Chapter

17

Neural Networks for Software Reliability Engineering

Nachimuthu Karunanithi
Bellcore

Yashwant K. Malaiya
Colorado State University

17.1 Introduction

Artificial neural networks (or simply *neural networks*) are a computational metaphor inspired by studies of the brain and nervous systems in biological organisms. They are highly idealized mathematical models of the essence of our present understanding of how simple nervous systems work. Neural networks operate on the principle of learning from examples; no model is specified a priori. Neural networks are likened to nonparametric models in the statistical literature. Recent development in neural networks has shown that they can be applied in a variety of problem domains. For example, neural networks are used to solve complex nonlinear function approximation problems, difficult linearly inseparable pattern classification problems, speech recognition and control problems, and complex time-series modeling problems. Though the neural network technology has been applied in various fields, its utility in software engineering has not been completely explored.

The purpose of this chapter is to introduce how this newly emerging technology can be used in software reliability engineering applications. In particular, we demonstrate the utility of neural networks models for solving two problems in the area of software reliability engineering. In the first example, we illustrate how a neural network can be used as a general reliability growth model. We validate the utility of this general model by learning to predict the cumulative faults for several software projects. Our results suggest that neural network models can provide a better predictive accuracy than some of the analytic models. In the second example, we illustrate how a neural network can be used as a classifier to identify fault-prone (or change-prone) software modules from their static complexity metrics. We demonstrate the applicability of this approach using metrics data from a Medical Imaging System software. Our results suggest that the neural network classifier may provide an edge over simple classifiers in certain categories.

17.2 Neural Networks

Neural networks are computational systems based on mathematical idealization of our present understanding of biological nervous systems. However, the present neural network models do not take into consideration all scientific knowledge about biological nervous systems; rather they model only a few rudimentary characteristics that can be expressed in simple mathematical equations. In general, neural networks can be characterized in terms of the following three entities [Karu92d]:

- Models of *neurons,* i.e., characteristics of the processing unit
- Models of *interconnection structure,* i.e., the topology of the architecture and the strength of the connections that encode the knowledge
- A *learning algorithm,* i.e., the steps involved in adjusting connection weights

The neural network research community has introduced a variety of models for these entities. Consequently, there exists a variety of neural network models and learning algorithms. We review here some of the relevant neural network models and their characteristics.

17.2.1 Processing unit

There exist many mathematical idealizations for the processing unit (or artificial neuron). A typical processing unit is described in terms of: (1) a distinct set of *fan-in connections,* which specifies a set of weighted connections through which the unit receives input either from other units in the network or the external world; (2) a distinct set of *fan-out connections,* which specifies a set of weighted connections through which the unit sends its output either to other units in the network or to the external world; (3) a *fan-in function,* which integrates inputs to produce a net input to the processing unit; and (4) an *activation function,* which acts on the current net input to produce an output. A typical processing unit is illustrated in Fig. 17.1.

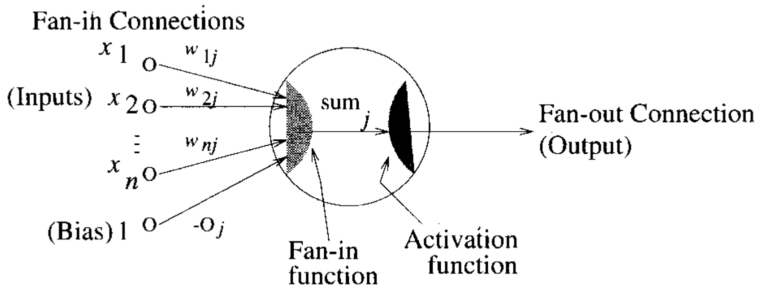

Figure 17.1 A typical processing unit.

Fan-in function. The purpose of the fan-in function is to integrate inputs received from other units or the external world, and then produce a net input to the processing unit. One of the most widely used fan-in functions is the dot-product function. Assume that w_{ij} is the fan-in weight vector of unit j from unit i, sum$_j$ is the net input to the unit, and x_i is the input from unit i, $1 \le i \le n$. Then the dot-product function is expressed as

$$\text{sum}_j = \sum_{i=1}^{n} w_{ij}x_i - \theta_j \tag{17.1}$$

where $-\theta_j$ represents the threshold, which is usually created by an adjustable weight from a special unit called *bias* unit whose output is always 1.

Activation function. The purpose of the activation function is to produce an activation value a_j for the unit j from the net input sum$_j$. The general form of an activation function is expressed as

$$a_j = F(\text{sum}_j) \tag{17.2}$$

The activation function can be linear or nonlinear. However, neural network models often employ a nonlinear activation function. There are several nonlinear functions that can be used in a neural network. We give a brief overview of some of the commonly used functions here.

The simplest function is the *linear threshold function.* The output response of the linear threshold function is defined as

$$F(\text{sum}_j) = \begin{cases} 0 & \text{if sum}_j \le 0 \\ C \cdot \text{sum}_j & \text{if } 0 < \text{sum}_j < L \\ 1 & \text{if sum}_j \ge L \end{cases} \tag{17.3}$$

where $C > 0$ is the slope constant and L, the threshold. Figure 17.2*a* depicts a linear threshold function with $L = 1$.

Another simplest nonlinear function is the *hard-limiting threshold function.* The hard-limiting threshold function is given by

$$F(\text{sum}_j) = \begin{cases} 1 & \text{if sum}_j \ge 0 \\ 0 & \text{otherwise} \end{cases} \tag{17.4}$$

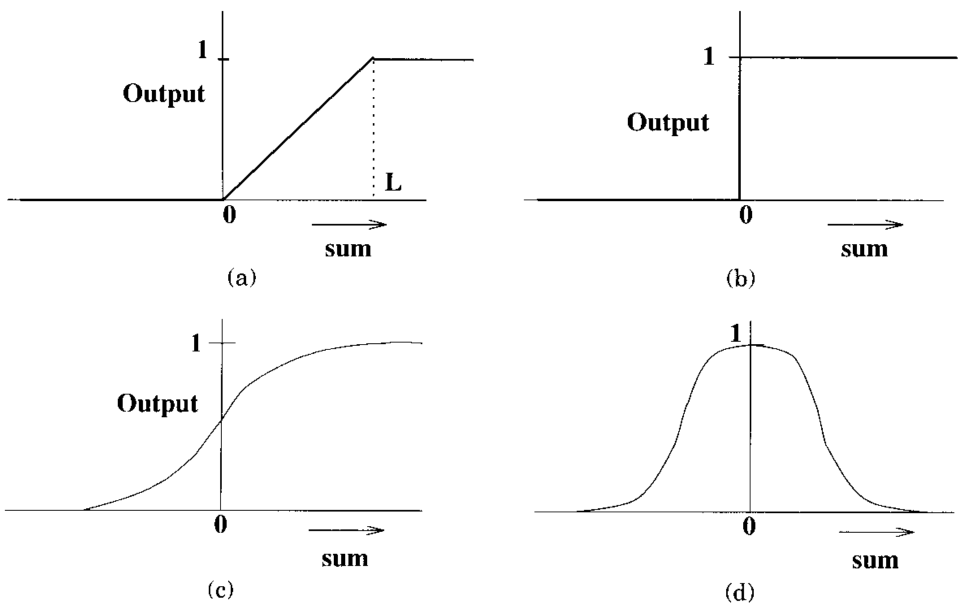

Figure 17.2 Output of typical activation functions.

The output behavior of this function is given in Fig. 17.2*b*.

The most widely used nonlinear activation function is the *logistic function*. The logistic function is defined as

$$F(\text{sum}_j) = 1/(1 + e^{-\text{sum}_j/T}) \tag{17.5}$$

where T is the so-called temperature coefficient, which controls the slope of the sigmoidal characteristics. The inverse of T is usually known as the *gain factor* of the activation function. The response of a sigmoid function is shown in Fig. 17.2*c*. One of the interesting features of the logistic function is that it produces a continuous output. A processing unit that incorporates a logistic activation function is often referred to as *sigmoidal unit*. The output of the sigmoidal unit is bounded between 0.0 and 1.0.

Another widely used nonlinear activation function is the *gaussian function*. A gaussian function with unit normalization is defined as

$$F_g(\text{sum}_j) = e^{[-(\text{sum}_j/\sigma)^2]} \tag{17.6}$$

where σ is a parameter which can be used to control the width of the bell-shaped response curve. The output of a typical gaussian function is illustrated in Fig. 17.2*d*.

17.2.2 Architecture

We can characterize the architecture of a neural network in terms of two attributes: (1) number of layers in the network and (2) the type of network topology used.

Number of layers. The units that act as an interface between the external input and the network constitute the *input layer.* Typically, the input layer is not involved in any useful computation; rather, each unit in the input layer acts as a distribution point for the external inputs. The units that output the network's response to the external world constitute the *output layer.* The layers that do not have direct communication with the external world are called *hidden layers.* The number of layers in a network may vary from a lower limit of two (one input and one output layer) to any higher positive integer. In the neural network literature, some authors do not consider the input units as a layer. Thus, a network is often called a *single-layer network* if it does not have a hidden layer. On the other hand, networks with one or more hidden layers are called *multilayer networks.* Hidden layers of a multilayer network allow the network to develop its own internal representation of the problem. Each hidden layer produces an internal representation from the input it receives. The number of hidden units in a hidden layer represents the dimensionality of the internal representation space. Thus, by controlling the number of hidden units in a hidden layer, we can expand or shrink the internal representation of a problem.

Type of connectivity. Based on the connectivity and the direction in which links propagate activation values, we can classify a multilayer network into one of two well-known classes of models: *feed-forward networks* and *recurrent networks.*

If a network employs only forward-feeding connections then it is called a *feed-forward network.* A feed-forward network can be either single layer or multilayer. Historically, a single-layer network with a hard-limiting threshold output is known as a *perceptron.* However, a single-layer network with sigmoidal unit is also commonly referred to as a *perceptron.* Perceptrons have limited applicability because they are capable of solving only linearly separable classification problems. Multilayer networks, on the other hand, are highly suited for problems that require nonlinear function mappings and linearly inseparable classification boundaries. A three-layer feed-forward network is shown in Fig. 17.3*a.* This feed-forward network has one unit each in the input and output layers, and three (hidden) units in the hidden layer. An additional unit, called the *bias unit,* is always used to supply threshold values (using adjustable links) to all hidden and output units in the network. For simplicity we have not included the bias unit in Fig. 17.3*a.* For a given problem, the number of units required for the input and the output layers are dictated by the dimensionality of the problem space. For example, if a problem has two independent variables and a dependent variable, then the network will have two input units and one output unit. However, for a given problem, both the number of hidden layers needed in the network and the number of units needed within each hidden layer of the network are dictated by its complexity and size.

In addition to feed-forward connections, we can also use recurrent connections to feed the previous time-step output values (i.e., the states of the units corresponding to the previous input pattern or the time step) of units either to the units in preceding layer(s) or back to the units themselves. Multilayer networks that employ feedback connections are referred to as *recurrent networks.* Based on how recurrent connections are established, we can classify a recurrent network into one of three categories: (1) *simple recurrent networks* proposed by Elman [Elma90], (2) *semirecurrent networks* proposed by Jordan [Jord86], and (3) *fully recurrent networks* [Will89]. In a fully recurrent network every unit receives input from all other the units in the network. In the simple recurrent network (or Elman network), the recurrent connections originate from the hidden units. In a standard semirecurrent network (also known as Jordan network) there is one-to-one feedback from the output units to "state units," and all these feedback links have a fixed weight of 1.0. Apart from the output feedback, each state unit receives self-feedback through a learnable link. Thus, the activation of a state unit is a function of the output of the network as well as its own activation at the previous time step. The state units, like the input units, propagate their activations in the forward direction to all the hidden units. For illustration, we show a modified Jordan-style recurrent network without self-feedback links of the state units in Figure 17.3*b*. Our experience shows that both feed-forward networks and the modified Jordan-style recurrent network are useful in software reliability engineering applications.

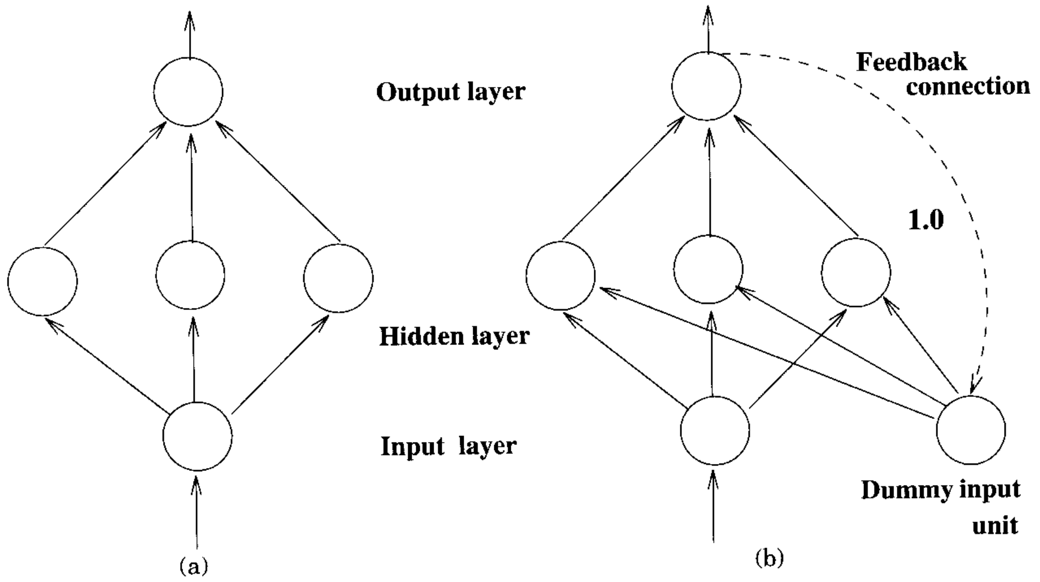

Figure 17.3 (*a*) A standard feed-forward network and (*b*) a modified Jordan network.

17.2.3 Learning algorithms

To solve a problem using a neural network, the network must first be taught solutions using a set of typical instances of input-output pairs known as the *training set.* The procedure by which the network is taught is known as the *learning algorithm.* During the training (or learning) phase, the strength of the interconnection links of the network are adjusted to reduce the residual error resulting from the training set. A variety of algorithms exist for training multilayer networks. Among them, the *back-propagation algorithm* is the most widely used. The back-propagation algorithm was proposed independently by several researchers [Werb74, Park82, Rume86]. In the mid-eighties, Rumelhart, Hinton, and Williams [Rume86] popularized the back-propagation algorithm by training multilayer networks to solve several interesting problems. Thus, the back-propagation algorithm has become one of the main driving forces behind the recent surge both in multilayer neural networks research and development of several variants of multilayer neural network training algorithms. A brief review of the back-propagation algorithm can be a valuable first step toward understanding other training algorithms and neural network models. Hence, we provide an overview of the back-propagation algorithm in the next section.

17.2.4 Back-propagation learning algorithm

The back-propagation learning algorithm is a class of supervised learning* algorithms in which the network weights are iteratively adapted using errors propagated back from the output layer. For the purpose of illustration, let us consider a simple three-layer feed-forward network constructed using sigmoidal units and trace how the back-propagation algorithm works.

Algorithm. (1) To start with, initialize the network weights with a set of random values. This step is called the *initialization phase.* Typically, a set of random values drawn from a small interval (for example, between −1.0 and 1.0) is used. However, one can also initialize the network weights with a set of values from a known distribution. (2) Next, in the *weight adjustment phase,* adjust the network weights incrementally over several iterations. In each iteration, adjust the weights in the direction of steepest decreasing gradient of the error surface (i.e., the surface formed by the sum of the square of the error between the desired output and the actual output for all patterns in the training

* A training algorithm is "supervised learning" if it requires actual outputs for each input pattern in the training set.

set). This iterative adjustment continues until either a minimum is reached in which the error is less than a prespecified tolerance limit, or until a set number of iterations has been reached. During each iteration, the sum of squared errors is calculated as follows:

$$E = \frac{1}{2} \sum_{p=1}^{P} \sum_{j=1}^{M} (y_j^p - o_j^p)^2 \tag{17.7}$$

where y^p is the desired output for the input vector in_p, o^p is the actual output of the network, M is the number of output units, and P is the number of patterns in the training set. This error is propagated back through the network to adjust weights using a gradient descent procedure. The gradient descent procedure changes the weights by an amount proportional to the partial derivative of the error with respect to each weight w_{ij}. Thus, the change in the weight of a link from unit i to unit j at the tth iteration is given by

$$\Delta w_{ij}(t) = \eta \cdot \delta_j \cdot in_i + \alpha \cdot \Delta w_{ij}(t-1) \tag{17.8}$$

where η, a proportionality constant called the *learning rate,* α, the momentum term, $\Delta w_{ij}(t-1)$, weight change during the previous iteration, in_i, the output activation of the unit i, and δ_j, partial derivative of the error with respect to the net input (sum_j) to the unit j. The error derivative δ_j for the jth unit in the output layer is given by

$$\delta_j = o_j(1 - o_j)(y_j - o_j) \tag{17.9}$$

and for the kth unit in the hidden layer is given by

$$\delta_k = h_k(1 - h_k) \sum_{j=1}^{M} (\delta_j \cdot w_{kj}) \tag{17.10}$$

where h_k is its activation.

Though the back-propagation algorithm can be used to train any multilayer network, one must be aware of several practical issues that could affect its efficiency. Since the back-propagation algorithm is a gradient descent optimization procedure, it is vulnerable to the problem of premature convergence. Premature convergence occurs whenever the algorithm gets stuck in a local minimum and the value of E is still higher than the allowed tolerance limit. Though several solutions have been proposed to improve/avoid premature convergence of the back-propagation algorithm [Jaco88], none of the solutions have been proved to work consistently across all error surfaces.

The second issue is specifying an optimum network architecture for a given problem. A network architecture is considered optimum if it has the minimum number of hidden units (and hence the minimum number of interconnection links) with which it can successfully learn the training set. If the network architecture is too small, then it may not be able to learn the entire training set. On the other hand, if the network architecture is too large, it may have too many degrees of freedom (weight or parameters) to learn the training set. Thus, if the network has too many weights, then it will memorize the training set rather than generalize it. Furthermore, an inappropriate network may consume an unreasonable amount of simulation resources. The back-propagation algorithm is applied to train the network with the assumption that the architecture of the network is specified either ahead of time by the user, or experimentally determined by trial and error. In the trial-and-error approach, the user may waste time experimenting with different architectures to find an appropriate architecture.

To address the limitations of the standard back-propagation algorithm, Scott Fahlman and colleagues [Fahl90] developed an efficient constructive training algorithm known as *cascade-correlation learning architecture* for feed-forward networks. We demonstrate the use of neural networks for software reliability applications using this algorithm. To understand how this algorithm works, we provide an overview of the algorithm in the following section.

17.2.5 Cascade-correlation learning architecture

The cascade-correlation algorithm combines two important ideas in its learning method: the *cascade architecture,* to add hidden units one at a time, and a *learning algorithm* (typically, Fahlman's "quickpropagation" [Fahl88]) to create, train, and install new hidden units. The cascade-correlation algorithm in effect grows a neural network. In the first stage of learning, the cascade-correlation uses no hidden units; this means that initially the net applies a simple perceptron-type network to learn as many training patterns as possible. When the algorithm can no longer reduce the error, a potential candidate hidden unit, which is separately optimized to maximize the correlation between its outputs and the residual error of the network over the entire training set, is added. When hidden units are trained, the weights connecting the output units are kept unchanged. After the hidden unit is connected to the output, training updates all the weights that directly go to the output layer. Once installed, each hidden unit becomes a new hidden layer in the network, and its incoming weights remain frozen for the rest of the training period. When subsequent hidden units are added, the outputs of the previously added hidden units become as additional inputs to the new hidden unit. This dynamic expansion of network architecture continues until the problem is solved.

The cascade-correlation algorithm consists of the following steps.

1. *Initialize the network.* Create a network architecture consisting of only input and output layers. Establish links from the input layer to the output layer and initialize them with random values.
2. *Train output layer.* Adjust weights feeding the output units. If the learning is complete (i.e., the error is below a preset limit) then stop; else if the error has not been reduced significantly for a certain number of consecutive epochs, or the maximum number of epochs allowed has been reached, then go to the next step.
3. *Initialize a candidate unit.* Create and initialize a candidate unit with random weights from all input units and all preexisting hidden units.
4. *Train the candidate unit.* Adjust candidate unit's weight to maximize the correlation between activation, v_c, of the candidate unit c and the residual error e_j at each output unit j.
   a. Compute correlation. The correlation is computed in terms of covariance,
   $$\text{cor}_c = \sum_{j=1}^{M} \left| \sum_{p=1}^{P} (v_c^p - \bar{V})(e_j^p - \bar{E}_j) \right| \tag{17.11}$$
   where $\bar{V} = \Sigma_{p=1}^{P} v_c^p / P$ and $\bar{E}_j = \Sigma_{p=1}^{P} e_j^p / P$
   b. Update candidate unit's weights using gradient ascent to maximize cor_c,
   $$\Delta w_{ci} = \alpha \cdot \frac{\partial \text{cor}_c}{\partial w_{ci}} \cdot i_i \tag{17.12}$$
   c. If the correlation at the current epoch is not significantly better than the previous epoch, then go to step 5; otherwise go back to step 4.*a*.
5. *Install the candidate unit.* Install the candidate unit as a hidden unit by establishing weights to output units and initializing them with the negative of the correlation values. Now freeze the incoming weights of the candidate unit and go to step 2.

The cascade-correlation learning algorithm has many advantages over the standard back-propagation algorithm. Three of the major advantages are: (1) the user need not specify the architecture; rather it is evolved automatically; (2) its learning speed is one or two orders of magnitude faster than the standard back-propagation algorithm; and (3) it is highly consistent in converging during training. Because of these advantages and our experience with the back-propagation algorithm, we use the cascade-correlation algorithm for all example demonstrations in this chapter.

17.3 Application of Neural Networks for Software Reliability

The problem of developing reliable software at a low cost still remains as an open challenge. To develop a reliable software system, we must address several issues. These include specifications for reliable software, reliable development methodologies, testing methods for reliability, reliability growth prediction modeling, and accurate estimation of reliability. Two of the problem areas in which the neural network is applicable are developing a general-purpose reliability growth model, and identifying change/fault-prone software modules early during the development cycle. This section reviews some of the limitations of the traditional modeling approaches used to solve these problems and argues that we can try the neural network approach as an alternative.

17.3.1 Dynamic reliability growth modeling

In current software-reliability research, one of the concerns is how to develop general prediction models. Existing models typically rely on a priori assumptions about the nature of failures and the probability of individual failures occurring. Furthermore, these models, referred to as *parametric models,* attempt to capture in two or three explicit parameters all the assumptions made about the software development process and environment. Because all of these assumptions must be made before the project begins, and because many projects are unique, the best that one can hope for is statistical techniques that predict failure on the basis of failure data from similar project histories. Though there is evidence to suggest that certain analytic models are better suited to certain types of software projects than other models, the issue of finding a common model for all possible software projects is yet to be solved.

Selection of a particular model is very important in software reliability growth prediction because both the release date and the resource allocation decision can be affected by the accuracy of predictions. Several solutions have been proposed to address the issue of model selection. For example, in Chap. 4 we suggest two alternatives: (1) try several software reliability growth models and select the one that gives highest confidence and (2) use a recalibration method to compensate for the bias of a model. The second alternative can be used either alone or in combination with the first solution. Alternatively, in Chap. 7 we propose a linear (or a weighted linear) combination of prediction results from models with opposite bias. Li and Malaiya [Li93] show that an adaptive prediction combined with preprocessing can enhance the predictive capability of the analytic models. All these solutions can broadly be termed *postprocessing* methods. Nevertheless, the issue of generalization of prediction models still remains open. An alternate approach is to use an adaptive model-building system that can develop its own model of the failure process from the actual characteristics of the given data set. In this chapter we demonstrate that such a general-purpose reliability growth model can be developed using the neural network approach.

17.3.2 Identifying fault-prone modules

Another concern in software reliability engineering is to identify potentially fault-/change-prone modules early during the development cycle. This concern is motivated by the developers' need to improve the overall reliability of the product by allocating more test efforts to potentially troublesome modules. Intrinsic complexity of the program texts, measured in terms of static complexity metrics, is often used as an indicator of troublesomeness of program modules. By controlling the complexity of software program modules during development, we can produce software systems that are easier to maintain and enhance (because simple modules are easier to understand). As seen in Chap. 12, static complexity metrics are measured from the passive program texts early during the development cycle and can be used as valuable indicators for allocating resources in future development efforts.

Several statistical regression and classification models have been suggested to predict the fault-proneness of software modules [Craw85, Shen85, Rodr87, Muns92, Lyu95b]. However, existing statistical models make simple assumptions that are often violated in practical measures. Furthermore, there are numerous metrics that are either redundant or that have some linear (or nonlinear) dependence among other metrics. High redundancy in the metric space and multiply-related metric dimensions may often result in unreliable predictive models. In this chapter, we demonstrate that neural networks can also be used as a classifier for identifying fault-prone software modules.

17.4 Software Reliability Growth Modeling

In this section we illustrate how neural networks can be used for predicting software reliability growth process. As pointed out earlier, the predictive capability of a neural network can be affected by which neural network model is used to model the failure data, how the input and output variables are represented to it, the order in which the input and output values are presented during training, and the complexity of the network. We address these issues in this section, and empirically show that neural networks can give accurate predictions across different software projects. Furthermore, we also perform an analysis to show that neural networks are capable of adapting their complexity to match the complexity of the training data set.

To illustrate how neural networks can be used as a prediction system, the following definitions are introduced [Karu91, Karu92a, Karu92b].

Definition 1

Sequential prediction: Given a sequence of inputs $((i_1, \ldots, i_t) \in I)$ and a corresponding sequence of outputs $((o_1, \ldots, o_t) \in O)$ up to the present time t and an input $(i_{t+d} \in I)$ belonging to a future instant $t+d$, predict the output $(o_{t+d} \in O)$.

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

We can represent a neural network as a mapping $\mathcal{NN}: I \mapsto O$ where I is an n dimensional input space and O is the corresponding M dimensional output space. Generally this mapping is accomplished using a multilayer network. The training procedure is a mapping operation $\mathcal{T}$: $I_k \mapsto O_k$ where $(I_k, O_k) = \{(i, o) \mid i \in I \textit{ and } o \in O\}$ is a subset of k input-output pairs sampled from the actual problem space (I, O). The function $\mathcal{T}$ represents an instance of $\mathcal{NN}$ that has learned to compute (or approximate) the actual mapping of the problem.

Definition 2

Neural network mapping: Sequential prediction can be formulated as a neural network mapping

$$\mathcal{P}: ((I_t, O_t), i_{t+d}) \mapsto o_{t+d}$$

in which (I_t, O_t) represents a sequence of t consecutive samples used for training and o_{t+d} the predicted output corresponding to a future input i_{t+d}.

Definition 3

Neural network software reliability growth model: Software reliability growth prediction can be expressed in terms of a neural network mapping as

$$\mathcal{P}: \{(T_k, F_k), t_{k+h}\} \mapsto \mu_{k+h}$$

where T_k is a sequence of cumulative execution time $(t_1, \ldots, t_k)$, F_k is the corresponding observed accumulated failures $(\mu_1, \ldots, \mu_k)$ up to the present time t_k used to train the network, $t_{k+h} = t_k + \Delta$ is the cumulative execution time at the end of a future test session $k+h$ and μ_{k+h} is the prediction of the network.

In the above definition $\Delta = \Sigma_{j=k+1}^{k+h} \Delta_j$ represents the cumulative execution time of h consecutive future test sessions. Note that each test session Δ_j can be either a fixed duration or a random interval.

17.4.1 Training regimes

A neural network's predictive ability can be affected by what it learns and in which sequence it learns. The notion of a training regime is introduced to distinguish the order in which data are presented during training. Two training regimes can be used in software reliability prediction: *generalization training* and *prediction training.* Figure 17.4 illustrates these training regimes.

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 and the dependent variables.

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

If we combine these two training regimes with the feed-forward network (FFN) and the modified Jordan network (JN), we get four neural network prediction models. Let us denote these models as *FFN-generalization, FFN-prediction, JN-generalization* and *JN-prediction.*

17.4.2 Data representation issue

The issue of data representation is concerned with the format used to represent the input-output variables of the problem to the neural net-

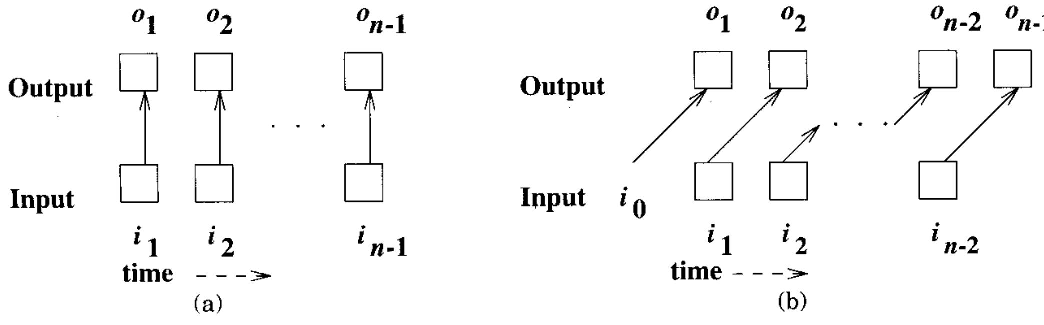

Figure 17.4 Two network-training regimes: (*a*) generalization training and (*b*) prediction training.

work. If a sigmoidal unit is used as the output unit, then its output response will be bounded between 0.0 and 1.0. Since the output variable of the software reliability growth model may vary over a large numerical value, it is necessary to scale the output variable to a range that conforms within the operational range of the output units. A simple representation is to use a direct scaling, which scales cumulative faults from 0.0 to 1.0. On the other hand, the input variables, in theory, need not be scaled to this range because the weights feeding the hidden units from the input layer can scale them appropriately. But, for practical reasons, it is a reasonable heuristic to scale also the input variables to an appropriate range. For the purpose of our illustration, we scaled both cumulative faults and cumulative execution time from 0.1 to 0.9 because our experience shows that (1) the network is less accurate in discriminating inputs whose scaled values are close to the boundary values 1.0 or 0.0, and (2) the sigmoidal unit's error derivative, which affects the rate of weight adaptation during training, becomes inconsequential when its output is close to 1.0 or 0.0. To scale the data, however, it is necessary to guess the appropriate maximum values for both the cumulative faults and the cumulative execution time [Karu92c].

17.4.3 A prediction experiment

To illustrate the predictive accuracy of neural network models, five well-known analytic software reliability growth models and failure history data sets from several different software projects were considered. These models include the *exponential* model proposed by Moranda [Mora75b], the *logarithmic* model proposed by Musa and Okumoto [Musa87], the *delayed S-shaped* model proposed by Yamada et al. [Yama83], the *inverse-polynomial* model proposed by Littlewood and Verrall [Litt74], and the *power* model proposed by Crow [Crow74]. All these models have two parameters and are nonhomogeneous Poisson process except for the *inverse-polynomial* model. Details of these models appear in Chap. 3.

The data sets used for illustration were collected from several different software systems, including 14 data sets (see the Data Disk). These data sets represent the observed failure history of those systems. Each data point within a set consists of two observations: the cumulative execution time and the corresponding accumulated number of defects disclosed.

17.4.3.1 Measures for evaluating predictability. In order to compare the predictive accuracy of different models, we have to use some meaningful quantitative measures. The software reliability research community uses a variety of metrics for comparison of models. We review three distinct approaches that are very common in software reliability research [Mala90a, Mala90b] and point out which one is relevant for our comparison.

Let the data be grouped into n points (t_i, D_i), $i = 1$ to n, where D_i is the cumulative number of defects found at time t_i, and t_i is the accumulated execution time (i.e., the time spent for testing the software) in disclosing D_i defects. Let μ_i be the projected number of defects at time t_i by a model.

1. *Goodness-of-fit.* In this approach, first a curve corresponding to a selected model is fitted to all the data points $\{t_i, \mu_i\}$, $i = 0, 1, \ldots, n$; then the deviation between the observed and the fitted values is evaluated by using the chi-square test or Kolmogorov-Smirnov test.
2. *Next-step predictability.* In this approach a partial data set corresponding to, say, $\{t_i, D_i\}$, $i = 1, \ldots, (l - 1)$ is used to predict the value of μ_l (cumulative number of defects after lth test interval). The predicted and the observed values of cumulative defects are then compared.
3. *Variable-term predictability.* In actual practice, the need to predict the behavior at a distant future of the test phase using present failure history is very important. This approach, used by Musa et al. [Musa87], makes projections of the final value of μ_n using a partial data set $\{t_i, D_i\}$, $i = 0, \ldots, l$, where $l < n$. The percentage prediction error in these projections can then be plotted against time.

A *two-component predictability measure* consisting of *average error* (AE) and *average bias* (AB) was used by Malaiya et al. [Mala90a, Mala92] to compare predictive capabilities of parametric models at different fault-density ranges. These measures are as follows. For a specific model m, let μ_{ij}^m be the projected total number of the defects to be detected at a future time t_j, where $j = i + d$. Using the projected value at each point i, we can calculate the model's prediction error $(\mu_{ij}^m - D_j)/D_j$, $i = 1$ to $n - 1$. Then predictability measures for a given data set s are given by

$$AE_s^m = \frac{1}{n-1} \sum_{i=1}^{n-1} \left| \frac{\mu_{ij}^m - D_j}{D_j} \right| \tag{17.13}$$

$$AB_s^m = \frac{1}{n-1} \sum_{i=1}^{n-1} \frac{\mu_{ij}^m - D_j}{D_j} \tag{17.14}$$

Two extreme prediction horizons of interest are: $j = i + d = i + 1$, the *next-step prediction,* and $j = i + d = n$, the *end-point prediction.* AE is a measure of how well a model predicts throughout the test phase, and

AB is the general bias of the model. AB can be either positive or negative depending on whether the model is prone to overestimation or underestimation. Note that the above AE measure can be used to compare the predictive accuracy of models within a single data set only. However, what we need is a normalized AE measure (normalized within each data set) with which competing models can be ranked across different software projects.

Let m $m_{\max}$ be the competing model that has the maximum average error measure for the data set s. Let $AE_s^{m_{\max}}$ be its average error measure. Then the normalized AE measure is given by

$$NAE_s^m = \frac{AE_s^m}{AE_s^{m_{\max}}} \tag{17.15}$$

Thus $NAE_s^m = 1.0$ when $m = m_{\max}$ (i.e, for the model which has the highest prediction error) and $0.0 < NAE_s^m < 1.0$ for $m \neq m_{\max}$. Note that more than one model may have the same value if their AE_s^m values are equal.

Let γ_s be the weighting (importance) factor of a data set s. Now an overall rank metric of the model m can be calculated as

$$R_m = \sum_{s=1}^{ND} \gamma_s \cdot NAE_s^m \tag{17.16}$$

where ND is the number of data sets used in comparing models. If we give equal weight to all data sets, then the above rank metric reduces to a simple sum

$$R_m = \sum_{s=1}^{ND} NAE_s^m \tag{17.17}$$

We can apply these metrics for both end-point predictions and next-step predictions.

17.4.3.2 Prediction results. We trained neural networks with the execution time as the input and the observed fault count as the target output. The training ensemble at time t_i consists of the complete failure history of the system since $t_{i=0}$ (i.e., since the beginning of testing). Any prediction of a neural network without proper training is equivalent to the network making a random guess. So we took care to see that the neural network was trained with at least some initial portion of the failure history. In all the experiments, we set the minimum size of the training set so it would contain at least the first three observed data points. We then increased the training set in steps of one point from this minimum set to all but the last point in the data set. At the end of each training, the network was fed with future inputs to measure its prediction of the total number of defects.

Table 17.1 summarizes the predictive accuracy of models in terms of the rank metric R_m defined earlier. This metric provides simple summary statistics. In order to find their relative accuracies, we also calculated the average and the standard deviation of R_m. The R_m values for end-point predictions of neural networks are well below the overall average of 6.10. This suggests that neural networks are better suited for end-point predictions than are analytic models. Jordan nets have R_m values 2.09 and 2.23, which are one standard deviation below that of the overall average of 6.10. The results suggest that the Jordan net models are better at end-point predictions than are all other models. One reason the Jordan nets exhibit such a low end-point prediction error may be that they are able to capture the correlation among points that are in the immediate past. In comparison, the analytic models have R_m values that are well above the average. This suggests that analytic models are either too pessimistic or too optimistic. Numbers in the "Rank" column represent the rank of these models based on their overall prediction accuracy.

Note that, except for the JN-prediction model, all models seem to have almost similar next-step prediction accuracy. This observation is further supported by the fact that all other models have R_m values that are bounded between one standard deviation above and below the average. The JN-prediction model has $R_m = 8.00$, which is one standard deviation below the overall average of 9.82, and suggests that this model has better next-step predictive accuracy than the other models compared.

In order to check whether the above observations have statistical significance, we performed further analysis using the analysis of variance (ANOVA) approach. In this approach we viewed the AE measures as outcomes of randomized block experiments in which the projects (or

TABLE 17.1 Summary of Normalized AE Measure

	End-point predictions		Next-step predictions	
Model	R_m	Rank	R_m	Rank
FFN-generalization	2.86	4	11.79	9
FFN-prediction	2.64	3	10.09	7
JN-generalization	2.09	1	9.81	5
JN-prediction	2.23	2	8.00	1
Logarithmic	6.29	5	8.78	2
Inverse polynomial	7.60	6	9.54	4
Exponential	8.20	7	8.96	3
Power	11.39	8	11.43	8
Delayed S-shape	11.59	9	9.95	6
Average	6.10		9.82	
Std	3.85		1.21	

data sets) were randomly selected and software reliability growth prediction models as "treatments" applied to each of the projects. Note that this approach accounts for the data set peculiarities that could affect performance of these competing models.

The overall average of AE values (i.e., the average of AE measures over all 14 data sets) are shown in Table 17.2. These values represent the predictive accuracy of models across different projects. The resulting F-statistics for projects and models are highly significant at the 1 percent level, and show that models have significantly different predictive accuracy. Since the models have significant F-statistics, we can use the least significant difference (LSD) procedure to differentiate among them in terms of their overall AE measure. Under LSD, two competing models are considered *significantly different* if their means in Table 17.2 differ by an amount equal to or greater than the value given by the LSD procedure. On the other hand, if the difference is less than the value given by the LSD procedure, then we can interpret that two models have similar predictive accuracy across all projects. Thus if a model X has an AE which is significantly lower than that of another model Y, then we can interpret, using > to denote significantly better, that {X} > {Y}.

The resulting LSD for the end-point prediction is 4.52 ($T_{5\%,104} = 1.98$). Thus, based on the averages in Table 17.2 we can conclude that {FFN-generalization, FFN-prediction, JN-generalization, JN-prediction} > {logarithmic, inv. polynomial, exponential} > {power, delayed S-shape}. Thus our analysis suggests that the neural network models are significantly better than analytic models in predicting end points.

The corresponding LSD value for the next-step prediction is 1.39 ($T_{5\%,104} = 1.98$). Based on this LSD value and the overall average of AE in Table 17.2, we can interpret that {JN-prediction, logarithmic, exponential} > {FFN-generalization, power}. Since the remaining four models do not show any significant difference in the next-step prediction accuracy, we can conclude that neither of these two competing approaches (neural network models versus analytic models) taken

TABLE 17.2 Overall Average of AE Measure

Model	End-point predictions	Next-step predictions
FFN-generalization	6.67	6.62
FFN-prediction	6.12	5.38
JN-generalization	4.75	5.47
JN-prediction	4.94	4.46
Logarithmic	14.23	5.04
Inverse polynomial	17.93	5.64
Exponential	18.45	5.19
Power	26.42	7.44
Delayed S-shape	25.61	5.80

alone have a distinct advantage. However, it should be noted that the JN-prediction model has a higher next-step prediction accuracy than all other models.

Since we used two different network architectures, we performed an additional ANOVA analysis to find out whether there is any significant difference among neural network models. The resulting LSD values (with a critical value of $T = 2.02$) are 1.16 for end-point predictions and 1.11 for next-step predictions, respectively. The LSD of 1.16 and the overall averages in Table 17.2 suggest that the Jordan network models are significantly better end-point predictors than the feed-forward network models. Furthermore, based on the LSD value of 1.11 for next-step predictions we conclude that the JN-prediction model is a better next-step predictor than the other neural network models. Thus, the Jordan network models are better predictors than the feed-forward network models.

In summary, our statistical analyses show that (1) on the average, neural networks models have better end-point prediction accuracy than analytic models, and (2) the type of neural network architecture can influence the predictive accuracy.

17.4.4 Analysis of neural network models

So far we have seen how we can use neural networks constructed using the cascade-correlation algorithm to accurately predict cumulative faults. As pointed out earlier, the number of hidden units added to the final network by the cascade-correlation algorithm may vary as the size and the complexity of the training set are changed. This is especially true in our reliability growth modeling problem because the number of points in the training set increased according to the time horizon that we took into consideration. Also, the number of hidden units varied from one data set to another because of their inherent peculiarity. Thus, most of the time the networks were constructed with zero, one, or two hidden units and occasionally with three or four hidden units. Thus, the models developed by the neural network approach are more complex than most analytic models that have two or three parameters.

17.5 Identification of Fault-Prone Software Modules

In this section, we demonstrate how neural networks can be used as a pattern classifier to identify fault-prone software modules early during the development cycle. There are two reasons for this demonstration: (1) to show that neural network classifiers can be developed as a complement to existing statistical classifiers and (2) to demonstrate that classifiers can be developed without the usual assumptions about the input metrics. The idea of using a neural network for classifying fault-prone software modules was first demonstrated by Koshgoftaar et al. [Kosh93c], using a multilayer network trained using a standard back-propagation algorithm. Here, we expand and evaluate the neural network approach using a perceptron network and a multilayer feed-forward network developed using a modified cascade-correlation algorithm [Karu93b, Karu93c]. We also address other issues such as selection of proper training samples and representation of software metrics.

17.5.1 Identification of fault-prone modules using software metrics

We can relate static complexity measures of program texts with faults found (or program changes made) during testing using two broad modeling approaches. In the *estimative approach,* regressions models are used to predict the actual number of faults that will be disclosed during testing [Craw85, Shen85, Khos90, Muns92, Lyu95b]. Regression models assume that the metrics that constitute the input variables are independent and identically distributed normals. However, most practical measures often violate the normality assumption. This in turn results in poor fit of the regression models and inconsistent predictions.

In the *classification approach,* software modules are categorized into two or more fault-prone classes. A special case of the classification approach is to categorize software modules into either low-fault (simple) or high-fault (complex) classes. The main rationale behind the two-class classification approach is that software managers are more often interested in getting an approximate classification from this type of model than an accurate prediction of the residual faults. Existing two-class categorization models are based on the linear discriminant principle [Rodr87, Muns92]. Linear discriminant models assume that the metrics are orthogonal and that they follow a normal distribution. However, it is often true that some of the real metrics are variants of an existing metric, and they tend to exhibit strong collinearity among themselves. We can reduce multicollinearity among metrics using either principal component analysis or some other dimensionality reduction techniques. However, the reduced metrics may not explain all the variability if the original metrics have nonlinear relationship.

17.5.2 Data set used

The metrics data used in this section were obtained from [Lind89] for Medical Imaging System software. As described in Sec. 12.4.2, the complete system consists of approximately 4500 modules amounting to about 400,000 lines of code written in Pascal, FORTRAN, and PL/M assembly language. From this set, a random sample of 390 high-level language routines was selected for the analysis. For each module in the sample, program changes were recorded as an indication of software fault. The number of changes in the program modules varied from zero to 98. In addition to changes, 11 software complexity metrics were extracted from each module. These metrics include [Lind89]: (1) total number of lines (code, comments, and declarations inclusive); (2) number of executable lines (comments and declarations excluded); (3) total number of characters; (4) number of comments; (5) number of comment characters; (6) number of code characters; (7) Halstead's length measure, which counts the total number of operators and operands; (8) Halstead's estimated length measure (calculated from an equation derived using the number of unique operands and operators); (9) Jensen's estimated length measure (calculated using the number of unique operands and operators); (10) McCabe's cyclomatic complexity (counts the number of paths through the code); and (11) Belady's bandwidth measure (computes the average nesting level of instructions).

For the purpose of our classification study, these metrics represent 11 input (both real and integer) variables of the classifier. We consider a software module as a low-fault-prone module (category I) if there is zero or one change and as a high-fault-prone module (category II) if there are 10 or more changes. We consider the remaining modules to be a medium-fault category. For the purpose of this study we consider only the low- and high-fault-prone modules. Our extreme categorization and deliberate discarding of program modules is similar to the approach used in other studies [Rodr87, Muns92]. After discarding medium-fault-prone modules, there are 203 modules left in the data set. Of 203 modules, 114 modules belong to the low-fault-prone category while the remaining 89 modules belong to the high-fault-prone category.

17.5.3 Classifiers compared

We demonstrate the utility of both statistical and neural network classifiers. We consider different classifiers because it could be useful in evaluating their relative merits and limitations. We use a simple gaussian classifier from the traditional statistical family, and both a perceptron and a multilayer feed-forward network from the neural network family. The perceptron classifier represents the simplest model of the neural network family, while the feed-forward network is a typical realization of a complex nonlinear classifier. Since we are interested in assigning modules into two extreme categories, we can use two output units in our neural nets corresponding to these categories. Thus, a given arbitrary vector **X** is assigned to category I if the value of the output unit 1 is greater than the output of unit 2, and to category II otherwise.

17.5.3.1 A perceptron classifier. A perceptron with a hard-limiting threshold can be considered as a realization of a simple nonparametric linear discriminant classifier. If we use a sigmoidal unit, then the continuous valued output of the perceptron can be interpreted as a likelihood or probability with which inputs are assigned to different classes. To train a perceptron network we can use back-propagation or quick-propagation procedures, or a simple optimization procedure. We chose the quick-propagation procedure, which is part of the cascade-correlation algorithm, as our training algorithm. In almost all our experiments, the perceptron learned about only 75 to 80 percent of the training set. This implies that the rest of the training samples may not be linearly separable.

17.5.3.2 A multilayer network classifier. To evaluate whether a multilayer network can perform better than the other two classifiers, we repeated the same set of experiments using feed-forward networks constructed using the cascade-correlation algorithm [Fahl90]. Our initial results suggested that the multilayer networks constructed by the cascade-correlation algorithm are not capable of producing a better classification accuracy than the other two classifiers. An analysis at the end of the training suggested that the resulting networks have too many free variables (i.e., due to too many hidden units). A further analysis of the rate of decrease in the residual error versus the number of hidden units added to the networks revealed that the cascade-correlation algorithm is capable of adding more hidden units to learn individual training patterns at the later stages of the training phase than in the earlier stages. This happens if the training set contains patterns that are interspersed across different decision regions, or what might be called *border patterns* [Ahme89]. Such an unrestricted growth of the network can be a disadvantage in certain applications. The larger the size of the network the less likely that the network will be able to generalize. Thus it is necessary to constrain the growth of the network during training.

17.5.3.3 Minimal network growth using cross-validation. One standard approach used in statistical model fitting is to incorporate a cross-validation during parameter estimation. (See Stone [Ston74] and Efron [Efro79] for the basic theory and details.) The idea of using cross-validation to construct minimal networks has been extended to neural networks by several researchers within the context of the standard back-propagation algorithms [Morg89, Weig90]. Here, we demonstrate the utility of cross-validation within the context of the cascade-correlation algorithm. The modified cascade-correlation algorithm incorporates the cross-validation check during the output layer training phase to constrain the growth of the size of the network. The specific method that we employ is to divide each training set S into two sets: (1) a *training set,* used both to add hidden units and to determine weights and (2) a *cross-validation set,* which is used to decide when to stop the algorithm. To stop the network construction, the performance on the cross-validation set is monitored during the output training phase. We stop training as soon as the residual error of the cross-validation set stops decreasing from the residual error at the end of the previous output layer training phase. Another issue that needs to be addressed in using cross-validation is selecting an appropriate cross-validation set. As a rule of thumb, for each training set of size S, we randomly pick one-third for cross-validation and the remaining two-thirds for training. The resulting network learned about 95 percent of the training patterns. As shown in Sec. 17.5.7, the cross-validated construction considerably improves the classification performance of the networks on the test set.

17.5.3.4 A minimum distance classifier. In order to compare neural network classifiers with statistical linear discriminant classifiers we also implemented a simple minimum distance based a two-class gaussian classifier of the form [Nils90]:

$$|\mathbf{X} - \mathbf{C}_i| \triangleq ((\mathbf{X} - \mathbf{C}_i)(\mathbf{X} - \mathbf{C}_i)^t)^{1/2} \tag{17.35}$$

where $\mathbf{C}_i$, $i = 1, 2$ represent the prototype points for categories I and II, $\mathbf{X}$ is an 11-dimensional-metrics vector, and t is the transpose operator. The prototype points $\mathbf{C}_1$ and $\mathbf{C}_2$ are calculated from the training set based on the normality assumption. In this approach a given arbitrary input vector $\mathbf{X}$ is placed in category I if $|\mathbf{X} - \mathbf{C}_1| < |\mathbf{X} - \mathbf{C}_2|$ and in category II otherwise.

17.5.4 Data representation

All raw component metrics had distributions that are asymmetric with a positive skew (i.e., long tail to the right), and they had different numerical ranges. Note that asymmetric distributions do not conform to the normality assumption of a typical gaussian classifier. First, we transformed each metric using a natural logarithmic base to remove the extreme asymmetry of the original distribution of the individual metric. Next, we divided each metric by its standard deviation of the training set to mask the influence of the individual component metric on the distance score, These transformations considerably improved the performance of the gaussian classifier. To be consistent in our comparison, we used the same (log) transformation and scaling of inputs for other classifiers.

17.5.5 Training data selection

We had two objectives in selecting training data: (1) to evaluate how well a neural network classifier will perform across different-sized training sets and (2) to select the training data that are as unbiased as possible. The first objective was motivated by the need to evaluate whether a neural network classifier can be used early in the software development cycle. Thus, the classification experiments were conducted using training sets of increasing size $S = \frac{1}{4}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \frac{9}{10}$ fraction of 203 samples belonging to categories I and II. The remaining $(1 - S)$ fraction of the samples was used for testing the classifiers. In order to avoid bias in the training data, we randomly selected 10 different training samples for each fraction **S.** This resulted in 60 (that is, 6×10) different training and test sets.

17.5.6 Experimental approach

Since a neural network's performance can be affected by the weight vector used to initialize the network, we repeated the training experiment 25 times with different initial weight vectors for each training set. This resulted in a total of 250 training trials for each value of S. The results reported here for the neural network classifiers represent a summary of statistics for 250 experiments. The performance of the classifiers is reported in terms of classification errors. There are two types of classification errors that a classifier can make: a type I error occurs when the classifier identifies a low-fault-prone (category I) module as a high-fault-prone (category II) module; a type II error is produced when a high-fault-prone module is identified as a low-fault-prone module. From a software manager's point of view, these classification errors will have different implications. Type I misclassification will result in waste of test resources (because modules that are less fault-prone may be tested longer than what is normally required). On the other hand, type II misclassification will result in releasing products that are of inferior quality. From reliability point of view, a type II error is a more serious error than a type I error.

17.5.7 Results

First, we provide a comparison between the multilayer networks developed with and without cross-validation. Table 17.3 compares the complexity and the performance of the multilayer networks developed with and without cross-validation. Columns 2 through 7 represent the size and the performance of the networks developed by the cascade-correlation without cross-validation. The remaining six columns correspond to the networks constructed with cross-validation. Hidden unit statistics for the networks suggest that the growth of the network can be considerably constrained by adding cross-validation during the output layer training. With cross-validation, the rate of growth of the network is not severely affected by the increase in the size of the training set. Networks without cross-validation, on the other hand, may grow linearly depending on the size and the nature of the classification boundary of the training set. The corresponding error statistics for both the type I and type II errors suggest that an improvement in classification accuracy can be achieved by cross-validating the size of the networks. Another side benefit with cross-validation is that the training time of the cascade-correlation algorithm can be reduced significantly (because the algorithm adds only a few hidden units).

Next, we compare the classification accuracy of classifiers in terms of type I and type II errors. Table 17.4 illustrates the comparative results for type I error. The first three columns in Table 17.4 represents the size of the training set in terms of S as a percentage of all patterns, the size of the training set in terms of number of patterns and the size of the test set, respectively. Column 4 represents the number of test patterns belonging to category I. The remaining six columns represent the type I error for the three classifiers in terms of percentage mean error and standard deviation. The percentages of errors were obtained by dividing the number of misclassifications by the total number of test patterns belonging to category I. These results show that the gaussian and the perceptron classifiers may provide better classification than multilayer networks at the early stages of the software development cycle. However, the difference in performance of the gaussian classifier is not consistent across all values of S. For example, the gaussian clas-

TABLE 17.3 A Comparison of Nets With and Without Cross-Validation

	Without cross-validation						With cross-validation					
Training set size	Hidden units		Type I error		Type II error		Hidden units		Type I error		Type II error	
S (%)	Mean	Std.	Mean	Std.	Mean	Std.	Mean	Std.	Mean	Std.	Mean	Std.
25	5.1	1.5	24.64	7.2	16.38	6.4	1.9	1.3	20.19	5.4	12.11	4.7
33	6.2	1.8	20.24	8.4	17.27	5.5	2.2	1.0	18.24	5.5	12.40	4.1
50	7.4	1.8	18.30	7.4	18.65	6.4	2.0	0.9	17.41	5.6	15.04	5.2
67	9.7	1.7	15.78	6.5	18.05	7.1	2.7	1.1	14.32	5.8	14.08	5.5
75	10.4	1.8	14.54	7.6	16.85	7.3	2.7	1.3	13.27	7.0	13.84	5.4
90	11.2	1.6	10.33	7.2	17.73	8.3	2.9	1.2	9.77	9.4	15.47	5.1

sifier did not improve its accuracy when $S = 90\%$. The neural network classifiers, on the other hand, seem to improve their performance with an increase in the size of the training set. Among neural networks, the perceptron classifier seems to do better classifications than a multilayer net.

Table 17.5 illustrates the comparative results for type II errors. Column 4 represents the number of category II patterns in the test set. The remaining six columns represent the error statistics for type II errors. The mean values of the error statistics suggest that a multilayer network classifier, on the average, may provide a better classification of category II modules than the other two classifiers. This is an important result from the reliability perspective. Furthermore, the difference in performance between the multilayer network and the other two classifiers is consistent across all training sets. Unlike in category I classification, both the neural networks and the gaussian classifiers seem not to improve their classification accuracy as the training set size is increased. This may partly be attributed to the fact that the number of test inputs for category II is lower than for category I.

In summary, the multilayer neural network classifiers may provide better classification of category II modules than other classifiers, and the perceptron (or the gaussian) classifier may be a useful candidate for classifying category I modules.

TABLE 17.4 A Summary of Type I Error

	No. of patterns		Test patterns	Error statistics					
				Gaussian		Perceptron		Multilayer nets	
S	Training set	Test set	Category I (%)	Mean	Std.	Mean	Std.	Mean	Std.
25	50	153	86	13.16	4.7	16.17	5.5	20.19	5.4
33	66	137	77	11.44	4.0	11.74	3.9	18.24	5.5
50	101	102	57	12.45	3.2	11.58	3.2	17.41	5.6
67	136	67	37	9.46	4.1	10.14	3.9	14.32	5.8
75	152	51	28	8.57	5.4	9.15	5.8	13.27	7.0
90	182	21	12	14.17	7.9	4.03	4.3	9.77	9.4

TABLE 17.5 A Summary of Type II Error

	No. of patterns		Test patterns	Error statistics					
				Gaussian		Perceptron		Multilayer nets	
S	Training set	Test set	Category II (%)	Mean	Std.	Mean	Std.	Mean	Std.
25	50	153	67	15.61	4.2	15.98	7.8	12.11	4.7
33	66	137	60	15.46	4.6	15.78	6.6	12.40	4.1
50	101	102	45	16.01	5.1	16.97	6.8	15.04	5.2
67	136	67	30	16.00	5.4	16.11	7.6	14.08	5.5
75	152	51	23	17.39	5.8	18.39	6.3	13.84	5.4
90	182	21	9	21.11	6.3	19.11	5.6	15.47	5.1

17.6 Summary

We demonstrated the applicability of neural networks for modeling software reliability growth and to classify error-prone software modules. In both applications, the neural network offers an alternative to conventional analytic models that are obtained using empirical methods or developed using some a priori assumptions.

Though we have demonstrated the application of neural nets in the context of two problems related to software reliability engineering, their applicability need not be restricted to only these problems in software reliability engineering. With the availability of public-domain and commercial neural network software, the neural network approach may be used as a tool in other software engineering problems, such as identification of software reuse components, document understanding, test-case generation, and partitioning of test cases.

Problems

17.1 Obtain the public domain cascade-correlation algorithm simulator from the following Internet ftp address.

> `pt.cs.cmu.edu (128.2.254.155)`, *login:* `anonymous`, *password:* `your address`, and *directory:* `/afs/cs/project/connect/code/supported`

Get the shell archive file "`cascor-v1.2.shar`" or the latest version. Familiarize yourself with the simulator by running the example XOR problem (File name: `xor.data`) that comes with the simulator. Also observe how the training and test data are represented, how the initial network is specified, the important parameters of the simulator, and how various parameters are set.

17.2 Obtain the data set Data2 in the Data Disk, scale it, and construct a training set for the generalization training regime (refer to Fig. 17.4) with the entire data set. Train a neural network using the cascade-correlation simulator and answer the following.

a. How many hidden units did the simulator add to successfully train the network?

b. What is the number of weights in the network?

c. What are the values of the final weight vector?

(*Hint:* To scale the data, first separate the cumulative execution time and cumulative faults, and then scale them between 0.1 and 0.9 such that the highest value corresponds to 0.9 while the lowest value corresponds to 0.1.)

17.3 For the trained network in Prob. 17.2, use the training set as the test set and record the network outputs. Rescale the recorded outputs back to the original scale and plot a fitness curve with execution time on the x axis and cumulative faults on the y axis. Now plot a similar curve with the original data. Visually observe the deviation between these two curves.

17.4 For the data set Data2 in Prob. 17.2, construct a training set for the *prediction training* regime (refer to Fig. 17.4 for details). Repeat the experiments of Probs. 17.2 and 17.3 with this training set.

17.5 In Prob. 17.3, a feed-forward neural network was trained using Data2. In this exercise, use the same procedure, data set, and the steps outlined below to test the next-step prediction accuracy of the FFN-generalization model. First, construct a training set with the first three pairs of Data2. The corresponding test set will contain one input representing the execution time of the fourth observation. After training the network, test the network by feeding the fourth execution time as input. Record the network output. Next, construct another training set by including the fourth pair of the data into the previous training set. The corresponding test will consist of the fifth observation of the execution time. Again train and test a network. Repeat this step by incrementally adding all but the last observation to the training set. Thus, there will be $n - 3$ predicted values (where n is the number of points in Data2). Finally, evaluate the predictive accuracy of the network using:

a. A graph similar to the one in Prob. 17.3.

b. The average error (AE) and the average bias (AB).

17.6 Repeat the steps used in Prob. 17.5 and evaluate the end-point prediction accuracy of the FFN-generalization model using Data2. Note that the training set will have to be expanded as in Prob. 17.5, while the test set will always contain the last observation of the data set.

17.7 Consider the data set Data2 and evaluate the predictive accuracy of the JN-generalization model. Note that the training set for the JN-generalization model has to be changed to accommodate an additional input corresponding to the output of the network from the previous time step (i.e., the previous input). A special case of training the JN-generalization model, known as *teacher forced training,* is to use the actual output from the data set rather than the output of the network corresponding to the previous time step. Thus, each point in the training set will consist of two inputs, corresponding to the current execution time and the cumulative fault of the previous time step, and an output for the current cumulative fault. Evaluate both the next-step and the end-point prediction accuracies.

17.8 Repeat the steps used in Prob. 17.6 and evaluate the JN-prediction model using the data set Data2.

17.9 Collect software complexity metrics data from a project in your organization and the associated failure history.

17.10 Use the data set obtained in Prob. 17.9 and conduct a two-class classification experiment as performed in Sec. 17.5.

17.11 Use the data set obtained in Prob. 17.9 and conduct a three-class classification experiment by including the medium-fault category that we have omitted in Sec. 17.5. Observe the following:

a. Classification accuracy of the network
b. Increase/decrease in complexity of the network

17.12 Multilayer neural network models solve problems by creating an internal representation of the input variables. It is a useful exercise to analyze their internal representations and understand how such internal representations can be related to external inputs. For example, in our application involving identification of fault-prone software modules, it was not obvious what type of associations the neural network classifier created among the input metrics. Provide an analysis of the internal representation of the neural network in terms of

a. Its association with inputs metrics
b. The final weight vector of the network
c. Its association with the output categories

17.13 We used neural network models to associate static complexity metrics with the number of faults (and hence fault-prone categories). Instead, develop models to associate

a. Software complexity metrics to maintenance metrics
b. Design metrics with quality and maintainability objectives