APPLYING NEURAL NETWORKS FOR SOFTWARE RELIABILITY PREDICTION

Nachimuthu Karunanithi, Darrell Whitley and Yashwant K. Malaiya

Computer Science Department, Colorado State University

Fort Collins, CO 80523.

Tel: (303) 491-6228 Fax: (303) 491-6639

E-mail: *karunani/whitley/malaiya@CS.ColoState.Edu*

Software reliability prediction research focuses mainly on developing analytic models that require *a prior* assumptions about the software development environments, the nature of software failures and the stochasticity of individual software 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 *a priori* assumptions about the software development environment and external parameters. 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. Given only the past failure history of a software system, the artificial neural network models automatically develops its own internal model of the failure process and predicts future failures with a better accuracy than some of the well-known analytic models.

Artificial Neural Networks are a computational metaphor which has been inspired by studies of brain and nervous system in biological organisms. Recent advances in this field have shown that this new approach can be used in applications that involve predictions. An interesting and difficult application is *time series* prediction. Prediction of software reliability growth can be considered analogous to predicting a complex sequential process.

In this paper we illustrate how artificial neural networks can be used both to learn to model the underlying failure process of a software system and to predict future failures. The illustration uses a data set from a real software project.

1 Neural Network Models

In this section we provide a background on artificial neural networks. Also we discuss the issue of network architecture and the problem of training data presentation.

. . . . . .

Insert Figure 1 about here.

. . . . . .

1.1 Neural Networks: A Background

Informally, neural networks can be defined as highly idealized mathematical models of our present understanding of simple biological nervous systems. Basic characteristic entities of a neural network are:

1. a large number of simple processing units called *neurons* that perform a local computation on their input to produce an output;

2. a large number of weighted interconnections among neurons that encode the knowledge of the network;

3. a learning algorithm that can lead to automatic development of internal representations.

One of the most widely used processing unit model is based on the *logistic* function. The resulting sigmoidal transfer function is given by:

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

where *sum* is the aggregate of weighted inputs. The actual input/output response of a sigmoidal unit is shown in Figure 1. Figure 1a shows how the *sum* is computed as a weighted sum of inputs. Note that the sigmoidal unit is non-linear and continuous.

There exists a variety of neural network models and learning procedures. (Readers not familiar with this field may refer to Lippmann[4] or an introductory book on neural networks for more details). 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 restrict our descriptions to feed-forward networks and a variant of recurrent networks known as *Jordan Networks*. We selected these two neural network models because in our previous research

we found them to be more accurate in software reliability predictions than other network models.[2] [3]

. . . . . .

Insert Figure 2 about here.

. . . . . .

A typical feed-forward neural network consists of three types of layers: an *input* layer of neurons that receive inputs (suitably encoded beforehand) from the outside world, an *output* layer of neurons that send outputs to the external world, and one or more *hidden* layers of neurons that have no direct communication with the external world. The function of hidden layer neurons is to receive inputs from the 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 perform any computation; they merely copy the input values and associate them with weights feeding the neurons in the (first) hidden layer. One of the important architectural constraints imposed on the feed-forward networks is that their links can propagate activations only in the forward direction. On the other hand, the Jordan networks have both forward connections as well as feedback connections. A typical 3-layer feed-forward network and a Jordan network are shown in Figure 2. Note that the feedback connection in Figure 2b is from the output layer to the hidden layer through a "dummy input" unit. The dummy input unit receives as input at time t the actual output of the output unit at time $t-1$. That is, the output of the additional input unit is the same as the output of the network corresponding to the previous input pattern. In Figure 2b the dashed line represents a fixed connection of strength 1.0 and solid lines represent the trainable connections. Thus, the Jordan network in Figure 2b is equivalent to a special case of the feed-forward network in Figure 2a with an additional input. However, if a network has multiple output units then the number of dummy units required will be equal to the number of output units.

1.2 Training of Neural Networks

Before we can use a neural network to predict future failures the network must be trained using part of the failure history of the software system. In our case, each cumulative execution time and the corresponding cumulative faults represent a training pair. Training of a neural network involves adjusting its connection strengths. Most feed-forward networks and recurrent networks are trained using a class of training algorithm known as a *supervised learning algorithm*. Under supervised learning, the network weights are adjusted using a quantified error feedback. Among supervised learning algorithms the *back-propagation algorithm*[8] is one of the most widely used algorithms. The Back-propagation training algorithm is an iterative procedure in which the network weights are adjusted 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 network weights are initialized with a set of small random values (between +1.0 and -1.0). During each epoch the network is presented with a sequence of training pairs; next a sum squared error between the training outputs and the network's outputs is calculated. Then the gradient of the sum squared error (with respect to weights) is used to adapt the network weights in such a fashion that the error measure gets reduced in future epochs. The training terminates when the sum squared error is reduced below a specified tolerance limit.

. . . . . .

Insert Figure 3 about here.

. . . . . .

1.3 How to Specify a Suitable Architecture?

With the standard back-propagation learning algorithm the architecture of the network must be fixed *a priori*. Both the accuracy of predictions and computational resources needed for simulation can be affected if the architecture is not a suitable one. Thus, for any given application the problem of specifying a suitable network architecture must be addressed first. A possible solution for this problem is discussed next.

Cascade-Correlation Algorithm: Recently Fahlman *et al.*[1] developed an efficient constructive algorithm known as the "Cascade-Correlation learning architecture" for dynamically constructing feed-forward neural networks. This algorithm combines the idea of incremental architecture and learning into a single training algorithm. In brief, this algorithm starts with a minimal network (consisting of an input and an output layer) and dynamically trains and adds new hidden units one by one, until it builds a suitable multi-layer architecture. We use this algorithm for constructing both feed-forward networks and Jordan networks. A typical feed-forward network developed by the Cascade-Correlation algorithm is shown in Figure 3. Architecturally, the cascade network is different from the standard feed-forward networks in that the former has additional feed-forward connections between the input/output layers as well as among hidden units. In the experiments reported here all neural networks use one output unit. On the input layer the feed-forward nets use one input unit whereas the Jordan nets use two units.

. . . . . .

Insert Figure 4 about here.

. . . . . .

1.4 How to Present Training Data?

The predictive capability of a neural network can be affected by what the network learns and in what sequence it learns. Here we illustrate two training regimes that can used in software reliability prediction. They are called *Generalization Training* and *Prediction Training*. These training regimes differ from each other depending on the way in which training data are presented. Figure 4 shows these training regimes.

Generalization Training: This is the standard way in which most feed-forward networks are trained. 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: This is the general approach employed in 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 if we combine these two training regimes with the feed-forward network and the Jordan network we can obtain four different neural network prediction models. We denote these models as *FFN-Generalization*, *FFN-Prediction*, *JordanNet-Generalization* and *JordanNet-Prediction*.

2 Neural Network as a Predictor

The problem of software reliability prediction can be stated as follows: given a sequence of cumulative execution time $((i_1, ..., i_k) \in I_k(t))$ and the corresponding observed accumulated faults $((o_1, ..., o_k) \in O_k(t))$ up to the present time t, and $(i_{k+h}(t+\Delta))$ representing the cumulative execution time at the end of a future test session $k+h$, predict the corresponding cumulative number of faults $o_{k+h}(t+\Delta)$. For the *prediction horizon* $h = 1$, the prediction is called the *next-step prediction* (or *short-term prediction*) and for $h = n(\geq 2)$ consecutive test intervals, it is known as the *n-step ahead prediction* (or *long-term prediction*). Here $\Delta = \sum_{j=k+1}^{k+h} \Delta_j$ represents the cumulative execution time of h consecutive future test sessions.

We can formulate our software reliability prediction problem in terms of a neural network mapping:

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

where, $(I_k(t), O_k(t))$ represents the failure history of the software system at time t used in training the network and $o_{k+h}(t+\Delta)$ the network's prediction. After a neural network is successfully trained, the network can be used to predict the total number of faults to be detected at the end of a future test session $k+h$ by feeding $i_{k+h}(t+\Delta)$ as its input.

3 Prediction Experiment

Data Set Used: In order to illustrate the predictive accuracy of neural networks we have chosen a real test/debug data set reported in Table 4 in Tohma *et al.*[9] In this data the execution time was reported in terms of days and the faults in terms of cumulative faults at the end of each day. The total test/debug time was 46 days and there were 266 faults at the end.

Data Representation: If we use sigmoidal units to construct a network then the actual output of the network will be bounded between 0.0 and 1.0. So before we attempt to use a neural network it may be necessary to represent the input/output variables of the problem in a range that is suitable for the neural network. In the simplest representation we can use a direct scaling so that the execution time and cumulative faults are scaled over a 0.0 to 1.0 range. Instead, we scale both the execution time and the cumulative faults over a 0.1 to 0.9 range for the following reasons: i) the network's ability to discriminate inputs that are close to the boundary values (i.e, inputs whose scaled values are close to 1.0 or 0.0) tend to be less accurate and ii) the error derivative of the sigmoidal unit, which affects the rate of weight adaptation during training, becomes inconsequential when the output of the sigmoidal unit is close to 1.0 or 0.0. However it should be noted that our direct scaling requires either the complete failure history of the system or guessing an appropriate maximum values for both the cumulative execution time and the cumulative faults.

Training Ensemble Size: The neural networks cannot predict future faults without learning the failure history (or at least some part of the history) of the software system. Any prediction without training is equivalent to making a random guess. In our experiment we restricted the minimum size of the training ensemble to three data points. We incremented the training set size from three to 45 in steps of two.

Monte-Carlo Experiment: In most training methods, the neural network weights are initialized with random values at the beginning of training. This results in the network converging to different sets of weights at the end of each training session. Thus, it is possible to get different prediction results at the end of each training session. We can compensate

such variations in predictions by taking an average over a large number of Monte-Carlo trials. In our experiment we trained the network with 50 different random seeds for each training set size and averaged their predictions.

Prediction Results: After training the neural network with a failure history up to time t (where t is less than the total test/debug time of 46 days), we can use the network to predict the cumulative faults at the end of a future test/debug session. To evaluate the neural networks we can use the following extreme prediction horizons: the *next-step* prediction (i.e, at $t = t + 1$) and the *end-point* prediction (i.e, at $t = 46$). Since we already have the actual cumulative faults for those two future test/debug sessions we can compute the network's prediction error at time t. Then the relative prediction error is given by *(predicted faults - actual faults)/actual faults*. Figures 5 and 7 show the relative prediction error curves of the neural network models. In these figures the percentage prediction error is plotted against the percentage normalized execution time $t/46$.

. . . . . .

Insert Figures 5 and 6 about here.

. . . . . .

Figures 5 and 6 show the relative error curves for *end-point* predictions of neural networks and five well-known analytic models. Results from the analytic models are included because they can provide a better basis for evaluating neural networks. (Refer to Malaiya *et al.*[6] for details about the analytic models and model fitting procedures.) Note that both figures have the same scale. These graphs suggest that the neural networks are more accurate than the analytic models.

Table 1. Average and Maximum Errors in End-Point Predictions

Model Used	Average Error			Maximum Error		
	1st Half	2nd Half	Overall	1st Half	2nd Half	Overall
Neural Network Models						
FFN-Generalization	7.34	1.19	3.36	10.48	2.85	10.48
FFN-Prediction	6.25	1.10	2.92	8.69	3.18	8.69
JordanNet-Generalization	4.26	3.03	3.47	11.00	3.97	11.00
JordanNet-Prediction	5.43	2.08	3.26	7.76	3.48	7.76
Analytic Models						
Logarithmic	21.59	6.16	11.61	35.75	13.48	35.75
Inverse Polynomial	11.97	5.65	7.88	20.36	11.65	20.36
Exponential	23.81	6.88	12.85	40.85	15.25	40.85
Power	38.30	6.39	17.66	76.52	15.64	76.52
Delayed S-shape	43.01	7.11	19.78	54.52	22.38	54.52

A summary of Figures 5 and 6 in terms of average and maximum error measures are given in Table 1. The columns under "Average Error" represent the following: *1st half*, the average prediction error of a model in the first half of the test/debug session; *2nd half*, the average prediction error of a model in the second half of the test/debug session; and *overall*, the average prediction error for the entire test/debug period. These average error measures also suggest that neural networks are more accurate than the analytic models. The first half results are interesting because the neural network models have average prediction errors that are less than 8% of the total defects disclosed at the end of the test/debug process. This an important result from software managers point of view because such reliable predictions at early stages of testing can be very helpful for making long-term planning.

Among the neural network models the difference in accuracy is not significant, whereas the analytic models exhibit considerable variations. Among the analytic models the Inverse Polynomial model and the Logarithmic model seem to perform reasonably well. The other three columns in Table 1 represent the maximum prediction errors. The maximum prediction errors give an indication of how unrealistic a model can be. These values also suggest

that the neural network models have less worst case predictions than the analytic models at various phases of the test/debug process.

. . . . . .

Insert Figure 7 about here.

. . . . . .

Figure 7 represents the *next-step* predictions of both the neural networks and the analytic models. These graphs suggest that the neural network models have only slightly less next-step prediction accuracy than that of the analytic models.

Table 2. Average and Maximum Errors in Next-Step Predictions

Model	Average Error			Maximum Error		
Used	1st Half	2nd Half	Overall	1st Half	2nd Half	Overall
Neural Network Models						
FFN-Generalization	8.61	2.40	4.59	17.51	4.95	17.51
FFN-Prediction	8.02	3.05	4.80	17.74	6.64	17.74
JordanNet-Generalization	6.92	3.73	4.86	12.11	8.24	12.11
JordanNet-Prediction	6.59	3.35	4.50	11.11	7.30	11.11
Analytic Models						
Logarithmic	4.94	2.31	3.24	5.95	7.56	7.56
Inverse Polynomial	4.76	2.24	3.13	6.34	7.83	7.83
Exponential	5.70	2.33	3.52	10.17	7.42	10.17
Power	4.59	2.44	3.20	8.59	7.12	8.59
Delayed S-shape	6.17	2.12	3.55	13.24	7.98	13.24

Table 2 shows the summary of Figure 7 in terms of average and maximum errors. Since average errors of the neural network models in the first half are above the analytic models by only 2 to 4 percentage and the difference in the second half are less than 2% it suggests that these two competing approachs are not significantly different. But the worst case prediction

errors may suggest that the analytic models have a slight edge over the neural network models. However, the difference in overall average errors is less than 2%, which suggests that both the neural network models and the analytic models have a similar next-step prediction accuracy.

4 How different are Neural Network models?

So far we have seen how neural networks can be used for software reliability predictions and how superior their performances are. Now let us look into the neural networks to understand what the underlying computations are and how they can be related to equivalent analytic models. In order to compare competing models we can use the number of parameters as a measure of complexity. Thus, a model is considered more complex if it has more parameters than its competitor.

Since we have used the Cascade-Correlation algorithm for evolving network architecture, the number of hidden units used to learn the problem varied depending on the size of the training set. On an average, the neural networks used 1 hidden unit when the normalized execution time is below 60% to 75% and zero hidden unit afterwards. However, occasionally 2 or 3 hidden units were used before the training was complete.

Models developed by the FFN-Generalization method can be expressed as shown below. With no hidden unit, the actual computation performed by the network is equivalent to a simple sigmoidal expression.

$$o_i = \frac{1}{1 + e^{-(w_0 + w_1 \cdot t_i)}}$$

where w_0 and w_1 are weights from the bias unit and the input unit respectively. This is equivalent to a two parameter sigmoidal model, whose $\mu(t_i)$ is given by,

$$\mu(t_i) = \frac{1}{1 + e^{(\beta_0 + \beta_1 \cdot t_i)}}$$

where t_i is the cumulative execution time at the end of ith test session, and β_0 and β_1 are parameters. We can easily see that $\beta_0 = -w_0$ and $\beta_1 = -w_1$. Thus training of neural

networks (finding weights) is equivalent to estimating these parameters.

Now consider the case where the network used one hidden unit. The model developed by the network is equivalent to a three parameter model of the form,

$$\mu(t_i) = \frac{1}{1 + e^{(\beta_0 + \beta_1 \cdot t_i + \beta_2 \cdot h_i)}}$$

where β_0, β_1 and β_2 are parameters of the model determined by weights feeding the output unit. In this model $\beta_0 = -w_0$, $\beta_1 = -w_1$ and $\beta_2 = -w_h$ (the weight from the hidden unit). However, the activation of the hidden unit h_i is an intermediate value computed using another two parameter sigmoidal expression,

$$h_i = \frac{1}{1 + e^{-(w_3 + w_4 \cdot t_i)}}$$

Thus there are five parameters in the model corresponding to the 5 weights in the network.

Similarly, we can express the models developed by the FFN-Prediction model as follows. For the network with no hidden unit the equivalent two parameter model is,

$$\mu(t_i) = \frac{1}{1 + e^{(\beta_0 + \beta_1 \cdot t_{i-1})}}$$

where the t_{i-1} is the cumulative execution time at $(i-1)_{th}$ instant. And for the network with 1 hidden unit the equivalent five parameter model is,

$$\mu(t_i) = \frac{1}{1 + e^{(\beta_0 + \beta_1 \cdot t_{i-1} + \beta_2 \cdot h_i)}}$$

Though we have not shown a similar comparison between Jordan network models and equivalent analytic models, it is quite straight forward to extend the above expressions. However, the models developed by the Jordan network can be more complex because of the feedback connection and the weights from the additional input unit.

The above expressions imply that the neural network approach develops models that can be relatively complex. These expressions also suggest that the neural networks used models of varying complexity at different phases of testing. In contrast, the analytic models have only 2 or 3 parameters and their complexity remain static. Thus the main advantage in neural networks approach is that the complexity of the models are automatically adjusted to match the complexity of the failure history.

5 Final Remarks

We have demonstrated how different neural network models and training regimes can be used for software reliability prediction. Results obtained with a real test/debug data suggests that the neural network models are better at end-point predictions than the analytic models. Though the results presented here are for only a single data set, the results are consistent with 13 other data sets that we have tested.[3]

The major advantages in using the neural network approach are: i) it is a "black box" approach and the user need not know much about the underlying failure process of the project, i) easy adaptation of models of varying complexity at different phases of testing within a project as well as across different software projects, and ii) simultaneous construction of a model and estimation of its parameters if we use a training algorithm such as the Cascade-Correlation. On the other hand, one main drawback with the neural network models is that we may not be able to ascribe physical interpretations to their weights. In contrast, it is seldom possible to find an analytic model whose parameters lack physical interpretation. Thus, we want to remind the readers that this is a new approach and further research is needed to gain more insights.

Acknowledgements

The authors would like to thank the anonymous reviewers for their useful comments and suggestions. This research was supported in part by the NSF grant IRI-9010546, and in part by a project funded by the SDIO/IST and monitored by ONR.

References

[1] S. E. Fahlman and C. Lebiere, "The Cascaded-Correlation Learning Architecture", Tech. Rep. CMU-CS-90-100, School of Computer Science, Carnegie Mellone University, Pittsburg, PA, Feb. 1990.

[2] N. Karunanithi, Y. K. Malaiya, and D. Whitley, "Prediction of Software Reliability Using Neural Networks", *Proc. of the International Symposium on Software Reliability Eng.*, May 1991, pp. 124-130.

[3] N. Karunanithi, D. Whitley and Y. K. Malaiya, "Prediction of Software Reliability Using Connectionist Approachs", To appear in a special issue of the *IEEE Trans. Software Eng.*, July 1992.

[4] R. P. Lippmann, "An Introduction to Computing with Neural Nets", *IEEE ASSP Magazine*, No 4, Apr. 1987, pp. 4-22.

[5] Y. K. Malaiya and P. K. Srimani, Eds., **Software Reliability Models: Theoretical Developments, Evaluation and Applications**, IEEE Computer Society Press, 1990.

[6] Y. K. Malaiya, N. Karunanithi, and P. Verma, "Predictability Measures for Software Reliability Models", To appear in the *IEEE Trans. Reliability Eng.*.

[7] J. D. Musa, A. Iannino, and K. Okumoto, **Software Reliability - Measurement, Prediction, Applications**, McGraw-Hill, 1987.

[8] D. Rumelhart, G. Hinton, and R. Williams, "Learning Internal Representations by Error Propagation", **Parallel Distributed Processing, Vol. I**, MIT Press, 1986, pp. 318-362.

[9] Y. Tohma, H. Yamano, M. Ohba, and R. Jacoby, "Parameter Estimation of the Hyper-Geometric Distribution Model for Real Test/Debug Data", *Tech Rep. 901002, Department of Computer Science, Tokyo Institute of Technology*, 1990.

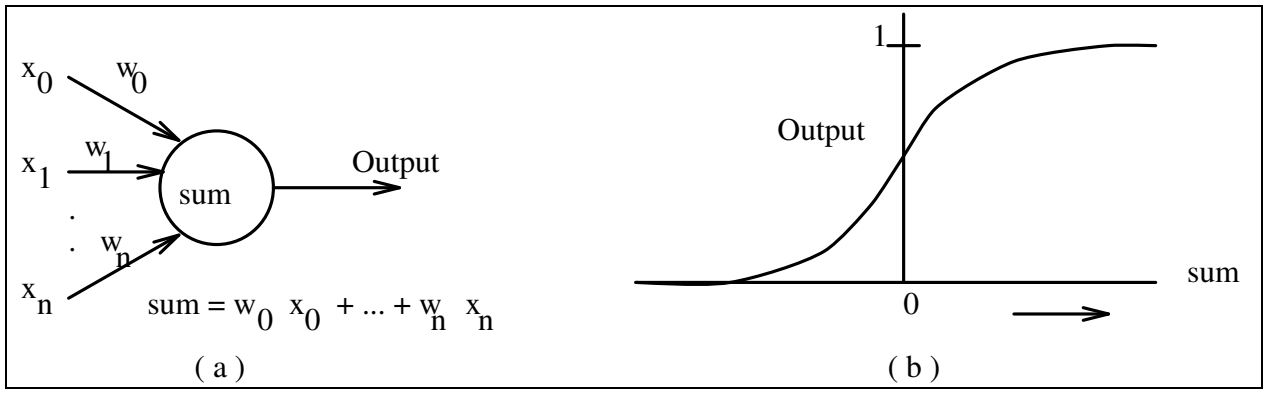

Figure 1: A typical sigmoidal unit (a), and its input/output response (b).

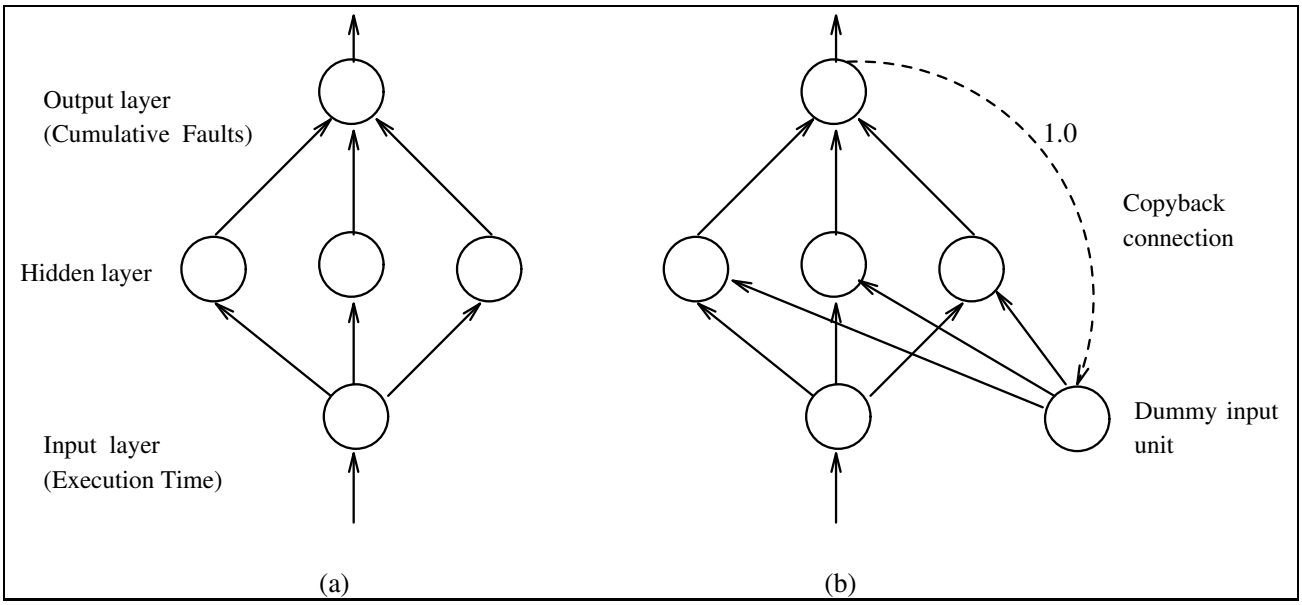

Figure 2: Typical neural networks: (a) A feed-forward network and (b) A Jordan network.

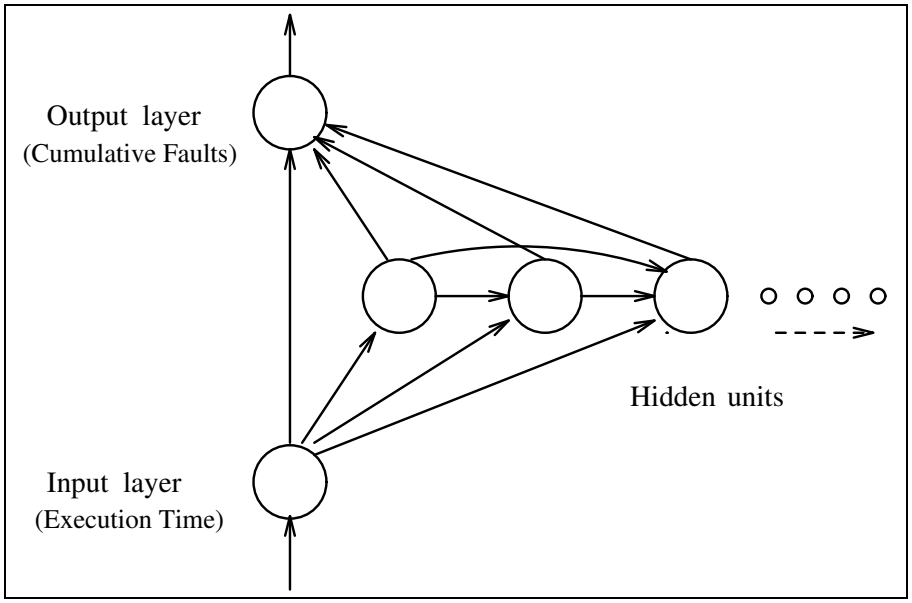

Figure 3: A typical feed-forward network developed by the Cascade-Correlation algorithm.

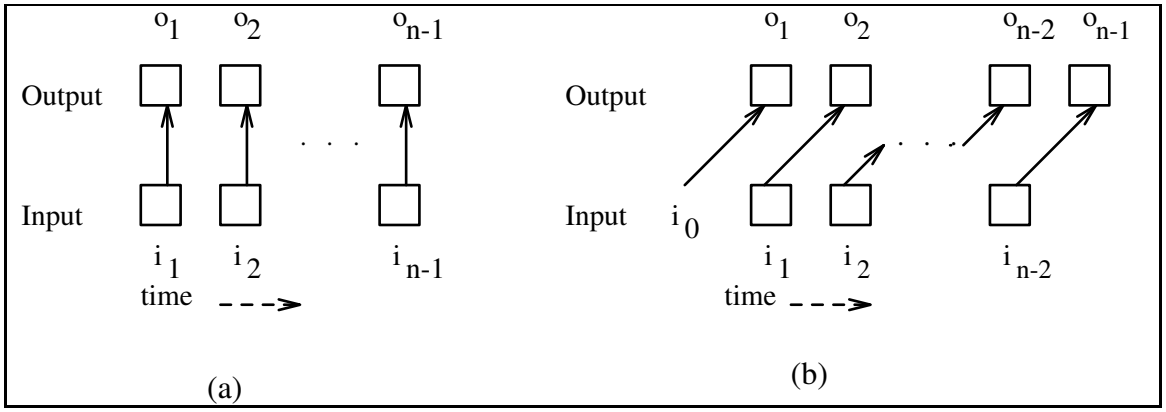

Figure 4: Different training regimes: (a) Generalization Training and (b) Prediction Training.

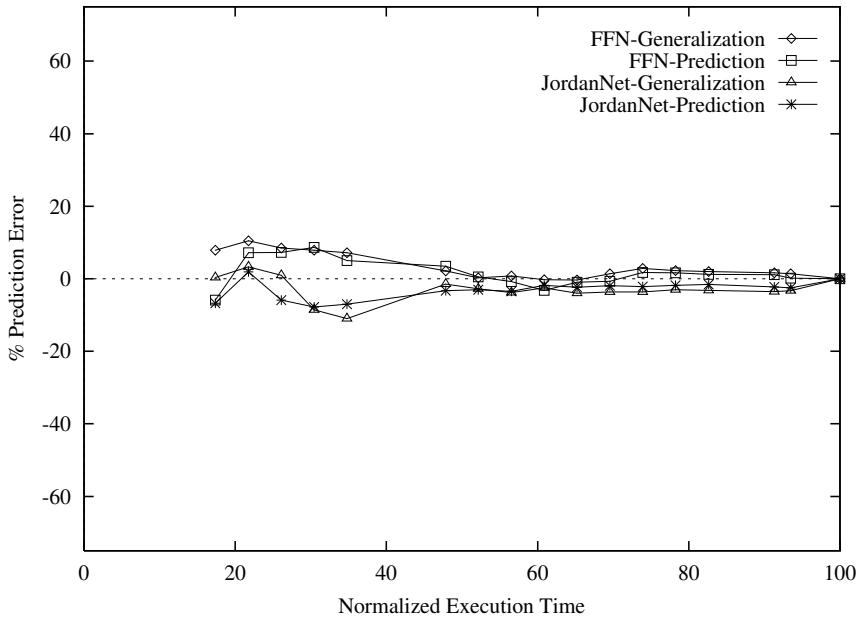

Figure 5: End-point predictions of neural network models.

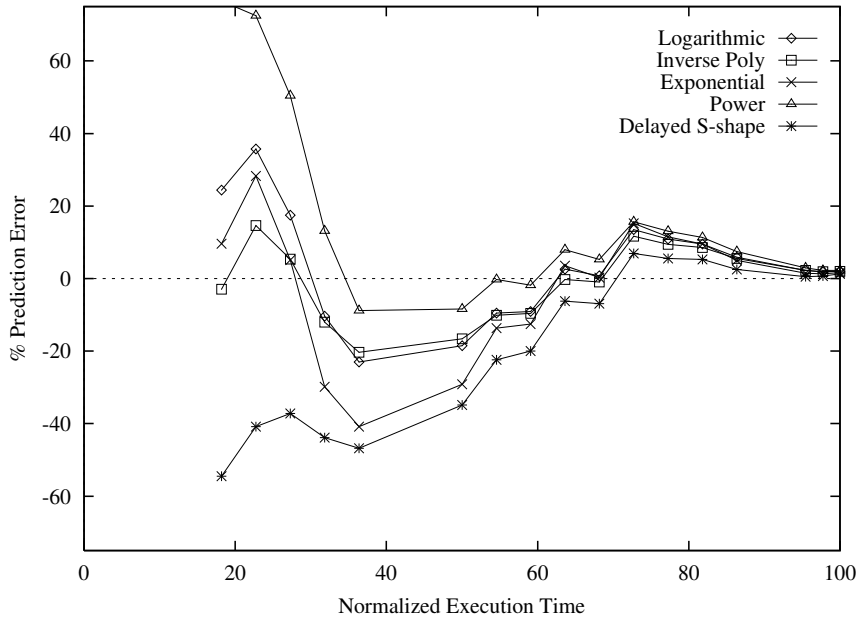

Figure 6: End-point predictions of analytic models.

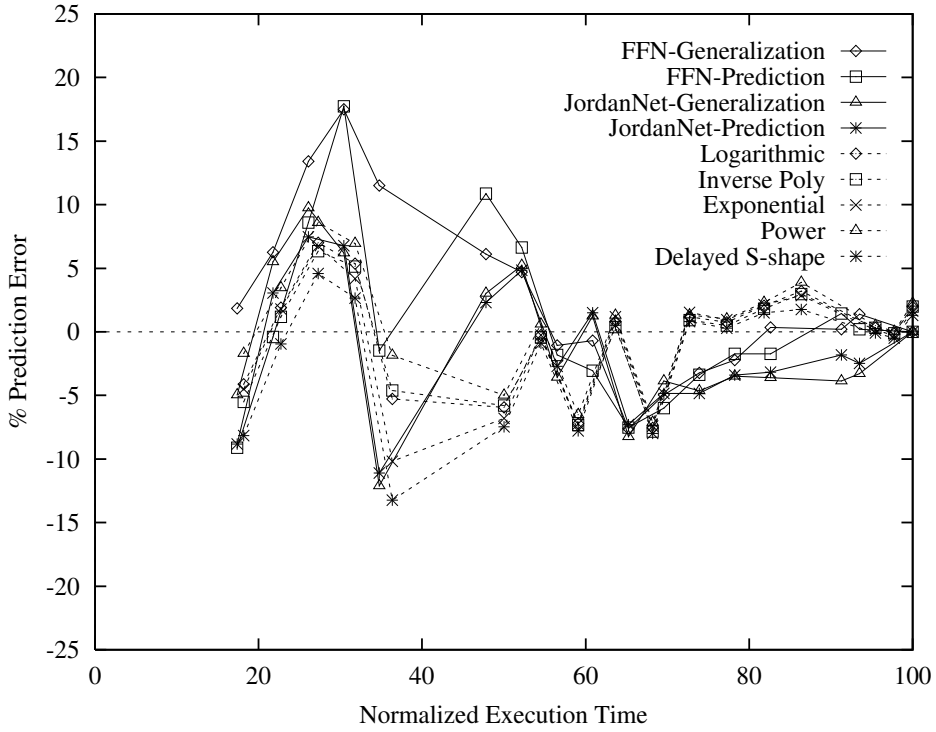

Figure 7: Next-step predictions of neural nets Vs analytic models

Biographies

Nachimuthu Karunanithi received B.E.(Honors) in Electrical Eng. from P.S.G Tech., Madras University, and M.E. from Anna University, Madras, in 1982 and 1984 respectively. Till July 1987 he was with C-DOT, New Delhi, India, where he participated in the design and development of software for a real-time Digital Switching System. He is currently a Ph.D. candidate in the Department of Computer Science at Colorado State University. His research interests are neural networks, genetic algorithms and software reliability modeling. He is a member of the IEEE TCSE subcommittee on software reliability engineering.

Darrell Whitley is an Assistant Professor in the Department of Computer Science at Colorado State University. He has published more than 30 papers in the areas of neural networks and genetic algorithms. He serves on the Governing Board of the International Society for Genetic Algorithms and is Program Chair for the 1992 Workshop on Combinations of Genetic Algorithms and Neural Networks as well as Program Chair for the 1992 Foundations of Genetic Algorithms Workshop. He received a Ph.D. in Anthropology in 1985 and an MS in Computer Science in 1987 from Southern Illinois University at Carbondale.

Yashwant K. Malaiya is a Professor in the Department of Computer Science and also in the Department of Electrical Engineering at Colorado State University. He has published widely in the areas of software and hardware reliability, fault modeling, testing and testable design. He has also been a consultant for industry. He was the general chair of the 24th Int. Symp. on Microarchitecture and the program chair of 5th Int. Conf. on VLSI Design. He has co-edited the IEEE CS Tech. Series book "Software Reliability Models, Theoretical Developments, Evaluation and Applications". He is the chair of TC on Microprogramming and Microarchitecture and a vice-chair of the TCSE subcommittee on software reliability engineering. He received B.Sc. from Govt. Degree College, Damoh, M.Sc. from University of Saugor and M.Sc. Tech. from BITS, Pilani in India. In 1978 he received Ph.D. in Electrical Engineering from Utah State University. He was with State University of New York at Binghamton during 1978-82.

Address questions about this article to the authors at Computer Science Dept., Colorado State University, Fort Collins, CO 80523; CSnet: karunani@cs.colostate.edu.