ELSEVIER

The Journal of Systems and Software 58 (2001) 47–62

The Journal of Systems and Software

www.elsevier.com/locate/jss

On the neural network approach in software reliability modeling ☆

Kai-Yuan Cai [a,*], Lin Cai [a], Wei-Dong Wang [a], Zhou-Yi Yu [a], David Zhang [b]

[a] Department of Automatic Control, Beijing University of Aeronautics and Astronautics, Beijing 100083, China

[b] Department of Computing, The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong

Received 1 March 2000; received in revised form 1 June 2000; accepted 1 August 2000

Abstract

Previous studies have shown that the neural network approach can be applied to identify defect-prone modules and predict the cumulative number of observed software failures. In this study we examine the effectiveness of the neural network approach in handling dynamic software reliability data overall and present several new findings. Specifically, we find

1. The neural network approach is more appropriate for handling datasets with 'smooth' trends than for handling datasets with large fluctuations.
2. The training results are much better than the prediction results in general.
3. The empirical probability density distribution of predicting data resembles that of training data. A neural network can qualitatively predict what it has learned.
4. Due to the essential problems associated with the neural network approach and software reliability data, more often than not, the neural network approach fails to generate satisfactory quantitative results.

Keywords: Software reliability modeling; Neural network; Network architecture; Scaling function; Filtering; Empirical probability density distribution; Software operational profile

1. Introduction

Software reliability modeling is a subject of growing importance. It aims to quantify software reliability status and behavior and helps to develop reliable software and check software reliability. Quite a few methods have been proposed in software reliability modeling, including regression modeling methods, capture–recapture modeling methods, times-between-failures modeling methods, NHPP modeling methods, operational-profile modeling methods, etc. (Cai, 1998; Lyu, 1996; Musa et al., 1987; Xie, 1991). Each of them has achieved success to some extent in particular cases and conventional statistical theory plays an essential role. On the other hand, however, it has been realized that numerous factors may affect software reliability behavior, including software development methodology, software development environment, software complexity, software development organization and personnel, and so on (Cai, 1998; Neufelder, 1993). Normally, these factors are highly interactive and demonstrate non-linear patterns. This imposes severe limitations on existing statistical modeling methods that depend heavily upon the assumptions of independence and linearity. Consequently, neural network methods, which may handle numerous factors and approximate any non-linear continuous function in theory, have drawn people's attention in recent years (Karuanithi, 1993; Karuanithi and Malaiya, 1992; Karuanithi et al., 1992a,b; Khoshgoftaar and Lanning, 1995; Khoshgoftaar et al., 1992, 1993, 1994, 1995, 1997; Khoshgoftaar and Szabo, 1996; Sherer, 1995) and a modest amount of efforts have been devoted to them. In these efforts it was argued that neural network methods could be applied to estimate the number of software defects, classify program modules, and forecast the number of observed software failures, and they often offered better results than existing statistical methods. On the other hand, however, in applying neural network methods to estimate the number of software defects, we found that essential limitations were associated with the neural network methods as a result of the implicit assumptions the neural network methods took (Cai, 1998).

☆ Supported by the national key project of China and the national outstanding youth foundation of China.

* Corresponding author. Tel./Fax: +86-10-8231-7328.

E-mail address: kyc@ns.dept3.buaa.edu.cn (K.-Y. Cai).

0164-1212/01/$ - see front matter
PII: S 0 1 6 4 - 1 2 1 2 (0 1) 0 0 0 2 7 - 9

In this study we apply the neural network methods to handle dynamic data of software reliability including time between successive software failures as well as number of observed software failures, and see if our previous findings can be confirmed and furthered. Sections 2 and 3 present, respectively, the dynamic data of software reliability and the neural network architecture we use in the rest of this study. Sections 4–8 focus on the applications of the neural network methods to handling time between successive software failures. Section 4 presents the basic results. Sections 5 and 6 consider the effects of network architectures and scaling functions on the prediction behavior of the neural network methods, respectively. Section 7 considers the effect of a filtering scheme of transforming time between successive software failures into successive software failure times, while Sections 8 shows the qualitative behavior of the neural network methods in handling time between successive software failures overall. In Section 9 we transform time between successive software failure into number of software failures observed in successive time intervals and discuss if the neural network methods can work in handling the latter. In Section 10 we present our general discussions on the advantages and disadvantages of the neural network methods in software reliability modeling. Concluding remarks are included in Section 11.

2. Software reliability data

Software failures may occur in the execution process of software. In software testing phase, time is normally taken to detect and remove the software failure causing defect(s) and thus software reliability follows a growth trend. In software operation phase, the software failure causing defect(s) may or may not be detected or removed but time is needed to resume normal operation process of software. In software reliability modeling, time spending on detecting and removing software defects or on resuming normal operation process of software is usually ignored or assumed zero. In this way we can use Fig. 1 to depict a software failure process, where T_i denotes the time instant of the ith software failure and $X_i = T_i - T_{i-1}$. In statistical modeling methods $\{X_i\}$ is usually assumed to be a series of independent random variables. However in the neural network methods this assumption is not necessary.

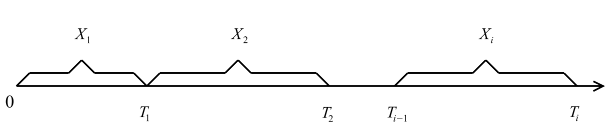

Fig. 1. Software failure process.

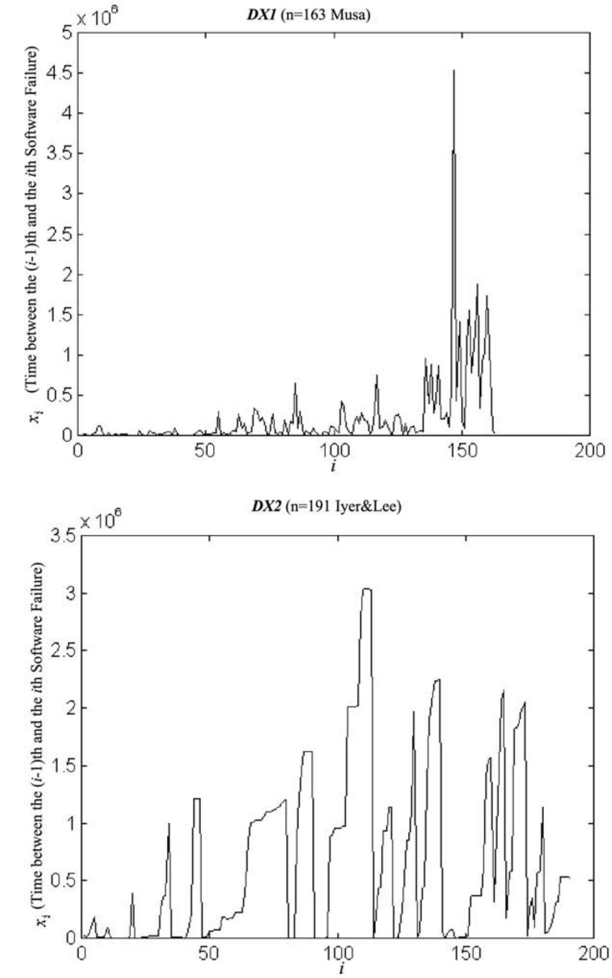

Fig. 2. Software reliability data in terms of time between successive software failures $\{x_i\}$.

Suppose $\{x_i\}$ is a realization of $\{X_i\}$. In this paper we employ two sets of software reliability data in $\{X_i\}$: $DX1$ and $DX2$ as shown in Fig. 2. $DX1$ is due to Musa collecting in a software testing phase and contains 163 data points (i.e., $x_1, x_2, \ldots, x_{163}$) (Musa, 1979), whereas $DX2$ is due to Iyer and Lee (1996) collecting in a software operation phase and contains 191 data points (i.e., $x_1, x_2, \ldots, x_{191}$). Appendix A tabulates these data.

3. Neural networks

An (artificial) neural network is a machine that is designed to model or mimic the way in which the brain performs a particular task or function of interest. It consists of neurons that are connected in a certain manner. A neuron serves as an information processing unit as depicted in Fig. 3. It has p inputs $(r_1, r_2, \ldots, r_p)$ and one output y_k, and consists of three basic elements: a set of synapses or connecting links in terms of weights $w_{k1}, w_{k2}, \ldots, w_{kp}$, an adder in terms of a summing junc-

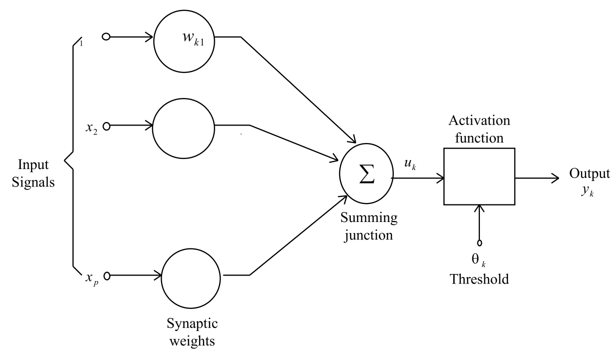

Fig. 3. Non-linear model of a neuron.

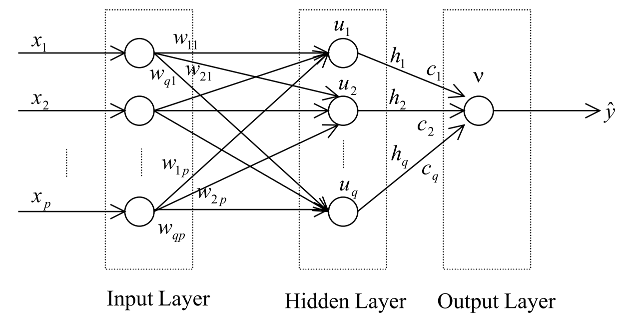

Fig. 4. Multilayer perception with one hidden layer and one output.

tion, and an activation function $\varphi(\cdot)$ for limiting the output y_k to, normally, a closed unit interval [0,1]. The subscript k denotes the neuron is the kth one in the neural network. Mathematically, we have

$$u_k = \sum_{j=1}^{p} w_{kj}x_j, \quad y_k = \varphi(u_k - \theta_k), \quad \theta_k \geqslant 0.$$

A commonly used neural network is the so-called multilayer feedforward network or multilayer perception. In this study we mostly use a multilayer perception with only one hidden layer and one output, as shown in Fig. 4, where each circle denotes a neuron. However the input layer (receiving inputs $x_1, x_2, \dots, x_p$) is a special one. Each neuron of the input layer only receives an input signal and passes it as output without any computation in the neuron. In this way

$$u_i = \sum_{j=1}^{p} w_{ij}x_j, \quad h_i = \varphi(u_i), \quad i = 1, 2, \dots, q,$$
$$v = \sum_{l=1}^{q} c_l h_l, \quad \hat{y} = \varphi(v).$$

Now given a vector input value $(x_1^{(k)}, x_2^{(k)}, \dots, x_p^{(k)})$ of $(x_1, x_2, \dots, x_p)$, if the corresponding output value $y^{(k)}$ of y is known, then $(x_1^{(k)}, x_2^{(k)}, \dots, x_p^{(k)}; y^{(k)})$ makes up a sample of the neural network. Suppose m samples of the neural networks are available, we need a learning algorithm to train the neural network, or to estimate or determine the network parameters $w_{ij}, \theta_{ij}, c_j; i = 1, 2, \dots, q,\ j = 1, 2, \dots, p$. In this study we assume that the activation function of each neuron is a so-called sigmoid function

$$\varphi(v) = \frac{1}{1 + \exp(-v)}$$

and employ the back-propagation algorithm to train the neural network which is implemented on the MATLAB version 4.2 environment. [1] In theory, a multilayer perception shown in Fig. 4 can approximate any continuous function $y = f(x_1, x_2, \dots, x_p)$ to any desired accuracy when $q = 2p + 1$.

4. Basic results

In software reliability modeling it is often required to forecast future software failures by use of observed software failures. Specifically, suppose x_i is the time between the $(i-1)$th software failure and the ith software failure, we need to forecast x_{i+1} by use of $\{x_1, x_2, \dots, x_i\}$, or in general, we need to forecast x_{i+k} by use of $\{x_i, x_{i+1}, \dots, x_{i+k-1}\}$, i.e., use the most recent k failures to forecast the next failure. This is equivalent to saying that we seek a functional relationship

$$x_{i+k} = f_{i,k}(x_i, x_{i+1}, \dots, x_{i+k-1}), \quad i = 1, 2, \dots$$

Obviously, $f_{i,k}$ should be non-linear and complicated. This offers a chance for neural networks to play a role, since a multilayer perception can serve as a non-linear functional approximator. In order to apply the neural network approach, we assume that $f_{i,k} \equiv f_k$, or $f_{i,k}$ is irrelevant of i.

In our case, let $k = 50$ at first, that is, we use the most recent 50 failures to forecast the next failure. The neural network we use is shown in Fig. 4, which contains 50 neurons in the input layer, [2] 101 neurons in the hidden layer, and one neuron in the output layer. In this way $(x_i, x_{i+1}, \dots, x_{i+49})$ and x_{i+50} make up a sample to train the neural network, with the former being the inputs, and the latter being the desired output. We choose 50 samples $(x_i, x_{i+1}, \dots, x_{i+49}; x_{i+50}), i = i_0, i_0 + 1, \dots, i_0 + 49$, to train the neural network and then use the trained neural network to forecast $x_{i_0+49+51}, x_{i_0+49+52}, \dots, x_{i_0+49+80}$. Specifically we have the following procedure for DX_1 which contains 163 failures in total:

1. Randomly choose [3] a i_0 from $1, 2, \dots, 83$;
2. Let $m = 0$;

[1] MATLAB is a popular software package used for control systems design.

[2] Other cases will be discussed in Section 5.

[3] $83 = 163 - 50 - 30$. i_0 cannot be greater than 83 since we have only 163 data points and each time use the most recent 50 failures to predict the next failure for 30 times, that is, $i_0 + 50 + 30 \leqslant 163$.

3. Use $(x_i, x_{i+1}, \ldots, x_{i+49}; x_{i+50}); i = i_0, i_0 + 1, \ldots, i_0 + 49$, to train the neural network;
4. Let $m = m + 1$;
5. Use $(x_{i_0+50}, x_{i_0+51}, \ldots, x_{i_0+50+49})$ as input to the trained neural network to forecast the next failure $x_{i_0+50+50}$;
6. Let $i_0 = i_0 + 1$;
7. If $m < 30$, go to step 4);
8. End.

The error tolerance bound for the back-propagation algorithm is 0.005. Here we adopt a linear scaling function to transform $\{x_{i+50}, i = i_0, i_0 + 1, \ldots, i_0 + 49\}$ into unit interval $[0, 1]$,

$$x^*_{i+50} = \frac{x_{i+50}}{\max\limits_{j=i_0,\ldots,i_0+49}\{x_{j+50}\}}$$

since the output of the neural network should lie in $[0, 1]$.

The above procedure is the so-called short-term prediction, that is, $(x_j, x_{j+1}, \ldots, x_{j+49})$ are used to predict x_{j+50}. A counterpart is the so-called long-term prediction, that is, once the predicted value of $x_{i_0+50+50}$, denoted by $\hat{x}_{i_0+50+50}$, is obtained, it is used as input to the trained network to generate the predicted value of $x_{i_0+50+51}$, denoted by $\hat{x}_{i_0+50+51}$. Further, $\hat{x}_{i_0+50+50}$ and $\hat{x}_{i_0+50+51}$ are used to predict $x_{i_0+50+52}$, and so forth. Figs. 5–8 depict the training and prediction results for $DX1$, including short-term and long-term.

For $DX2$, we can follow the same procedure formulated above except randomly choosing i_0 from $1, 2, \ldots, 111$, since there are 191 failures in total $(191 - 50 - 30 = 111)$. The training and prediction results are shown in Figs. 9–12. Table 1 summarizes the training and prediction results of $DX1$ and $DX2$, where the relative RE is defined as

$$\mathrm{RE} = \left|\frac{\hat{x}_i - x_i}{x_i}\right| \times 100\%$$

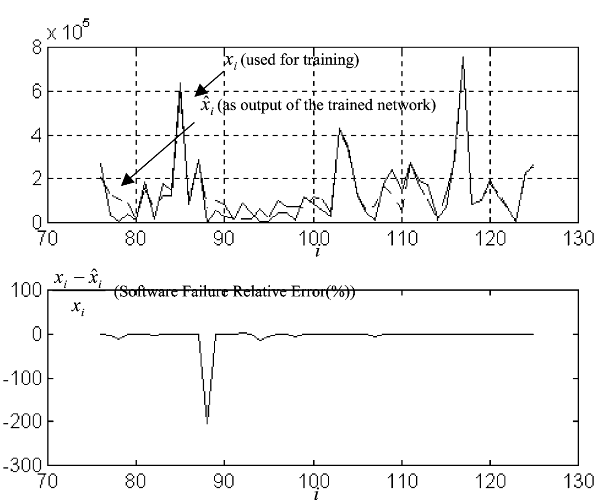

Fig. 5. Training results of the first software reliability dataset ($DX1$).

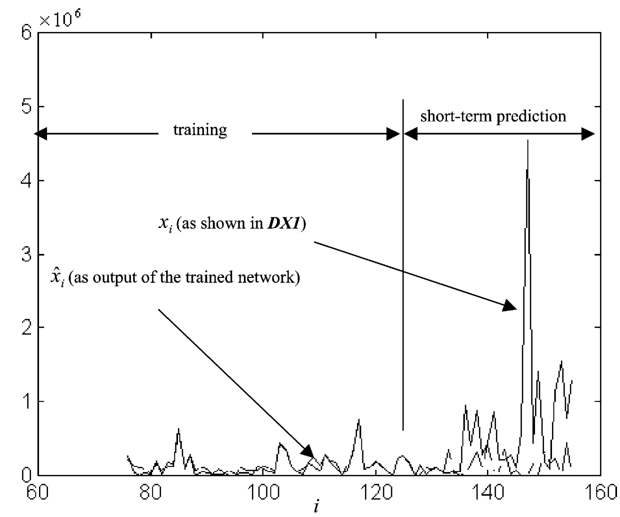

Fig. 6. Training and short-term prediction results of the first software reliability dataset ($DX1$).

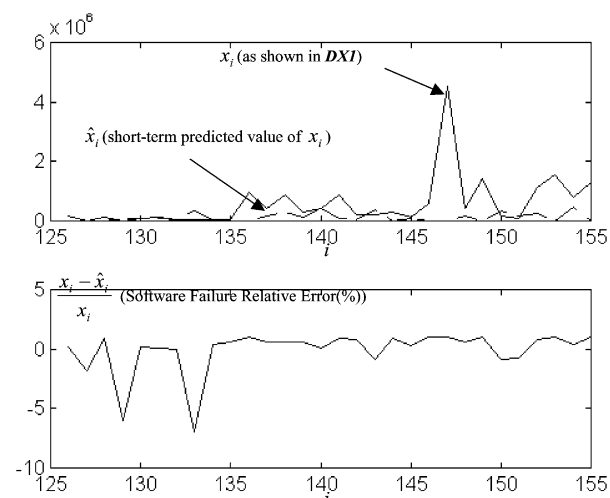

Fig. 7. Short-term prediction results of the first software reliability dataset ($DX1$).

with x_i being the actual value, and let $\hat{x}_i$ be the corresponding trained or predicted value.

From Figs. 5–12 and Table 1, we observe:

1. In the 50 training samples of $DX1$, only 52% trained outputs have relative error less than 20%; in the 50 training samples of $DX2$, 74% trained outputs have relative error less than 20%. This suggests that the training accuracy is not high.
2. In the 30 short-term predictions of $DX1$, only 7% predicted values have relative error less than 20%; things are similar for long-term predictions and $DX2$. This suggests that the prediction accuracy is low.
3. Short-term predictions are somewhat better than long-term predictions, but the improvements are minor; this can be explained by the fact that the short-term predictions are poor by themselves.

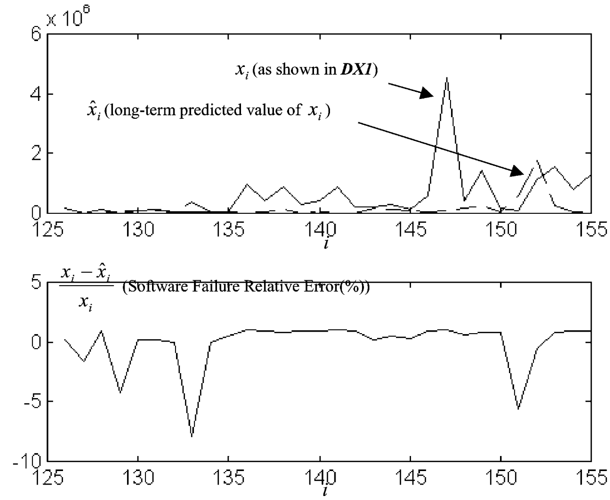

Fig. 8. Long-term prediction results of the first software reliability dataset ($DX1$).

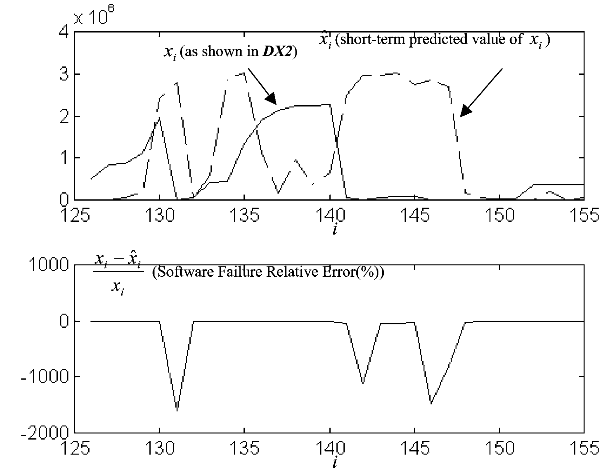

Fig. 11. Short-term prediction results of the second software reliability dataset ($DX2$).

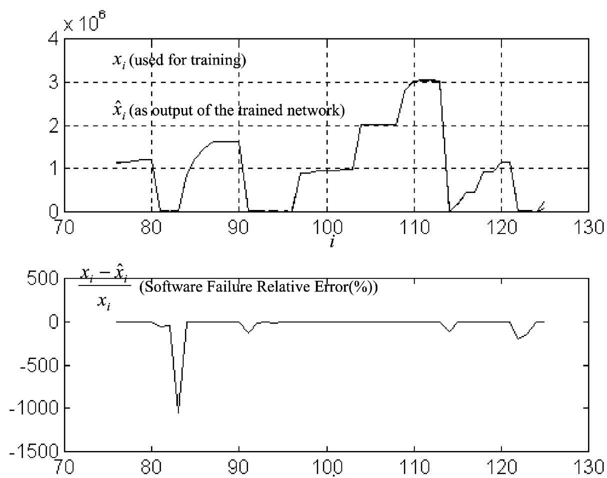

Fig. 9. Training results of the second software reliability dataset ($DX2$).

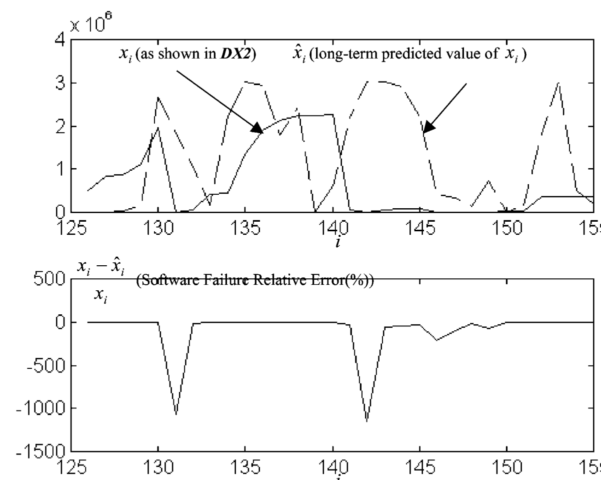

Fig. 12. Long-term prediction results of the second software reliability dataset ($DX2$).

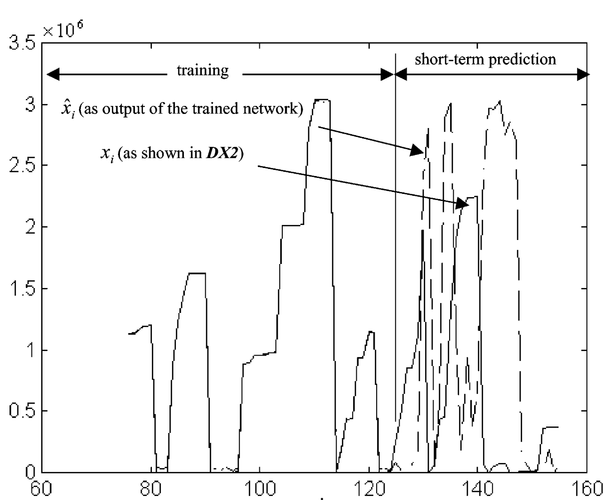

Fig. 10. Training and short-term prediction results of the second software reliability dataset ($DX2$).

Table 1
Training and prediction result of $DX1$ and $DX2$

Proportion of with RE $\leqslant$ 20%	Training dataset (%)	Short-term prediction dataset (%)	Long-term prediction dataset (%)
$DX1$ Set	52	7	7
$DX2$ Set	74	7	3

4. It seems that $DX2$ shows better training results than $DX1$. This may be due to the difference between $DX1$ and $DX2$. From Fig. 2 we see that $DX1$ has a more fluctuating pattern than $DX2$; it is reasonable to believe that neural networks can mimic a smooth pattern.
5. From other training results (not listed here for the sake of space limitation) we see that larger x_i often leads to smaller relative error; this may be explained

by the criterion of the back-propagation learning algorithm which attempts to minimize

$$E = \frac{1}{2} \sum_{j=i_0}^{i_0+49} (\hat{x}_j^* - x_j^*)^2.$$

5. Effects of network architectures

From the last section we see that the multilayer perception with one hidden and 50 input neurons does not offer satisfactory results for handling $DX1$ and $DX2$. In this section we examine if changes in network architectures can have positive effects on handling $DX1$ and $DX2$.

5.1. Number of input neurons

Refer to Fig. 3, we still use the neural network with $q = 2p + 1$ except that p may or may not be 50. Following the same procedure presented in Section 4 of training the neural network and forecasting the future failures, irrelevant of the value of p, we always use 50 samples to train the neural network. Table 2 summarizes the training and prediction results for $DX1$ with varying values of p. The corresponding error tolerance bound for the back-propagation algorithm is 0.02. We see that with increasing number of input neurons, the training and prediction results are not necessarily improved. This may be due to the nature of $DX1$: it seems that $DX1$ corresponds to a varying operational profile. As observed by Schneidewind in *statistical* software reliability modeling, not all the observed data should be used to forecast reliability behavior since old data may not be as representative of the current and future failure processes as recent data (Schneidewind, 1993).

5.2. Four-layer architecture

Instead of using the multilayer perception with one hidden layer, here we use a multilayer perception with two hidden layers and 50 input neurons. Each hidden layer consists of $2 \times 50 + 1 = 101$ neurons. Following the same procedure presented in Section 4 to process $DX1$, we arrive at Figs. 13 and 14. Surprisingly, we see that in-

Table 2
Training and prediction results for $DX1$ with a varying number of input neurons

Proportion of data with RE $\leqslant$ 100%	Training dataset (%)	Short-term prediction dataset (%)	Long-term prediction dataset (%)
Number of input neurons (p)			
20	64	10	10
30	80	30	23.3
40	82	16.67	16.7
50	98	73	70

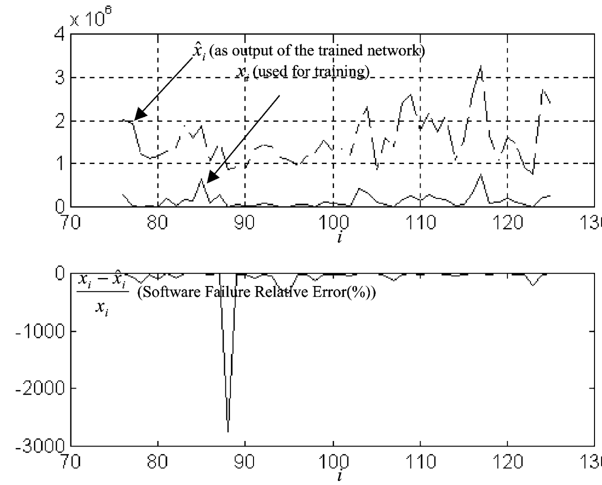

Fig. 13. Training results of the first software reliability dataset ($DX1$) by use of four-layer network.

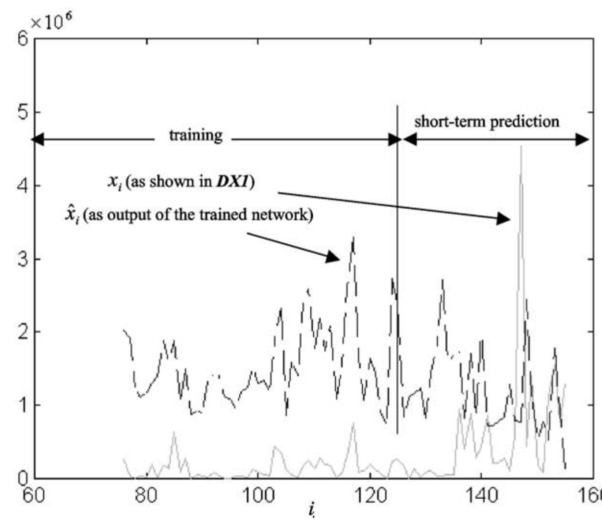

Fig. 14. Training and short-term prediction results of the first software reliability dataset ($DX1$) by use of four-layer network.

clusion of one more hidden layer does not help. Rather, it makes the training and prediction results worse. This is counter to intuition, since more hidden layer offers more network parameters and thus more degrees of freedom for the neural network to approximate a given functional relationship. A possible explanation for the surprising results may be that more neurons mean more non-linear functional transformations. We note that each neuron contains an activation function.

5.3. Statistic

From Fig. 2 we see that $\{x_i\}$ is subject to large variations, or they are random in some sense. This leads us to wondering if inclusion of some probabilistic statistic in the neural network can help. Specifically, we use the mean of input signals as an extra input signal.

Table 3
Training and prediction results for $DX1$ with and without the use of a probabilistic statistic

Proportion of dataset with RE $\leqslant$ 20%	Training dataset (%)	Short-term prediction dataset (%)	Long-term prediction dataset (%)
$p = 50$	53	6	6
$p = 51$	51	6	4

$$MX_i = \frac{x_i + x_{i+1} + \cdots + x_{i+49}}{50}.$$

In this way the neural network has 51 neurons in the input layer, and thus $(x_i, x_{i+1}, \ldots, x_{i+49}, MX_i; x_{i+50})$ makes up a training sample. Following the same procedure presented in Section 4 of using the neural network with $p = 50$ and with $p = 51$, respectively, as shown in Fig. 3, to process $DX1$, we arrive at Table 3; the error tolerance bound for the back-propagation algorithm is 0.005. The case of $p = 50$ means that MX_i is not used, whereas the case of $p = 51$ corresponds to using MX_i, [4] we see that using a probabilistic statistic to mimic the 'random' nature of $\{x_i\}$ does not help.

5.4. Difference signals

A possible reason for a neural network not generating satisfactory results is that it is not smart or predictive enough. In order to improve the predictiveness of a neural network, we wonder if difference signals can help. [5] We may employ difference signals as input signals to a neural network to 'predict' the next failure. Specifically, let

$$\Delta x_i = x_{i+1} - x_i.$$

First, we use

$$\{\Delta x_i, \Delta x_{i+1}, \ldots, \Delta x_{i+48}, \Delta x_{i+49}\},$$
$$i = i_0, i_0 + 1, \ldots, i_0 + 49,$$

to train a multilayer perception with one hidden layer, 49 input neurons (for $\Delta x_i, \Delta x_{i+1}, \ldots, \Delta x_{i+48}$) and one output neuron (for Δx_{i+49}), and eventually to generate a $\Delta\hat{x}_{i+49}$. Then let $\tilde{x}_{i+50} = x_{i+50} + \Delta\hat{x}_{i+49}$. We may treat $\tilde{x}_{i+50}$ as a predicted value of x_{i+50}. In turn, we use a second multilayer perception to process $DX1$. The procedure is the same as that presented in Section 5.3 except using $\tilde{x}_{i+50}$ to replace MX_i. This leads to Table 4; the corresponding error tolerance bound for the back-propagation algorithm is 0.005. Obviously, using difference signals helps very little.

[4] The results corresponding to $p = 50$ are slightly different from those presented in Table 1. One cannot guarantee that a neural network can generate the same training and prediction results in two runs.

[5] This is inspired by the idea of PID control in control engineering. In a PID controller, 'D' denotes the difference and attempts to 'predict' the future behavior of a controlled object.

Table 4
Training and prediction results for $DX1$ with and without using difference signals

Proportion of dataset with RE $\leqslant$ 20%	Training dataset (%)	Short-term prediction dataset (%)	Long-term prediction dataset (%)
Without using difference signals	52	7	7
With using difference signals	67	4	3

In dealing with the first multilayer perception, since Δx_i may be negative or positive, we adopt a scaling function

$$\Delta x_i^* = \frac{\Delta x_i}{2 \times \max_j |\Delta x_j|} + \frac{1}{2}.$$

Then Δx_i^* always lie in the unit interval $[0, 1]$.

6. Effects of scaling functions

A scaling function transforms an observed (actual) value of x_i into the unit interval [0,1]. In Section 4 we adopt a linear scaling function. In this section we consider the non-linear scaling function. Specifically, in addition to the linear scaling function shown in Section 4, we employ logarithmic and exponential functions as follows

$$\Delta x_{i+50}^* = \ln(x_{i+50}) / \max_{j=i_0,\ldots,i_0+49} \{ \ln(x_{j+50}) \},$$
$$\Delta x_{i+50}^* = \exp(-x_{i+50}/a),$$

where a is a parameter we need to further specify. Following the same procedure presented in Section 4 for processing $DX1$ and $DX2$ except adopting a different scaling function, we arrive at Tables 5 and 6.

From Tables 5 and 6 we see that non-linear scaling functions improve the training results of $DX1$, but have little essential effect on the prediction results of $DX1$ and the training and prediction results of $DX2$. Actually, linear scaling function does not affect the data structures of $DX1$ and $DX2$, whereas non-linear scaling functions will do a reverse. The non-linear scaling functions reduce the large fluctuations of $DX1$ smaller, or the transformed

Table 5
Training and prediction results of $DX1$ by use of different scaling functions

Proportion of data with RE $\leqslant$ 20%	Training dataset (%)	Short-term prediction dataset (%)	Long-term prediction dataset (%)
Scaling function			
Linear	52	7	5
Logarithmic	82	7	3
Exponential ($a = 2 \times 10^6$)	78	7	3

Table 6
Training and prediction results of $DX2$ by use of different scaling functions

Proportion of data with RE $\leqslant$ 20%	Training dataset (%)	Short-term prediction dataset (%)	Long-term prediction dataset (%)
Scaling function			
Linear	74	7	3
Logarithmic	88	13	7
Exponential ($a = 2 \times 10^6$)	74	7	7

Table 7
Training and prediction results of $DX1$ by use of exponential scaling function with different values of a

Proportion of data with RE $\leqslant$ 20%	Training dataset (%)	Short-term prediction dataset (%)	Long-term prediction dataset (%)
a			
50 000	78	10	6.7
2×10^5	92	10	3
2×10^6	78	7	3

dataset of $DX1$ demonstrates a smooth pattern, and thus improve the training results of the neural network. Since $DX2$ has a rather smooth data structure (trend) in itself, it is reasonable to believe that non-linear scaling functions do not lead $DX2$ to a better (smoother) data structure and thus the corresponding training results of the neural network can hardly be improved. As to the prediction results, we need to have inverse transformations of the scaling functions to obtain them. So, a non-linear scaling function does not necessarily improve the prediction results of a neural network.

An advantage of the exponential scaling function is that it always transforms x_i into [0,1] and it is not necessary to care for the maximum value of x_i. An uncertainty is that we need to further specify the parameter a. Table 7 tabulates the training and prediction results of $DX1$ by use of the exponential scaling function with different values of a. Large values of a tend to make the resulting dataset of transformed $DX1$ uniformly distributed over the unit interval [0,1] and thus improve the training results of the neural network. Here we follow the same procedure as that presented in Section 4 except using a different scaling function.

7. Effects of filtering

Noises may be associated with observed values of a real signal. A filtering scheme attempts to eliminate the noises from the observed values of a signal and obtain trustable values of the signal. [6] Actually, the non-linear scaling function plays a role of filtering in some sense and reduces high-frequency noises. Since the results, and especially the prediction results presented in the previous sections are not very satisfactory, in this section we examine if alternative filtering scheme can help. Specifically, we transform $\{x_i\}$ into $\{t_i\}$ with

$$t_i = \sum_{j=1}^{i} x_j.$$

This can be viewed as a low-frequency pass filter. Then $DX1$ and $DX2$ are transformed into datasets $DT1$ and $DT2$, respectively. From Fig. 15 we see that $DT2$ demonstrates a rather stable pattern, whereas the trend of $DT1$ changes dramatically around t_{140}. Recall that $DX1$ was collected in a software testing phase, and $DX2$ was collected in a software operational phase, we may argue that the software corresponding to $DT1$ was tested under a varying test profile, or there were at least two test profiles, whereas the software corresponding to $DT2$ runs under an unchanged operational profile. We may

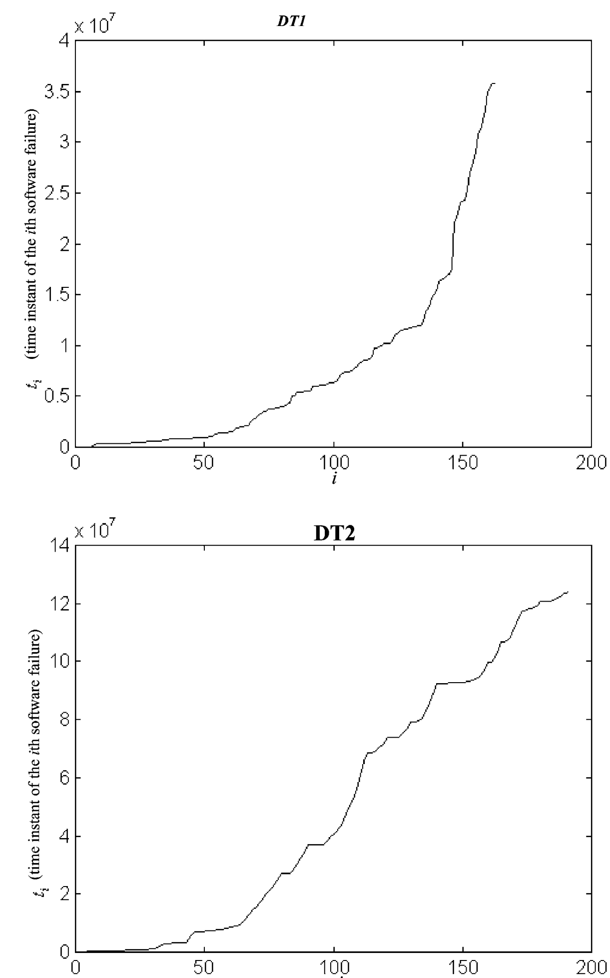

Fig. 15. Cumulating patterns of the first and second software reliability datasets.

[6] Filtering is an essential concept in control engineering. It plays a very important role in modern control theory and systems.

Table 8
Training and prediction results of $DT1$ and $DT2$

Proportion of data with RE $\leqslant$ 20%	Training dataset (%)	Short-term prediction dataset (%)	Long-term prediction dataset (%)
Dataset			
DT_1	100	93	40
DT_2	100	77	57

also argue that $DT1$ corresponds to a reliability growth trend, whereas $DT2$ demonstrates a stable reliability level. From these observations we may further argue that the pattern of $\{t_i\}$ may serve as a pragmatic tool for detecting change in software operational (test) profile. A pattern of stable increment in $\{t_i\}$ corresponds to an unchanged software operational (test) profile; a pattern of unstable increment in $\{t_i\}$ corresponds to a varying software operational (test) profile. On the other hand, we also note that $DT1$ is 'smoother' than $DT2$, or larger fluctuations are associated with $DT2$. Applying the same procedure presented in Section 4 to $DT1$ and $DT2$ by use of the linear scaling function, we arrive at Table 8. The neural network provides very good training results and satisfactory short-term prediction results. Even for the long-term predictions, the accuracy has been greatly improved in comparison with that for $DX1$ and $DX2$.

Further we note that the short-term predictions of $DT1$ are better than those of $DT2$, whereas an opposite situation emerges for the long-term predictions. This may be explained as follows: short-term predictions focus on local patterns of datasets, whereas $DT1$ has better or smoother local pattern than $DT2$ that is more fluctuating locally. On the other hand, long-term predictions are more concerned with global patterns of datasets. Since $DT1$ changes dramatically around $i = 140$, whereas no dramatic fluctuations happen in $DT2$, we may say that $DT2$ has a better or smoother global pattern than $DT1$. In other words, neural networks generate satisfactory results for smooth datasets and poor results for datasets with large fluctuations.

Then can the attractive results of $DT1$ and $DT2$ be used to improve the training or prediction results of DX_1 and DX_2? We apply the neural network approach to $DT1$ and $DT2$, and then use the training and prediction results of $DT1$ and $DT2$ to obtain the corresponding results of $DX1$ and $DX2$, respectively, employing the relationship $x_i = t_i - t_{i-1}$. In this way we arrive at Figs. 16 and 17, and Table 9. Compared to Figs. 6 and 10 and Table 1, we see that the results do not get improved; rather, they become worse in some sense.

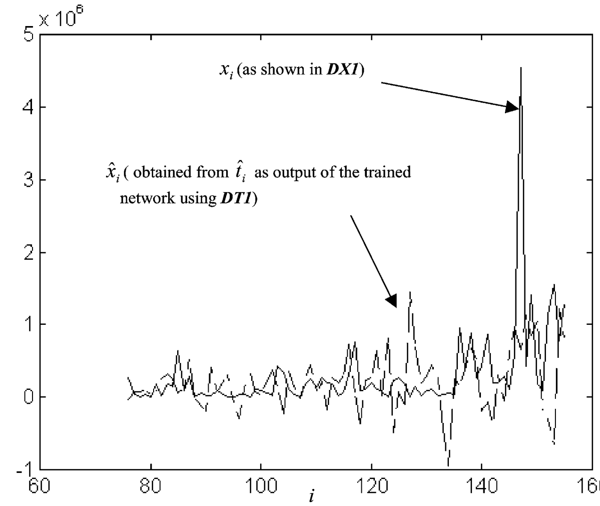

Fig. 16. Training and short-term prediction results of the first software failure dataset ($DX1$) via Its cumulating dataset ($DT1$).

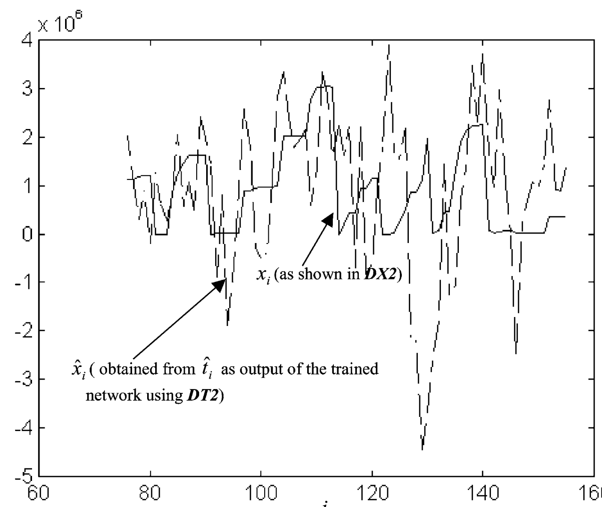

Fig. 17. Training and short-term prediction results of the second software failure dataset ($DX2$) via Its cumulating dataset ($DT2$).

Table 9
Training and prediction results of $DX1$ and $DX2$ by use of $DT1$ and $DT2$

Proportion of data with RE $\leqslant$ 20%	Training dataset (%)	Short-term prediction dataset (%)
Dataset		
$DX1$ by use of $DT1$	42	10
$DX2$ by use of $DT2$	47	3

8. Qualitative behavior

From Section 4 we note that since $DX1$ and $DX2$ are highly fluctuating, the prediction results are not quantitatively satisfactory. Then how about the qualitative behavior of the prediction results? By qualitative behavior we mean the overall pattern, e.g., probability distribution, rather than particular quantitative values of some variables of concern. We transform Figs. 6 and

10 into Figs. 18 and 19, respectively, to make up histograms. That is, we generate histograms for the datasets used for training the neural networks and for the short-term results predicted by the trained neural network, respectively. We see that empirical probability density histograms of the training data are similar to those of the short-term prediction results. Actually we realize this is also the case even if non-linear scaling functions are employed (refer to Section 6). In other words, the qualitative pattern of predictions mimics that of the training data in terms of probability distributions. What a neural network can predict is what it has learned. Neural networks demonstrate satisfactory qualitative behavior. This coincides with our previous observation of the neural network approach in processing static software reliability data: the neural network approach is good at module classification, although it is poor at quantitatively estimating the number of software defects (Cai, 1998).

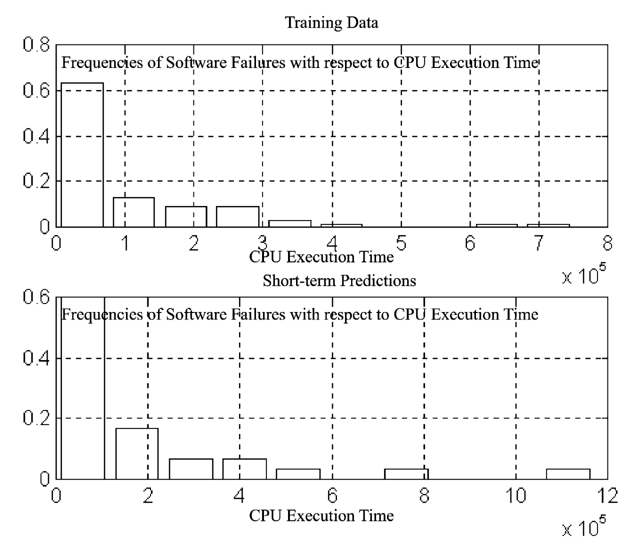

Fig. 18. The first software failure dataset ($DX1$): histogram of training data versus that of short-term predictions.

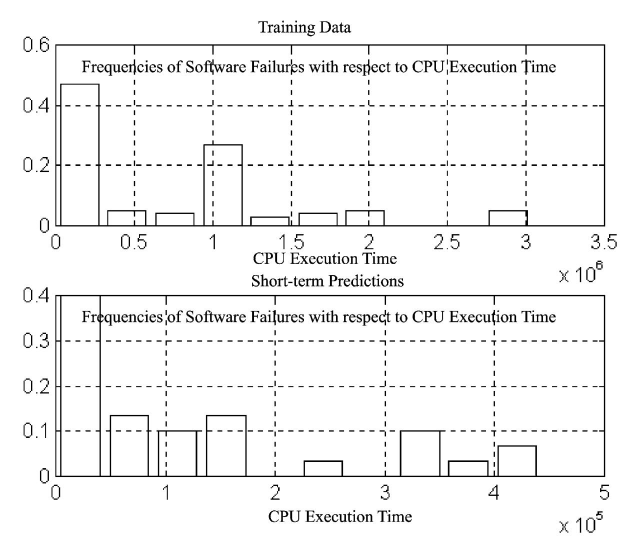

Fig. 19. The second software failure dataset ($DX2$): histogram of training data versus that of short-term predictions.

9. Software failure counts

Another type of software reliability data is represented in terms of the number of software failures observed in successive time intervals, as depicted in Fig. 20. In the first time interval of length τ_1, k_1 software failures are observed; in the second time interval of length τ_2, k_2 software failures are observed; and in the ith time interval of length τ_i, k_i software failures are observed. In this section we examine how well neural networks can process this type of software reliability data.

9.1. Basic results

For the convenience of comparisons with the results presented in the previous sections, here we still use the datasets $DX1$ and $DX2$. However we transform them into software failure counts. Specifically, for $DX1$, let $\tau = t_{163}/m = \tau_1 = \tau_2 = \cdots = \tau_m$. That is, the entire observed time span is divided into m intervals of equal length. For $DX2$, let $\tau = \tau_{191}/m = \tau_1 = \tau_2 = \cdots = \tau_m$. In this way $DX1$ and $DX2$ are transformed into $DK1$ and $DK2$, respectively, in terms of $\{k_1, k_2, \ldots, k_m\}$. Figs. 21 and 22 show $\{k_1, k_2, \ldots, k_m\}$ with $m = 80$ versus the observed time for $DK1$ and $DK2$, respectively. We see that $DK1$ and $DK2$ demonstrate different patterns.

In order to process $\{k_1, k_2, \ldots, k_m\}$, we still employ the neural network depicted in Fig. 4. However here the input layer consists of about $m/3$ neurons, say, p^* neurons. In this way $(k_i, k_{i+1}, \ldots, k_{i+p^*-1}; k_{i+p^*})$ makes up a training sample. About $m/3$ samples are used to train the neural network, and then the trained neural network is used to predict the number of software failures observed in the remaining about $m/3$ time intervals. Figs. 23 and 24 show the training and short-term prediction results for $DK1$ and $DK2$ with $m = 40$. Since k_i must be a non-negative integer, the actual outputs (predicted values) of the trained neural network have been rounded to the closest integers. Tables 10 and 11 summarize the training and prediction results for $DK1$ and $DK2$ with different values of m.

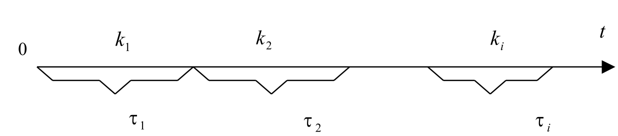

Fig. 20. Software failure counts.

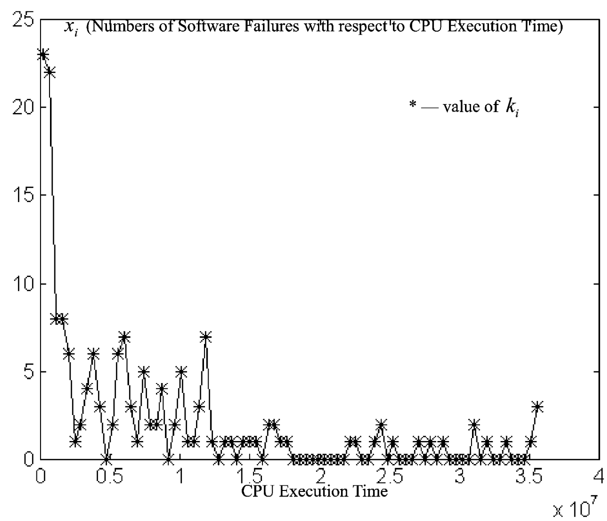

Fig. 21. Behavior of the first software failure count dataset ($DK1$) with $m = 80$ with respect to failure time.

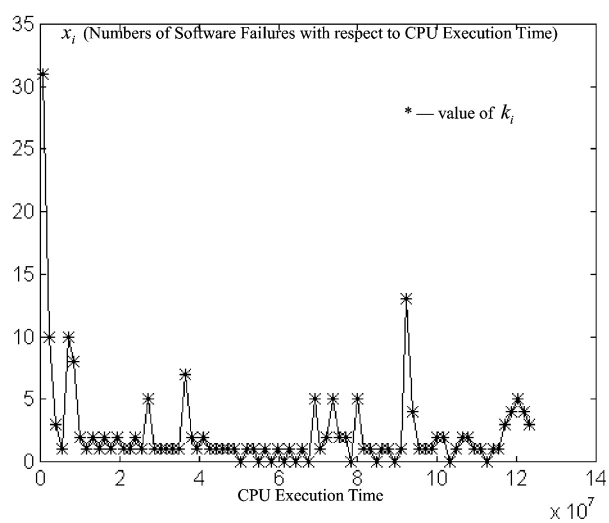

Fig. 22. Behavior of the second software failure count dataset ($DK2$) with $m = 80$ with respect to failure time.

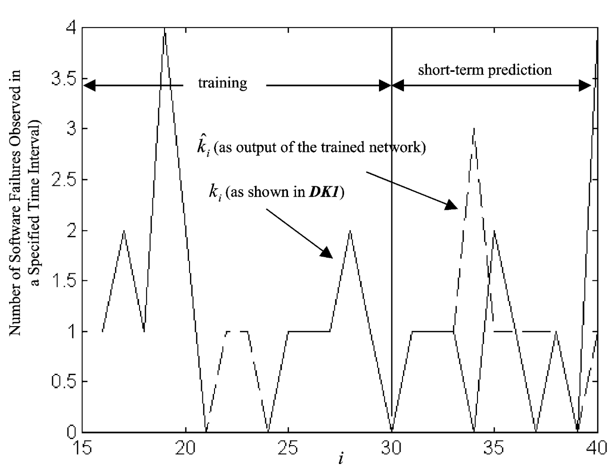

Fig. 23. Training and Short-term prediction results of the first software failure count dataset ($DK1$) with $m = 40$.

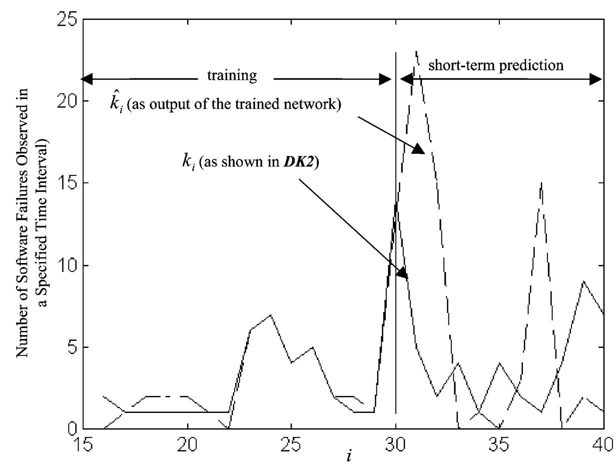

Fig. 24. Training and short-term prediction results of the second software failure count dataset ($DK2$) with $m = 40$.

Table 10
Training and prediction results of $DK1$ ($\hat{k}_i$ – predicted value of k_i)

Proportion of data with $k_i = \hat{k}_i$	Training dataset (%)	Short-term prediction dataset (%)	Long-term prediction dataset (%)
m			
40	93	20	20
80	93	52	64
100	89	66	69

Table 11
Training and prediction results of $DK2$ ($\hat{k}_i$ – predicted value of k_i)

Proportion of data with $k_i = \hat{k}_i$	Training dataset (%)	Short-term prediction dataset (%)	Long-term prediction dataset (%)
m			
40	67	20	0
80	83	16	8
100	77	29	31
120	92	32	38

Table 10 suggests that with increasing m, the prediction result for $DK2$ is improved. This can be interpreted as follows: with increasing m, the number of observed software failures in each time interval reduces and tends to be zero or one. Then the prediction task looks like a problem of classification, and a neural network can do this well. However the different pattern of $DK2$ makes the short-term prediction results for $DK2$ with $m = 80$ not get improved in comparison with those for DK_2 with $m = 40$. But in general, the increasing m improves the prediction behavior.

9.2. *Effects of filtering*

Similar to Section 7, we can sum up $k_1, k_2, \ldots, k_i$ and examine the effects of filtering. Let $kc_i = \sum_{j=1}^{i} k_j$, and in

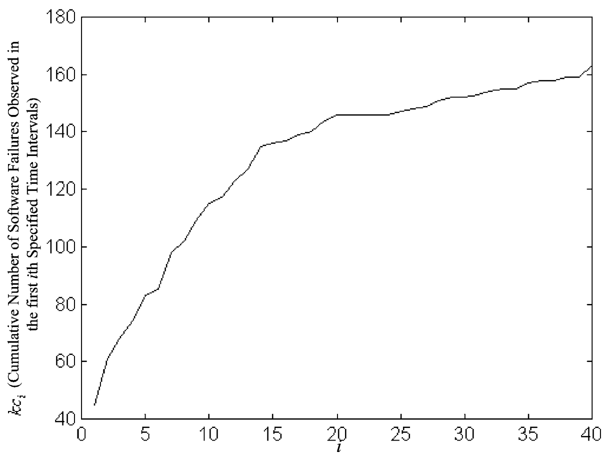

Fig. 25. Cumulating pattern of dataset of the first software failure count dataset ($DKC1$) with $m = 40$: kc_i with respect to i.

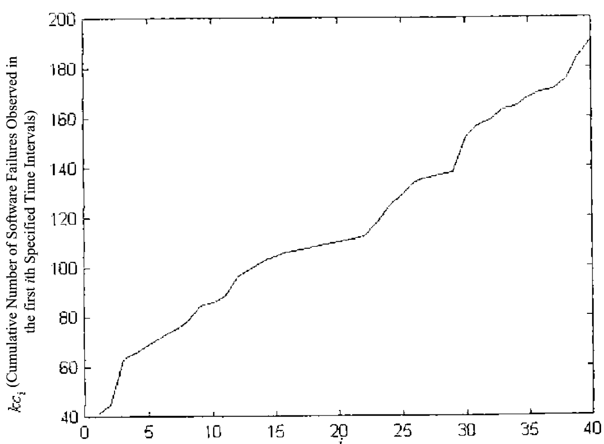

Fig. 26. Cumulating pattern of dataset of the second software failure count dataset ($DKC2$) with $m = 40$: kc_i with respect to i.

this way we transform $DK1$ and DK_2 into $DKC1$ and$DKC2$, respectively, in terms of $\{kc_1, kc_2, \ldots\}$. Figs. 25 and 26 show the patterns of $DKC1$ and $DKC2$, respectively, with $m = 40$. Following the same procedure of processing $DK1$ and $DK2$ as conducted in Section 9.1, except k_i being replaced by kc_i, we arrive at Tables 12 and 13. Here $\hat{kc}_i$ denotes the predicted value of kc_i (in integer), and the relative error is defined as

$$\mathrm{RE} = \left| \frac{kc_i - \hat{kc}_i}{kc_i} \right| \times 100\%,$$

whereas the predicted value of k_i can be determined via $\{\hat{kc}_i\}$ as

$$\hat{k}_i = \hat{kc}_i - \hat{kc}_{i-1}$$

We see that $DKC1$ can be processed by the neural network approach very well. This coincides with the previous observation by Karuanithi et al., 1992a,b. However things are somewhat different for DKC_2. The prediction behavior varies with m and may be unsatisfactory. On the other hand, refer to Tables 10 and 11, using $\{\hat{kc}_i\}$ to determine $\{\hat{k}_i\}$ does not help to get $\{\hat{k}_i\}$ improved.

10. General discussions

From the results presented in the previous sections, we can have the following observations.

1. The effectiveness of the neural network approach depends heavily on the nature of handled dataset. It may generate satisfactory results for 'smooth' datasets, but is poor at processing highly fluctuating datasets.

Table 12
Basic results of processing $DKC1$

m	Proportion of data with RE $\leqslant$ 20%			Proportion of data with $k_i = \hat{k}_i$, with $\hat{k}_i$ being determined via $\hat{kc}_i$	
	Training dataset (%)	Short-term prediction dataset (%)	Long-term prediction dataset (%)	Training dataset (%)	Short-term prediction dataset (%)
40	100	100	100	27	20
80	100	100	100	27	56
100	100	100	100	34	51

Table 13
Basic results of processing $DKC2$

m	Proportion of data with RE $\leqslant$ 20%			Proportion of data with $k_i = \hat{k}_i$, with $\hat{k}_i$ being determined via $\hat{kc}_i$	
	Training dataset (%)	Short-term prediction dataset (%)	Long-term prediction dataset (%)	Training dataset (%)	Short-term prediction dataset (%)
40	100	80	20	13	0
80	100	72	20	13	12
100	100	29	26	20	26
120	100	45	57	20	15

2. In general, the training results of a neural network are much better than the prediction results of the corresponding trained neural network. That is, a neural network may well describe what has happened, but in the mean time, fail to forecast what will happen.
3. Even if a neural network may fail to demonstrate satisfactory quantitative behavior, its qualitative behavior may still look good: – the empirical probability density distribution of the predicting data resembles that of the training data. In other words, a neural network can predict what it has learned.
4. A linear scaling function keeps the structure of training data unchanged, and things are different for a non-linear scaling function. However this does not mean that a non-linear scaling function will certainly improve the behavior of a neural network. Whether a linear or non-linear scaling function can help is largely subject to the nature of data structure after applying the scaling function.
5. Applying a filtering technique to a fluctuating dataset may make the dataset smooth and thus help a neural network to generate better results. However, compared to handling a fluctuating dataset directly without filtering, handling the dataset with filtering may impair the accuracy of the prediction results for the original data since a transformation (inverse to the filtering) must be carried out in this circumstance.
6. An advantage of the neural network approach is that numerous factors or signals can be taken into account simultaneously. For example, $x_1, x_2, \ldots x_i$ can be employed to forecast x_{i+k}, whereas i may be theoretically unbounded.
7. Compared to statistical techniques that normally make various unrealistic statistical assumptions, no assumption of this kind is made in the neural network approach. Further, a neural network represents a non-linear function.
8. However the neural network approach essentially follows a black-box philosophy. No causal relationships between available data and predicted data are considered even if they may exist. This means a portion of the useful information is ignored.
9. Although it has mathematically been shown that any continuous function can be approximated to any desired accuracy by a multilayer perception, the neural network approach lacks a sound theoretical formulation. Selection of a network architecture or determination of the number of neurons is largely a kind of art.
10. It seems that no explicit assumptions are made in the neural network approach. However several implicit assumptions have been taken. First, we assume

    $$x_{i+k} = f_{i,k}(x_i, x_{i+1}, \ldots, x_{i+k-1}); \quad i = 1, 2, \ldots$$

    That is, x_{i+k} must be uniquely determined by $\{x_i, x_{i+1}, \ldots, x_{i+k-1}\}$ or $f_{i,k}$ must exist. Second, $f_{i,k}$ must be irrelevant of i. Third, $f_{i,k}$ must be continuous. No sufficient evidence has been presented to justify these assumptions.
11. Neural networks are good at approximating a given and known non-linear function. However when we train a neural network, we are actually trying to identify a function $f_k = f_{i,k}$. The function is unknown beforehand. Being good at approximating does not certainly mean being good at identifying.
12. Also, being good at approximating does not certainly mean being good at forecasting. A neural network may have good training results and poor prediction results simultaneously, and we are more concerned with the prediction results.
13. The power of a neural network seemingly comes from the collective contribution of the numerous neurons. On the other hand, the numerous neurons also make the network outputs 'robust' to the variations of the network inputs. Increasing the number of neurons may make the network outputs less sensitive to the changes in network inputs. A neural network with too many neurons will not be swift enough to follow a fluctuating trend in the input data. This may partially explain why including more layer or input neurons does not necessarily help to handle fluctuating datasets (refer to Section 5).
14. Since the network parameters are initialized randomly in a learning algorithm, there is a problem of repeatability of training and prediction results of a neural network. That is, even if the same training sample set and the same learning scheme are employed, the same network outputs can hardly be generated twice.
15. There are some essential problems with software reliability data. First, software reliability data often demonstrate a highly fluctuating pattern, or they are 'over-random'. Second, varying software operational profiles may make software reliability data more fluctuating. Third, unexpected reasons may introduce sharp 'outliers' into software reliability data. Fourth, compared to the requirements of achieving accurate estimates of the numerous network parameters, the volume of software reliability data available is rather small.
16. Therefore, more often than not, the neural network approach does not generate satisfactory quantitative results, though the qualitative behavior should be appreciated.

11. Concluding remarks

Several studies showed that the neural network approach was good at identifying defect-prone modules and predicting the cumulative number of observed software failures (Karuanithi, 1993; Karuanithi and Malaiya, 1992). Our previous study revealed another scenario in applying the neural network approach to software defect predictions (Cai, 1998). It was observed that the neural network approach was quantitatively poor at predicting the number of software defects, but qualitatively good at classifying program modules. In this study we examine the effectiveness of the neural network approach in handling dynamic software reliability data. We use multilayer perceptions to handle time between successive software failures as well as number of software failures observed in successive time intervals. Moreover, we examine the effects of network architectures, scaling functions and filtering techniques. While confirming previous observations to some extent on the neural network approach in software reliability modeling, we have several new findings. Specifically and overall, we have the following major observations:

1. The effectiveness of the neural network approach is largely subject to the nature of the handled datasets. Datasets with smooth trends can be handled well, but things are quite different for datasets with large fluctuations.
2. The training results of a neural network is normally much better than the prediction results of the corresponding trained network. The best a neural network can predict is what it has learned.
3. Although neural network may fail to generate satisfactory quantitative results, it still demonstrates good qualitative behavior. The empirical probability density distribution of predicting data resembles that of training data.
4. Since implicit assumptions are included in the neural network approach and software reliability data may suffer large fluctuations and volume limitation, more often that not, the neural network approach does not generate satisfactory quantitative results for software reliability modeling.

Of course, what we have observed are limited by the datasets, mulitilayer perceptions and the simulation tool (MATLAB 4.2) we employed. More datasets and other types of neural networks and simulation tools should be tried to further justify (or refute) our findings. However due to the essential problems associated with the neural network approach and software reliability data, the observations presented in this paper should make sense.

Acknowledgements

The comments of two reviewers help the authors to improve the readability of the paper. The first author thanks the last author for his arrangement of the first author's visit to the Department of Computer Science, City University of Hong Kong, supported by a special foundation of the City University of Hong Kong.

Appendix A. Software reliability data

See Tables 14 and 15.

Table 14
$DX1$: Musa's data (1979)

i	x_i	i	x_i
1	320.00	83	178 200.00
2	1439.00	84	144 000.00
3	9000.00	85	639 200.00
4	2880.00	86	86 400.00
5	5700.00	87	288 000.00
6	21 800.00	88	320.00
7	26 800.00	89	57 600.00
8	113 540.00	90	28 800.00
9	112 137.00	91	18 000.00
10	660.00	92	88 640.00
11	2700.00	93	43 200.00
12	28 793.00	94	4160.00
13	2173.00	95	3200.00
14	7263.00	96	42 800.00
15	10 865.00	97	43 600.00
16	4230.00	98	10 560.00
17	8460.00	99	115 200.00
18	14 805.00	100	86 400.00
19	11 844.00	101	57 699.00
20	5361.00	102	28 800.00
21	6553.00	103	432 000.00
22	6499.00	104	345 600.00
23	3124.00	105	115 200.00
24	51 323.00	106	44 494.00
25	17 017.00	107	10 506.00
26	1890.00	108	177 240.00
27	5400.00	109	241 487.00
28	62 313.00	110	143 028.00
29	24 826.00	111	273 564.00
30	26 335.00	112	189 391.00
31	363.00	113	172 800.00
32	13 989.00	114	21 600.00
33	15 058.00	115	64 800.00
34	32 377.00	116	302 400.00
35	41 632.00	117	752 188.00
36	41 632.00	118	86 400.00
37	4160.00	119	100 800.00
38	82 040.00	120	194 400.00
39	13 189.00	121	115 200.00
40	3426.00	122	64 800.00
41	5833.00	123	3600.00
42	640.00	124	230 400.58
43	640.00	125	259 200.00
44	2880.00	126	183 600.00
45	110.00	127	3600.00
46	22 080.00	128	144 000.00
47	60 654.00	129	14 400.00
48	52 163.00	130	86 400.00
49	12 546.00	131	110 100.00
50	784.00	132	28 800.00
51	10 193.00	133	43 200.00
52	7841.00	134	57600.00
53	31 365.00	135	46 800.00
54	24 313.00	136	950 400.00
55	298 890.00	137	400 400.00
56	1280.00	138	883 800.00
57	22 099.00	139	273 600.00
58	19 150.00	140	432 000.00

Table 14 (Continued.)

i	x_i	i	x_i
59	2611.00	141	864 000.00
60	39 170.00	142	202 600.00
61	55 794.00	143	203 400.00
62	42 632.00	144	277 680.00
63	267 600.00	145	105 000.00
64	87 074.00	146	580 080.00
65	149 606.00	147	4 533 960.00
66	14 400.00	148	432 000.00
67	34 560.00	149	1 411 200.00
68	39 600.00	150	172 800.00
69	334 395.00	151	86 400.00
70	296 015.00	152	1 123 200.00
71	177 395.00	153	1 555 200.00
72	214 622.00	54	777 600.00
73	156 400.00	155	1 296 000.00
74	16 800.00	156	1 872 000.00
75	10 800.00	157	335 600.00
76	267 000.00	158	921 600.00
77	34 513.00	159	1 036 800.00
78	7680.00	160	1 728 000.00
79	37 667.00	161	777 600.00
80	11 100.00	162	57 600.00
81	187 200.00	163	17 280.00
82	18 000.00		

Table 15
$DX2$: Iyer and Lee's data (1996)

i	x_i	i	x_i
1	21 804.00	97	884 537.00
2	6358.00	98	893 733.00
3	27 294.00	99	950 561.00
4	127 631.00	100	951 988.00
5	189 694.00	101	956 058.00
6	3518.00	102	969 965.00
7	12 210.00	103	978 392.00
8	14 502.00	104	2 009 086.00
9	6562.00	105	2 010 192.00
10	97 212.00	106	2 012 919.00
11	22 276.00	107	2 014 680.00
12	1078.00	108	2 021 405.00
13	1169.00	109	2 788 848.00
14	656.00	110	3025 774.00
15	759.00	111	3 036 425.00
16	489.00	112	3 038 077.00
17	392.00	113	3 032 470.00
18	125.00	114	522.00
19	19 919.00	115	193 947.00
20	390 250.00	116	436 124.00
21	775.00	117	438 474.00
22	2777.00	118	931 130.00
23	3626.00	119	932 493.00
24	6400.00	120	1 142 273.00
25	10 602.00	121	1 142 142.00
26	13 974.00	122	170.00
27	19 877.00	123	196.00
28	21 377.00	124	6037.00
29	24 233.00	125	24 8967.00
30	25 312.00	126	509 844.00
31	261 648.00	127	849 549.00
32	356 934.00	128	853 489.00
33	372 627.00	129	1 114 172.00
34	996 308.00	130	1 965 556.00
35	5152.00	131	1745.00
36	5838.00	132	64 236.00
37	5840.00	133	430 794.00
38	5868.00	134	455 467.00
39	5656.00	135	1 336 603.00
40	897.00	136	1 895 883.00
41	19 872.00	137	2 128 076.00
42	10 9107.00	138	2 239 002.00
43	211 412.00	139	2 239 429.00
44	1 208 852.00	140	2 252 299.00
45	1 211 097.00	141	59 076.00
46	1 215 349.00	142	2646.00
47	5395.00	143	52 920.00
48	6697.00	144	69 995.00
49	11 840.00	145	70 489.00
50	61 291.00	146	1939.00
51	63 205.00	147	3329.00
52	68 671.00	148	6519.00
53	69 939.00	149	10 341.00
54	69 998.00	150	14 604.00
55	18 2314.00	151	19 913.00
56	183 681.00	152	363 130.00
57	169 033.00	153	364 654.00
58	169 635.00	54	365 201.00
59	172 938.00	155	367 205.00
60	222 835.00	156	368 904.00
61	223 496.00	157	624 031.00
62	224 238.00	158	1 404 616.00
63	229 871.00	159	1 548 712.00
64	495 696.00	160	1 569 953.00
65	852 460.00	161	318 386.00
66	1 008 395.00	162	938 843.00
67	1 009 559.00	163	1 472 080.00
68	1 021 474.00	164	2 058 785.00
69	1 022 778.00	165	2 160 478.00
70	1 029 531.00	166	175 109.00
71	1 060 974.00	167	577 275.00
72	1 095 758.00	168	587 033.00
73	1 098 899.00	169	1 814 556.00
74	1 103 357.00	170	1 823 969.00
75	1 105 899.00	171	1 874 995.00
76	1 133 337.00	172	1 993 574.00
77	1 135 500.00	173	2 056 510.00
78	1 167 069.00	174	19 299.00
79	1 194 139.00	175	266 187.00
80	1 197 940.00	176	351 597.00
81	598.00	177	90 093.00
82	598.00	178	56 4758.00
83	37.00	179	589 996.00
84	858 084.00	180	1 139 509.00
85	1 217 027.00	181	44 230.00
86	1 462 761.00	182	49 789.00
87	1 617 029.00	183	79 269.00
88	1 617 030.00	184	139 714.00
89	1 620 637.00	185	311 915.00
90	1 618 435.00	186	322 874.00
91	293.00	187	530 403.00
92	2161.00	188	530 403.00
93	3026.00	189	530 403.00
94	3117.00	190	530 403.00
95	4428.00	191	525 654.00
96	5954.00		

References

Cai, K.Y., 1998. Software Defect and Operational Profile Modeling. Kluwer Academic Publishers, Dordretch.

Iyer, R.K., Lee, I., 1996. Measurement-based analysis of software reliability. In: Lyu, M.R. (Ed.), Handbook of Software Reliability Engineering. McGraw-Hill, New York, pp. 303–358.

Karuanithi, N., 1993. A neural network approach for software reliability growth modeling in the presence of code churn. In: Proceedings of the Fourth International Symposium on Software Reliability Engineering. pp. 310–317.

Karuanithi, N., Malaiya, Y.K., 1992. The scaling problem in neural networks for software reliability prediction. In: Proceedings of the Third International Symposium on Software Reliability Engineering. pp. 76–82.

Karuanithi, N., Whitley, D., Malaiya, Y.K., 1992a. Prediction of software reliability using connectionist models. IEEE Transactions on Software Engineering 18 (7), 563–574.

Karuanithi, N., Whitley, D., Malaiya, Y.K., 1992b. Using neural networks in reliability prediction. IEEE Software 9 (4), 53–59.

Khoshgoftaar, T.M., Allen, E.B., Hudepohl, J.P., Aud, S.J., 1997. Application of neural networks to software quality modeling of a very large telecommunications system. IEEE Transactions on Neural Networks 8 (4), 902–909.

Khoshgoftaar, T.M., Lanning, D.L., 1995. A neural network approach for early detection of program modules having high risk in the maintenance phase. Journal of Systems and Software 29, 85–91.

Khoshgoftaar, T.M., Pandya, A.S., More, H.B., 1992. A neural network approach for predicting software development faults. In: Proceedings of the Third International Symposium on Software Reliability Engineering. pp. 83–89.

Khoshgoftaar, T.M., Lanning, D.L., Pandya, A.S., 1993. A neural network modeling for detection of high-risk program. In: Proceedings of the Fourth International Symposium on Software Reliability Engineering. pp. 302–309.

Khoshgoftaar, T.M., Lanning, D.L., Pandya, A.S., 1994. A comparative study of pattern recognition techniques for quality evaluation of telecommunications software. IEEE Journal on Selected Areas in Communications 12 (2), 279–291.

Khoshgoftaar, T.M., Szabo, R.M., 1996. Using neural networks to predict software faults during testing. IEEE Transactions on Reliability 45 (3), 456–462.

Khoshgoftaar, T.M., Szabo, R.M., Guasti, P.J., 1995. Exploring the behavior of neural network software quality models. Software Engineering Journal, 89–96.

Lyu, M.R., 1996. Handbook of Software Reliability Engineering. McGraw-Hill, New York.

Musa, J.D., 1979. Software Reliability Data. Bell Telephone Laboratories, London.

Musa, J.D., Iannino, A., Okumoto, K., 1987. Software Reliability: Measurement, Prediction, Application. McGraw-Hill, New York.

Neufelder, A.M., 1993. Ensuring Software Reliability. Marcel Dekker, New York.

Schneidewind, N.F., 1993. Software reliability model with optimal selection of failure data. IEEE Transactions on Software Engineering 19 (11), 1095–1104.

Sherer, S.A., 1995. Software fault prediction. Journal of Systems and Software 29, 97–105.

Xie, M., Software Reliability Modeling. World Scientific, Singapore, 1991.

Kai-Yuan Cai is a Cheung Kong Scholar (Chair Professor), jointly appointed by the Ministry of Education of China and the Li Ka Shing Foundation, Hong Kong. He has been a professor at Beijing University of Aeronautics and Astronautics (BUAA) since 1995. He was born in April 1965 and entered BUAA as an undergraduate student in 1980. He received his B.S. degree in 1984, M.S. degree in 1987, and Ph.D. degree in 1991, all from BUAA. He was a research fellow at the Centre for Software Reliability, City University, London, and a visiting scholar at the Department of Computer Science, City University of Hong Kong.
Dr. Cai has published over 40 research papers. He is the author of three books: *Software Defect and Operational Profile Modeling* (Kluwer, Boston, 1998); *Introduction to Fuzzy Reliability* (Kluwer, Boston, 1996); *Elements of Software Reliability Engineering* (Tshinghua University Press, Beijing, 1995, in Chinese). He serves on the editorial board of the international journal *Fuzzy Sets and Systems* and is the editor of the *Kluwer International Series on Asian Studies in Computer and Information Science* (http://www.wkap.nl/series.htm/ASIS). He is one of the 1998-year recipients of the Outstanding Youth Investigator Award of China. His main research interests include computer systems reliability and intelligent control.

Lin Cai is now with a Beijing-based company. She received her B.S. degree in control engineering from Beijing University of Aeronautics and Astronautics in 1998.

Wei-Dong Wang is now with a Beijing-based company. He received his B.S. degree in control engineering from Beijing University of Aeronautics and Astronautics in 1999.

Zhou-Yi Yu received his B.S. degree in control engineering from Beijing University of Aeronautics and Astronautics in 1999. He is now working towards his Ph.D. degree in control engineering at BUAA.

David Zhang (M'92–SM'95) graduated in computer science from Peking University in 1974 and received his M.Sc. and Ph.D. degrees in computer science and engineering from Harbin Institute of Technology (HIT) in 1983 and 1985, respectively. From 1986 to 1988, he was a postdoctoral fellow at Tsinghua University and became an Associate Professor at Academia Sinica, Beijing, China. In 1988, he joined the University of Windsor, Ontario, Canada, as a visiting research fellow in electrical engineering. In 1995, he received his second Ph.D. in electrical and computer engineering at University of Waterloo, Ontario, Canada. Then, he was an Associate Professor in City University of Hong Kong. Currently, he is a professor in Hong Kong Polytechnic University, as well as a Founder and Director of both Biometrics Technology Centres supported by UGC/CRC, Hong Kong Government, and National Natural Scientific Foundation (NSFC) of China, respectively. He is also a Guest Professor and Supervisor of PhD in HIT. He is a Founder and Editor-in-Chief, *International Journal of Image and Graphics*, and an Associate Editor, *Pattern Recognition* and *International Journal of Pattern Recognition and Artificial Intelligence*. He has been listed in 1999 Marquis Who's Who in the World (16th). His research interests include automated biometrics-based identification, neural systems and applications, and parallel computing for image processing & pattern recognition. So far, he has published over 150 papers including four books around his research areas and given over five keynotes or invited talks or tutorial lectures, as well as served as program/organizing committee members and session co-chairs at international conferences in recent years. In addition, he has developed some applied systems and received several recognisable project awards. He is a Senior Member in IEEE.