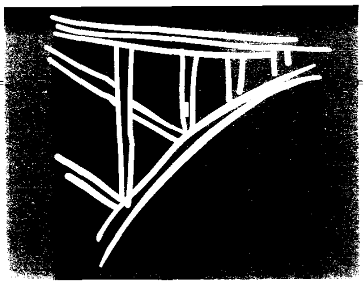


Using Neural Networks in Reliability Prediction

NACHIMUTHU KARUNANITHI, DARRELL WHITLEY, and YASHWANT K. MALAIYA, Colorado State University

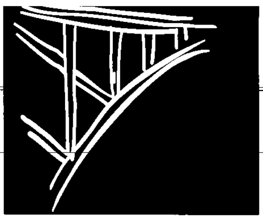
◆ *The neural-network model requires only failure history as input and predicts future failures more accurately than some analytic models. But the approach is very new.*

In current software-reliability research, the concern is how to develop general prediction models. Existing models typically rely on assumptions about development environments, the nature of software failures, and the probability of individual failures occurring. Because all these assumptions must be made before the project begins, and because many projects are unique, the best you can hope for is statistical techniques that predict failure on the basis of failure data from similar projects. These models are called reliability-growth models because they predict when reliability has grown enough to warrant product release.

Because reliability-growth models exhibit different predictive capabilities at different testing phases both within a project and across projects, researchers are finding it nearly impossible to develop a universal model that will provide accurate predictions under all circumstances.

A possible solution is to develop models that don't require making assumptions about either the development environment or external parameters. Recent advances in neural networks show that they can be used in applications that involve predictions. An interesting and difficult application is time-series prediction, which predicts a complex sequential process like reliability growth. One drawback of neural networks is that you can't interpret the knowledge stored in their weights in simple terms that are directly related to software metrics — which is something you can do with some analytic models.

Neural-network models have a significant advantage over analytic models, though, because they require only failure history as input, no assumptions. Using that input, the neural-network model automatically develops its own internal model of the failure process and predicts



future failures. 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 experiments, we found this to be true.

TAILORING NEURAL NETWORKS FOR PREDICTION

Reliability prediction can be stated in the following way. Given a sequence of cumulative execution times $(i_1,\ldots,i_k) \in I_k(t)$, and the corresponding observed accumulated faults $(o_1,\ldots,o_k) \in O_k(t)$ up to the 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 the next-step prediction (also known as 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. A type of long-term prediction is *endpoint* prediction, which involves predicting an output for some future fixed point in time. In endpoint prediction, the prediction window becomes shorter as you approach 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 a 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 the network's prediction.

Training the network is the process of adjusting the neuron's (neurons are defined in the box below) interconnection strength using part of the software's failure history. After a neural network is trained, you can use it to predict 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.

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

To provide a more suitable means of selecting the appropriate network architecture for a project, Scott Fahlman and colleagues[1] developed the cascade-corre-

WHAT ARE NEURAL NETWORKS?

Neural networks are a computational metaphor inspired by studies of the brain and nervous system in biological organisms. They are highly idealized mathematical models of how we 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 A shows the actual I/O response of this unit model, where Sum is computed as a weighted sum of inputs. The unit is nonlinear and continuous.

Richard Lippman describes many neural network models and learning procedures.[1] Two well-known classes suitable for prediction applications are feed-forward networks and recurrent networks. In the main text of the article, we are concerned with feed-forward networks and a variant class of recurrent networks, called Jordan networks. We selected these two model classes because we found them to be more accurate in reliability predictions than other network models.[2,3]

REFERENCES

1. R. Lippmann, "An Introduction to Computing with Neural Nets," *IEEE Acoustics, Speech, and Signal Processing*, Apr. 1987, pp. 4-22.
2. N. Karunanithi, Y. Malaiya, and D. Whitley, "Prediction of Software Reliability Using Neural Networks," *Proc. Int'l Symp. Software Reliability Eng.*, May 1991, pp. 124-130.
3. N. Karunanithi, D. Whitley, and Y. Malaiya, "Prediction of Software Reliability Using Connectionist Approaches," *IEEE Trans. Software Eng.* (to appear).

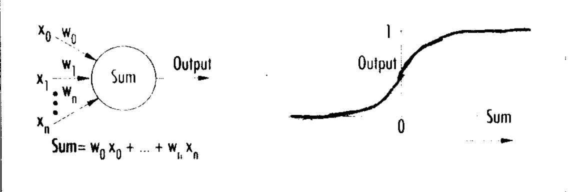


***Figure A.** A typical transfer-function unit and its I/O response.*

lation learning algorithm. The algorithm, which dynamically constructs feed-forward neural networks, combines the idea of incremental architecture and learning in one training algorithm. It starts with a minimal network (consisting of an input and an output layer) and dynamically trains and adds hidden units one by one, until it builds a suitable multilayer architecture.

As the box on the facing page describes, we chose feed-forward and Jordan networks as the two classes of models most suitable for our prediction experiments. Figure 1a shows a typical three-layer feed-forward network; Figure 1b shows a Jordan network.

A typical feed-forward neural network comprises a layer of neurons that receive inputs (suitably encoded) from the outside world, a layer that sends outputs to the external world, and one or more layers that have no direct communication with the external world.

The hidden layer of neurons receives inputs from the previous layer and converts them to an activation value that can be passed on as input 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.

The cascade-correlation algorithm combines the idea of incremental architecture and learning in one training algorithm.

Feed-forward networks can propagate activations only in the forward direction; Jordan networks, on the other hand, have both forward and feedback connections.

The feedback connection in the Jordan network in Figure 1b is from the output layer to the hidden layer through a recurrent input unit. At time t, the recurrent unit receives as input the output unit's output at time $t - 1$. That is, the output of the additional input unit is the same as the output of the network that corresponds to the previous input pattern.

In Figure 1b, the dashed line represents a fixed connection with a weight of 1.0. This weight copies the output to the additional recurrent input unit and is not adjusted during training. The solid lines in the figure represent the trainable connections. Thus, the Jordan network in Figure 1b is a special case of the feed-forward network in Figure 1a with an additional input. However, if a network has multiple output units, it will require an equal number of recurrent input and output units.

We used the cascade-correlation algorithm to construct both feed-forward and Jordan networks. Figure 2 shows a typical feed-forward network developed by the cascade-correlation algorithm. The cascade network differs from the feed-forward network in Figure 1a because it has feed-forward connections between I/O layers, not just among hidden units.

In our experiments, all neural networks use one output unit. On the input layer the feed-forward nets use one input unit; the Jordan networks use two units, the normal input unit and the recurrent input unit.

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.

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 if you combine these two training regimes with the feed-forward network and the Jordan network, you get four

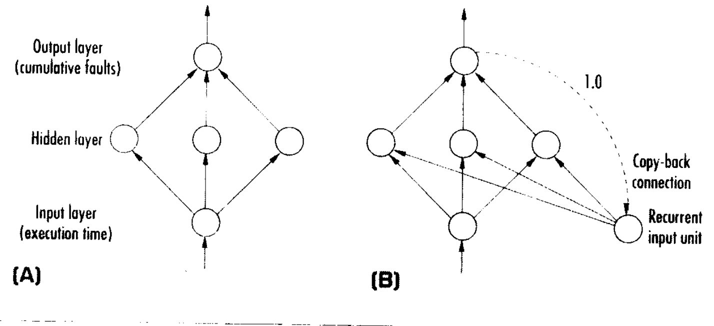


Figure 1. (A) A standard feed-forward network and (B) a Jordan network.

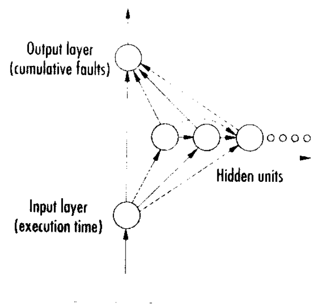

Output layer (cumulative faults)
Hidden units
Input layer (execution time)

Figure 2. A feed-forward network developed by the cascade-correlation algorithm.

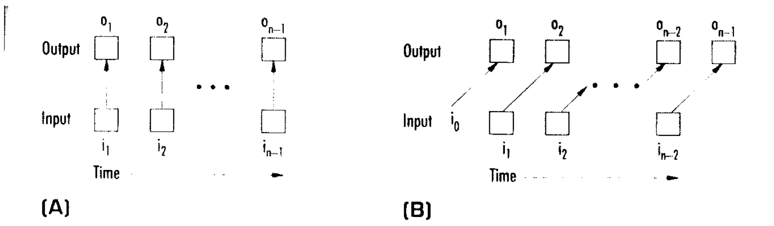

Output
Input
Time
(A)
(B)
o_1 o_2 o_{n-1} o_{n-2}
i_0 i_1 i_2 i_{n-1} i_{n-2}

Figure 3. Two network-training regimes: (A) generalization training and (B) prediction training.

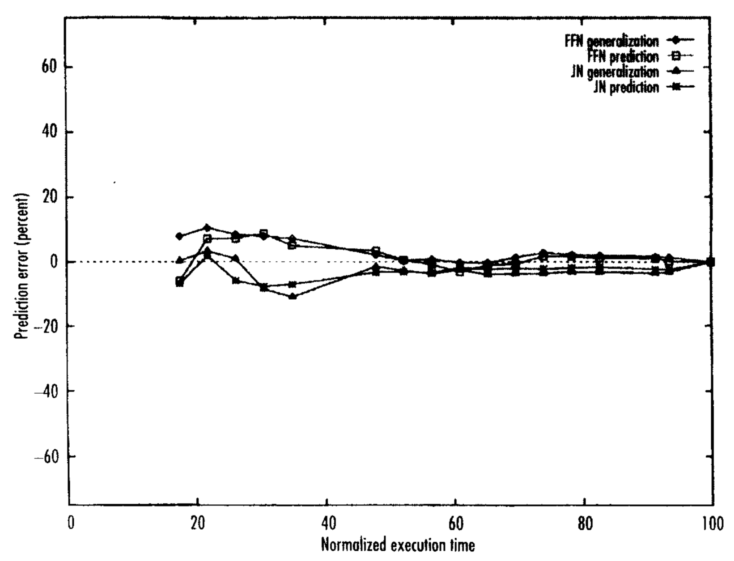


Figure 4. Endpoint predictions of neural-network models.

neural network prediction models: FFN generalization, FFN prediction, JN generalization, and JN prediction.

Training the network. Most feed-forward networks and Jordan 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.[2]

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 a 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 outputs and the network's actual outputs. It uses the gradient of the sum squared error (with respect to weights) to adapt the network weights so that the error measure is smaller in future epochs.

Training terminates when the sum squared error is below a specified tolerance limit.

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 neural networks.[4] In this data (Tohma's Table 4), 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 this simple representation. We scaled both cumulative execution time and cumulative faults from 0.1 to 0.9 because

♦ The network is less accurate in discriminating inputs 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 adaptation 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's complete failure history or you must guess the appropriate maximum values for both the cumulative execution time and the cumulative faults.

Neural networks 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 the training ensemble to three data points and incremented the training-set size from three to 45 in steps of two. This type of grouping is common and helps remove noise.

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 sets 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 a large number of trials. In our experiment, we trained the network with 50 random seeds for each training-set size and averaged their predictions.

Results. After training the neural network with a failure history up to time t (where t is less than the 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 sessions, you can compute the network's prediction error at t. Then the relative prediction error is given by (predicted faults – actual faults)/actual faults.[4]

Figures 4 and 6 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 $t/46$.

Figures 4 and 5 show the relative error curves for endpoint 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. Yashwant Malaiya and colleagues give details about the analytic models and fitting procedures.[5,6] The graphs suggest that neural networks are more accurate than analytic models.

Table 1 gives a summary of Figures 4 and 5 in terms of average and maximum error measures. The columns under Average error represent the following:

- ♦ *First half* is the model's average prediction error in the first half of the testing and debugging session.
- ♦ *Second half* is the model's average prediction error in the second half of the testing and debugging session.
- ♦ *Overall* is the model's average prediction error for the entire testing and debugging session.

These average error measures also suggest that neural networks are more accurate than analytic models. First-half results are interesting because the neural-network models' average prediction errors are less than eight percent of the total defects disclosed at the end of the testing and debugging session.

This result is significant because such reliable predictions at early stages of testing can be valuable in 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 maximum prediction errors in the table show how unrealistic a model can be.

These values also suggest that the neural-network models have fewer worst-case predictions than the analytic models at various phases of testing and debugging.

Figure 6 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 the analytic models.

TABLE 1
AVERAGE AND MAXIMUM ERRORS IN ENDPOINT PREDICTIONS

Model	Average error			Maximum error		
	1st half	2nd half	Overall	1st half	2nd half	Overall
Neural-net 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
JN generalization	4.26	3.03	3.47	11.00	3.97	11.00
JN 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

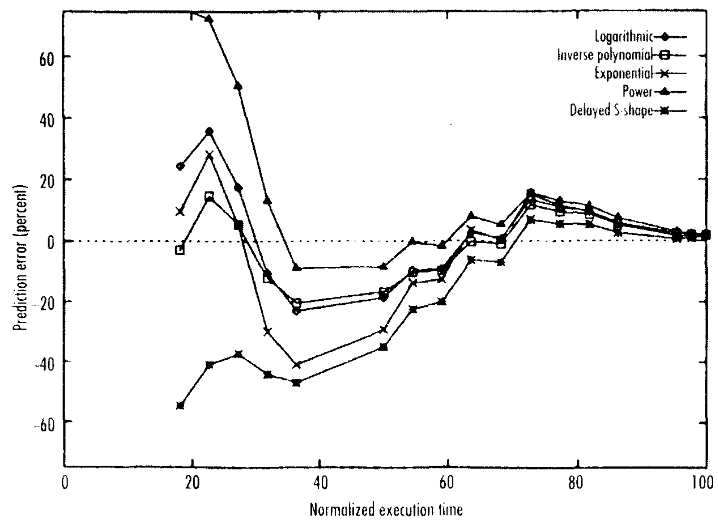


Figure 5. Endpoint predictions of analytic models.

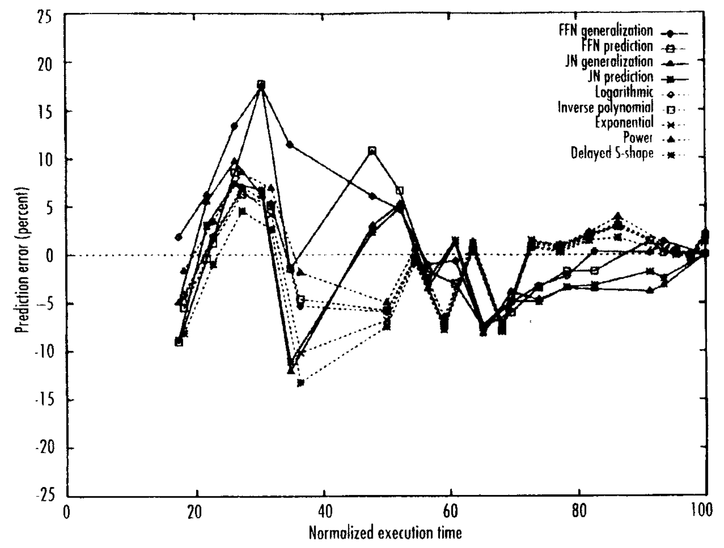


Figure 6. *Next-step predictions of neural-network models and analytic models.*

TABLE 2
AVERAGE AND MAXIMUM ERRORS IN NEXT-STEP PREDICTIONS

Model	Average error 1st half	Average error 2nd half	Average error Overall	Maximum error 1st half	Maximum error 2nd half	Maximum error Overall
Neural-net 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
JN generalization	6.92	7.73	4.86	12.11	8.24	12.11
JN 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 6 in terms of average and maximum errors. Since the neural-network models' average errors are above the analytic models in the first half by only two to four percent and the difference in the second half is less than two percent, these two approaches don't appear to be that different. But 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 two percent, which suggests that both the neural-network models and the analytic models have a similar next-step prediction accuracy.

NEURAL NETWORKS VS. ANALYTIC MODELS

In comparing the five analytic models and the neural networks in our experiment, we used the number of parameters as a measure of complexity; the more parameters, the more complex the model.

Since we 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 average, the neural networks used one hidden unit when the normalized execution time was below 60 to 75 percent and zero hidden units afterward. However, occasionally two or three hidden units were used before training was complete.

Though we have not shown a similar comparison between Jordan network models and equivalent analytic models, extending the feed-forward network comparison is straightforward. However, the models developed by the Jordan network can be more complex because of the additional feedback connection and the weights from the additional input unit.

FFN generalization. In this method, with no hidden unit, the network's actual computation is the same as a simple logistic 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, and t_i is the cumulative execution time at the end of ith test session.

This expression is equivalent to a two-parameter logistic-function model, whose $\mu(t_i)$ is given by

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

where β_0 and β_1 are parameters.

It is easy to see that $\beta_0 = -w_0$ and $\beta_1 = -w_1$. Thus, training neural networks (finding weights) is the same as estimating these parameters.

If the network uses one hidden unit, the model it develops is the same as a three-parameter model:

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

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

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

Thus, the model has five parameters that correspond to the five weights in the network.

FFN prediction. In this model, 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 the i–1th instant.

For the network with one hidden unit, the equivalent five-parameter model is

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

Implications. These expressions imply that the neural-network approach develops models that can be relatively complex. These expressions also suggest that neural networks use models of varying complexity at different phases of testing. In contrast, the analytic models have only two or three parameters and their complexity remain static. Thus, the main advantage of neural-network models is that model complexity is automatically adjusted to the complexity of the failure history.

We have demonstrated how you can use neural-network models and training regimes for reliability prediction. Results with actual testing and debugging data suggest that neural-network models are better at endpoint predictions than analytic models. Though the results presented here are for only one data set, the results are consistent with 13 other data sets we tested.[3]

The major advantages in using the neural-network approach are

♦ It is a black-box approach; the user need not know much about the underlying failure process of the project.

♦ It is easy to adapt models of varying complexity at different phases of testing within a project as well as across projects.

♦ You can simultaneously construct a model and estimate its parameters if you use a training algorithm like cascade correlation.

We recognize that our experiments are only beginning to tap the potential of neural-network models 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. ♦

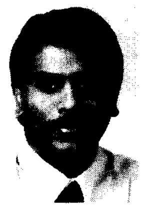
ACKNOWLEDGMENTS

We thank *IEEE Software* reviewers for their useful comments and suggestions. We also thank Scott Fahlman for providing the code for his cascade-correlation algorithm.

This research was supported in part by NSF grant IRI-9010546, and in part by a project funded by the SDIO/IST and monitored by the Office of Naval Research.

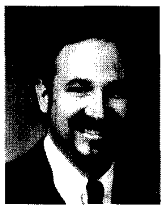
REFERENCES

1. S. Fahlman and C. Lebiere, "The Cascaded-Correlation Learning Architecture," Tech. Report CMU-CS-90-100, CS Dept., Carnegie-Mellon Univ., Pittsburgh, Feb. 1990.
2. D. Rumelhart, G. Hinton, and R. Williams, "Learning Internal Representations by Error Propagation," in *Parallel Distributed Processing, Volume I*, MIT Press, Cambridge, Mass., 1986, pp. 318-362.
3. Y. Tohma et al., "Parameter Estimation of the Hyper-Geometric Distribution Model for Real Test/Debug Data," Tech. Report 901002, CS Dept., Tokyo Inst. of Technology, 1990.
4. J. Musa, A. Iannino, and K. Okumoto, *Software Reliability — Measurement, Prediction, Applications*, McGraw-Hill, New York, 1987.
5. Y. Malaiya, N. Karunanithi, and P. Verma, "Predictability Measures for Software Reliability Models," *IEEE Trans. Reliability Eng.* (to appear).
6. *Software Reliability Models: Theoretical Developments, Evaluation and Applications*, Y. Malaiya and P. Srimani, eds., IEEE CS Press, Los Alamitos, Calif., 1990.



Nachimuthu Karunanithi is a PhD candidate in computer science at Colorado State University. His research interests are neural networks, genetic algorithms, and software-reliability modeling.

Karunanithi received a BE in electrical engineering from PSG Tech., Madras University, in 1982 and an ME in computer science from Anna University, Madras, in 1984. He is a member of the subcommittee on software reliability engineering of the IEEE Computer Society's Technical Committee on Software Engineering.



Darrell Whitley is an associate professor of computer science at Colorado State University. He has published more than 30 papers on neural networks and genetic algorithms.

Whitley received an MS in computer science and a PhD in anthropology, both from Southern Illinois University. He serves on the Governing Board of the International Society for Genetic Algorithms and is program chair of both the 1992 Workshop on Combinations of Genetic Algorithms and Neural Networks and the 1992 Foundations of Genetic Algorithms Workshop.

Yashwant K. Malaiya is a guest editor of this special issue. His photograph and biography appear on p. 12.

Address questions about this article to Karunanithi at CS Dept., Colorado State University, Fort Collins, CO 80523; Internet karunani@cs.colostate.edu.