



SOFTWARE RELIABILITY PREDICTION USING NEURAL NETWORK

Mamta Arora¹, Shailee Choudhary²

¹Assistant Professor(IT) ² Head Of Department(IT)
Manav Rachna College of Engineering, Faridabad
India (Delhi)

Abstract: Software reliability growth models have achieved considerable importance in estimating reliability of software products. This paper explores the use of feed-forward neural networks as a model for software reliability growth prediction. To empirically evaluate the predictive capability of this new approach data sets from different software projects are used. The neural networks approach exhibits a consistent behavior in prediction and the predictive performance is comparable to that of parametric model.

Keywords: Software Reliability, Software Reliability Growth Models, Neural Networks, Predictability Measures

I. Introduction

Software reliability is becoming more and more important in software industry various techniques are required to discover faults in the development of software. Software reliability can be defined as the probability of a software system to perform its specified functions correctly over a long period of time or for different input set under the usage environments similar to that of its target customer [1]. However, as reliability of software is measured in terms of failure it is impossible to measure reliability before the software is developed completely, software reliability is the most extensively studied quality among all the quality attributes [2].

In recent years researchers have proposed several different software reliability growth models [3,4,5]. Since all the existing software reliability models are based on some key assumptions about the software development environments, the nature of software failures and stochasticity of individual failure occurrence. These analytic models exhibit different predictive capabilities at different phases of testing within a project as well as across different projects. Thus, developing an universal analytic model for accurate predictions in all circumstances may not be practical.

One possible approach to improve the adaptability of software reliability growth prediction models is to develop models that are free of both priori assumptions about the software development environment and external parameters. Recent advances in neural networks show that they can be used in applications that involve prediction. This paper illustrates an adaptive modeling approach based on "Artificial Neural Networks" and demonstrates how this new approach can be applied to predict software reliability. Recently, artificial neural networks have been used both to estimate parameters of a formal model and to learn to emulate the process model itself in order to predict future outcomes. Werbose [6] uses back propagation networks as an alternative to regression techniques to identify sources of forecast uncertainty in a recent gas market model. Shadmehr et al. [7] use a multilayer feed-forward network for both estimating model parameters of pharmacokinetic system and predicting the noise present in the measured data samples. They indicate the neural network's performance was better than the maximum likelihood estimator.

Neural network models have significant advantage over analytic models, though, because they require only failure history as input, no assumptions. Using that input, neural network model automatically develops its own internal model of failure process and predicts future failure. Because it adjusts model complexity to match the complexity of the failure history, it can be more accurate than some commonly used analytic models. In our experiment we found this to be true. The data set used in this experiment uses data set from real project.

II. Neural Network

Neural networks are a computational metaphor inspired studies of the brain and nervous system in biological organisms. They are highly idealized mathematical models of how understand the essence of these simple nervous systems.

The basic characteristics of a neural network are

- It consists of many simple processing units, called neurons, that perform a local computation on their input to produce an output.
- Many weighted neuron interconnections encode the knowledge of the network.
- The network has a learning algorithm that lets it automatically develop internal representations

One of the most widely used processing unit models is based on the logistic function. The resulting transfer function is given by

$$\text{output} = \frac{1}{1 + e^{-\text{sum}}}$$

where sum is the aggregate of weighted inputs. Figure 1 shows the actual I/O response of this unit model, where Sum is computed as weighted sum of inputs. The unit is nonlinear and continuous.

There exists a variety of neural network models and learning procedures. Two well-known classes of neural networks that can be used for prediction applications are: feed-forward networks and recurrent networks. In this paper we use feed-forward networks and learning procedure is back propagation algorithm which comes under the category of supervised learning.

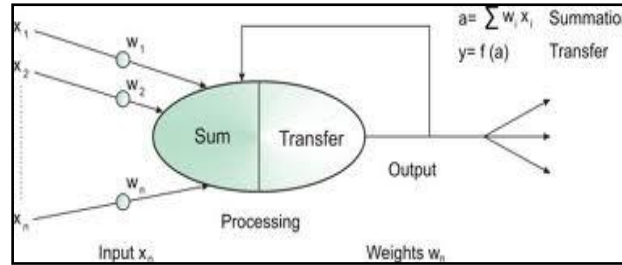


Figure 1: A typical transfer unit

III. TAILORING NEURAL NETWORKS FOR PREDICTION

Reliability prediction can be stated in the following way. Given a sequence of cumulative execution times $(i_1, \dots, i_k) \in I_k(t)$, and the corresponding observed accumulated faults $(o_1, \dots, o_k) \in O_k(t)$ up to present time t , and the cumulative execution time at the end of a future test session $k + h$, $i_{k+h}(t + \Delta)$, predict the corresponding cumulative faults $o_{k+h}(t + \Delta)$.

For the prediction horizon $h = 1$, the prediction is called next term prediction (also known as short term prediction), and for $h = n (\geq 2)$ consecutive test intervals, it is known as n -step-ahead prediction, or *long-term* prediction. A type of long term prediction is *endpoint* prediction, which involves predicting an output for some future fixed point in time. In end-point prediction, the prediction window becomes shorter as you approached the fixed point of interest.

Here

$$\Delta = \sum_{j=k+1}^{k+h} \Delta_j$$

represents the cumulative execution time of h consecutive future test sessions. You can use Δ to predict the number of accumulated faults after some specified amount of testing. From the predicted accumulated faults, you can infer both the current reliability and how much testing may be needed to meet the particular reliability criterion.

This reliability-prediction problem can be stated in terms of neural network mapping:

$$P: \{(I_k(t), O_k(t)), i_{k+h}(t + \Delta)\} \rightarrow o_{k+h}(t + \Delta)$$

where $(I_k(t), O_k(t))$ represents the failure history of the software system at time t used in training the network and $O_{k+h}(t + \Delta)$ is network's prediction.

Training the network is the process of adjusting the neuron's interconnection strength using part of the software's failure history. After a neural network is trained, it can be used for predicting the total number of faults to be detected at the end of a future test session $k + h$ by inputting $i_{k+h}(t + \Delta)$.

The three steps of developing a neural network for reliability prediction are specifying a suitable network architecture, choosing the training data, and training the network.

A. Specifying an architecture

Both prediction accuracy and resource allocation to simulation can be compromised if the architecture is not suitable. Many of the algorithm used to train the neural networks require you to decide the network architecture ahead of time or by trial and error.

An architecture can be defined by following either *static* or *dynamic* approach. In the *static* approach the network architecture is defined in advance and remains invariant throughout the training phase. Training algorithms, such as back propagation, typically assumes that network architecture is statically defined before training begins. In contrast, a *dynamic* approach automatically evolves a suitable network architecture during the training phase itself. In this paper we are using *static* approach.

A typical feed-forward neural network consist of three types of layers: an input layer of neurons that receive inputs from outside world, an output layer of neurons that send outputs to the external world, and one or more hidden layer of neurons that have no direct communication with the external world as shown in Figure 2. The function of the hidden layer neurons is to receive inputs from previous layer and convert them to an activation value that can be passed on as inputs to the neurons in the next layer. The input layer neurons do not involve in any computation; they merely copy the input values and distribute them as input to the neurons in the first hidden layer. One of the important architectural constraint imposed on feed forward networks model is that their link can propagate only in the forward direction.

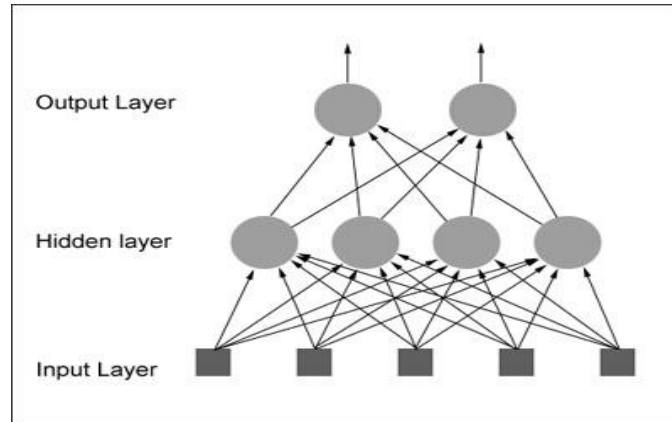


Figure 2: A Standard Feed Forward Network

B. Choosing Training Data

A neural network's predictive ability can be affected by what it learns and in what sequence. Figure 3 shows two reliability-prediction regimes :generalization training and prediction training.

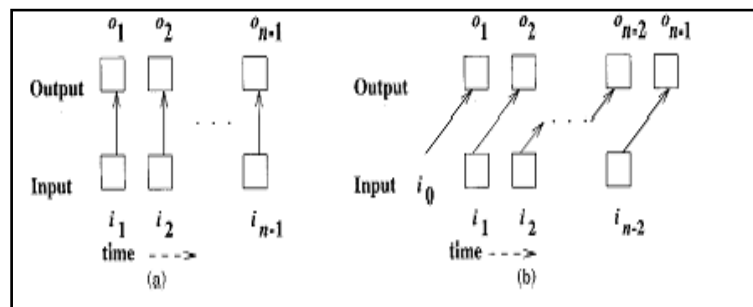


Figure 3: Two network training regimes : (a) generalization training (b) prediction training

Generalization training is the standard way of training feed-forward networks. During training, each input i_t at time t is associated with the corresponding output o_t . Thus the network learns to model the actual functionality between the independent (or input) variable and the dependent (or output) variable.

Prediction training, on the other hand, is the general approach for training recurrent networks. Under this training, the value of the input variable i_t at time t is associated with the actual value of the output variable at time $t+1$. Here, the network learns to predict outputs anticipated at the next time step.

Thus combination these two training regimes with the feed-forward network, gives two neural network prediction models: FFN generalization and FFN prediction.

C. Training the network

Most feed forward networks are trained using a supervised learning algorithm. Under supervised learning, the algorithm adjusts the network weights using a quantified error feedback. There are several supervised learning algorithms, but one of the most widely used is back propagation-an iterative procedure that adjusts network weights by propagating the error back into the network.

Typically, training a neural network involves several iterations (also known as epochs). At the beginning of training, the algorithm initializes network weights with a set of small random values (between +1.0 and -1.0). During each epoch, the algorithm presents the network with the sequence of training pairs. We used cumulative execution time as input and the corresponding cumulative faults as the desired output to form a training pair. The algorithm then calculates a sum squared error between the desired output and the network's actual output. It uses the gradient of the sum squared error (with respect to weights) to adapt the network weights so that the error measure is small in future epochs. Training terminates when the sum squared error is below a specified tolerance limit.

IV. PREDICTION EXPERIMENT

We used the testing and debugging data from an actual project described by Yoshiro Tohma and colleagues to illustrate the prediction accuracy of the neural network. In this data (Tohma's table4), execution time was reported in terms of days and faults in terms of cumulative faults at the end of each day. Total testing and debugging time was 46 days, and there were 266 faults.

If you use logistic-function units to construct a network, the network's output will be bounded between 0.0 and 1.0. so before you attempt to use a neural network you may have to represent the problem's I/O variables in a range suitable for the neural network. In the simplest representation, you can use a direct scaling, which scales execution time and cumulative faults from 0.0 to 1.0.

We did not use the simple representation. We scaled both cumulative execution time and cumulative faults from 0.1 to 0.9 because

- The network is less accurate in discriminate input close to the boundary values (inputs whose scaled values are close to 1.0 or 0.0)
- The unit's error derivative, which affects the rate of weight adaption during training, becomes inconsequential when the unit's output is close to 1.0 or 0.0.

To scale the data, however, you must have either the software complete failure history or you must guess the appropriate maximum values for both the cumulative execution time and the cumulative faults.

Neural network cannot predict future faults without learning the software's failure history (or at least some part of it). Any prediction without training is equivalent to making a random guess. We restricted the minimum size of training ensemble to 3 data points and incremented the training set size from 3 to 45 in steps of two. This type of grouping is common and helps to remove noise.

A. Method

Most training methods initialize neural network weights with random values at the beginning of training, which causes the network to converge to different weight set at the end of each training session. You can thus get different prediction results at the end of each training session. To compensate for these prediction variations, you can take an average over large number of trails. In our experiment, we trained the network with 40 random seeds for each training-set size and averaged their predictions.

B. Results

After training the neural network with the failure history up to time t (where t is less than total testing and debugging time of 46 days), you can use the network to predict the cumulative faults at the end of a future testing and debugging session.

To evaluate neural networks, you can use the following extreme prediction horizons: the next-step prediction (at $t=t+1$) and the endpoint prediction (at $t=46$).

Since you already know the actual cumulative faults for those two future testing and debugging session, you can compute the network's prediction error at t . Then the relative prediction error is given as:

$$\text{prediction error} = \frac{\text{actual faults} - \text{predicted faults}}{\text{actual}} \text{ faults}$$

Figure 4 shows the relative prediction error curve of the feed forward neural network model. In this figure the percentage prediction error is plotted against the execution time.

Figure 5 shows the actual values and predicted errors shown by feed forward network. the network is trained using 80% of the values and validated with the 20% of data.

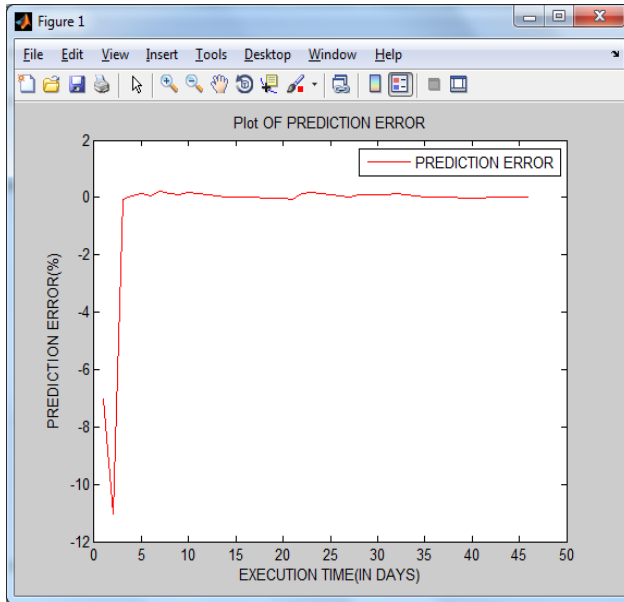


Figure 4: prediction Error

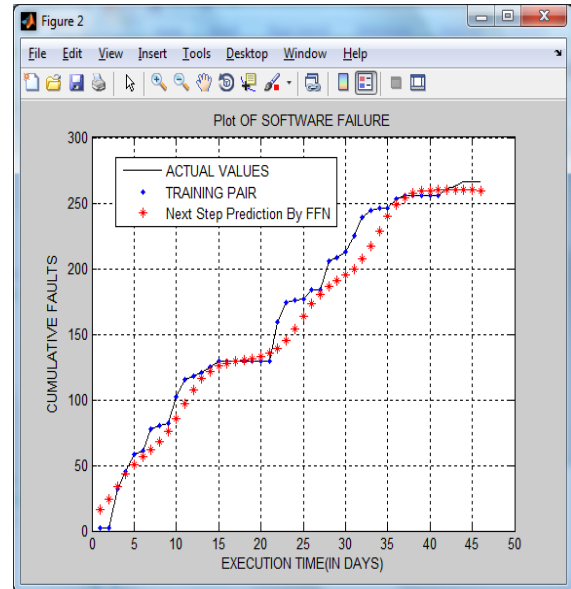


Figure 5: Results of Actual Failure& predicted failure by Feed Forward network

V. Conclusion

In this paper we have demonstrated how you can use neural-network models and training regimes for reliability prediction. Though the results presented here are for only one data set, the results are consistent with 5 other data sets we tested. The main advantage using neural network approach is it is a black-box approach; the user need not to know much about underlying failure process of the project. We recognize that our experiments are only beginning to tap the potential of feed forward neural model in reliability, but we believe that this class of models will eventually offer significant benefits. We also recognize that our approach is very new and still needs research to demonstrate its practicality on a broad range of software projects.

REFERENCES

- [1] Tian, J, "Software Quality Engineering", John Wiley and Sons Inc. 2005. A. N. Netravali and B. G. Haskell, *Digital Pictures*, 2nd ed., Plenum Press: New York, 1995, pp. 613-651.
- [2] Liang T., Afzel, N. "On-line prediction of software reliability using an evolutionary" connectionist model", *Journal of Systems and Software*, vol. 77, pp. 173-180, 2005R. Nicole, "Title of paper with only first word capitalized," accepted for publication in *IEEE Trans. Broadcast Technology*.
- [3] A. L. Goel, "Software Reliability Models: Assumptions, Limitations, and Applicability," *IEEE Trans. Software Engg.*, Vol SE-11, No 12, 1985 Dec, pp 1411-1423.
- [4] Y. K. Malaiya and P. K. Srimani, Ed., "Software Reliability Models: Theoretical Develop- ments, Evaluation and Applications," IEEE Computer Society Press, 1990.
- [5] J. D. Musa, A. Ianino and K. Okumoto, *Software Reliability - Measurement, Prediction, Applications*, RlcGraw-Hill, 1987.
- [6] P. Werbos, "Generalization of Backpropaga- tion with Application to Recurrent Gas Market Model,)' *Neural Networks*, 1988, Vol. 1, pp 339-356.
- [7] R. Shadmehr and D. Z. D'Argenio, "A Comparison of a Neural Network Based Estimator and Two Statistical Estimators in a Sparse and Noisy Data Environment," *Proc. IJCNN, Washington D.C.*, 1990 June, Vol.1, pp 289-292.