Iannino, and K. Okumoto, “Software Reliability, Measurement, Prediction and Application”, McGraw-Hill, (1987).
3. M. Xie, “Software Reliability Modeling”, World Scientific Publishing, (1991).
4. C. Y. Huang, M. R. Lyu, and S. Y. Kuo, “A Unified Scheme of Some Non homogenous Poisson Process Models for Software Reliability Estimation,” IEEE Trans. Software Eng., 29 (3) (20030 261-269.
5. A. L. Goel, and K. Okumoto, “Time-Dependent error-Detection Rate Model for Software Reliability and Other Performance Measures,” IEEE transactions on Reliability, R-28 (3) (1979) 206-211.
6. M. Ohba, et al., “S-shaped Software Reliability Growth Curve: How Good Is It?” , COMPSAC’82, (1982) 38-44.

7. S. Yamada, and S. Osaki, "Reliability Growth Models for Hardware and Software Systems Based on Non homogeneous Poisson Processes: a Survey," Microelectronics and Reliability, 23 (1983) 91-112.
8. S. Yamada, and S. Osaki, "S-shaped Software Reliability Growth Model with Four Types of Software Error Data", International Journal of Systems Science, 14 (1983) 683-692.
9. Z. Jelinski, and P. B. Moranda, "Software Reliability Research," Statistical Computer Performance Evaluation (ed. Freiberger, W.), Academic Press, New York, (1972) 465-484.
10. N. Karunanithi, Y. K. Malaiya, and D. Whitley, "Prediction of Software Reliability Using Neural Networks," Proc. 1991 IEEE International Symposium on Software Reliability Engineering, (1991) 124-130.
11. N. Karunanithi, D. Whitley, and Y. K. Malaiya, "Using Neural Networks in Reliability Prediction," IEEE Software, 9(4) (1992) 53-59.
12. N. Karunanithi, D. Whitley, and Y. K. Malaiya, "Prediction of Software Reliability Using Connectionist Models," IEEE Transaction On Software Engineering, 18 (7) (1992) 563-574.
13. T. M. Khoshgoftaar, R. M. Szabo, and P. J. Guasti, "Exploring the Behavior of Neural-network Software Quality Models," Software Engineering Journal, 10 (3) (1995) 89–96.
14. R. Sitte, "Comparison of Software Reliability Growth Predictions: Neural networks vs. parametric recalibration", IEEE Transactions on Reliability, 48 (3) (1999) 285-291.
15. T. M. Khoshgoftaar, E. B. Allen, J. P. Hudepohl, and S. J. Aud, "Software Metric-based Neural-network Classification Models of a Very Large Telecommunications System," in Applications and Science of Artificial Neural Networks II, S. K. Rogers and D. W. Ruck, Eds. Orlando, FL: SPIE-Int. Soc. Opt. Eng., Vol. 2760 of Proc. SPIE, (1996) 634–645.
16. J. P. Hudepohl, S. J. Aud, T. M. Khoshgoftaar, E. B. Allen, and J. Mayrand, "EMERALD: Software Metrics and Models on the Desktop", IEEE Software, 13(5) (1996) 56–60
17. T. Dohi, Y. Nishio, and S. Osaki, "Optimal Software Release Scheduling Based on Artificial Neural Networks", Annals of Software Engineering, 8 (1999) 167-185.
18. K. Y. Cai, L. Cai, W. D. Wang, Z. Y. Yu, and D. Zhang, "On the Neural Network Approach in Software Reliability Modeling", The Journal of Systems and Software, (2001) 47-62.
19. S. L. Ho, M. Xie, and T. N. Goh, "A Study of the Connectionist Models for Software Reliability Prediction, Computers and Mathematics with Applications, (46) (2003) 1037-1045.
20. Y. Takada, K. Matsumoto and K. Torii, "A Software-Reliability Prediction Model Using a Neural Network", System and Computers" 25 (14) (1994) 22-31
21. P. Guo and M. R. Lyu, "A pseudo inverse learning algorithm for feedforward neural networks with stacked generalization applications to software reliability growth data", Neuro computing, 56 (2004) 101-121
22. L. Tian and A. Noore, "Evolutionary neural network modeling for software cumulative failure time prediction", Reliability Engineering & System safety, 87(1) (2005) 45-51
23. N. Gupta and M. P. singh, "Estimation of Software Reliability with Execution time Model using the Pattern Mapping Technique of Artificial Neural Network", Computer and Operation Research 32 (2005) 187-199
24. C. Smidts, M. Stutzke and R.W. Stoddard, "Software reliability modeling: An approach to early reliability prediction", IEEE Transactions on Reliability 47 (3) (1998) 268-278.
25. M. Xie, G. Y. Hong and C. Wohlin, "Software reliability prediction incorporating information from a similar project", Journal of System and Software, 49 (1) (1999) 43-48.
26. Q. P. Hu, M. Xie and S. H. Ng, "Software reliability prediction improvement with prior information incorporated", Proceeding of the 12th ISSAT International Conference on Reliability and Quality in Design 06 Chicago USA, (2006) 303-307
27. L. Tian, and A. Noore, "On-line prediction of software reliability using an evolutionary connectionist model", The journal of System and Software 77 (2005) 173-180.
28. C. G. Bai, Q. P. Hu, M. Xie, S. H. Ng, "Software failure prediction based on markov Bayesian network model", The Journal of System and Software 74 (3) (2005) 275-282.
29. M. Reformat, "A fuzzy-based multi model system for reasoning about the number of software defects", International Journal of Intelligent Systems 20 (11) (2005) 1093-1115.
30. P. F. Pai, W. C. Hong, "Software Reliability forecasting by support vector machines with simulated vector machines with simulated annealing algorithms", 79 (2006) 747-755.
31. Y. S. Su and C. Y. Huang, "Neural-Network-based approaches for software reliability estimation using dynamic weighted combinational models", Journal of System and Software, 80 (4) (2006) 606-615.
32. S. Kanmani, V. R. Uthariaraj, V. Sankaranarayanan, and P. Thambidurai, "Object-oriented software failure fault prediction using neural networks", Information and Software Technology 49 (2007) 483-492.
33. H. Madsen, P. Thyregod, B. Burtschy, G. Albeanu, F. Popentiu, "On using soft computing techniques in software reliability engineering", International Journal of Reliability, Quality and Safety Engineering, 13 (1) (2006) 61-72.
34. R. Schalkoff, "Patten Recognition: Statistical, Structural and Neural Network." Wiley India, ISBN 978-81-265-1370-3, (2007).
35. A. R. Webb, "Statistical Pattern Recognition.", John Wiley and Sons Ltd., 2nd, ISBN 0-470-84514-7,(2002).
36. K. Hornik, M. Stinchcombe, and H. White, "Approximation capabilities of multiplayer feedforward networks", Neural Networks, 4 (1989) 251- 257.
37. S. Haykin, "Neural Networks", Second Edition, Pearson Education (Singapore), (2004).
38. T. Kohonen, "Correlation matrix memories.", IEEE Trans Computers, 21 (1972) 353–359.

39. J. B. MacQueen, “Some methods for classification and analysis of multivariate observations.”, Proc 5th Berkeley Symposium on Math Statistics and Probability, University of California Press, Berkeley, (19670 281–297.
40. D. H. Ackley, G. E. Hinton, and T. J. Sejnowski, “A learning algorithm for Boltzmann machines.”, Cognitive Science, 9 (1985) 147–169
41. S. Kirkpatrick, J. Gelatt, M. P. Vecchi, “Optimization by simulated annealing. Science”, 220 (1983) 671–680.
42. T. Kohonen, “A Simple Paradigm for the Self-Organized Formulation of Structured Maps,” Competition and Cooperation in Neural Nets, (Eds.) S. Amari, M. Arbib, Berlin, Springer-Verlag, 5 (1982).
43. J. J. Hopfield, “Neural Networks and Physical Systems with Emergent Collective Computational Abilities”. Proceedings of the National Academy Sciences (USA), 79 (1982), 2554 – 2558.
44. J. J. Hopfield, “Neurons with Graded Responses Have Collective Computational Properties Like Those of Two-State Neurons”, in Proceedings of The National Academy Sciences (USA), 81 (1984) 3088 – 3092
45. A. G. Barto, “Reinforcement learning and adaptive critic methods”, D. A. White, D. A. Sofge, (eds) Handbook of intelligent control: neural, fuzzy, and adaptive approaches, Van Nostrand Reinhold, New York, (1992) 469–471.
46. A. G. Barto, R. S. Sutton, and C. W. Anderson” Neuron like adaptive elements that can solve difficult learning control problems”, IEEE Trans System Man Cybern, 13 (1983) 834–846.
47. L. P. Kaelbling, M. H. Littman, and A. W. Moore, “Reinforcement learning: A survey”, Journal of Artificial Intelligence Research, 4 (1996) 237–285.
48. P. J. Werbos, “Backpropagation through time: What it does and how to do it”, Proc. IEEE, 78 (10) (1990) 1550–1560.
49. H. Atlan, and I. R. Cohen, “Theories of immune networks. Spriner-Verlag, Berlin (1989)
50. F. J. Pineda, “Generalization of back-propagation to recurrent neural networks.”, Physical Rev Letter, 59 (1987) 2229–2232.
51. M. L. Minsky and S. Papert, “Perceptrons.”, MIT Press, Cambridge, MA, (1969).
52. M.R. Lyu and A. Nikora, “Using Software Reliability Models More Effectively,” in IEEE Software, (1992) 43-53.
53. K. Kanoun and J. C. Laprie, “Software Reliability Trend Analyses from Theoretical to Practical Considerations”, IEEE Transactions on Software Engineering, 20 (9) (1994) 740-747.