Application of Cascade-Correlation Neural Networks

to Nonlinear System Identification

by

Klaus C. Mueller

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Electrical Engineering

APPROVED:

Dr. H. F. VanLandingham, Chairman

Dr. John S. Bay

Dr. William T. Baumann

December 1994

Blacksburg, Virginia

Application of Cascade-Correlation Neural Networks

to Nonlinear System Identification

by

Klaus C. Mueller

Dr. H. F. VanLandingham, Chairman

Electrical Engineering

(ABSTRACT)

Much research in recent years has been done in applying artificial neural networks to the problem of nonlinear system identification. The most common neural network architecture, the multilayer feed-forward network, trained with the backpropagation algorithm, has been shown to be capable of universal function approximation which makes it applicable to a much wider range of problems than other nonlinear identification techniques. While these neural networks show great potential, they still suffer several drawbacks, such as slow convergence toward a solution. New neural network architectures have been proposed in an attempt to overcome these limitations. This study examines one such architecture, Cascade-Correlation, and its usefulness in system identification applications, particularly the nonlinear case.

Acknowledgment

This paper could not have been completed without the help and support of many people. I would first like to express my gratitude to Dr. Hugh F. VanLandingham for his time, advice and guidance which made this paper possible. I would like to thank Dr. John S. Bay and Dr. William T. Baumann for being on my committee. My appreciation also to Dr. Scott Fahlman and R. Scott Crowder of Carnegie Mellon University for providing the source code for Cascade Correlation.

Many friends have contributed invaluable assistance and support. Special thanks go out to Tanya Cartwright and Susan Bixler for their valuable input and support. Thanks also to the employees of A.L.I. Computers and to Kirsten Shifflett, Lee Worley, John Wolfram, Scott Baker, Mark Stolt and Larry Mallak for their continuing support and friendship.

I would especially like to express my gratitude to my parents and brothers for their lifetime support. Herzlichen Dank für eure Unterstützung und Liebe, Mama und Papa. This thesis is dedicated to you.

Table of Contents

List of Figures and Tables

Figures

Tables

Chapter 1. Introduction

1.1 Motivation

The first step in any engineering problem is to understand the system being dealt with. This can often be accomplished through some knowledge of the physical laws that govern the system. In many cases, however, the underlying physics of the system are very complex or are not fully understood. In these situations it is desirable to create a model from empirical data using system identification techniques. Such techniques are well established for linear systems and easily applicable to a wide range of problems. On the other hand system identification techniques for nonlinear systems, which are more common in real-world situations, often require extensive computations and are applicable to only a small range of problems. Using artificial neural networks for nonlinear system modeling overcomes many of the limitations inherent in other "conventional" nonlinear identification methods. Neural network architectures such as multilayer feed-forward networks are basically nonlinear maps capable of universal function approximation which makes them an ideal candidate for nonlinear system identification applications. The backpropagation algorithm is used to adjust the parameters of these networks based on a sequence of input/output data pairs without the need of any *a priori* assumptions about the physical system. The network can in effect "learn" to model the system with a minimum of pre-existing knowledge about the system.

Although neural network system identification techniques show considerable improvement over more conventional methods, traditional neural network architectures and algorithms also exhibit several limitations. The backpropagation algorithm, for example, suffers from slow convergence toward a solution and in some instances may fail to con-

verge at all. It is also difficult to find an optimal network structure for a given problem. This can result in a network that is inadequate to fully model the underlying system. It may also over-train and become just a high order curve fit of the training data set. The internal structure is determined before the network is trained and cannot be altered without retraining the entire network. This makes it difficult to determine the ideal structure without extensive trial and error.

1.2 Objective

Several new algorithms have been proposed in recent years to try to overcome some of the limitations of backpropagation feed-forward networks. Among these are a variation on backpropagation known as "quickprop" and a self-constructing network architecture, cascade-correlation, proposed by Scott Fahlman.[1],[2] Dynamic network construction, as used by cascade-correlation, starts with a minimal network structure and adds elements during training as needed until an adequate network is created. For a variety of classification problems neural networks using these approaches have shown considerable performance increases over standard backpropagation networks. Their usefulness in system identification and related function approximation applications have yet to be explored. This study will examine some of the advantages and disadvantages the cascade-correlation architecture offers as applied to nonlinear dynamic system identification. Several examples are simulated in order to explore the network's effectiveness in modeling nonlinear systems.

1.3 Overview

This study is presented in four sections. Chapter 2 introduces the concepts behind "traditional" artificial neural network architectures, notably the multilayer feed-forward network, and the backpropagation algorithm. The chapter also explores some of the issues behind using neural networks for system identification applications as shown in previous literature.

Chapter 3 details the cascade-correlation architecture and the "quickprop" training algorithm. Algorithm implementation issues are also discussed. Previous results and comparisons to backpropagation networks are presented.

Chapter 4 combines cascade-correlation with the neural network system identification approach outlined in Chapter 2 and applies them to several nonlinear identification examples. The results are discussed and analyzed. The final chapter includes conclusions on the results of this study and suggests some areas for further study.

Chapter 2. Neural Networks and System Identification

While many real-world problems deal with nonlinear systems, conventional methods for nonlinear system identification are often not very practical for application to these problems. Artificial neural networks can overcome many of the limitations that plague conventional nonlinear identification techniques, and thus show great potential as universally applicable nonlinear system identifiers. This chapter details "traditional" backpropagation networks – multilayer feed-forward neural networks trained with the backpropagation algorithm. Several of the issues concerning their applicability and implementation for system identification are then addressed.

2.1 Artificial Neural Networks

Artificial neural networks were developed in an attempt to create intelligent systems that mimic some of the properties exhibited by biological neural networks such as learning, adaptability, and generalization. The earliest foundations for artificial neural networks were laid by D.O. Hebb in 1949. He proposed a learning law that became the starting point for current network training algorithms.[3] Subsequent research in the 1950's and 1960's centered on networks consisting of a single layer of artificial neurons. Interest waned in 1969 when Minsky and Papert [4] showed these networks were incapable of solving several basic problems such as the "exclusive-or" gate. No techniques were available at the time for training networks that contained more than a single layer of neurons. This changed in the early 1980's when Werbos [5], Parker [6], and Rumelhart, Hinton and Williams [7]

independantly proposed the backpropagation algorithm. This algorithm allows for the training of multiple layers of neurons and it has since been successfully applied to a wide range of problems, including pattern identification, speech recognition, and nonlinear function approximation.

Two general types of artificial neural networks that are commonly used are recurrent networks and feed-forward networks. Recurrent networks contain feedback connections between the network outputs, inputs, and/or hidden units. Two examples of recurrent networks are the Hopfield and the Bidirectional Associative Memory (BAM) networks.[8] From a systems theoretic point of view, these networks represent nonlinear dynamic feedback systems and they are most often used as associative memories and for the solution of optimization problems. Feed-forward networks have no feedback connections and can be viewed as representing static nonlinear maps. They are used mostly for pattern recognition problems. Multilayer feed-forward networks, as described below, are also predominantly used for system identification of nonlinear dynamic systems.[9]

2.1.1 feed-forward neural networks

The fundamental building block of a feed-forward network is the neuron shown in Figure 2.1. A neuron is supplied inputs from either the input layer or from a previous layer of neurons. Each input to the neuron, i, is multiplied by a weight coefficient, w_i. The inputs are then summed and passed through a nonlinear activation function. The activation function $f(x)$ serves to transform an unbounded input signal to the node into a bounded output signal. The nonlinear nature of the activation function is the key feature in artificial neural networks that allows nonlinear function approximation. Any bounded monotone-increasing function can serve as an activation function, the most common being the sigmoidal, the hyperbolic tangent, and the gaussian functions. The output neurons may have nonlinear or

linear activation functions. Linear activations may be more appropriate for function approximation applications, particularly system identification, as the dynamic range of the system may be greater than one.

A layer consists of several neurons in parallel, with each neuron receiving inputs from each of the nodes in the previous layer. A multilayer feed-forward network is created by using several layers together as shown in Figure 2.2. This simple network contains one hidden layer with three neurons, an input layer, and an output layer. More complex networks can be constructed by adding an arbitrary number of hidden layers and an arbitrary number of neurons in each layer.

The output of a multilayer network is computed by first presenting activations to the network inputs. These signals are propagated through the hidden layer(s) to the output layer. A network with one hidden layer is represented mathematically by:

Input layer to hidden layer

$$\tilde{\mathbf{x}} = \mathbf{W1}^T \mathbf{u} \tag{2.1a}$$
$$\mathbf{x} = f(\tilde{\mathbf{x}})$$

Hidden layer to output layer

$$\tilde{\mathbf{y}} = \mathbf{W2}^T \mathbf{x} \tag{2.1b}$$
$$\hat{\mathbf{y}} = f(\tilde{\mathbf{y}})$$

where **u** is the vector of inputs,

x is the vector of hidden layer outputs,

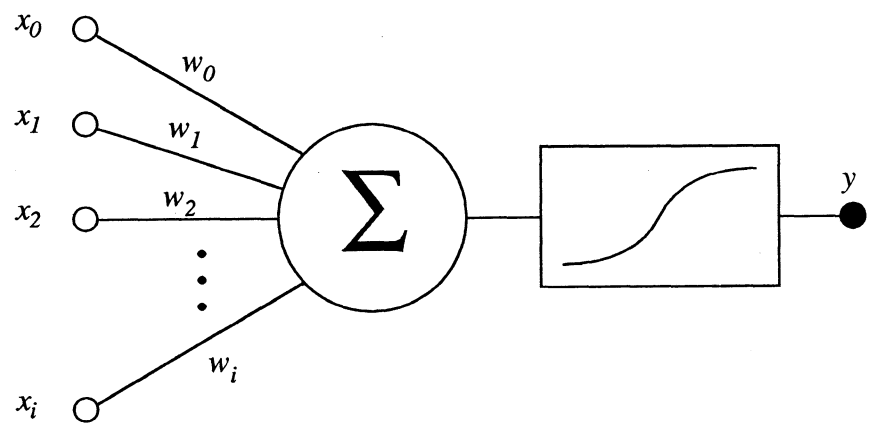


Figure 2.1: The artificial neuron.

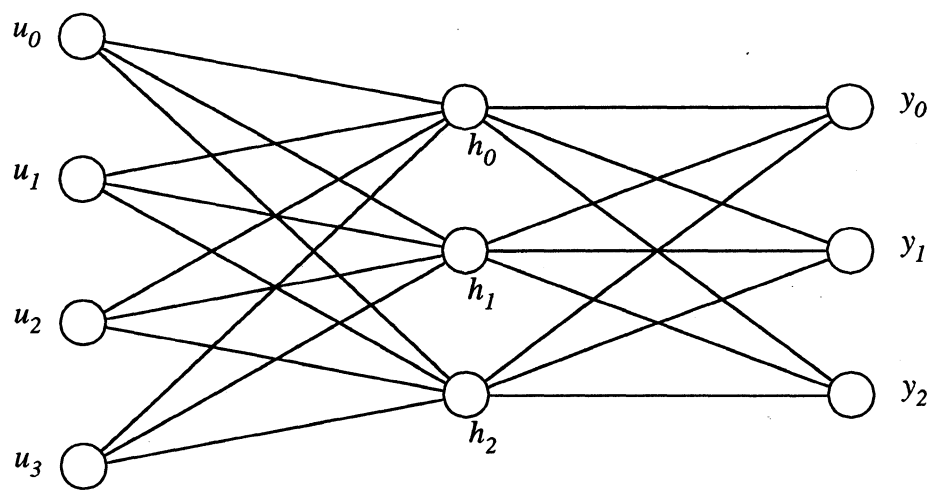


Figure 2.2 A multilayer feed-forward neural network.

$\hat{\mathbf{y}}$ is the vector of outputs,

W1 and **W2** are the weight matrices, and

$f(\bullet)$ is the nonlinear squashing function.

The weight coefficients are determined by using an algorithm such as backpropagation to "train" the network to learn the relationship between a set of input and output data.

2.1.2 the backpropagation algorithm

The biggest advance in artificial neural networks came with the introduction of the backpropagation algorithm for training the weights of multilayer feed-forward networks. Before Rumelhart, Hinton and Williams [7] and others proposed backpropagation in the mid-1980's, no methods were available for training multilayer networks. Backpropagation is an example of a supervised learning algorithm where the data sets for training consist of a set of inputs and a corresponding set of outputs measured from the system to be approximated. The inputs are applied to the network and the resulting outputs are compared to their desired values. The resulting error is used to adjust the weights in an attempt to minimize the error.

The backpropagation algorithm trains the weights of a multilayer network by comparing the *i*th output of the network $\hat{y}_i(k)$ to the corresponding desired output value $y_i(k)$. The instantaneous summed squared error at k over all the outputs is computed by

$$E(k) = \tfrac{1}{2}\sum_{i=1}^{C}\left[\hat{y}_i(k) - y_i(k)\right]^2 \qquad (2.2)$$

where C is the number of output nodes.

The weights in each layer are adjusted using a gradient descent approach that is a nonlinear extension of the least-mean-squares algorithm proposed by Widrow [10], which has been shown to be effective for linear estimation. The gradient descent "delta" rule attempts to move down the slope of the error function, $E(k)$, toward the global minimum, by computing the partial derivative of the error with respect to the weight coefficient $W_{ij}(k)$. The resulting equation for the change in weights is

$$\Delta W_{ij}(k) = -\eta_k \frac{\partial \mathrm{E}(k)}{\partial W_{ij}(k)}. \tag{2.3}$$

The parameter η_k is the step size, also known as the learn rate. It defines how large a step the algorithm should take down the slope.

The weight updates for the output layer are computed by expanding equation (2.3) so that it becomes

$$\Delta W2_{ij} = -\eta_2 \frac{\partial \mathrm{E}}{\partial \hat{y}_j} \frac{\partial \hat{y}_j}{\partial \tilde{y}_j} \frac{\partial \tilde{y}_j}{\partial W2_{ij}} = \eta_2 \ \delta 2_j \, x_i \tag{2.4a}$$

where $$\delta 2_j = (\hat{y}_j - y_j) \cdot f'(\tilde{y}_j). \tag{2.4b}$$

The error computed in (2.2) is then propagated back to the hidden layer in order to update its weights using the "delta" rule:

$$\Delta W1_{ij} = -\eta_1 \frac{\partial \mathrm{E}(k)}{\partial x_i(k)} \frac{\partial x_i}{\partial \tilde{x}_i} \frac{\partial \tilde{x}_i}{\partial W1_{ij}} = \eta_1 \, \delta 1_j u_i \tag{2.5a}$$

where $$\delta 1_j = f'(x_j)\sum_{i=1}^{C} \delta 2_i W 2_{ji}. \qquad (2.5b)$$

The algorithm is initialized by selecting random values for the initial weights. The weight updates are then repeated for each input/output pair in the training data set. A pass over the entire data set constitutes an epoch, after which the root-mean-squared error is measured over the data set. The training process is repeated until this error falls below a defined threshold. At this point training is considered successful and is stopped.

2.1.3 variations on backpropagation

Although backpropagation is an effective means for training the weights of multi-layer feed-forward networks, it has several limitations and problems. The biggest drawback is a very slow rate of convergence toward a solution. Training may require tens of thousands of epochs or more for any but the simplest of problems. Often the algorithm may stagnate or become trapped in local minima in the error surface, and fail to converge. Many adaptations on backpropagation have been suggested in an effort to overcome these problems and to increase overall performance.

One simple method to decrease the likelihood that backpropagation will become trapped is to add a "momentum" term to the "delta" rule [7]:

$$\Delta W_{ij}(t+1) = -\eta_k \frac{\partial \mathrm{E}}{\partial W_{ij}} + \alpha \Delta W_{ij}(t). \qquad (2.6)$$

Multiplying the "momentum coefficient" α to the previous weight change and adding it to the "delta" rule in this manner gives training some impetus to move out of a local minimum where it may otherwise stagnate.

Backpropagation is also very sensitive to the selection of parameters such as the learn rate. Taking a small enough step will guarantee that training will converge to a (local) minima and in some cases to the global minima or at least to a sufficiently close solution. This will also result in a very slow convergence. Too large a step size can cause the algorithm to overshoot the solution. Several methods have been suggested for adjusting the learn rate during training, either algorithmically or heuristically, with varying degrees of success.[11]-[13] Another approach is to adjust the learn rate based on the second derivative. An algorithm based on this concept is "quickprop" suggested by Scott Fahlman [2] and described in detail in the next chapter.

A useful adaptation of feed-forward networks is to superimpose linear feedthru connections, directly linking the input nodes to the output nodes, as shown in Figure 2.3. This feedthru layer can aid in learning any linearities in the mapping. This proves especially useful for system identification applications.[14]

2.2 The System Identification Problem

System identification is concerned with methods for determining a mathematical representation, or model, for a physical system. These systems are defined by a set of measurable causes, or inputs, that are mathematically related to a set of measurable effects, or outputs. Creating a model of such systems is valuable in aiding its prediction and control, and in understanding the system's overall behavior. System identification has applications to a wide range of problems, including chemical processes, biomedical systems, and electric power systems.

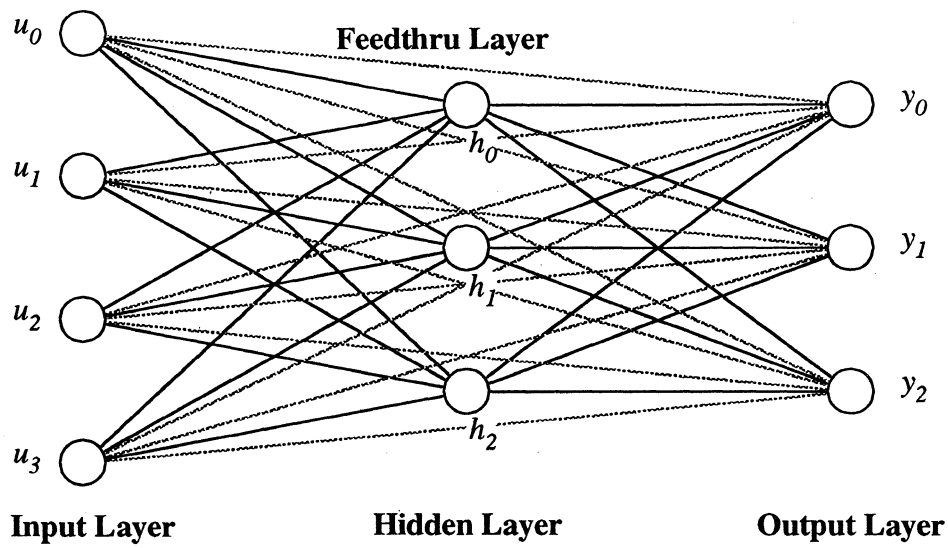


Figure 2.3: A multilayer feed-forward neural network with linear feedthru connections.

There are two methods for creating a model of a physical system. The first approach is to establish a set of equilibrium equations based on the physical laws that apply to the system of interest. This requires some knowledge about the underlying physics and dynamics of the system which are not always readily available, especially for complex systems. The second approach is called the "black-box" approach which attempts to create a model using only observations of the system's inputs and outputs. These observations can be obtained from experimental measurements, requiring only little knowledge about the system's inner mechanics to create a good model. Model parameters are adjusted based on the difference between the model's output and the desired plant output. While the former of these methods may provide a more complete understanding of the underlying physics of a system, the latter method is often more practical, especially for very complex systems.

Before a model of a system can be created, some assumptions about the nature of the system must be made. The first, and most important assumption, is determining whether to describe it using a linear or a nonlinear model. Linear systems are generally represented using either a state space model

$$\begin{aligned} \mathbf{x}[k+1] &= \mathbf{A}\mathbf{x}[k] + \mathbf{B}\mathbf{u}[k] \\ \mathbf{y}[k] &= \mathbf{C}\mathbf{x}[k] + \mathbf{D}\mathbf{u}[k] \end{aligned} \tag{2.7}$$

or the autoregressive-moving average (ARMA) model

$$\begin{aligned} y(k) = \alpha_1 y(k-1) + \alpha_2 y(k-2) + \ldots + \alpha_n y(k-n) + \\ \beta_0 u(k) + \beta_1 u(k-1) + \ldots + \beta_m u(k-m) \end{aligned} \tag{2.8}$$

Techniques for determining the coefficients for these models, such as least-squares techniques, maximum likelihood estimation [15], and pseudo-observability indices [14], are well established. Linear models can be created easily for a large variety of systems.

While "black-box" methods for system identification are readily available for the linear case, many real-world physical systems are nonlinear. Linear models can be used on these systems by restricting the range of perturbation to a small and fixed area of interest, often around an equilibrium point. Such models, however, are valid over this small range only and are therefore limited in usefulness. In order to obtain a more complete model, nonlinear system identification techniques are desirable. Several such techniques exist for nonlinear systems. The most common are the Volterra-Wiener series [16],[17] and the Extended Kalman Filter.[18] In theory, any non-explosive, time-invariant causal system with a fading memory can be approximated by a Volterra-Wiener series. In practice this technique is restricted to systems with "mild" polynomic nonlinearities and where second or third order models are sufficient. However more complex systems often have intensive computational requirements. Similar problems also limit the application of the Extended Kalman Filter. Other nonlinear techniques require restrictive assumptions about the nature of the nonlinearities and about the underlying differential equations, or their implementation may be highly complex and require excessive computation. Techniques for nonlinear modeling using artificial neural networks have shown great potential in overcoming these limitations, they and show promise as a universally applicable nonlinear system identification technique.

2.2.1 system identification problem statement

This study is concerned with causal time-invariant nonlinear dynamical plants with a uniformly bounded input, $u(k)$, and a plant output, $y_p(k)$, that is assumed to be stable.

The objective is to construct a suitable identification model $\hat{P}(u)$, which when presented with the same input $u(k)$ as the plant $P(u)$, produces an output, $\hat{y}_p(k)$, which approximates $y_p(k)$ to a desired precision, ε as defined by

$$\left\|\hat{y}_p - y_p\right\| = \left\|\hat{P}(u) - P(u)\right\| \leq \varepsilon, \quad u \in U \tag{2.9}$$

This is represented by the block diagram shown in Figure 2.4

The models used to accomplish this are nonlinear extensions of the linear case of equations (2.7) and (2.8). The nonlinear state space equations are represented as

$$\begin{aligned} x(k+1) &= \Phi[\cdot] \\ y(k) &= \Psi[\cdot] \end{aligned} \tag{2.10}$$

while the nonlinear extension of the ARMA model (NARMA) is represented by

$$y(k) = f(y(k-1),\ldots,y(k-m);u(k),\ldots,u(k-n)) \tag{2.11}$$

where $\Phi(\cdot)$, $\Psi(\cdot)$ and $f(\cdot)$ represent nonlinear functions.

System identification techniques can be applied either "on-line" or "off-line." "On-line" identification attempts to create a model while observing the input/output relation during real-time operation. A more accurate model can usually be constructed with data collected during previous observations and then used to adjust the model parameters "off-line." This method has an added advantage because the data may be processed and optimized for the identification process, which further increases the likelihood that an accurate model will be created.

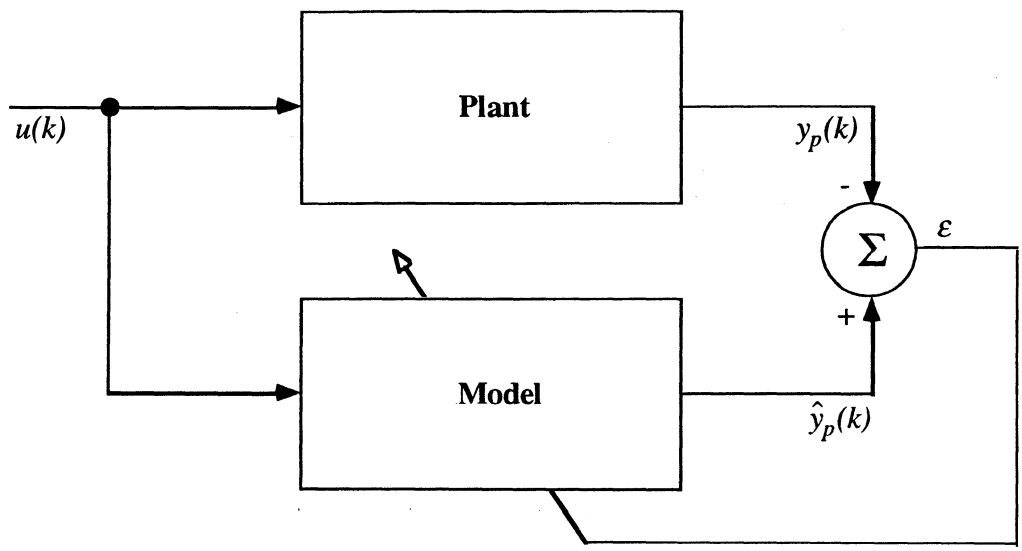


Figure 2.4: "Black-box" identification block diagram.

2.3 System Identification Using Neural Networks

As mentioned in section 2.2, none of the "traditional" techniques for nonlinear systems identification are very practical *and* applicable to a wide range of problems. The resurgence of interest in artificial neural networks in recent years has shown great potential in providing a method that fills these requirements. This section will explore some of the considerations required for using feed-forward neural networks to identify dynamic systems.

2.3.1 artificial neural networks are universal approximators

Interest for using artificial neural networks for system identification was spurred by the work of Hornik, Stinchcombe and White.[19] They showed that a multilayer feed-forward network, with as few as one hidden layer and using an arbitrary activation function, is capable of approximating any Borel measurable function to any degree of accuracy, provided a sufficient number of neurons are available in the hidden layer. Multilayer feed-forward neural networks can therefore be considered universal approximators. Failure for a network to correctly approximate a function can be attributed to inadequate learning, an inadequate number of hidden units, or the presence of a stochastic, rather than a deterministic relation, between inputs and outputs in the training data set. This conclusion is supported by Chen and Chen [20], who further showed that this approximation capability is applicable to the approximation of the output of dynamic systems.

These results are significant because they suggest that feed-forward neural networks can be trained to approximate any bounded, continuous, time-invariant, and causal nonlinear function. Further, the nonlinearities are embedded in the activation functions of the neural network, so there is no need for numerical approximation techniques, such as a trun-

cated Taylor series expansion, to determine the system nonlinearities. Neural networks, therefore, avoid much of the computational complexity and expense that limit other methods.

2.3.2 neural network system identification structures

While feed-forward networks are universal function approximators, from a systems theory point of view they are static nonlinear maps. The output of the network is simply a nonlinear expansion of the input nodes. In order to model nonlinear dynamic systems, it is necessary to somehow incorporate the dynamics into the neural network model. This is accomplished by feeding back past values of the system's inputs and outputs to derive part of the network's input vector. The resulting network resembles the NARMA model of equation (2.11) with the function $f(\bullet)$ representing the neural network model. The number of delays used for the system inputs and outputs define the order, m and n, of the model. Determining the proper order for the model is important. A neural network model cannot generate any missing dynamic terms and therefore an inadequate model may result if the order is insufficient for the problem. Too large an order may also produce adverse effects such as intensifying noise and other inaccuracies in the model.[14],[21]

The NARMA model of (2.11) is a very general representation and therefore it suits most applications. However, very little is known about the way it can be controlled. To overcome this limitation, Narendra and Parthasarty [9],[22] introduced several other models which use neural networks for the representation of SISO plants. These models are generalizations to nonlinear cases of models used in adaptive systems literature for the identification and control of linear systems.

Model I: system with linear output dynamics

$$y_p(k+1) = \sum_{i=0}^{n-1} \alpha_i y_p(k-i) + g[u(k), u(k-1), \ldots, u(k-m+1)] \tag{2.12}$$

Model II: system with linear input dynamics

$$y_p(k+1) = f[y_p(k), y_p(k-1), \ldots, y_p(k-n+1)] + \sum_{i=0}^{m-1} \beta_i u(k-i) \tag{2.13}$$

Model III: system with separate nonlinear input and output dynamics

$$y_p(k+1) = f[y_p(k), y_p(k-1), \ldots, y_p(k-n+1)] + g[u(k), u(k-1), \ldots, u(k-m+1)] \tag{2.14}$$

Feed-forward networks are used as subsystems in these models for the nonlinear functions represented by $f(\bullet)$ and $g(\bullet)$. The linear portions of these models, whose coefficients are determined separately from the neural network training, may prove more mathematically tractable for applications where control is an important objective.

A more general model than those above can be implemented by a multilayer feed-forward network using a superimposed linear feedthru layer as described in section 2.2.3. This model can be represented in the form

$$\hat{y}_p(k+1) = \sum_{i=0}^{n} \alpha_i y_p(k-i) + \sum_{i=0}^{m} \beta_i u(k-i) + f[y_p(k), y_p(k-1), \ldots, y_p(k-n); u(k), u(k-1), \ldots, u(k-m)] \quad (2.15)$$

In addition to the capability of a universal nonlinear approximator, this network structure simultaneously learns any linearities in the system even if they are not explicitly known. The linear coefficients therefore need not be determined separately from the network training. The linear portion of the network may also aid in controller design. Moreover, if no hidden units are used, this network reduces to a linear ARMA model with the coefficients determined by the "delta" rule. This model can therefore easily approximate linear systems as well as nonlinear systems.[14]

2.3.3 "black-box" identification using neural networks

The basic approach for "black-box" identification of linear and nonlinear systems is identical for conventional and neural network methods. The identification procedure consists of three steps: determining a model structure, determining the parameters of the model, and determining the validity of the model. This section describes some of the considerations when applying neural networks as "black-box" identifiers.

Model Structure

Defining a model structure for neural network techniques involves determining both the dynamic structure (or order) needed to approximate the system, and the internal structure of the network, i.e., the number and types of layers and the number of nodes needed in each layer. If the order of the neural network model is insufficient, the network will be incapable of accurately modeling the dynamics. Choosing too large an order, while most

likely reducing the internal network complexity and speeding training, will greatly enhance the effects of any noise and other inaccuracies. It may also lead to network models that are overfitted, where the model tends to become just a high dimensional curve fit to the training data. Such a model may predict accurately over the range of the training data set, but it may not be a good representation of the underlying system dynamics. Determining the proper order is mostly a matter of trial and error. An arbitrarily small order is picked for the initial model. If this network model proves to be insufficient, a higher order needs to be selected and the network retrained.

The complexity of the internal network structure of a multilayer feed-forward network can also affect the approximation ability of a network model and its generalization capabilities. While one hidden layer is sufficient for most problems, judging the number of necessary hidden nodes is mostly guesswork. The effect of the hidden units is similar to that of the order; too few may produce an insufficient model, too many may result in overfitting.

Model Parameters

Once the model's order and internal structure are fixed, the parameters of the model need to be found. In the neural network case, this involves training the connection weights using backpropagation or a similar algorithm to learn a training data set. The key to implementing a good identifier is to provide enough information in the training data set so that the network has an adequate description of the reachable input and output spaces. The data must be created so the network has full knowledge of the system dynamics. It must also cover the permissible minimum and maximum values so proper scaling can be determined. A good method for achieving this is to present the system with a white noise input and measuring the resulting outputs.

Model Validation

When training of the network reaches a point where it is considered successful, the validity of the model must be tested to determine its overall accuracy and to detect any inadequacies in the fit. The validity of a model may vary with the method used for prediction. "One-step-ahead" prediction (Figure 2.5a) uses delayed values of the system's inputs and outputs to predict what the next output value will be:

$$\hat{y}(t) = f(u(t-1),\ldots,u(t-n_u), y(t-1),\ldots,y(t-n_y)). \qquad (2.16)$$

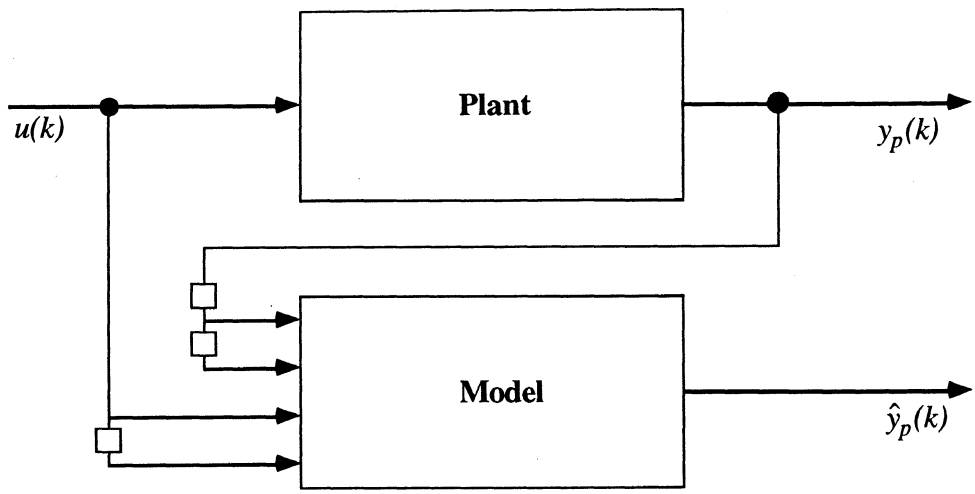
Often this method of prediction will produce good results over the test data set even if the model is not completely accurate because the model parameters were estimated by minimizing "one-step-ahead" prediction errors.

A more telling measure of predictive capability of a model is to compute the "model predicted output" (Figure 2.5b) where the predicted output values are fed back to the networks input vector:

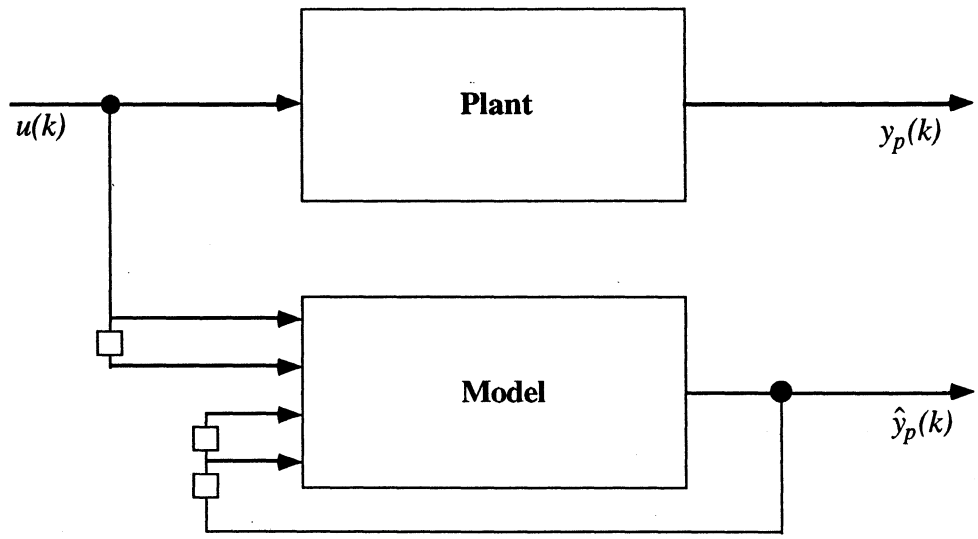
$$\hat{y}(t) = f(u(t-1),\ldots,u(t-n_u), \hat{y}(t-1),\ldots,\hat{y}(t-n_y)) \qquad (2.17)$$

Since the predicted output is based on past predictions of the output and any errors in prediction will tend to effect future predictions, this method will often quickly indicate inadequate models.

The validity can be tested by running the model using diverse test data sets, preferably including data measured over a different range from that used for training. A more formal approach, suggested by Billings et. al. [20], computes the various correlations of the inputs and the residual errors to perform a model validity test.



(a) "one-step-ahead" prediction



(b) "model-predicted output"

Figure 2.5: Model prediction block diagrams.

$$
\begin{aligned}
&\phi_{\varepsilon\varepsilon}(\tau) = E[\varepsilon(t-\tau)\varepsilon(t)] = \delta(\tau) \\
&\phi_{u\varepsilon}(\tau) = E[u(t-\tau)\varepsilon(t)] = 0, \forall\tau \\
&\phi_{u^2\varepsilon}(\tau) = E[(u^2(t-\tau) - \overline{u}^2(t))\varepsilon(t)] = 0, \forall\tau \\
&\phi_{u^2\varepsilon^2}(\tau) = E[(u^2(t-\tau) - \overline{u}^2(t))\varepsilon^2(t)] = 0, \forall\tau \\
&\phi_{\varepsilon(\varepsilon u)}(\tau) = E[\varepsilon(t)\varepsilon(t-1-\tau)u(t-1-\tau)] = 0, \tau \geq 0
\end{aligned}
\tag{2.18}
$$

These normalized correlations between the two sequences are computed by

$$
\phi_{\psi_1\psi_2}(\tau) = \frac{\sum_{t=1}^{N-\tau} \psi_1(t)\psi_2(t+\tau)}{\left[\sum_{t=1}^{N} \psi_1^2(t) \sum_{t=1}^{N} \psi_2^2(t)\right]^{1/2}}. \tag{2.19}
$$

This test is based on the assumption that if a model is adequate then the residual errors, $\varepsilon(t)$, should be unpredictable from all linear and nonlinear combinations of past inputs and outputs.

Since the correlations will never be exactly zero for all cases, a 95% confidence band is used to determine if the estimated correlations are significant or not. Even though these tests cannot be considered 100% conclusive because they were derived for the class of analytic nonlinear systems, and neural network models can approximate a larger range of functions, they are useful tools in determining the adequacy of the model's representation of the system dynamics.

2.4 Conclusion

Multilayer feed-forward neural networks have been shown to have great potential for nonlinear system identification applications. Govind and Ramamoorthy [16] compared the neural network technique to the "traditional" Volterra-Wiener series. They showed that the neural network techniques generally produced approximations as good as or better than other techniques. The backpropagation algorithm also provides a method for determining the model parameters that is much simpler and less computationally expensive than those required for the traditional techniques. Most importantly, the neural network models are applicable to a much wider range of nonlinear systems. In addition to being universal approximators, neural networks can model nonlinear systems with no assumptions of the underlying model structure other than the system order, which can be determined experimentally. Neural network models are also as easily applicable to linear systems as to nonlinear systems, especially when a linear feedthru layer is included.

While multilayer feed-forward networks trained with backpropagation have shown good results as nonlinear models and are a considerable improvement over more conventional nonlinear system identification techniques, they are still hindered by several limitations and problems. Backpropagation is very sensitive to user-selectable parameters such as the learn rate. Even with optimal parameter selections, the algorithm tends to converge slowly to the solution (global minimum) and in the process may become trapped in local minima, thus failing to converge at all. Another difficulty arises in determining an adequate internal structure (the number of hidden nodes and layers) of the neural network required for a good approximation. Inappropriate network sizes can lead to inadequate representations of the system. Large networks may tend to overfit the data, while small networks may not model the system fully. New neural network algorithms and architectures, such as cascade-correlation, may overcome some of these limitations.

Chapter 3. Cascade-Correlation Neural Networks

Although multilayer feed-forward networks have been shown to have excellent application to nonlinear system identification problems, they suffer from several major limitations. Among these are the difficulty in determining the number of hidden layers and hidden nodes suitable for a given problem. The backpropagation training algorithm tends to require long training times before converging to a solution. It can also be very sensitive to algorithm parameters and initial weight selections.

Traditional multilayer feed-forward neural networks have a static internal structure – the number of hidden layers and hidden nodes are determined, often guessed, before the network is trained. Adding or removing hidden nodes or layers requires retraining the entire network. Therefore it is difficult to efficiently determine an optimal structure for a particular application. Using an "nonoptimal" network structure can result in a network that is either insufficient to correctly approximate a system, or it is too large and overfits the data.

These problems suggest a need for new algorithms and architectures. One solution is to allow for dynamic configuration of hidden nodes. There are two general approaches to solving this problem: pruning and network construction. Pruning starts with a large, excessive number of hidden nodes and then attempts to remove any unimportant connections or nodes. Examples of this approach are the "optimal brain damage" technique [29] and the "Frobenius approximation reduction" method.[30] The network construction approach starts with a minimal network and adds hidden nodes as called for. This approach is much more practical and efficient since hidden units are trained only as needed. The cascade-correlation algorithm is an example of a network construction architecture.

This chapter first describes the cascade-correlation architecture and algorithms, including the "quickprop" algorithm used for weight training. Implementation details, such as the effect of algorithm parameters, are also described and the general properties of this class of artificial neural networks are explored. This chapter concludes with a discussion of previous applications of the cascade-correlation algorithm to various problems, and some proposed variations of the algorithm.

3.1 The Cascade-Correlation Architecture

The cascade-correlation architecture is a multilayer feed-forward neural network that uses a network construction approach to determine a minimal structure required for a given application. It was proposed by Fahlman and Lebiere [1] to surmount some of the problems and limitations associated with backpropagation networks that have a static structure.

The cascade-correlation architecture attempts to solve what Fahlman terms the "moving target problem." The backpropagation algorithm adjusts all of the weights in the network simultaneously so that each hidden unit sees a constantly changing environment, thus wasting much effort in its attempt to assume a useful role in solving the overall problem. Cascade-correlation overcomes this by combining a cascade architecture in which hidden units are added one at a time, and a learning algorithm that attempts to maximize the magnitude of the correlation between a new unit's output and the residual error to be eliminated. Once a new hidden unit is added, the input connection weights are frozen, requiring training of only the current candidate unit. The cascade-correlation architecture also uses direct connections between a new hidden unit and any existing units, enabling it to make use of

any pre-existing information in the network. A cascade-correlation network structure with three hidden units is shown in Figure 3.1.

3.1.1 the cascade-correlation algorithm

The initial network structure (Figure 3.3a) consists of only a set of inputs units, including a bias term, connected with adjustable weights to a set of outputs units that have either linear or nonlinear activation functions. No hidden units are included in this initial configuration. The weights of this network are trained over the entire training data set using either the "delta" rule or Fahlman's "quickprop" until no significant error reduction occurs. Then, once again the training set is applied to the network in order to measure the error. If the performance is satisfactory, below a defined precision threshold, training is stopped, otherwise a new hidden unit is added.

A new hidden unit is created by starting with a candidate unit that receives connections from all network inputs and from all existing hidden units. The candidate unit is not connected to the active network until training of the unit's input weights is completed. The training data sets are presented to the candidate unit with the goal of maximizing S, the sum over all the output units of the magnitude of the correlation between V, the candidate units value and E_o, the residual output error:

$$S = \sum_o \left| \sum_p (V_p - \overline{V})(E_{p,o} - \overline{E}_o) \right| \tag{3.1}$$

where o is the network output at which error is measured,

p is the current training pattern, and

$\overline{V}$ and $\overline{E}_o$ are the values of V and E_o averaged over all training patterns.

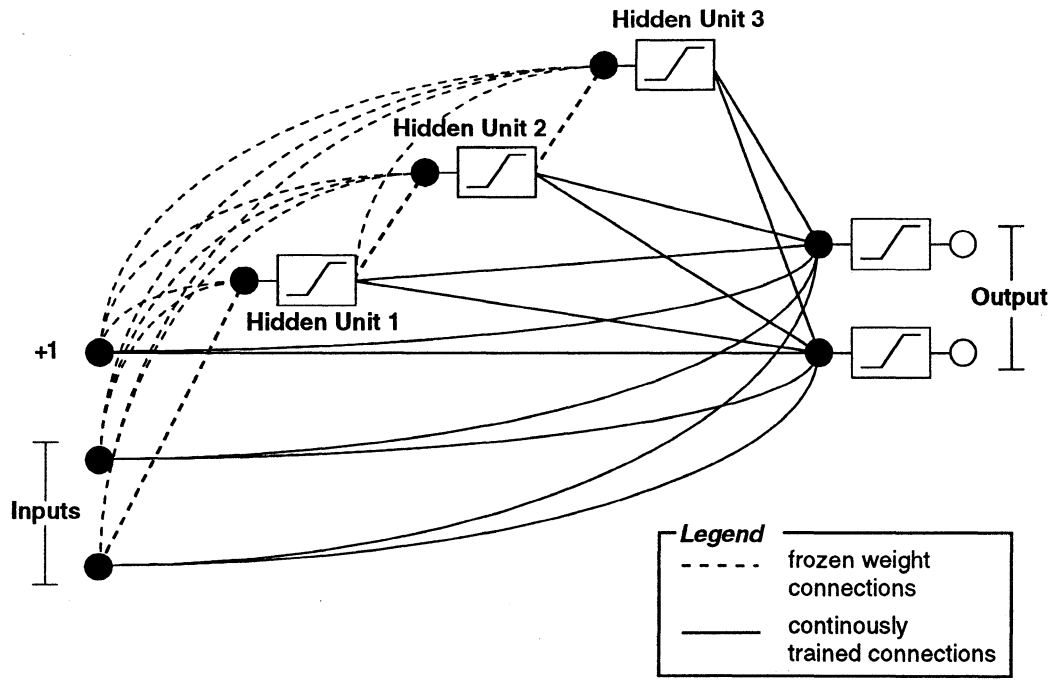


Figure 3.1: The cascade-correlation network architecture.

S is maximized by using a gradient ascent method over all training patterns on (3.1)

$$\frac{\partial S}{\partial w_i} = \sum_{p,o} \sigma_o (E_{p,o} - \overline{E}_o) f'_p I_{i,p} \qquad (3.2)$$

where σ_o is the sign of the correlation between candidate value and output o,

f'_p is the derivative for pattern p of the candidate unit's activation function with respect to the sum of its inputs, and

$I_{i,p}$ is the input the candidate unit receives from input i for pattern p.

The weights are updated by using the partial derivative (3.2) with either the "delta" rule or "quickprop." When S stops improving noticeably, the candidate unit's input weights are frozen, i.e., these weights are not adjusted when subsequent units are trained, and the unit is added to the active network. The algorithm repeats by retraining all of the output units.

A pool of candidate units, with each unit having different random initial weights while receiving the same input signals and the same residual error, can be trained simultaneously. The best unit from the pool is then added to the network (Figures 3.3b-e). This can reduce the chance that a useless unit, that may have become trapped during training, is installed in the network. With large candidate pools, many different areas of the weight space can be explored simultaneously, which improves the likelihood that an accurate and minimal network is created. If enough memory is available, the unit values and output errors can be cached for an entire epoch which makes it unnecessary to recompute these values for each instance and reduces the number of required computations.

The activation functions used for the hidden units are similar to those used in standard feed-forward networks: the sigmoid, the hyperbolic tangent, and the gaussian func-

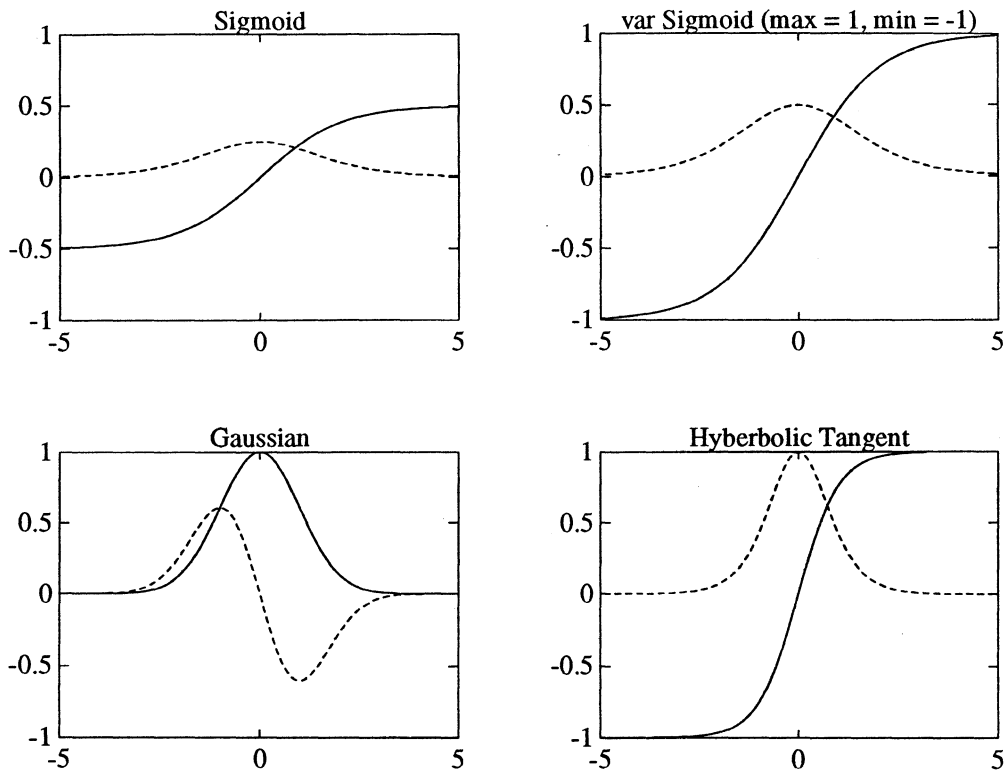


Figure 3.2: Nonlinear activation functions.
(dashed lines represent first derivatives of activations)

tions, shown in Figure 3.2 with their first derivatives (dashed lines). A pool of candidate units may contain units with different activation functions. The algorithm can then effectively choose the activation that best suits the problem.

To ensure the smallest possible network configuration is chosen without over-training the network, a validation epoch may be run after training of the output units is completed. The network is presented with a data set that is unique from the training set and the resulting error is measured. If the performance is not better than that of the previous validation epoch, a copy of the current state of the network is stored and training continues. If the performance does not improve over the best score to date for a specified number of training cycles (determined by a patience parameter), the network is returned to the stored "best" state and training stops.

3.1.2 "QuickProp" weight adaptation

Another problem with backpropagation networks lies in selecting a good step size (learn rate) for the gradient descent algorithm. The ideal step size minimizes training time while not overshooting the solution. Several approaches to find the ideal step size have been suggested, including adjusting it heuristically based on the history of the computation, and making use of the second derivative of the error with respect to each weight.

Fahlman's "quickprop" algorithm [2] uses an approach that approximates the second order information. This has the advantage that the second derivative need not be explicitly calculated, which would only add the computational overhead. The second order approximation is based on two assumptions: that the error vs. weight curve for each weight can be approximated by a parabola, and that the change in the slope of the error curve for each weight is not affected by the other simultaneously changing weights. The previous and current error slopes and the weight change between points are used to determine a

parabola as an approximation of the second derivative. The algorithm then jumps directly to the minimum point of this parabola. The equation for the weight updates thus becomes

$$\Delta w(t) = \frac{S(t)}{S(t-1) - S(t)} \Delta w(t-1) \tag{3.6}$$

where $S(t)$ and $S(t-1)$ are the current and previous values of $\partial E / \partial w$.

Fahlman determined that even though the assumptions are somewhat risky and the estimation is quite crude, it is very effective when applied iteratively. A step size parameter is still required to bootstrap the algorithm and to restart the process for any weights that had a previous step size of zero, but the algorithm is much less sensitive to the selection of this parameter because the step size is determined mainly by (3.6).

3.1.3 error measures

The "quickprop" algorithm requires a measure for determining when training has stagnated or when performance is sufficiently accurate. This is accomplished in cascade-correlation using the true error and error index measures described below.[31]

True Error

The true error is the squared error over all the outputs:

$$E_{true} = \sum_{k} (\hat{y} - y)^2 \tag{3.7}$$

where y is the desired output,

$\hat{y}$ is the network output, and

k is the number of outputs.

Error Index

The error index is the normalized root mean squared error, used to determine when the overall network performance is sufficiently accurate within defined constraints. The error index is defined as

$$E_{index} = \frac{\sqrt{E_{true}/N}}{\sigma} \tag{3.8a}$$

where N = (# of training points) * (# of outputs)

σ is the standard deviation of the data set computed by:

$$\sigma = \sqrt{\frac{N\sum y^2 - \left(\sum y\right)^2}{N(N-1)}}. \tag{3.8b}$$

3.1.4 algorithm parameters

Combining the cascade-correlation and "quickprop" algorithms introduces a number of additional parameters than are required for a backpropagation network. While none of these parameters are as sensitive as the learn rate is to backpropagation, proper values are desired for optimal algorithmic efficiency. Below is a listing of these parameters and a description of their effect on training. Included are parameters for additional enhancements to the training algorithms suggested by Fahlman in [2].

Cascade-Correlation Parameters

• **bias:** bias input to the network (usually 1 or 0).

• **error index threshold:** the desired network accuracy as compared to the error index (equation 3.8) measured during training. If the error index falls below this value training is stopped and considered successful.

• **patience:** the number of epochs training can continue without improving noticeably before going to the next step in the algorithm (either adding a new candidate unit, or training the output connections.)

• **candidate unit epochs:** the maximum number of epochs to train the candidate pool before adding the best unit to the network. In most cases training stagnates before this point is reached.

• **output epochs:** the maximum number of epochs to train the output units before training new candidate units.

• **maximum units:** the maximum number of hidden units to add to the network.

• **new unit type:** the nonlinear activation function used for the new candidate units. Types can be any of those shown in Figure 3.2, or VARIED which places an equal number of units with each activation function into the candidate pool.

• **candidate pool size:** the number of candidate units to train simultaneously when adding an new unit.

"QuickProp " Parameters

• **maximum growth factor μ:** no weight step is allowed to be greater in magnitude than μ times the previous step for that weight; if the step is too large, use μ times the previous step as the new step. This prevents the step size from becoming infinitely large. A

value around 1.75 is suggested by Fahlman. Too large a value may cause the network to behave chaotically and result in failure to converge.

• **weight decay term:** a small value added to the slope computed for each weight in order to keep the weights from growing too large.

• **learn rate ε:** adds ε times the current slope to the Δw value computed by the quadratic formula (3.6), unless the current slope is opposite in sign from the previous slope; in which case the quadratic term is used alone. This also allows for a way to initialize the "quickprop" process.

• **initial weight range r:** weights are initialized using uniformly distributed random values in the specified range $-r \leq w_{ij} \leq r$.

• **sigmoid-prime offset:** small value (usually 0.1) added to the derivative of sigmoid-prime activation function so it does not go to zero (see [2]). For cascade-correlation this parameter is applied only to output layer training because it seems to interfere with the correlation machinery.

3.2 Properties of Cascade-Correlation Networks

In many ways the nature of the cascade-correlation architecture is very different from that of traditional static feed-forward networks. This new class of networks may present additional advantages over backpropagation networks beyond decreased training times. This section will examine some of the properties of cascade-correlation networks that make it unique from traditional feed-forward network structures.

3.2.1 cascade connections

Unlike standard feed-forward networks, the cascade-correlation architecture uses cascaded connections between hidden units, i.e., each hidden unit receives inputs not only from the input layer, but also from any pre-existing units in the network. These cascade connections allow a candidate unit to use any existing information in the network produced by previous units in its attempt to reduce the residual error, thus avoiding redundancies. Cascading several hidden units also allows the network to produce more complex outputs than is otherwise possible. The activation functions of several connected units can easily produce a square wave shape or other complex mappings that are impossible using only a single nonlinear activation.

3.2.2 using pools of candidate units

Other advantages of cascade-correlation learning originate with the use of a pool of candidate units and the freezing of connection weights once a new unit is added. By training several candidate units simultaneously, with each unit differing only in initial weights and possibly in the activation function used, the algorithm can explore several areas of weight space simultaneously. This would suggest using large candidate pools increases the chance that an accurate and minimal network will be found. On a computer that allows only serial processing, training larger candidate pools tends to increase the required training time. The real power of this approach would be evident when parallel processing is used, which allows the candidate units to truly be trained simultaneously.

3.2.3 freezing hidden unit weight connections

Once a new hidden unit is added, the input weights to that unit are frozen. Only the output unit's weights are retrained. This has several advantages. First, since only one unit

is being trained at any time, the environment that unit sees stays constant throughout its training unlike backpropagation, where simultaneous training of hidden units constantly changes the environment that each unit is attempting to adapt to. Adjusting the weights of only one hidden unit at a time is also computationally much more efficient.

As a candidate unit is being trained, it trys to learn a "feature" of the system in its attempt to become useful toward a solution. Freezing the weights ensures each of these "feature detectors" does not change once it has been added. This minimizes the chances the network will over-parameterize the data. If subsequent hidden units do result in a network that overfits the data set, they can easily be removed, restoring the network to an earlier state that better represents the underlying system.

3.3 Applications of Cascade-Correlation Networks

The two classes of problems feed-forward neural networks are commonly applied to are pattern classification and function approximation. This section describes some of the problems in each of these classes cascade-correlation networks have been successfully applied to by other researchers.

3.3.1 classification

The problem of classification involves training a network to successfully separate a set of input data into two or more distinct groups, an example of which is the two-spiral problem. The network must learn to identify which of two intertwined spirals a given data point belongs to. This problem has been shown to be a very difficult one for standard backpropagation to solve. Fahlman and Lebiere [2] showed that a cascade-correlation net-

work can learn this task quickly and easily. They also compared the ability of the two training algorithms to solve the N-parity problem. The cascade-correlation network found a more compact network while training up to 50 times faster than a backpropagation network.

Yang and Honavar [32] applied cascade-correlation to a series of real-world classification problems. Their results supported the conclusions reached by Fahlman's examples with the algorithm showing training times of 1 to 2 orders of magnitude faster than backpropagation while requiring substantially smaller network structures. As may be expected, better performance was reported with larger candidate pools.

Whitley and Karananithi [33] examined some of the generalization characteristics of cascade-correlation, i.e., the ability of a network to correctly process data that lie outside the range used for the training data set. They applied cascade-correlation to a large signal processing classification problem. In addition to training times of up to 100 times faster, the cascade-correlation networks exhibited much better generalization capabilities than a backpropagation network. For another complex two-dimensional problem, the cascade-correlation network easily found a solution while the backpropagation network failed to converge. Whitley's conclusions about cascade-correlation's generalization capabilities are supported by the work of Sjoogard.[34]

3.3.2 function approximation

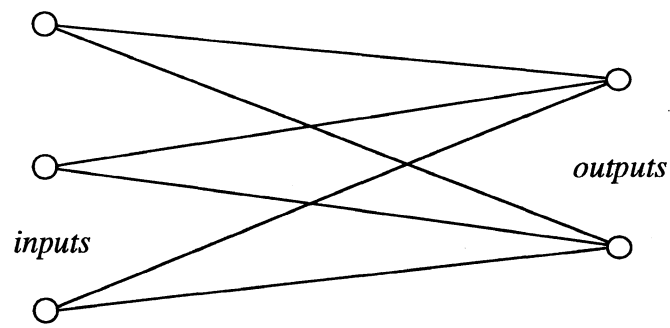
Function approximation problems, as the name suggests, apply neural networks to approximate continuous smooth functions. System identification is one such problem. To date not much research has been done on the application of cascade-correlation networks to these types of problems. Smith and Chase [35] use a cascade-correlation network for an on-line structural system identification problem. The network was to emulate linear regression

techniques in the frequency domain to identify stiffness and damping matrix coefficients. The cascade-correlation network showed considerable improvement in speed over the linear regression approach making it more feasible for real-time applications.

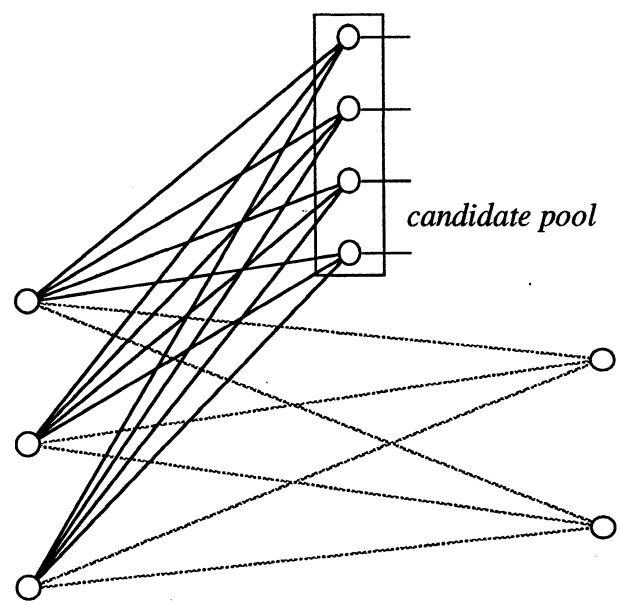
Nelson, Ensley and Rogers [36] applied cascade-correlation networks to predict aperiodic or chaotic functions, in particular the Mackey-Glass equations. The predictive accuracy of their networks varied with the order of the model. Bad models resulted from too few or too many feedback inputs. The predictive accuracy of the networks was also shown to increase with larger training data sets.

3.4 Conclusions

The cascade-correlation algorithm presents a novel neural network architecture that builds a network by adding hidden units as needed. This allows for automatic determination of a minimal network configuration, and for more efficient training of connection weights. The result, when applied to a variety of applications, has been considerably faster training than backpropagation training of static feed-forward neural networks. In some cases the cascade-correlation network easily determined solutions where backpropagation networks failed. The generalization capabilities of the cascade-correlation networks have also shown improvement. These results suggest considerable promise for this new architecture in overcoming some of the problems that plague the use of traditional static backpropagation feed-forward neural networks.

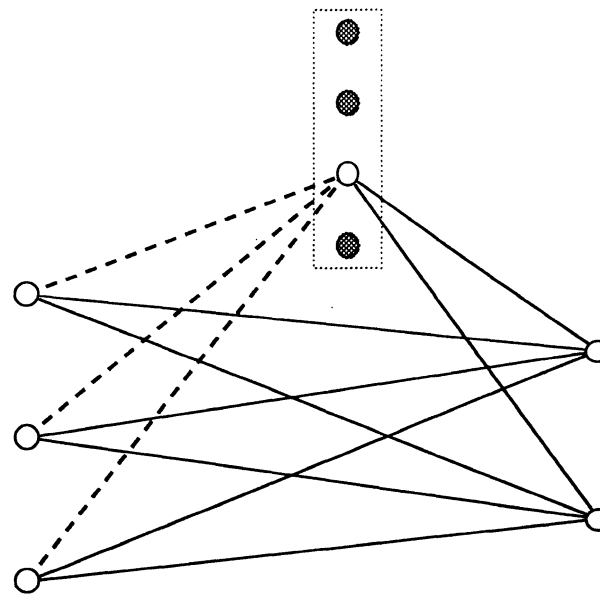


(a) The initial network consisting of only linear connections between the system inputs and outputs. The connection weights are adjusted until training stagnates.

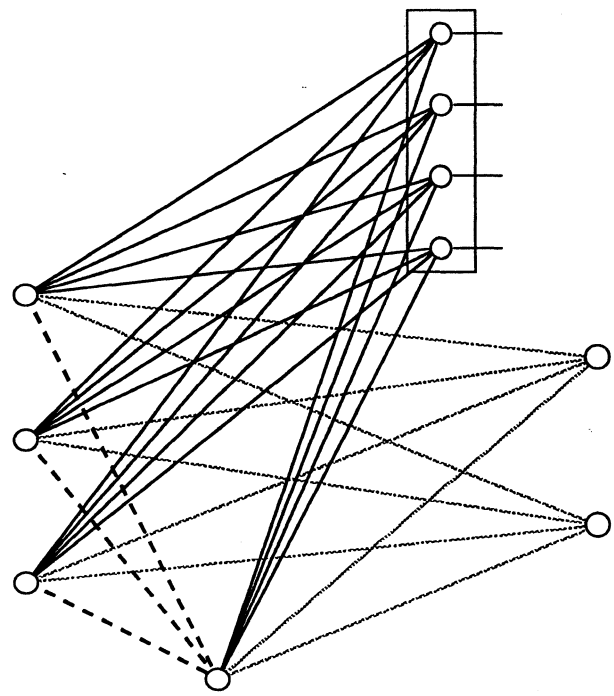


(b) A pool of candidate units, each with different initial weights, is created and their input weights are trained. Each candidate unit receives the same input/output signals and residual errors.

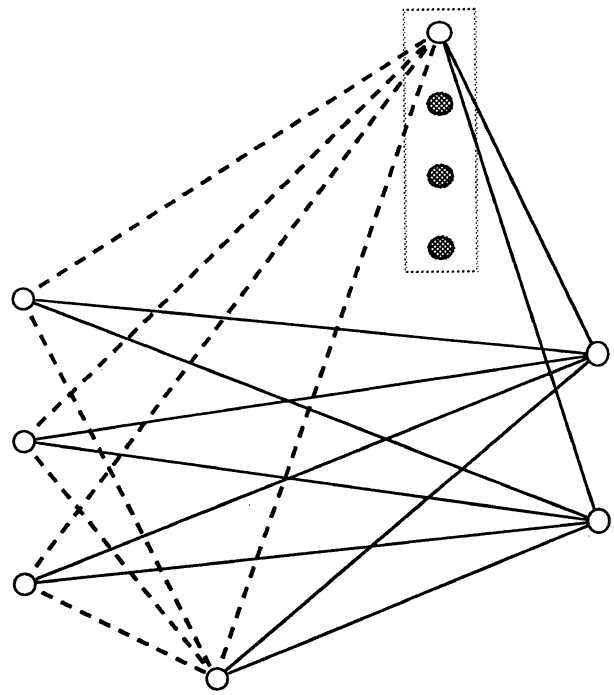
Figure 3.3: Construction of a cascade-correlation network.



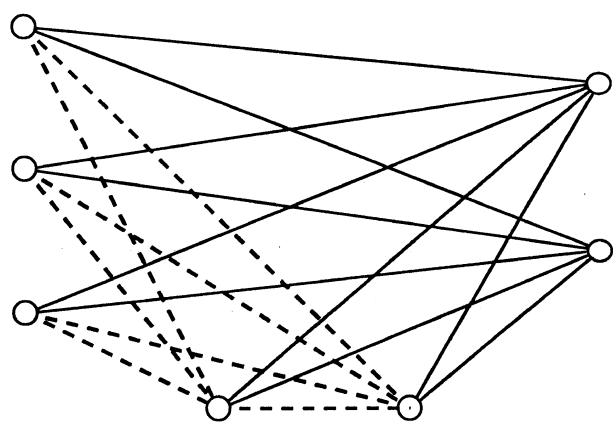
(c) Once training has stagnated, the best candidate unit is selected and connected to the network. Its input weights are frozen and all output weights are trained.



(d) Training a second layer of candidate units.



(e) The best unit is added to the network and all output weights are retrained.



(f) The final network with 3 inputs, 2 outputs and 2 hidden units.

Chapter 4. Examples and Analysis

Chapter 2 introduced general neural network structures and their application to system identification problems. Neural network techniques offer many advantages over conventional system identification methods, particularly for nonlinear problems, but practical application of these techniques can be difficult. Determining an appropriate structure for a multilayer feed-forward network is an art at best – more often it is simply a matter of trial and error. Once the number of layers and nodes per layer are determined, adjusting the connection weights with backpropagation may require considerable time before converging to the solution, if learning does not stagnate first. The backpropagation algorithm is nevertheless considerably less computationally intensive than other "conventional" nonlinear identification techniques.

The cascade-correlation architecture, detailed in Chapter 3, presents an algorithm that overcomes some of the limitations of traditional backpropagation networks. Cascade-correlation networks use a network construction approach to training. This eliminates the guesswork in determining an appropriate number of hidden units. Applications to various classification problems have shown considerable decreases in training time compared to backpropagation networks.

This chapter examines the application of cascade-correlation to several simple nonlinear system identification problems – a second order polynomial, a chaotic time series, and forward and inverse robot arm kinematics. The effectiveness of training will be examined in order to gauge the applicability of the cascade-correlation architecture towards identification problems.

4.0.1 algorithm implementation

The "CasCor" computer program used for these simulations is an adaptation of the UNIX C code, written by Matt White [31], to an Apple Macintosh interface. The program incorporates all the features outlined in Chapter 3, including pools of candidate units, caching of unit values during an epoch, and availability of test and validation epochs. The candidate units may use gaussian, hyperbolic tangent, or one of three sigmoidal activation functions. Connection weights are adjusted using the "quickprop" algorithm,proposed by Scott Fahlman,[2] along with several other algorithmic enhancements to improve learning.

4.0.2 methodology

The cascade-correlation models created in this chapter are based on the neural network system identification models described in Chapter 2. The system dynamics are incorporated into the model by feeding back delayed values of the output along with delayed values of the system inputs, as shown in Figure 4.1. Data sets for the following examples were created using the C code in Appendix B. Separate data sets were created for training and testing. The test data set is used for the validation epochs, which also serves to compute the "one-step-ahead" (OHA) prediction error (Equation 2.16) of the network. The "model-predicted-output" (MPO) of the network models, as defined by Equation 2.17, is measured by starting the trained network with an initial value and then feeding the predicted values back into the network inputs.

Plots of the training errors, validation errors, and hidden unit correlations over the course of training are shown for several cases. Each point on these plots represents the error or correlation after training of the output units or of the hidden units was completed. A circle marks the state of the network selected by the validation process as the final network.

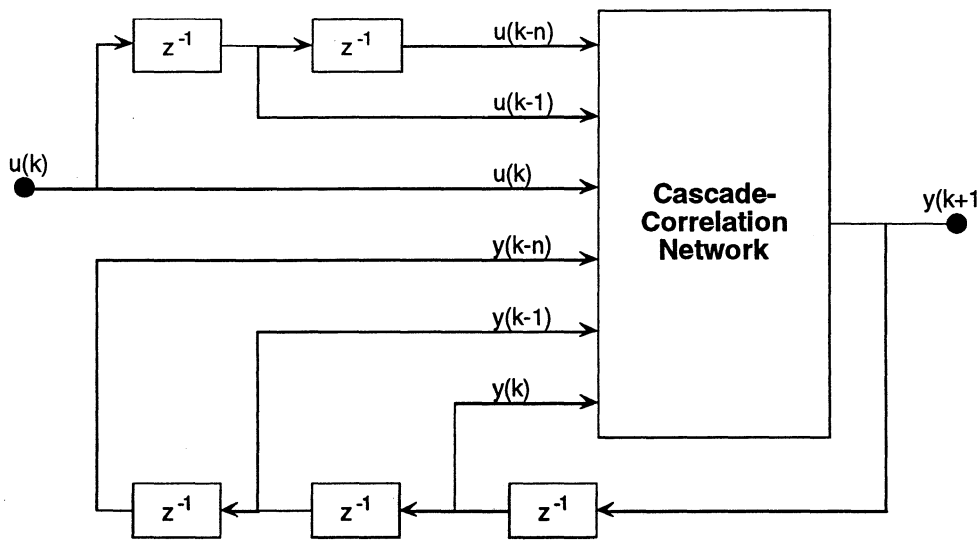


Figure 4.1: Identification model using cascade-correlation neural network. *The system's inputs and outputs are delayed and fed-back to form the network model's inputs.*

4.1 Second Order Polynomial

A simple nonlinear identification problem is the second-order polynomial described by:

$$y_k = 0.8y_{k-1} + 0.2y_{k-1}^2 + u_k - 0.5u_{k-1} \qquad (4.1)$$

A training data set, plotted in Figure 4.2, was created by applying a uniformly random input, $u_k \in [-0.5, 0.5]$, to the system starting at an initial value of $y_0 = 0$. A separate data set for validation was created with an input $u_k = 0.02 \cdot k$ and an initial conditions $y_0 = 0$ and $u_0 = 0$. The training data set contains 50 data points and the validation set contains 25 points. The model prediction capabilities of the network models were tested using $y_0 = 0.5$ and an input of $u_k = 0.02 \cdot k$ for $k = 0$ to 49.

4.1.1 training using squared inputs

One technique to improve the training of artificial neural networks is to preprocess the inputs before they are presented to the network. For example, the y_{k-1} input can be squared so the network's input vector becomes

$$u_{nn} = \begin{bmatrix} u_k & u_{k-1} & y_{k-1} & y_{k-1}^2 \end{bmatrix}, \qquad (4.2)$$
$$y_{nn} = [y_k].$$

Squaring this input value incorporates the function's nonlinearity in the network input vector, effectively reducing identification, in this instance, to a linear problem. The training algorithm must now identify only the polynomial's coefficients, which requires only a linear network. Since no hidden units are needed, training converges very quickly,

achieving a true error of 1.50474e-11 in just 86 epochs. The model-predicted-output (MPO) of the cascade-correlation model, shown in Figure 4.3, is indistinguishable from the desired output of the system. The network weights, shown in Table 4.1, are identical to the polynomial coefficients.

4.1.2 training using delayed inputs

When the identification problem is linear, training is straightforward and the algorithm quickly converges to a solution. The real promise of neural networks, however, lies in their ability to model nonlinear systems. The nonlinear dynamics are incorporated in the model by delaying the system's inputs and outputs to form the network input vector, as shown in Figure 4.1. The network input vector thus becomes:

$$u_{nn} = [u_k \quad u_{k-1} \quad y_{k-1} \quad \cdots \quad u_{k-n} \quad y_{k-n}] \tag{4.3}$$
$$y_{nn} = [y_k].$$

Figures 4.4 and 4.5 show the MPO and OSA predictions for the best model trained. The order, n, of this network is one. Gaussian activation functions are used for the hidden units, while linear activations are used for the output units. Both prediction methods show the model to be a good approximation of the polynomial system. A plot of the error index values of the training and validation sets, Figure 4.6, shows that the validation error index fluctuates wildly during learning, while the training error index steadily decreases until leveling off. A total of 26 units are added to the network before the validation process returns the network to its minimum error state of five hidden units.

Many other networks were trained for this example. The errors and learning statistics (i.e. training time, the number of hidden units, and the number of training epochs) are

tabulated in Table 4.2. A candidate pool size of 10 was used for all cases. The results in this table suggest several things about the effectiveness of training. First, networks trained using similar parameters varied greatly in their accuracy as models. Best results were consistently obtained using validation epochs and gaussian activation functions. In several instances, the validation error would reach a minimum, and then fluctuate for the remainder of training while the output error continued to decrease. This suggests the validation epoch works well in keeping the model from becoming simply a high order curve fit of the training data.

Table 4.3 shows the results of three sets of networks that were trained for the polynomial example using different candidate unit pool sizes. The networks were trained using identical parameters, varying only in randomly selected initial weights and in the size of the candidate pools. Five networks were trained for each of the pool sizes of 10, 30 and 50 candidate units. The averages and standard deviations are tabulated for each of these cases. Increasing the candidate pool size decreases the average error, particularly the MPO error, suggesting the networks learn more consistently with larger pool sizes. The improvement in prediction does not come without a cost, evident in the longer training times required as more candidate units are trained simultaneously.

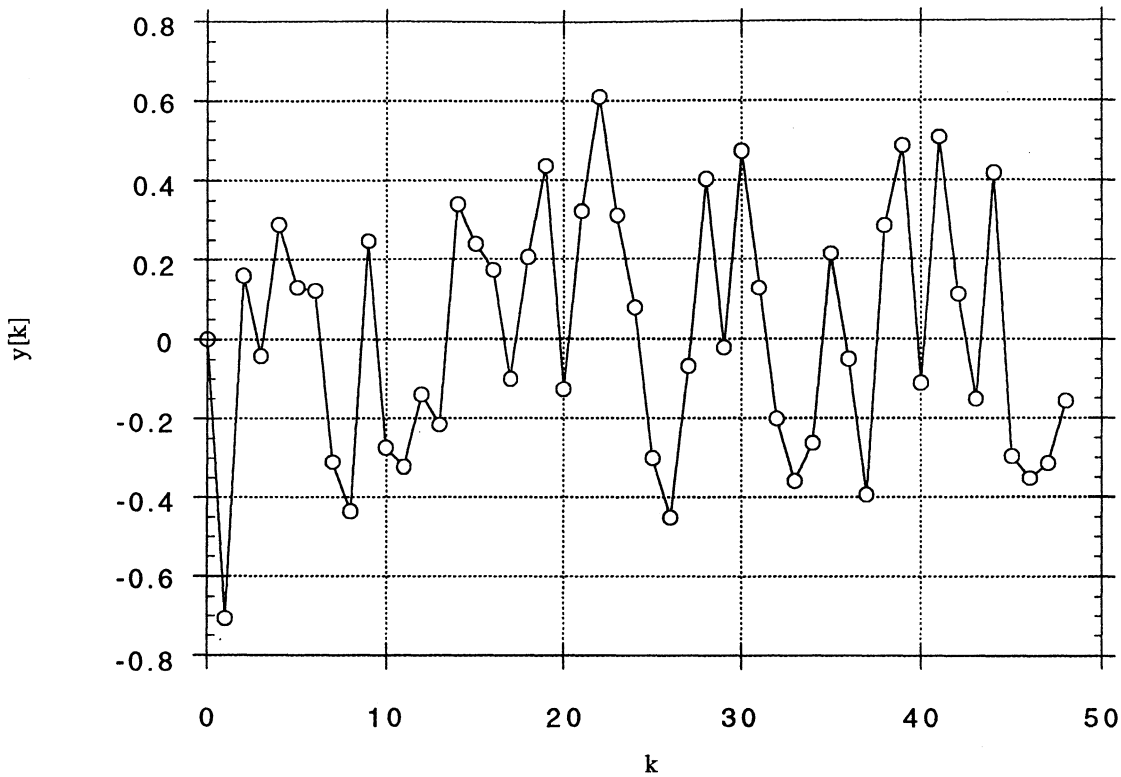


Figure 4.2: 2nd order polynomial training data set.
Data set created using uniformly random $u[k]$ *in the range* [-0.5, 0.5] *and an initial value of* $y[0] = 0$.

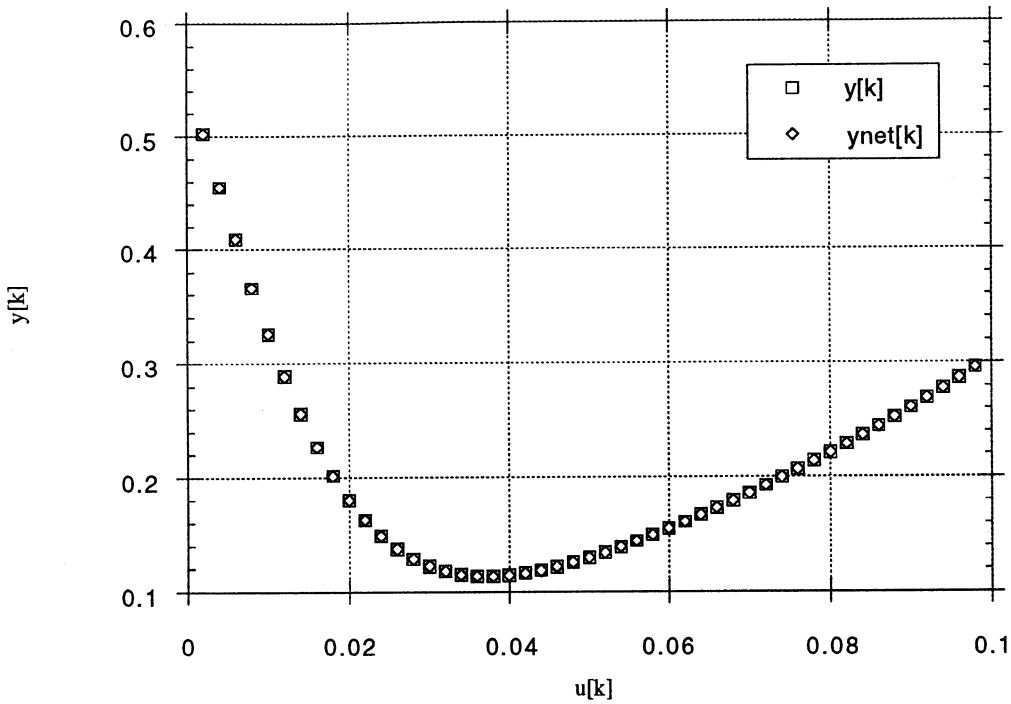


Figure 4.3: Model-predicted-output of 2nd order polynomial model.
Example A - neural network model input vector includes $y[k\text{-}1]^2$ term.

Table 4.1: Polynomial coefficients and CCNN weights.
Example A - neural network model input vector includes $y[k\text{-}1]^2$ term.

	bias (1.0)	u[k]	u[k-1]	y[k-1]	y[k-2]
weights	0.000001	0.999998	-0.49998	0.799981	0.199995
coefficients	0	1	-0.5	0.8	0.2

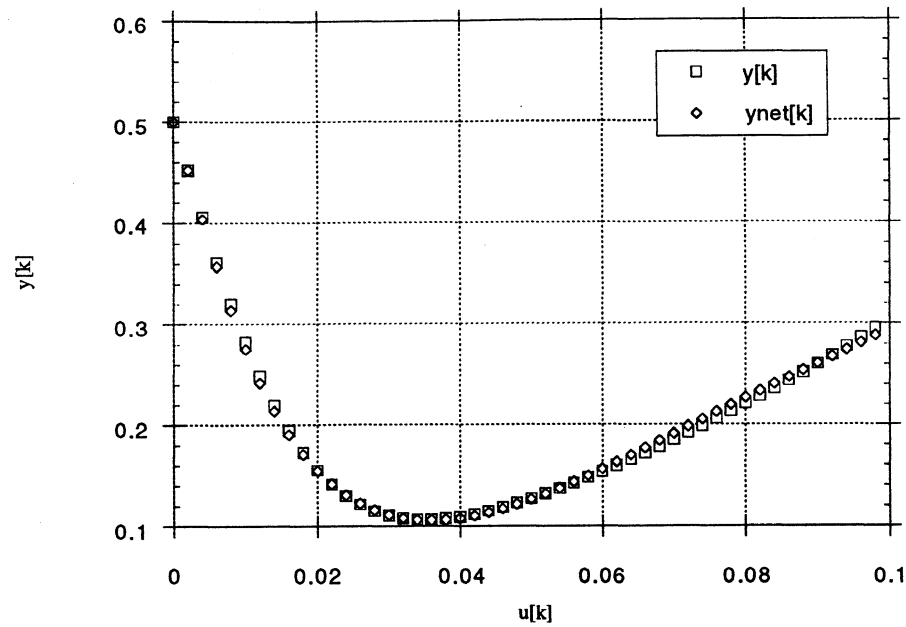


Figure 4.4: Model-predicted output of 2nd order polynomial model.
Example B - one delayed input/output: $E_{MPO} = 7.1781x10^{-5}$

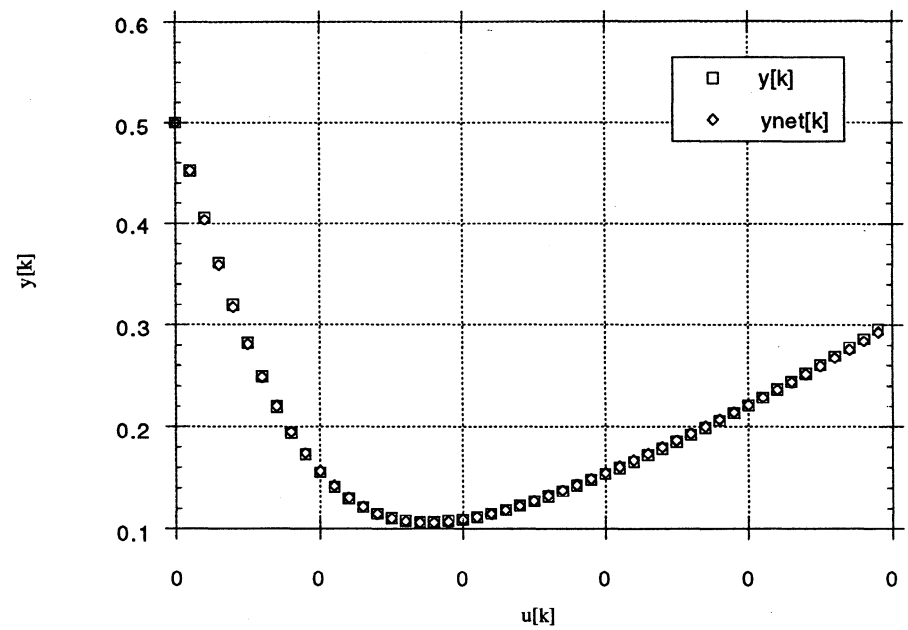


Figure 4.5: One-step-ahead prediction of 2nd order polynomial model.
Example B - one delayed input/output: $E_{OSA} = 2.7813x10^{-5}$

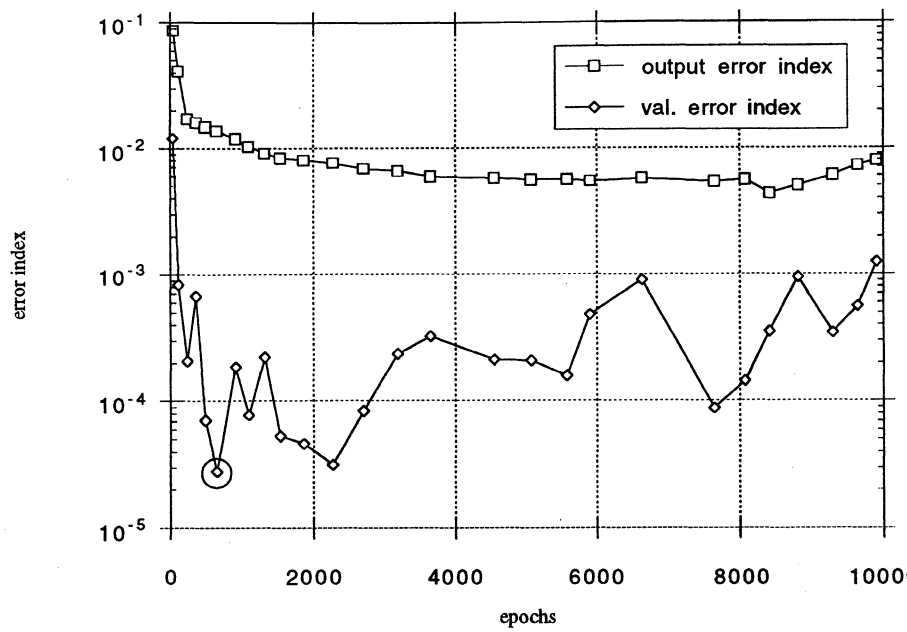


Figure 4.6: Output training and validation epoch errors.
Example B - the circle marks the minumum state of the network as determined by the validation epoch.

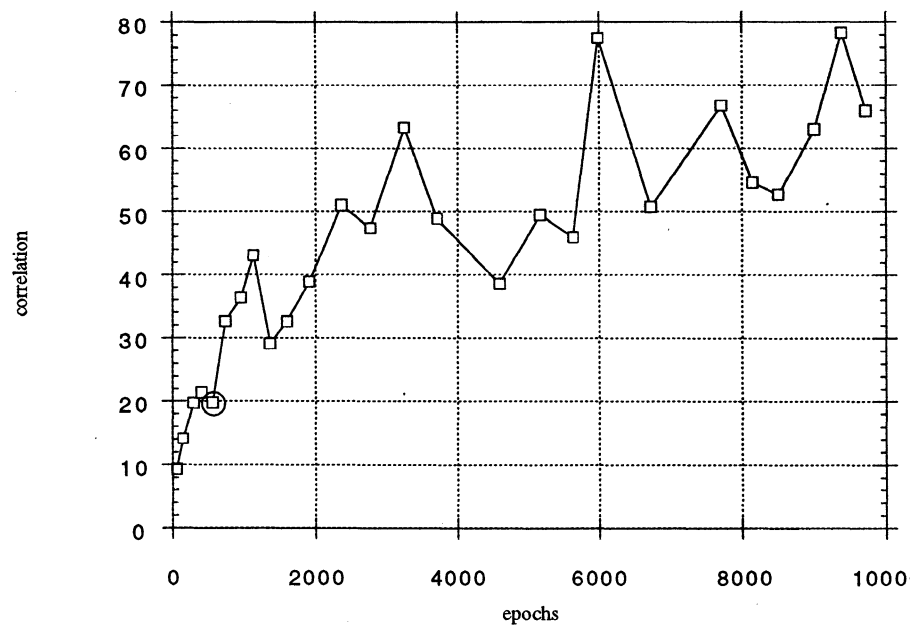


Figure 4.7: Hidden unit correlations.
Example B - error measured after hidden unit added to network.

Table 4.2 2nd-order polynomial models.

	True Error			train. time	hidden	max	new	
	OSA	MPO	epochs	(h:mm:ss)	units	units	unit	Notes
1	2.7913E-05	0.000717809	9905	0:47:18	5/50	50	G	
2	7.2860E-06	0.003307029	7694	0:32:32	11/30	30	G	
3	3.7276E-05	0.005747048	2775	0:22:05	2/30	30	G	
4	1.2060E-05	0.00660488	8506	0:51:34	14/30	30	G	
5	5.3909E-05	0.008476427	3586	0:26:00	4/25	25	G	randwire, n=3
6	2.9355E-05	0.009213129	4722	0:42:05	8/25	25	G	randwire, n=3
7	7.3476E-05	0.01276018	3661	0:23:47	5/20	20	G	
8	5.5201E-04	0.014187725	8159	0:24:14	20/20	20	S	no validation
9	4.5510E-04	0.022415927	2227	0:11:47	10/10	10	G	no validation
10	1.9850E-03	0.026172219	8723	0:25:55	20/20	20	V	
11	1.8585E-04	0.041340465	1819	0:10:52	10	10	S	no validation
12	6.4477E-04	0.042675869	4567	0:16:02	4	30	V	
14	3.4994E-04	0.073038528	10006	4:45:00	30	30	G	
15	2.5616E-04	0.086564475	30403	9:06:07	3	50	G	randwire, n=0
16	6.5942E-04	0.096842323	3614	0:13:49	2	30	V	
17	2.3356E-03	0.116495608	885	0:04:55	5	5	V	no validation, randwire n=0
18	3.1158E-04	0.122071171	6888	1:54:29	3	30	G	
19	1.0814E-03	0.170630795	16407	0:39:25	25	10	T	no validation
20	2.4840E-03	0.230748827	2990	0:03:22	10	10	T	no validation
21	2.1509E-02	0.246104554	1352	0:02:47	10	10	T	no validation
22	3.6768E-04	0.257430497	1000	0:01:46	10	10	T	
24	8.5992E-06	0.332586	6936	0:38:07	12	30	G	

Table 4.3: Second-order polynomial models, trained using candidate unit pool sizes of 10, 30 and 50 hidden units.

Candidate Unit Pool Size: 10

trial	OSA error	MPO error	epochs	train time	hid. units
A	1.10E-05	1.38E-02	6845	0:35:07	12
B	9.56E-06	2.55E-02	11760	0:59:51	20
C	2.53E-03	2.84E-01	13808	1:03:27	20
D	2.30E-03	7.02E-01	10142	0:51:12	20
E	1.31E-03	1.44E+00	12325	0:57:00	20
average	*1.23E-03*	*4.93E-01*	*10976.00*	*0:53:19*	*18.40*
std. dev.	*1.21E-03*	*5.98E-01*	*2655.91*	*0:11:07*	*3.58*

Candidate Unit Pool Size: 30

trial	OSA error	MPO error	epochs	train time	hid. units
A	1.45E-04	1.39E-02	4593	0:56:57	3
B	5.19E-05	1.16E-03	4789	0:52:59	2
C	8.89E-04	2.51E-01	3876	1:02:20	2
D	4.31E-04	4.24E-02	4262	0:57:24	2
E	9.37E-05	2.52E-03	4438	0:55:06	2
average	*3.22E-04*	*6.21E-02*	*4391.60*	*0:56:57*	*2.20*
std. dev.	*3.50E-04*	*1.07E-01*	*347.59*	*0:03:29*	*0.45*

Candidate Unit Pool Size: 50

trial	OSA error	MPO error	epochs	train time	hid. units
A	6.98E-05	8.85E-03	9900	2:28:11	5
B	1.50E-05	2.42E-02	5014	2:07:11	3
C	3.76E-06	1.94E-02	10526	3:40:17	11
D	1.56E-03	6.07E-02	7447	3:11:15	11
E	6.93E-05	8.63E-03	10352	3:20:48	11
average	*3.44E-04*	*2.44E-02*	*8647.80*	*2:57:32*	*8.20*
std. dev.	*6.81E-04*	*2.14E-02*	*2379.41*	*0:38:35*	*3.90*

4.2 *Chaotic Time Series*

Another simple, but useful, example is the chaotic time series described by

$$x_{k+1} = 4x_k(1 - x_k) \tag{4.4}$$

A chaotic system is an interesting example because is very sensitive to initial conditions, and to inaccuracies in prediction, which tend to propagate exponentially. This makes it very difficult to predict the behavior of these systems accurately for very long.

Several different data sets were used for training the network. Two data sets were created using sequential data points starting with an arbitrary initial condition of 0.2 – one set containing 50 points, the second containing 200 data points. A plot of 100 points of the sequential series is shown in Figure 4.8. A third data set was created using 200 uniformly random numbers for xk., in the range [0,1]. The resulting network models were tested by computing the OSA prediction errors from $xk. = 0$ to 1 in 0.01 increments, and the MPO error computed for $k = 0$ to 24, starting with an initial condition of 0.2.

Figure 4.9 shows the network model output demonstrating the best overall OSA error, with the corresponding sequential series MPO shown in Figure 4.10. This model predicts well until about $k = 9$. The output of a model with a considerably larger OSA error is plotted in Figures 4.13 and 4.14. Although the larger OSA error seems to suggest the model is not as accurate, the MPO improved, predicting very well up to $k = 14$.

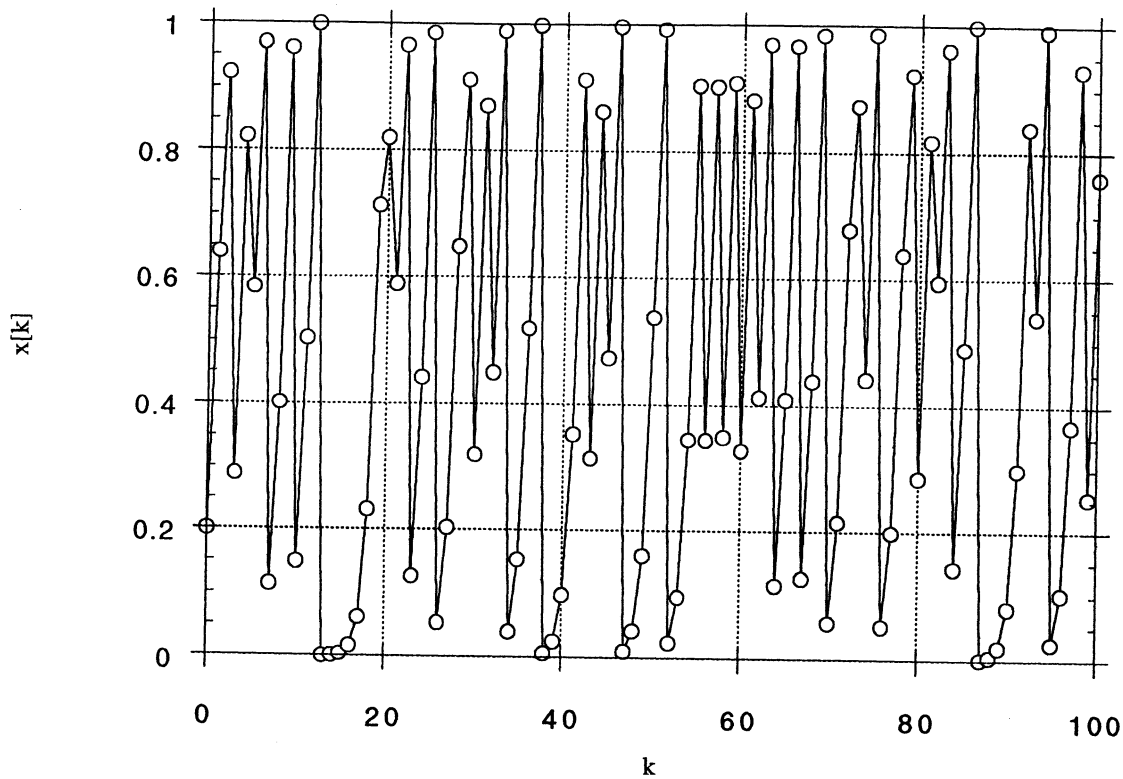


Figure 4.8: Chaos series training data set.
Using an initial condition of $x[k] = 0.2$.

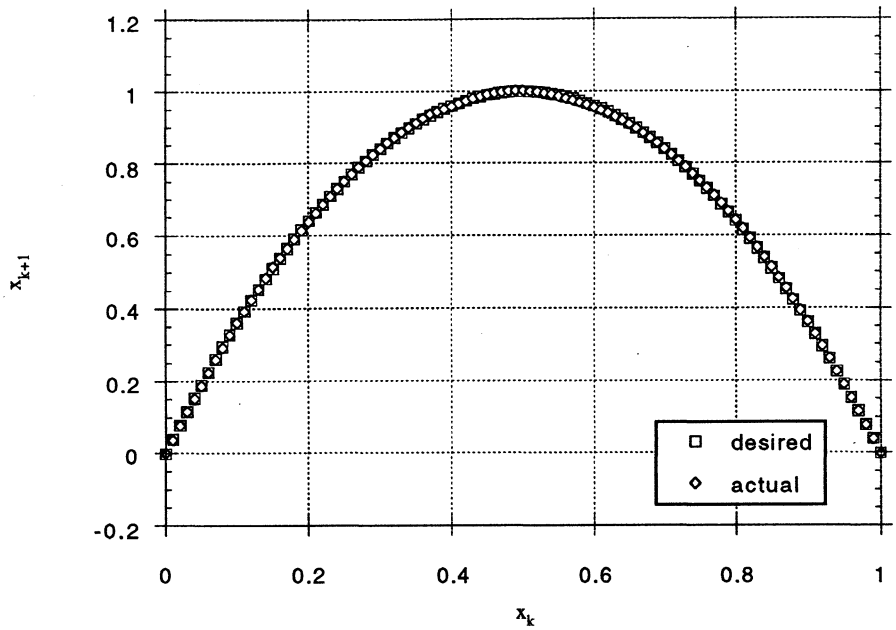


Figure 4.9: One-step-ahead prediction of chaotic time series.
Example C - Model using squared input: $E_{OSA} = 4.1288x10^{-4}$

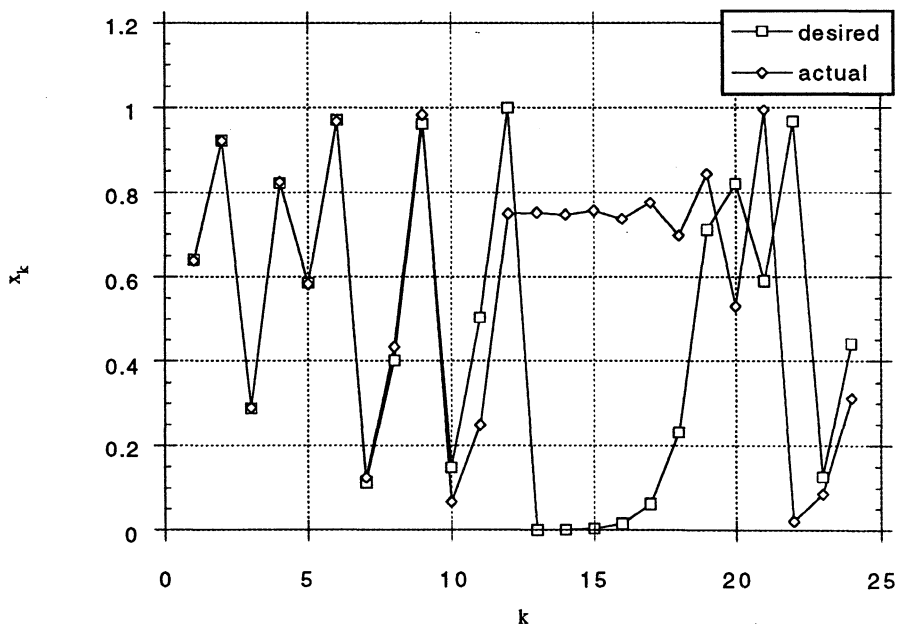


Figure 4.10: Model-predicted-output of chaotic time series.
Example C - Model using squared input: $E_{MPO} = 4.24951563$

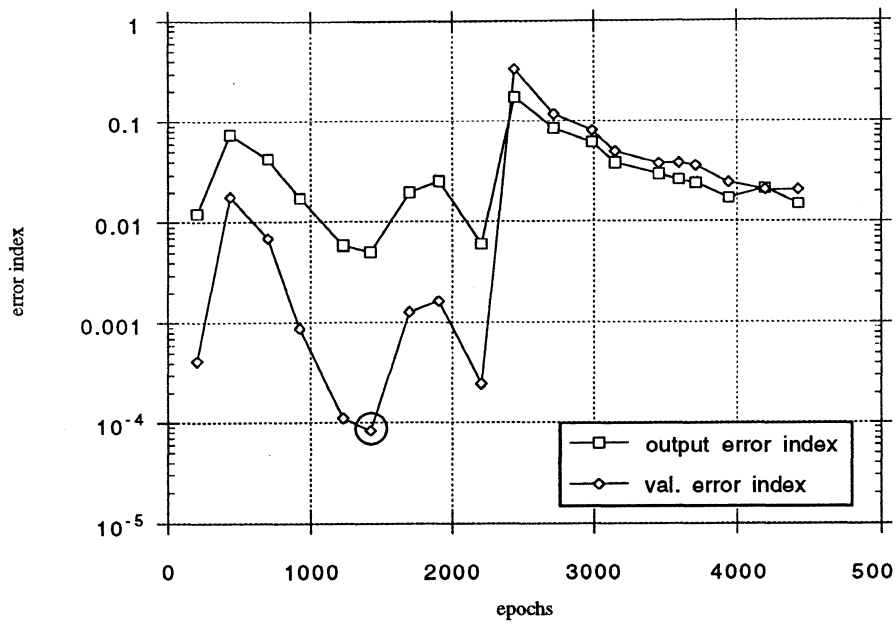


Figure 4.11: Output training and validation epoch errors.
Example C - Network with five hidden units

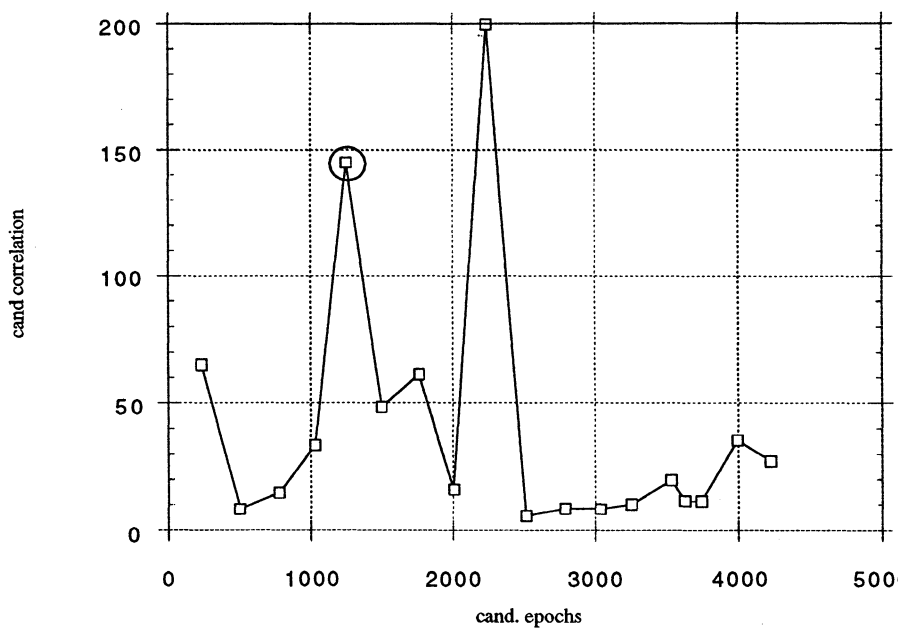


Figure 4.12: Hidden unit correlations.
Example C

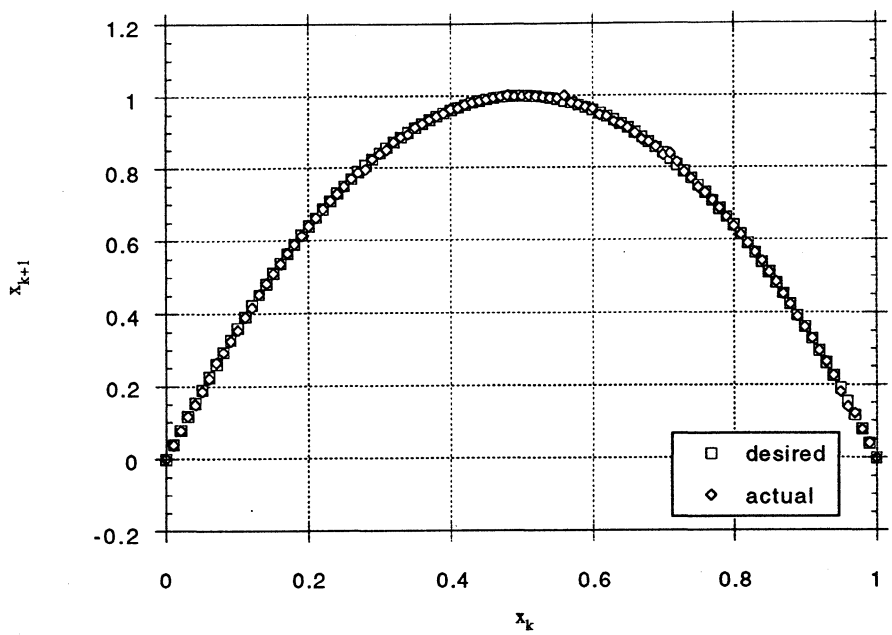


Figure 4.13: One-step-ahead prediction for chaotic time series.
Example D - 50 hidden units, $E_{OSA} = 1.48615x10^{-3}$

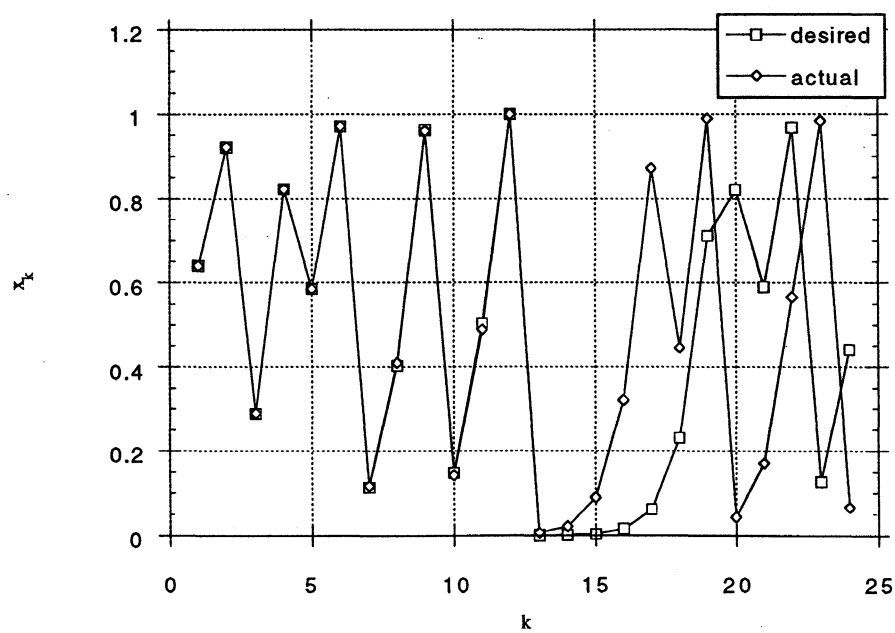


Figure 4.14: Model-predicted-output for chaotic time series.
Example D - $E_{MPO} = 2.69352183$

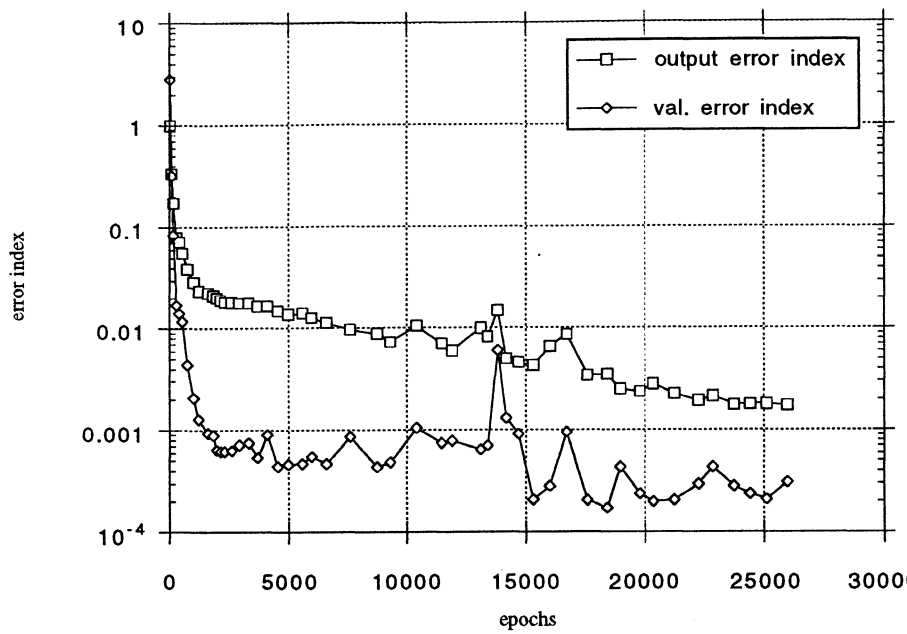


Figure 4.15: Output training and validation epoch errors.
Example D - 50 hidden units

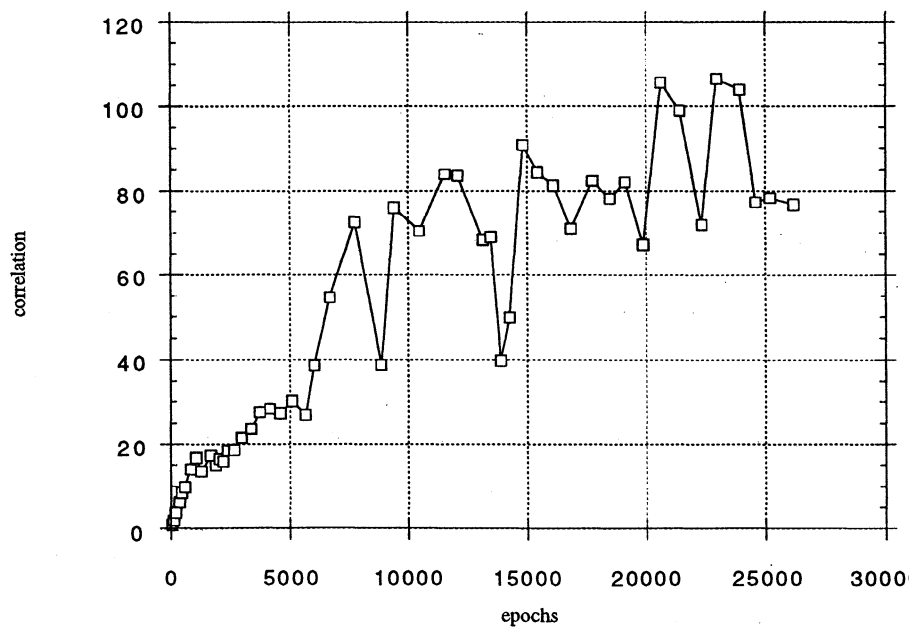


Figure 4.16: Hidden unit correlations.
Example D

4.3 Robot Arm

Another nonlinear identification application involves the modeling of forward and inverse kinematics for a robot arm. The system to be modeled is a 2-degree-of-freedom robot arm, shown in Figure 4.17, with its kinematics described by

$$\begin{aligned} x_{11} &= \cos(\theta_1) \\ x_{12} &= \sin(\theta_1) \\ x_{21} &= x_{11} + \cos(\theta_1 + \theta_2) \\ x_{22} &= x_{12} + \sin(\theta_1 + \theta_2) \end{aligned} \tag{4.5}$$

where $\theta = (\theta_1, \theta_2)$ are the joint angles and,

$\mathbf{X} = (x_{11}, x_{12}, x_{21}, x_{22})$ are the Cartesian coordinates.

Two models are to be created – one to identify the forward kinematics: $\theta \Rightarrow \mathbf{x}$, and one to identify the inverse kinematics: $\mathbf{x} \Rightarrow \theta$.

Training and test data sets for the forward and inverse kinematics are created as shown in Figures 4.18 and 4.19. Test data points are selected between training data points as well outside the region of training, to see how well the models can generalize. The outputs for two models of the forward kinematics are shown in Figures 4.20 and 4.21. Two models of the reverse kinematics are plotted in Figures 4.22 and 4.23. The plots are separated into two regions, indicated by a dashed line. Region A indicates the range of points covered by the training data sets, while the points in Region B lie outside this range. All the network models perform well over Region A. The models accurately interpolated points not explicitly part of the training data set in this Region. The accuracy quickly deteriorates for points that lie in Region B.

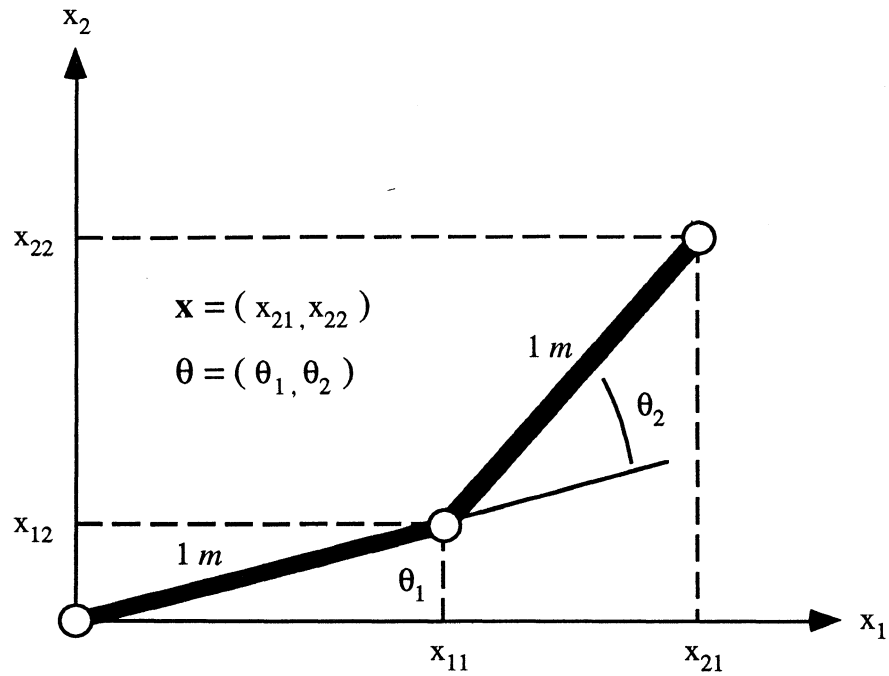


Figure 4.17: A 2-degree-of-freedom robot arm.

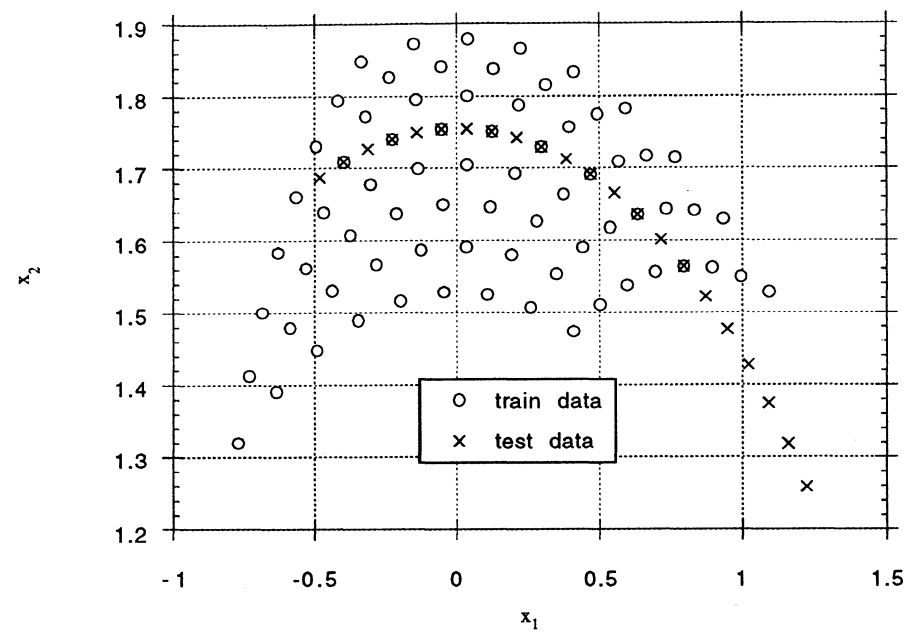


Figure 4.18: Output of robot arm train and test data sets - forward kinematics.

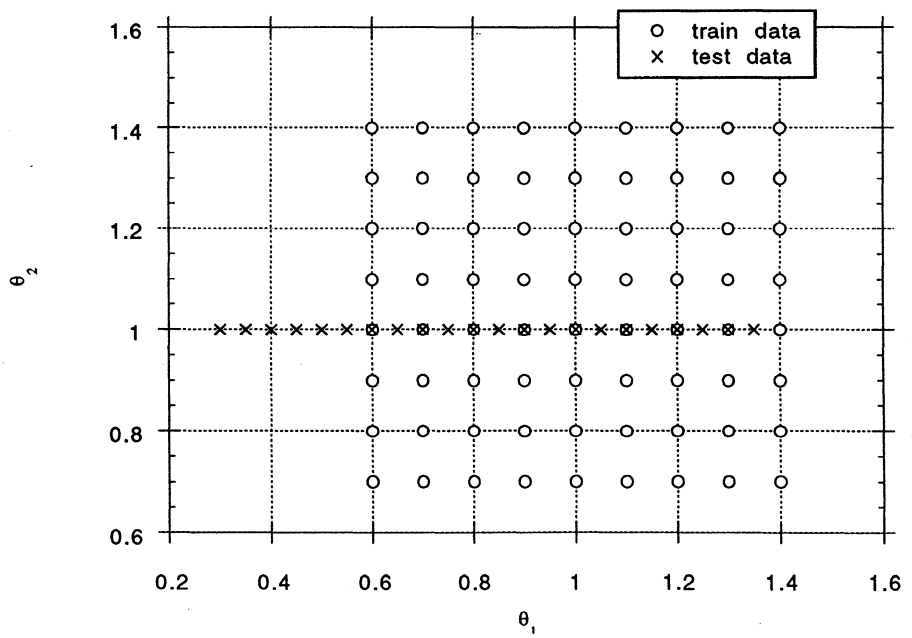


Figure 4.19: Output of robot arm train and test data sets - reverse kinematics.

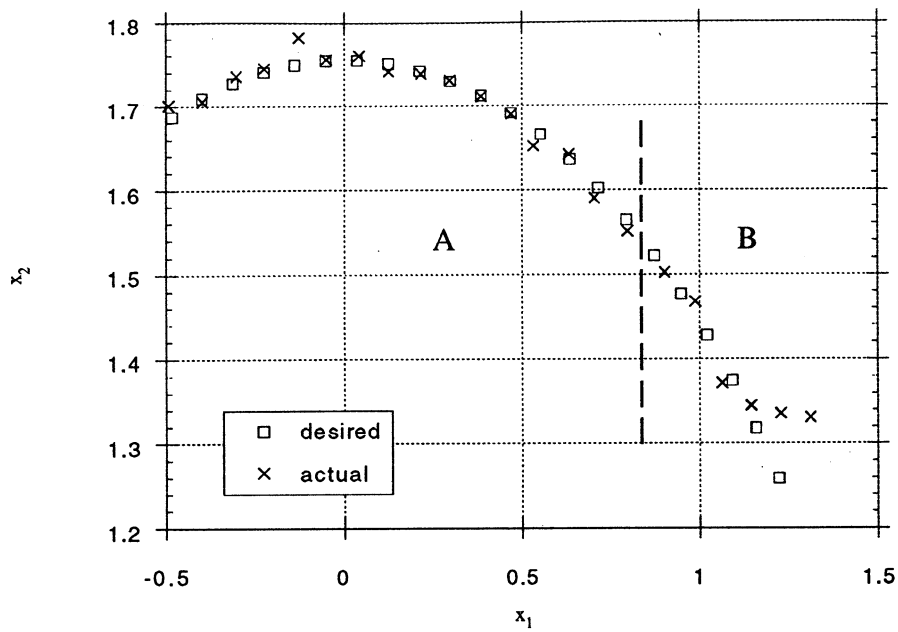


Figure 4.20: Predicted forward kinematics of robot arm model - Example E.
Data points to right of dashed line lie outside of training data set.
Network with 18 hidden units: $E_{true} = 0.03238377$

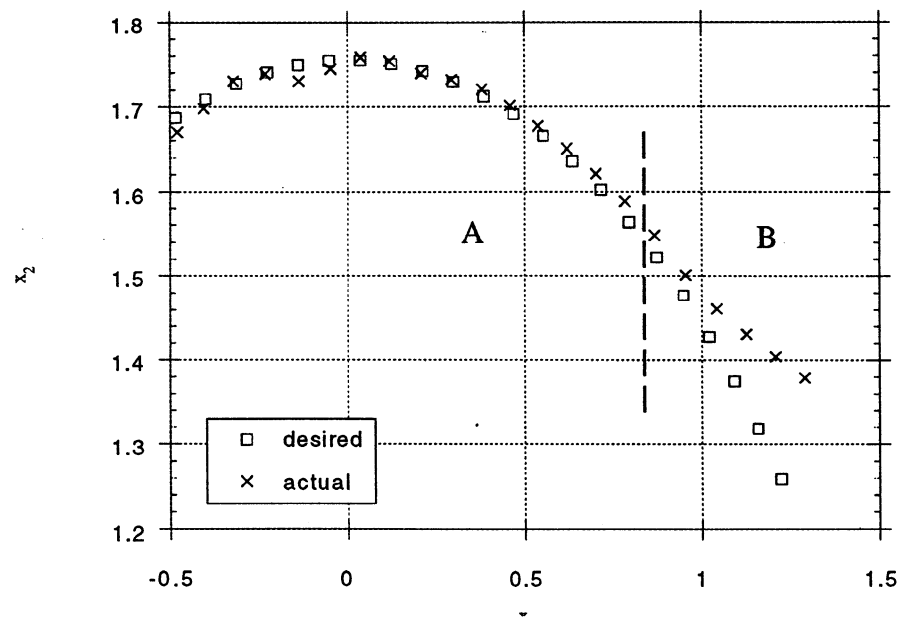


Figure 4.21: Predicted forward kinematics of robot arm model - Example F.
Network with 7 hidden units: $E_{true} = 0.03930181$

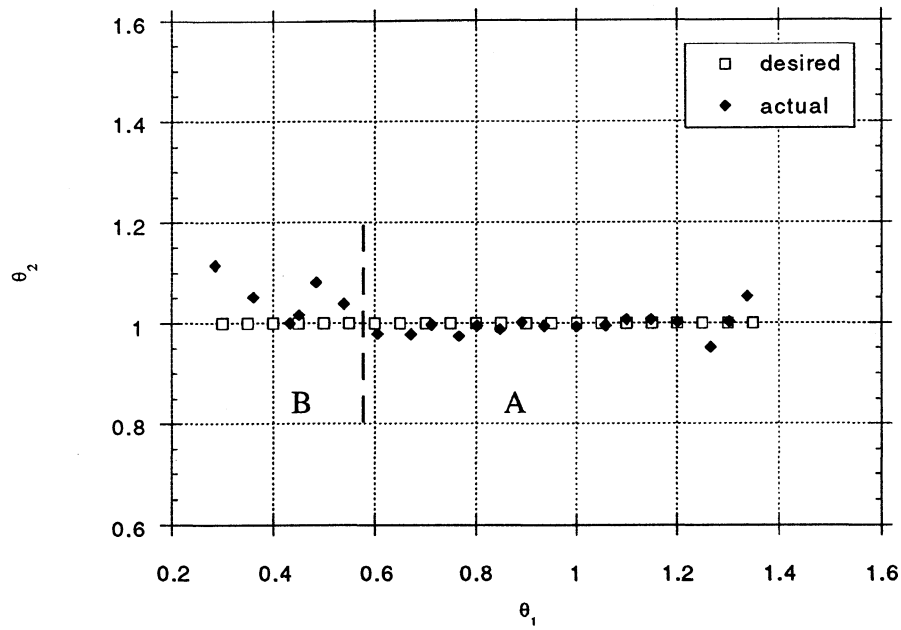


Figure 4.22: Predicted reverse kinematics of robot arm model - Example G.
Data points to left of dashed line lie outside of training data set.
Network with 15 hidden units: $E_{true} = 0.0346352$

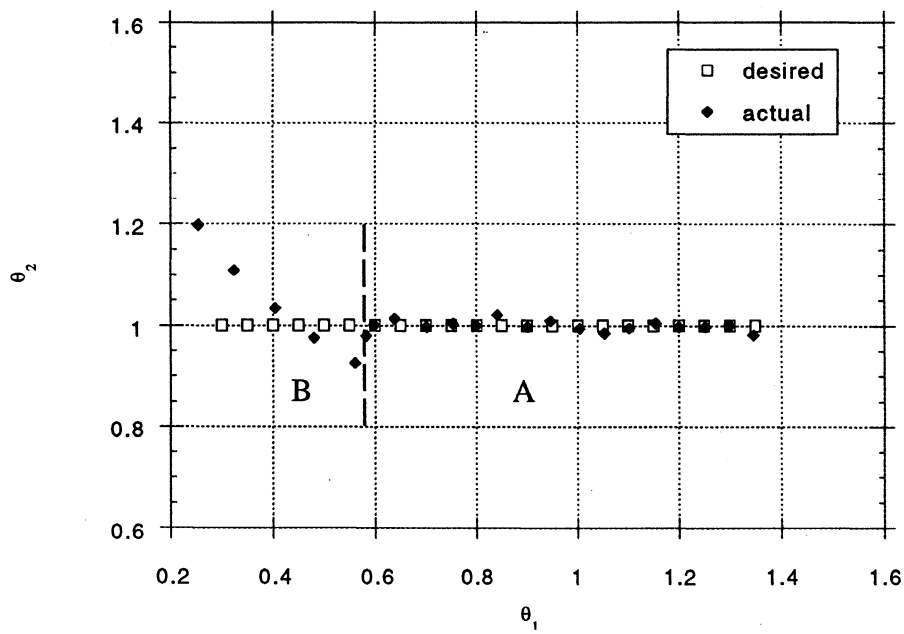


Figure 4.23: Predicted reverse kinematics of robot arm model - Example H.
Network with 36 hidden units: $E_{true} = 0.0693078$

4.4 *Evolution of a Cascade-Correlation Network*

The modular method in which a cascade-correlation network is constructed allows for a convenient look at how the network learns to model a function. This section examines how a network model is built for the chaotic time series problem. A network is trained using a data set of 100 uniformly random points in the range [0,1]. The network's output is tested using an input, $x[k]$ from 0 to 1 in 0.01 increments, similar to that used in section 4.2. The input vector contains $x[k]$ and a bias input of 1, which is to be mapped to the output $x[k+1]$. Linear activations are used for the output nodes of the networks. The candidate unit pool contains two candidate units for each of the nonlinear activation functions.

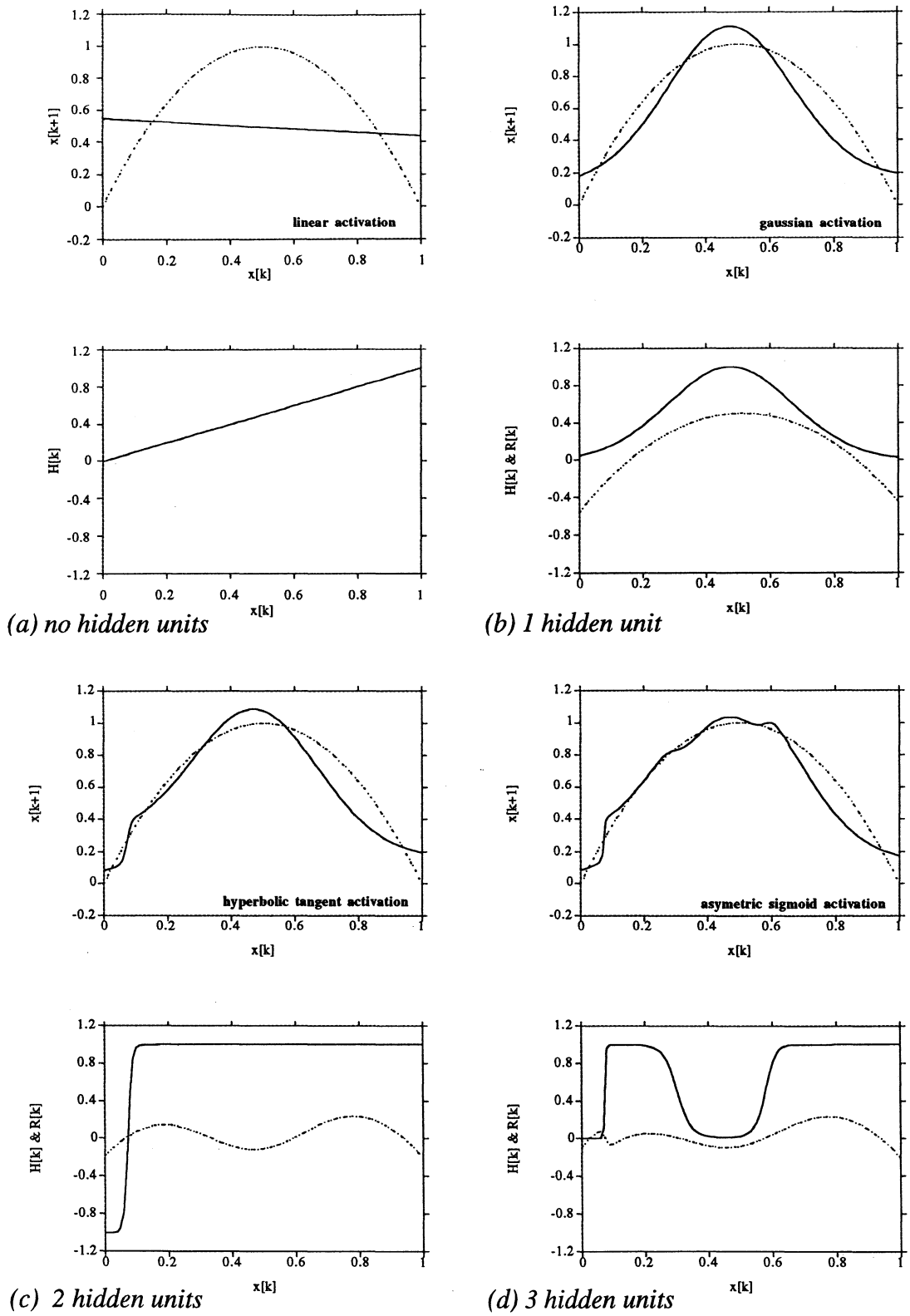
The outputs of the test epochs for each added hidden unit added to the network are shown in Figures 4.24(a)-(n). Two plots are provided for each unit – the upper plot shows the output of the network (solid line) compared to the desired response (dashed line), while the lower plot shows the hidden unit's output (solid line) with the residual error the new unit is attempting to minimize (dashed line).

A large part of the residual error is eliminated within the first 1000 epochs, out of a total of 7739 epochs used for training the network. Six of the 15 hidden units are added to the network during this time. Several properties of the cascade-correlation architecture ensure quick training during the initial stages. First, only one layer of weights is being adjusted at any given time during training. Since hidden unit connection weights are frozen once they are added to the network they are not constantly retrained. The error is not propagated back through existing layers, thus saving a great deal of computation effort. The cascade connections between hidden units also have a major effect. The plots of the hidden unit outputs, such as Figures 4.24(d) and 4.24(f), show that more complex outputs are possible because of the interconnections between hidden units than are possible with

only a single activation function. Each additional unit can have therefore have a greater effect in reducing the residual error.

The same properties that result in efficient training early on can affect training adversely as the residual error becomes smaller. Even though the output error over the training data set continues to decrease, the error over the test data stagnates, staying fairly constant over the final 5000 epochs. Comparing the output of the hidden units added during the latter stages of training to the residual error to be minimized provides some clues why training tends to slow. The initial stages of training are very effective in minimizing the residual error, but as this error becomes small the task for each new unit becomes increasingly difficult. This is due in part to the nature of the algorithm for training candidate hidden units. The residual error is minimized by maximizing the correlation between the candidate unit's output and the error. This criterion seems to train the connection weights so the hidden units operate mostly in their saturation regions. The complex outputs produced by the cascade connections add to the problem. The hidden unit outputs tend to jump from one extreme to the other, as shown in Figures 4.24(j), 4.24(k) and 4.24(l). This has very little effect in eliminating any of the residual error, and in some instances it introduces discontinuities and other errors. The result is training that is very "zig-zag" and slow to converge. The final network model, as shown in Figure 4.24(m), is "unsmooth" over parts of the output.

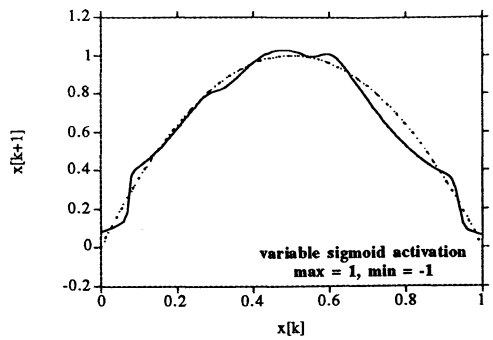
Training for this example was stopped after 15 units were added. Adding additional units might eventually smooth out the "rough spots," as was the case for the network model shown in Figure 4.13, which contained 50 hidden units. Large networks present other problems, most notably the high fan-in of connections from preexisting units to the new candidate unit. The deep networks this creates not only require increased training times for each candidate unit, it also makes hardware implementation impractical.

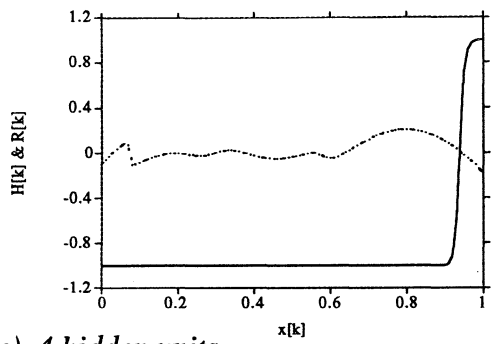


(a) no hidden units *(b) 1 hidden unit*

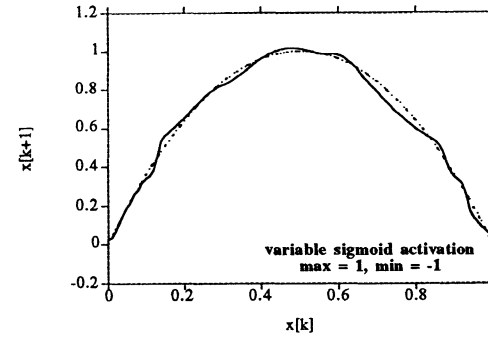
(c) 2 hidden units *(d) 3 hidden units*

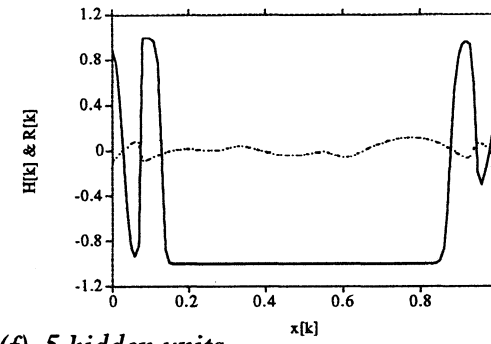
Figure 4.24: Evolution of a cascade correlation network.



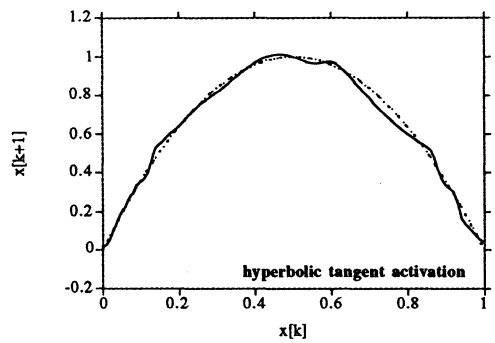


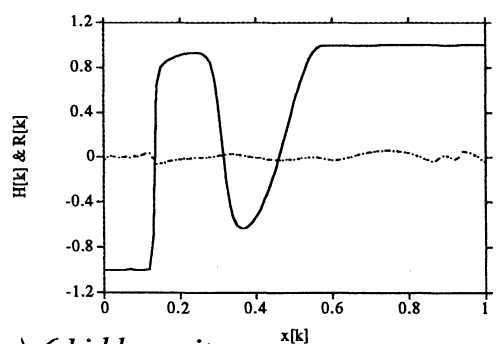
(e) 4 hidden units



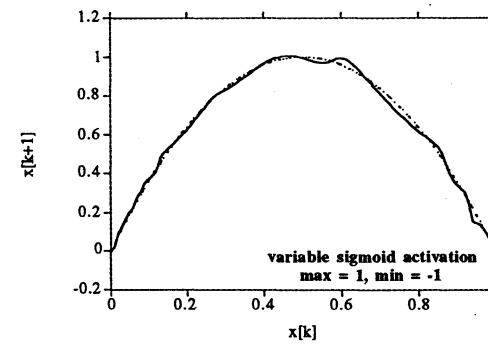


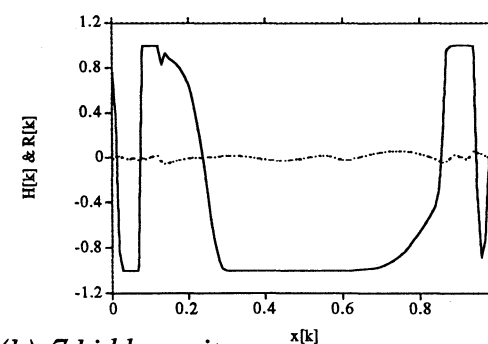
(f) 5 hidden units



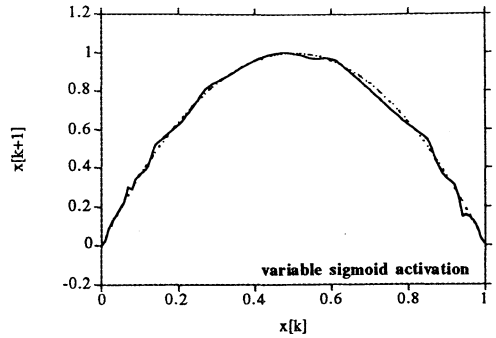


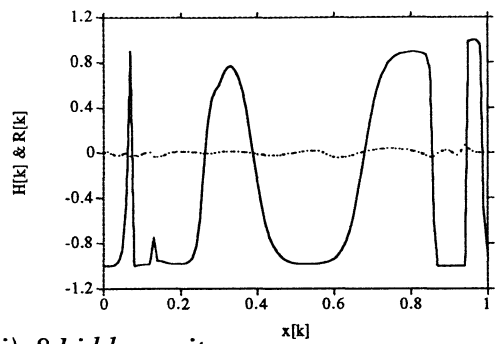
(g) 6 hidden units



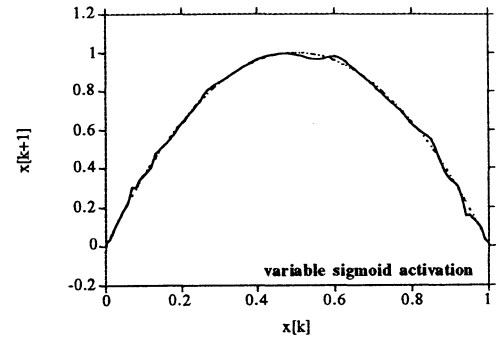


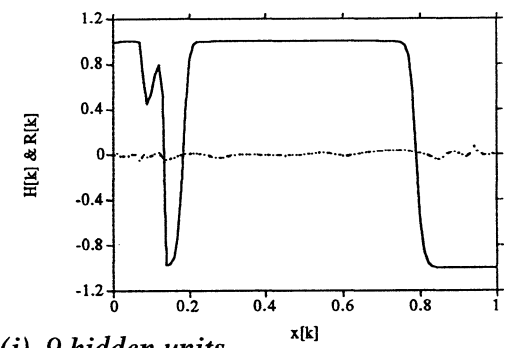
(h) 7 hidden units



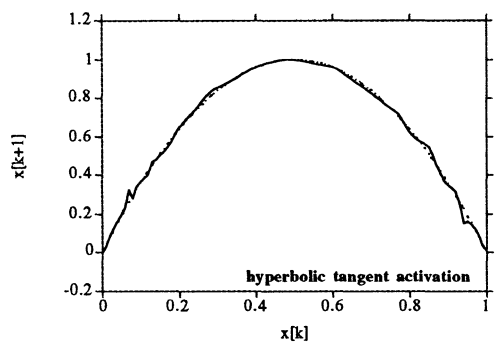


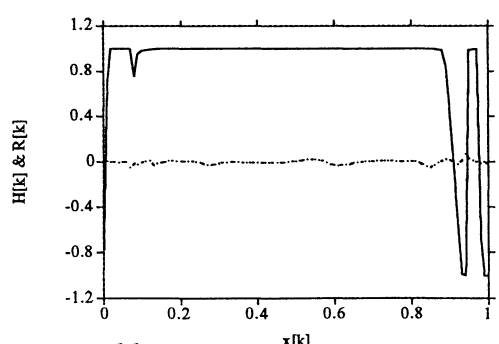
(i) 8 hidden units



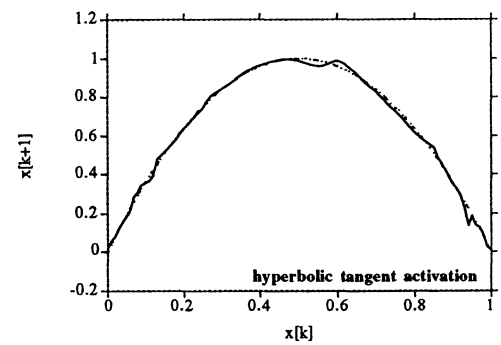


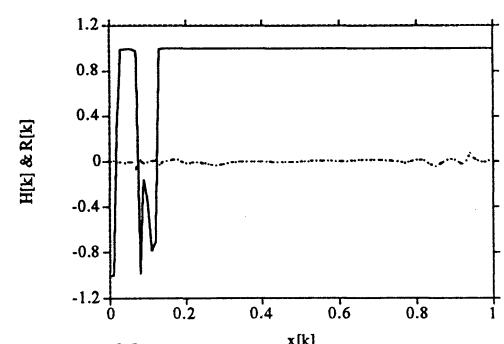
(j) 9 hidden units



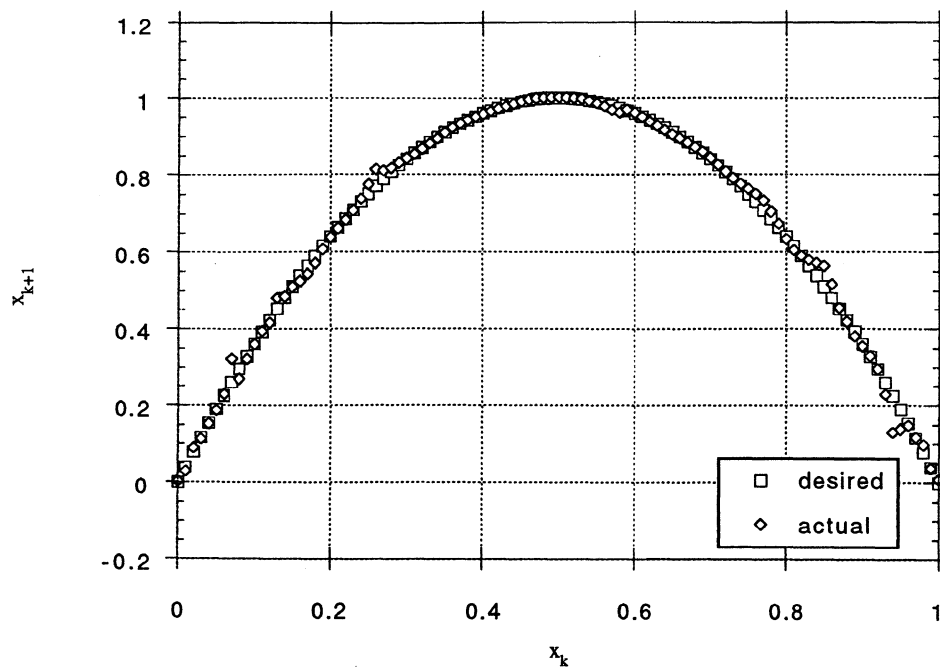


(k) 10 hidden units

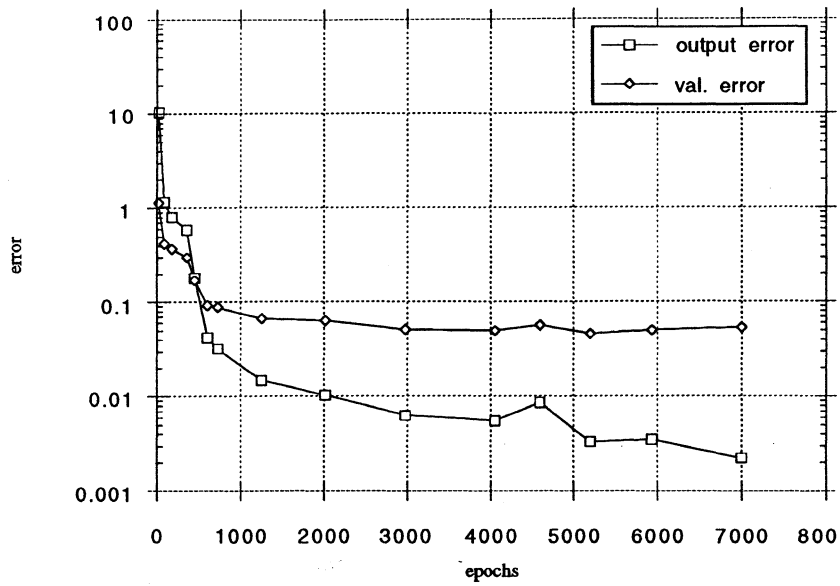




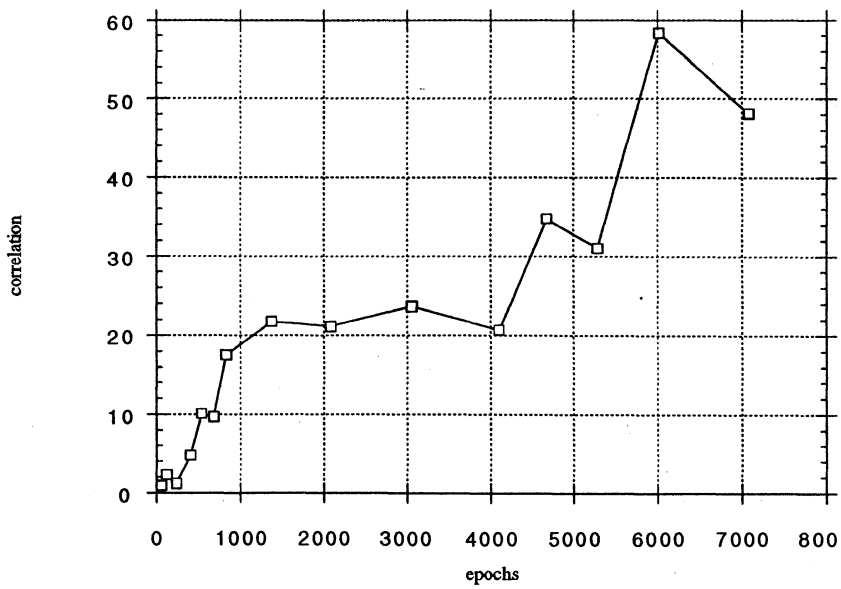
(l) 11 hidden units



(m) Final network output - 15 hidden units



(n) Output units training and validation errors



(o) Hidden unit correlations

4.5 *Comments on Cascade-Correlation Training*

The models shown for the previous examples represent the best results from a number of training trials run with varying degrees of success. This section deals with some of the factors that influence the effectiveness and consistency of cascade-correlation training. Some suggestions that may improve the algorithm for system identification problems are also offered.

4.5.1 algorithm parameters

Most of the algorithmic parameters had very little effect on how well a network trained. The learn rate, for example, is much less sensitive for cascade-correlation than it is in backpropagation. This is due, in large part, to the adaptive step sizes used by "quickprop." The specified learn rate is used only to initialize the algorithm, or when the step size determined by "quickprop" becomes zero. Other parameters such as the maximum growth factor and the weight decay term simply keep the weights from becoming too large. Different values for the parameters described in Section 3.1.4 had little noticeable effect on the efficiency of training. Care should be taken, however, to ensure the maximum number of epochs allowed for each stage of training is sufficiently large so training mostly stagnates before reaching this point.

One parameter that does seem to effect training is the candidate unit pool size. Although training times were longer for larger pool sizes, the results in Table 4.3 suggest large pools of candidate increase the consistency of training. Using a large pool improves the chances a good model is created. Most of the networks trained for these examples used only pools of 10 to 40 hidden units. Using pools of candidate units also becomes more attractive if parallel processing is available, allowing for subgroups of candidate units in the

pool to be trained simultaneously. Using such techniques would allow large candidate pools to be trained much more quickly.

4.5.2 validation

The algorithm consistently chose better models when validation epochs were used. In many cases the validation error would reach a minimum and then either stay constant or fluctuate wildly while the training error continued to steadily decrease. This suggests subsequent hidden units acted only to overfit the network to the training data. The use of validation therefore ensures the network does not overtrain, which is especially important for function approximation applications. The resulting networks, however, were not always accurate models. Validation does not necessarily produce an optimal or minimal network for a given problem. Network sizes varied greatly even when all other algorithm parameters were kept constant.

4.5.3 cascade connections between hidden units

Several trials were run with networks that did not contain the cascade connections between hidden units. The results suggest how important these connections are to the cascade-correlation architecture for quick and efficient training. A network trained for the second-order polynomial problem, using a candidate pool size of 10 units, required over 30000 epochs and nine hours of training time, before validation selected a final network containing three hidden units. A similar network containing three hidden units, but with cascaded connections, required only a tenth the number of epochs and a total training time of 20 minutes to produce a more accurate model.

4.5.4 improving cascade-correlation training

The examples presented in this chapter suggest the cascade-correlation architecture can be used effectively as a nonlinear system identification technique. The results also show, however, that there are several problems with the algorithm. These include the high fan-in of connections to new units in large networks and the stagnation of learning in some networks.

One problem with cascade-correlation networks, particularly with large networks, is a high fan-in of connections to new hidden units. Connecting all hidden units together can lead to very deep networks with many layers. While some information received from preexisting units is useful to a candidate unit, too much or unrelated information may adversely affect the role the new unit takes on towards the overall solution. This has the effect of degrading the efficiency and the learning speed as a network gets larger. Limiting the number of connections could be especially useful during the latter stages of training, when the residual error is small, helping new units to be more effective. Two recent papers have suggested variations of cascade-correlation that limit the number of cascade connections between hidden units to reduce the fan-in. One such method is "limited fan-in random wiring," suggested by Klagges and Soegtrop.[37] The idea is to limit the number of connections between units to a user defined value, c. The cascade connection patterns to a candidate unit are randomly selected (random-wired) from all preexisting units. Each candidate in a pool receives different random connection patterns. Selecting the best combination of random connections is accomplished by using a large number of candidate pools. Klagges and Soegtrop claimed this method produced more accurate and more compact networks, with better generalization capabilities, on the two-spiral problem.

"Iterative atrophy" is a similar approach to limiting the number of connections between hidden units proposed by Smotroff, Friedman and Connolly.[38] This approach re-

moves less useful connections through a process of competition and atrophy. The saliency of the weights is analyzed as the candidate units are trained. Connections that are determined to be weak are eliminated from the network. One advantage of "iterative atrophy" is the more analytical approach the algorithm takes to eliminating connection weights, as opposed to the random elimination of connections used by Klagges and Soegtrop.

Freezing the weights of the hidden units once they are added aids the cascade-correlation to train more quickly than backpropagation networks. Not only do candidate units see a constant environment, but training is performed on only one layer of weights. In some instances, it might be beneficial to perform some "backfitting" on all existing weight connections. This might help a cascade-correlation network to more efficiently eliminate the small residual errors that occur during the latter parts of training. One simple approach to this is to use a hybrid of the cascade-correlation and backpropagation networks. Training should proceed with cascade-correlation training until it is stopped by validation or otherwise stagnates. The remaining error would then be minimized using backpropagation to "fine-tune" all the weights in the network, particularly the hidden units' weights. How effective this may be in practice remains to be seen.

Chapter 5. Conclusions

In this paper, the application of artificial neural networks to the problem of nonlinear system identification is examined. Neural networks have been of special interest in recent years for identification and control of nonlinear systems because they operate as model-free approximators: i.e., very little *a priori* knowledge is required about the dynamics of the system. Neural network techniques are therefore well suited for universal application to nonlinear systems while "traditional" identification algorithms are often limited to specific nonlinear systems or are too computationally intensive for practical applications.

Feed-forward neural networks have shown several limitations when dealing with nonlinear dynamic systems. The structure of feed-forward networks, i.e., the number of hidden neurons and hidden layers, is determined before training begins and is kept static. Units and layers cannot be added or removed without retraining the entire network. This is further complicated by the difficulty of determining a suitable number of units for a given problem – in most cases it is a matter of guesswork. The backpropagation algorithm, used for training the network connection weights, contributes to the problems with a tendency to converge very slowly to a solution. Backpropagation is also sensitive to the choice of initial conditions and to algorithm parameters such as the learn rate.

Many variations of feed-forward networks and backpropagation have been suggested to overcome these problems. Several of these approaches are incorporated into the cascade-correlation architecture. Cascade-correlation dynamically constructs a network by adding hidden units as needed to a simple initial network. An appropriate structure is thereby determined by the algorithm as training progresses. A variation of backpropagation, called "quickprop," that incorporates second-order derivative information in choosing the gradient-descent path to allow for adaptive step sizes down the error surface, is used to train

connection weights. The combination of these algorithms results in a neural network architecture that has demonstrated considerable performance increases, while less sensitive to user parameters and initial conditions, over traditional static feed-forward networks for a wide variety of problems.

The cascade-correlation network produced good models for three different test cases. Compact, accurate models were trained quickly for each problem. The use of candidate pools, cascade-connections between hidden units, and a validation technique to avoid over-generalization combine to give the cascade-correlation more flexibility and faster training than is possible with traditional backpropagation networks.

In several instances, the cascade-correlation algorithm had difficulty converging to a solution. Training often reached a minimum and deteriorated for subsequent epochs. For other cases training would "oscillate" around the solution with added hidden units contributing new errors while eliminating existing ones. This can be attributed, in part, to a lack of sensitivity of the correlation training mechanism used to train new hidden unit input weights. Improved accuracy was achieved using larger pools of candidate units at the cost of increasing the required training time. Several suggestions were made on algorithm improvements and hybrids that may further improve cascade-correlation training. Despite the problems, the combination of the network construction nature of the algorithm and faster training make cascade-correlation neural networks an attractive option for application to nonlinear system identification and control problems.

Attention Patron:

Page 80 omitted
from numbering

Attention Patron:

Page 81 omitted
from numbering

References

[1] Scott E. Fahlman and Christian Lebiere, "The Cascade-Correlation Learning Algorithm," Artificial Neural Networks: Concepts and Theory, IEEE Computer Society Press, 1992, pp 286-294.

[2] Scott E. Fahlman, "An Empirical Study of Learning Speed in Back-Propagation Networks," CMU Technical Report CMU-CS-88-162, 1988.

[3] D.O. Hebb, Organization of Behavior, New York: Science Editions, 1961.

[4] M.L. Minsky and S. Papert, Perceptrons: An Introduction of Computational Geometry, M.I.T. Press, Cambridge, MA, 1969; 2nd ed., 1988.

[5] P.J. Werbos, "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences," Ph.D. Dissertation in Statistics, Harvard University, August 1974.

[6] D.B. Parker, "Learning Logic," Invention Report S81-64, File 1, Office of Technology Licensing, Stanford University, Stanford, CA, 1982.

[7] D.E. Rumelhart, G.E. Hinton, and R.J. Williams, "Learning Internal Representations by Error Propagation," in Rumelhart, D.E. and McClelland, J.L., Parallel Distributed Processing: Explorations in the Microstructure of Cognition, MIT Press, 1986.

[8] Philip D. Wasserman, Neural Computing: Theory and Practice, Van Nostrand Reinhold, New York, 1989.

[9] Kumpati S. Narendra and Kannan Parthasarathy, "Identification and Control of Dynamical Systems Using Neural Networks," *IEEE Trans. on Neural Networks*, Vol. 1, No. 1, March 1990, pp 4-27.

[10] B. Widrow and S.D. Stearns, Adaptive Signal Processing, Prentice-Hall, Englewood Cliffs, N.J., 1985.

[11] Robert A. Jacobs, "Increased Rates of Convergence Through Learning Rate Adaptation," *Neural Networks*, Vol.1, 1988, pp 295-307.

[12] Plaut, D. C., Nowlan, S. J., and Hinton, G. E., "Experiments on Learning by Back-Propagation," Technical Report CMU-CS-86-126, Carnegie-Mellon University, Computer Science Dept., Pittsburgh, PA, 1986.

[13] Franzini, M. A., "Speech Recognition with Back Propagation," In *Proceedings, Ninth Annual Conference of IEEE Engineering in Medicine and Biology Society,* 1987.

[14] H. F. Vanlandingham, S. Bingulac and M. Tran, "A Comparison of Conventional and Neural Network Approaches to System Identification," *Control-Theory and Advanced Technology*, Vol. 9, No. 1, pp 77-97, March 1993.

[15] N.K. Sinha and B. Kuszta, Modeling and Identification of Dynamic Systems, Van Nostrand Reinhold, New York, 1983.

[16] Girish Govind and P.A. Ramamoorthy, "Multi-layered Neural Networks for Arbitrary Approximation: An Explanation and Simulations," Intelligent Engineering Systems Through Artificial Neural Networks, ASME Press, New York, 1991, pp 589-594.

[17] S.A. Billings, "Identification of Nonlinear Systems - A Survey," *IEE Proceedings, Part D*, Vol. 127, No. 6, November 1980, pp 272-283.

[18] J.D. Donne and U Ozguner, "A comparative study of neural vs. conventional methods for modeling and prediction," *Proceedings of the 1992 IEEE International Symposium on Intelligent Control*, p. 548-53.

[19] Kurt Hornik, Maxwell Stinchcombe and Halbert White, "Multilayer Feedforward Networks are Universal Approximators," *Neural Networks*, Vol. 2, 1989, pp 359-366.

[20] Tianping Chen and Hong Chen, "Approximations of Continous Functionals by Neural Networks with Application to Dynamic Systems," *IEEE Trans. on Neural Networks*, Vol. 4, No. 6, November 1993, pp 910-918.

[21] S.A. Billings, B. Jamaluddin, and S. Chen, "Properties of neural networks with applications to modelling non-linear dynamical systems," *International Journal of Control*, Vol. 55, No. 1, p. 193-224.

[22] K.S. Narendra and K. Parthasarthy, "Neural networks in dynamical systems," *SPIE Vol 1196 Intelligent Control and Adaptive Systems*, 1989, pp 230-241.

[23] S. Chen and S.A. Billings, "Neural networks for nonlinear dynamic system modelling and identification," *International Journal of Control*, Vol. 56, No. 2, p. 319-46.

[24] S. Chen, S.A. Billings and P.M. Grant, "Non-linear system identification using neural networks," *International Journal of Control*, Vol. 51, No. 6, p. 1191-214.

[25] John G. Kuschewski, Stefen Hui, and Stanslaw H. Zak, "Application of Feedforward Neural Networks to Dynamical System Identification and Control," *IEEE Trans. on Control Systems Technology*, Vol. 1, No. 1, March 1993, pp 37-49.

[26] Peter M. Mills and Albert Y. Zomaya, "A Neural Network Approach to On-Line Identification of Nonlinear Systems," *Cybernetics and Systems: An International Journal*, Vol. 24, 1993, pp 171-195.

[27] L.H. Tsoukalas and A. Akonomopoulus, "Modeling Complex Systems with Artificial Neural Networks," Intelligent Engineering Systems Through Artificial Neural Networks, ASME Press, New York, 1991, pp 581-587.

[28] Kumpati S. Narendra and Kannan Parthasarthy, "Gradient Methods for the Optimization of Dynamical Systems Containing Neural Networks," *IEEE Trans. on Neural Networks*, Vol. 2, No. 2, March 1991, pp 252-262.

[29] Y. LeCun, J.S. Denker, S.A.Solla, "Optimal Brain Damage", Advances in Neural Information Processing II, edited by D.S. Touretzky, pp 398-605, Morgan Kaufmann, 1990.

[30] S.Y. Kung, Y.H. Hu, "A Frobenius Approximation Reduction Method for Determining Optimal Number of Hidden Units", *International Joint Conference on Neural Networks*, Seattle, pp. II 163-168, 1991.

[31] Matt White, computer code CASCOR, Carnegie Mellon University, 1993 (available via anonymous ftp from FTP.CS.CMU.EDU /AFS/CS/PROJECT/CONNECT/ CODE.)

[32] Jihoon Yang and Vasant Honavar, "Experiments with the Cascade-Correlation Algorithm," *1991 IEEE International Joint Conference on Neural Networks-Singapore*, Vol. 3 pp 2428-33.

[33] D. Whitley and N. Karunanithi, "Generalization in Feed Forward Neural Networks," *IJCNN-91-Seattle: International Joint Conference on Neural Networks*, vol. 2, pp 77-82.

[34] Steen Sjogaard, "A Conceptual Approach to Generalization in Dynamic Neural Networks," *Proceedings ofthe IEEE-SP Workshop - Neural Networks for Signal Processing II*, Denmark, 1992, pp 59-68.

[35] A. Smith, J. Chase, "Use of the Cascade-Correlation Neural Network for System Identification," *Third International Conference on Adaptive Structures (Proceedings)*, pp 619-630.

[36] Dale E. Nelson, D. David Ensley, Steven K. Rogers, "Prediciton of chaotic time series using Cascade Correlation: Effects of number of inputs and training set size," *SPIE Vol. 1709 Applications of Artificial Neural Networks III*, 1992, pp 823-829.

[37] Henrik Klagges and Micheal Soegtrop, "Limited Fan-in Random Wired Cascade-Correlation," IBM Research Division, Physics Group Munich Technical Report, (available from anonymous ftp from ftp.cis.ohio-state.edu, /pub/neuroprose.)

[38] Ira G. Smotroff, David H. Friedman and Dennis Connolly, "Large Scale Networks Via Self Organizing Hierarchical Networks," *Proceedings - SPIE The International Society for Optical Engineering, Vol. 1469 Applications of Artificial Neural Networks*, pp 544-550.

[39] Bart Kosko, Neural Networks for Signal Processing, Prentice Hall, Englewood Cliffs, NJ, 1992.

[40] Jenq-Neng Hwang, Shih-Shien You, Shyh-Rong Lay and I-Chang Jou, "What's Wrong with A Cascaded Correlation Learning Network: A Projection Pursuit Learning Perspective," University of Washington Technical Report, (available from anonymous ftp from ftp.cis.ohio-state.edu, /pub/neuroprose.)

[41] Bart Kosko, Neural Networks and Fuzzy Systems, Prentice Hall, Englewood Cliffs, New Jersey, 1992.

Appendix A - Cas-Cor C source code

```
/*      Cascade Correlation Learning Algorithm

        v1.1
        Matt White   (mwhite+@cmu.edu)
        5/31/94

        QUESTIONS/COMMENTS: neural-bench@cs.cmu.edu

        This code has been placed in public domain by its author.  As a
        matter of simple courtesy, anyone using or adapting this code is
        expected to acknowledge the source.  The author would like to hear
        about any attempts to use this system, successful or not.

        This code is currently being maintained by the site contact listed
        above.  If you find a bug, add a useful feature, or discover a hack
         that will increase system performance, please contact the person at
         the address listed above so that the distribution source may be
        modified accordingly.

        This code is a re-engineered version of the C port, by Scott
        Crowder, of the original Lisp code by Scott Fahlman.  Features have
        been added to allow data sets from the CMU learning benchmark
        database to run on this system.

        For an explanation of this algorithm and some results, see "The
        Cascade-Correlation Learning Architecture" by Scott E. Fahlman and
        and Christian Lebiere in D. S. Touretzky (ed.), "Advances in Neural
        Information Processing Systems 2", Morgan Kaufmann, 1990.  A somewhat
        longer version is available as CMU Computer Science Tech Report
        CMU-CS-90-100, available for FTP from FTP.CS.CMU.EDU in directory
        '/afs/cs/project/connect/tr'.

        RELEASE NOTES

        This is the new version of the older Crowder version of the same
        simulator.  Although we no longer support the Crowder version, it is
        still available in the 'code/old' directory.

        The major differences between this program and the previous one are
        that we now support the CMU Learning Benchmark Format.  This program
        accepts both IO and SEQUENCE data files.  Online help is now available
        at the interaction prompt.  In addition, the output has been cleaned
        extensively to make it more readable.  If you find a bug (or just have
        a suggestion) in this code, please send mail to the address above.

        To build this program, unpack the archive and type 'make'.  If you
        desire to use a compiler different from 'cc', add a line that says
        'CC=<compiler>' to the top of the Makefile.  You can also specify
        optimizations on the 'CFLAGS' line.

        Revision Log
        ~~~~~~~~~~~~
        5/31/94         1.1     Added a function to dump results of the test
                                epoch to a file
        3/30/94         1.0.4   Fixed a bug that caused activation prime for
                                varsigmoid units in the candidate layer to be
                               calculated incorrectly.  Thanks to Hugo Silva
                               for pointing this out.
        1/10/94         1.0.3   Fixed a bug that caused validation not to work
                                correctly in multiple trial runs.
        12/7/93         1.0.2   Fixed a bug which could cause learning
```

```
                              disabilities on 64bit machines.
        10/15/93        1.0.1 Fixed inconsistant entries in ParmTable.
                              Fixed bug that caused the last parameter in a
                              configuration file not to be read.
                              Fixed a bug that caused varied candidates
                              not to work.
                              Changed call to 'difftime' to a more low level
                              form.  Apparently, 'difftime' is not available
                              on Sun workstations.
        9/24/93         1.0   Initial release
*/


/*      Include Files  */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <math.h>
#include <time.h>
#include <signal.h>

#include "tools.h"
#include "parse.h"
#include "interface.h"
#include "cascor.h"


/*      Global Variable Declarations  */

net_data     net;              /*  Internal activation and weights        */
net_info     netConfig;        /*  Information on network topology        */
output_data  out;              /*  Information about the network outputs  */
cand_data    cand;             /*  Information about the cadidate units   */
error_data   error;            /*  Network error calculations                  */
parm_data    parm;             /*  Network run parameters               */
cache_data   cache;            /*  Cached activation and error values          */
alt_data_t   val,              /*  Validation data set                         */
             test;             /*  Test data set                      */
run_res_t    runResults;           /*  Totalled results from all trials     */

int       Ncand,                   /*  Number of candidates being trained          */
          Nunits,              /*  Number of units in the network         */
          Ninputs,             /*  Number of inputs to the network        */
          Noutputs,            /*  Number of outputs from the network       */
          NtrainPts,           /*  Number of training points, minus segment */
                               /* markers                                 */
          NtrainOutVals,           /*  Number of training points, times the     */
                               /* number of outputs                       */
          epoch;                   /*  Epoch that has just been calculated      */
unsigned long          connx;  /*  Number of connection crossings         */
boolean   isSeq,                   /*  Is this a sequence data set?          */
          dataLoaded,          /*  Has the data been loaded?              */
          interact,            /*  Interact with the user?                */
          helpAvail,           /*  Is help available?                     */
          interruptPending;    /*  Is there an interrupt pending?         */
```

```
/*      Macro Definitions      */

/*      RANDOM_WEIGHT -  Returns a random value between plus and minus 'x'  */

#define RANDOM_WEIGHT(x) ( x * (random() % 1000 / 500.0) - x )


/*      NO_CACHE -  Signals insufficient memory and then turns off the cache */

#define NO_CACHE  {                                                          \
                  fprintf ( stderr, "\nInsufficient memory for cache.\n" );  \
                   fprintf ( stderr, "Shutting cache down.\n" );              \
                   destroy_cache ( );                                        \
                  parm.useCache = FALSE;                                     \
                   return FALSE;                                                 \
                   }


/*      Function Prototypes    */

void      init_prog           ( void );      /*  Initialization functions */
void      init_parms          ( void );
void      init_vars           ( int * );
void      build_net           ( int );
void      init_net            ( int );
void      init_error          ( error_data * );
void      init_cand           ( void );

void      train_outputs             ( status_t * );       /*  Output training    */
void      output_epoch        ( void );      /* functions          */
void      adjust_weights      ( void );

void      train_cand          ( status_t * );       /*  Correlation machinery  */
void      correlation_epoch   ( void );
void      cand_epoch          ( void );
void      compute_correlations ( void );
void      adjust_correlations  ( void );
void      compute_slopes      ( void );
void      adjust_cand_weights  ( void );
void      install_cand        ( void );

void      validation_epoch    ( float, int, status_t * );
void      test_epoch          ( void );      /*  Network testing code  */
void      dump_results          ( FILE *, float *, float * );

boolean   build_cache         ( int );       /*  Cache code          */
void      destroy_cache             ( void );
void      compute_cache             ( void );
void      recompute_cache     ( int );

/*      Miscellanious useful functions       */

void      forward_pass        ( int, int, data_set );
void      setup_inputs        ( int, int, data_set );
void      output_pass         ( void );
void      compute_error             ( float *, error_data *, boolean, boolean );
void      quickprop           ( int, float *, float *, float *, float *,
                                  float, float, float, float );
float     activation          ( int, float );
float     activation_prime    ( int, float, float );
float     output_function     ( int, float );
float     output_prime        ( int, float );
```

```
float     std_dev              ( data_set, int, int );


void  main  ( int argc, char *argv [] )
{
  int          trial,                          /*  Current trial           */
               maxUnits;                       /*  Maximum units in network */
  time_t       startTime;                      /*  Time trial was started   */
  status_t     status  = TRAINING,     /*  Training status         */
              valStat         = TRAINING,     /*  Validation status       */
               candStat;                       /*  Candidate training      */
                                               /* status                   */

  /*  Identify the program, load data and parameters, check for user       */
  /* changes and then initialize the data structures                */

  prog_id           ( );
  init_prog          ( );
  init_parms         ( );
  exec_command_line  ( argc, argv );
  if  ( interact )
    change_parms ( FALSE );
  else
    list_parms  ( );
  init_vars          ( &maxUnits );
  build_net          ( maxUnits );

  /*  Repeatedly run trials until the max is reached  */

  for  ( trial = 0 ; trial < parm.Ntrials ; trial++ )  {

    /*  Initialize for this trial  */

    valStat = TRAINING;
    init_net       ( maxUnits);
    output_begin_trial  ( trial + 1, &startTime );
    if  ( parm.useCache )
      compute_cache  ( );

    /*  Keep training until the network reaches it's maximum size  */

    while  ( Nunits < maxUnits )  {

      /*  Train the outputs until they stagnate, time out or you win  */

      train_outputs ( &status );
      output_train_results  ( status );
      if  ( status == WIN )
        break;

      /*  Run validation epoch  */

      if  ( parm.validate )  {
        validation_epoch ( 0.49999, maxUnits, &valStat );
        if  ( valStat != TRAINING )
          break;
      }

      /*  Train a pool of candidate units and add the best one to the net  */

      init_cand      ( );
      train_cand     ( &candStat );
      install_cand   ( );
```

```
      output_cand_results  ( candStat );
    }

    /*  If we have a candidate with untrained outputs, train them  */

    if  ( (status != WIN) && (valStat == TRAINING) )
      train_outputs  ( &status );

    /*  Run test epoch  */

    if  ( parm.test )
      test_epoch  ( );

    /*  Output the results of this trial  */

    output_trial_results ( status, trial + 1, startTime );
    if  ( parm.saveWeights )
      save_weights ( parm.weightFile, interact, trial + 1, startTime );
  }

  output_run_results ( );
  close_help ( );
}


/********************* Initialization Functions      ******************************/

/*      INIT_PROG -  Initializes the random number generator and resets the
        'dataLoaded' flag.
*/

void  init_prog  ( void )
{
  time_t  timer;       /*  Used as a random number seed      */

  time     ( &timer ); /*  Seed the random number generator  */
  srandom  ( timer );

  dataLoaded  = FALSE; /*  Reset the data loaded flag              */
  interact    = FALSE; /*  Reset the interaction flag              */

  helpAvail = init_help ( HELP_FILE );

  interruptPending = FALSE;
  signal  ( SIGINT, trap_ctrl_c );
}


/*      INIT_PARMS -  Defaults all run parameters.  These parameters should
        do OK on two-spirals.
*/

void  init_parms  ( void )
{
  parm.Ntrials         = 1;    /*  Initialize the integers        */
  parm.maxNewUnits             = 25;
  parm.valPatience             = 12;
  parm.winRadius               = 3;

  parm.weightRange             = 1.0;  /*  Initialize the floating point    */
  parm.weightMult              = 1.0;  /* numbers                          */
  parm.sigPrimeOffset          = 0.1;
  parm.bias                    = 1.0;
```

```
  parm.dataFile                         = NULL; /*  Initialize the character strings */
  parm.weightFile               = NULL;

  parm.validate                         = DEF_VALIDATE;       /*  Initialize boolean      */
  parm.test                     = DEF_TEST;     /* variables                */
  parm.useCache                         = TRUE;
  parm.saveWeights              = FALSE;

  parm.candNewType              = SIGMOID;      /*  Initialize new unit type */

  parm.out.epochs               = 200;          /*  Initialize the output    */
  parm.out.patience             = 12;           /* parameters                */
  parm.out.sigMax               = BIN_POS;
  parm.out.sigMin               = BIN_NEG;
  parm.out.epsilon              = 1.0;
  parm.out.decay                = 0.0;
  parm.out.mu                   = 2.0;
  parm.out.changeThresh                 = 0.01;

  parm.cand.epochs              = 200;          /*  Initialize the candidate */
  parm.cand.patience            = 12;           /* parameters                */
  parm.cand.sigMax              = BIN_POS;
  parm.cand.sigMin              = BIN_NEG;
  parm.cand.epsilon             = 100.0;
  parm.cand.decay               = 0.0;
  parm.cand.mu                  = 2.0;
  parm.cand.changeThresh        = 0.03;

  error.indexThresh             = 0.2;          /*  Initialize the error     */
  error.scoreThresh             = 0.4;          /* parameters                */

  Ncand                         = 8;            /*  Initialize the number of */
                                                /* candidates                */
}


/*      INIT_VARS -  This is a second pass at initializing variables.
        Variables initialized here are those that require information stored in
        other variables to calculate.  Thus, we put this off until
        configuration is finalized.
*/

void init_vars ( int *maxUnits )
{
  /*  Calculate shrink factor and scaled epsilon for outputs and shrink  */
  /* factor for the candidates                                             */

  out.shrinkFactor      = parm.out.mu / (1.0 + parm.out.mu);
  out.scaledEpsilon     = parm.out.epsilon / (netConfig.train_pts -
                                              netConfig.train_seg);
  cand.shrinkFactor     = parm.cand.mu / (1.0 + parm.cand.mu);

  /*  Calculate standard deviation for all data sets being used           */

  error.stdDev          = std_dev( netConfig.train, netConfig.train_pts,
                                   NtrainOutVals );

  if  ( parm.validate )
    val.stdDev = std_dev ( val.data, val.Npts, val.NoutVals );
  if  ( parm.test )
    test.stdDev         = std_dev ( test.data, test.Npts, test.NoutVals );
```

```
  runResults.Nvictories              = 0;    /*  Reset run results for this run  */
  runResults.Nepochs          = 0;
  runResults.errorBits        = 0;
  runResults.Nunits           = 0;

  runResults.crossingsSec     = 0.0;
  runResults.percentCorrect   = 0.0;
  runResults.runTime          = 0.0;
  runResults.errorIndex              = 0.0;
  runResults.trueError        = 0.0;
  runResults.sumSqError              = 0.0;

  /*  Calculate the total number of units possible  */

  *maxUnits           = 1 + Ninputs + parm.maxNewUnits;
}


/*      BUILD_NET -  Allocate memory for all network data structures not yet
        allocated.  Allocation assumes that the network can grow to its
        maximum size.
*/

void  build_net  ( int maxUnits )
{
  char *fn     = "Build Net"; /*  Function Identifier       */
  int  i;                     /*  Indexing variable */

  if  ( parm.useCache )              /*  Build the cache    */
    build_cache ( maxUnits );

  /*  Allocate memory for internal activation and weights   */

  if  ( !( parm.useCache ) )
    net.values  = (float *)     alloc_mem ( maxUnits, sizeof( float ), fn );
  net.weights   = (float **)    alloc_mem ( maxUnits, sizeof( float * ), fn );
  net.unitTypes = (unit_type *) alloc_mem ( maxUnits, sizeof( unit_type ),fn );
  for  ( i = Ninputs + 1 ; i < maxUnits ; i++ )
    net.weights [i] = (float *) alloc_mem ( i, sizeof( float ), fn );

  /*  Allocate memory for the outputs                          */

  out.values   = (float *)  alloc_mem ( Noutputs, sizeof( float ), fn );
  out.weights  = (float **) alloc_mem ( Noutputs, sizeof( float * ), fn );
  out.deltas   = (float **) alloc_mem ( Noutputs, sizeof( float * ), fn );
  out.slopes   = (float **) alloc_mem ( Noutputs, sizeof( float * ), fn );
  out.pSlopes  = (float **) alloc_mem ( Noutputs, sizeof( float * ), fn );
  for  ( i = 0 ; i < Noutputs ; i++ )  {
    out.weights [i] = (float *) alloc_mem ( maxUnits, sizeof( float ), fn );
    out.deltas [i]  = (float *) alloc_mem ( maxUnits, sizeof( float ), fn );
    out.slopes [i]  = (float *) alloc_mem ( maxUnits, sizeof( float ), fn );
    out.pSlopes [i] = (float *) alloc_mem ( maxUnits, sizeof( float ), fn );
  }

  /*  Allocate memory for the candidate units                    */

  cand.values    = (float *)     alloc_mem ( Ncand, sizeof( float ), fn );
  cand.sumVals   = (float *)     alloc_mem ( Ncand, sizeof( float ), fn );
  cand.types     = (unit_type *) alloc_mem ( Ncand, sizeof( unit_type ), fn );
  cand.weights   = (float **) alloc_mem ( Ncand, sizeof( float * ), fn );
  cand.corr      = (float **) alloc_mem ( Ncand, sizeof( float * ), fn );
  cand.pCorr     = (float **) alloc_mem ( Ncand, sizeof( float * ), fn );
  cand.deltas    = (float **) alloc_mem ( Ncand, sizeof( float * ), fn );
```

```
  cand.slopes    = (float **) alloc_mem ( Ncand, sizeof( float * ), fn );
  cand.pSlopes   = (float **) alloc_mem ( Ncand, sizeof( float * ), fn );
  for  ( i = 0 ; i < Ncand ; i++ )  {
    cand.weights [i] = (float *) alloc_mem( maxUnits, sizeof( float ), fn );
    cand.corr [i]    = (float *) alloc_mem( Noutputs, sizeof( float ), fn );
    cand.pCorr [i]   = (float *) alloc_mem( Noutputs, sizeof( float ), fn );
    cand.deltas [i]  = (float *) alloc_mem( maxUnits, sizeof( float ), fn );
    cand.slopes [i]  = (float *) alloc_mem( maxUnits, sizeof( float ), fn );
    cand.pSlopes [i] = (float *) alloc_mem( maxUnits, sizeof( float ), fn );
  }

  /*  Allocate memory for error structure                    */

  if  ( !(parm.useCache ) )
    error.errors = (float *) alloc_mem ( Noutputs, sizeof( float ), fn );
  error.sumErr   = (float *) alloc_mem ( Noutputs, sizeof( float ), fn );

  /*  Allocate memory for validation connections             */

  if  ( parm.validate )  {
    val.bestOutConn = (float **)alloc_mem ( Noutputs, sizeof( float * ), fn );
    for  ( i = 0 ; i < Noutputs ; i++ )
      val.bestOutConn [i] = (float *)alloc_mem(maxUnits, sizeof( float ), fn);
  }
}


/*      INIT_NET -  Initializes the vital data structures for the network,
        preparing it for another trial.
*/

void  init_net  ( int maxUnits )
{
  int i,j;     /*  Indexing variables  */

  /*  Initialize the outputs  */

  for  ( i = 0 ; i < Noutputs ; i++ )  {
    for  ( j = 0 ; j <= Ninputs ; j++ )  {
      out.weights [i][j]      = RANDOM_WEIGHT( parm.weightRange );
      out.deltas [i][j]               = 0.0;
      out.slopes [i][j]               = 0.0;
      out.pSlopes [i][j]      = 0.0;
    }
    for  ( j = Ninputs + 1 ; j < maxUnits ; j++ )  {
      out.deltas [i][j]     = 0.0;
      out.slopes [i][j]     = 0.0;
      out.pSlopes [i][j]    = 0.0;
    }
  }

  /*  Initialize the cache    */

  if  ( parm.useCache )
    for  ( i = 0 ; i < netConfig.train_pts ; i++ )
      if  ( netConfig.train [i].inputs != NULL )  {
        for  ( j = 0 ; j < maxUnits ; j++ )
          cache.values [i][j] = 0.0;
        for  ( j = 0 ; j < Noutputs ; j++ )
          cache.errors [i][j] = 0.0;
      }

  /*  Initialize the global variables */
```

```
  Nunits = Ninputs + 1;
  epoch  = 0;
#ifdef CONNX
  connx = 0;
#endif
}


/*      INIT_ERROR -  This function resets the values contained in the data
        structure specified to their starting values
*/

void  init_error  ( error_data *error )
{
  int i;        /*  Indexing variable  */

  error -> bits                = 0;
  error -> trueErr     = 0.0;
  error -> sumSqErr    = 0.0;
  for  ( i = 0 ; i < Noutputs ; i++ )
    error -> sumErr [i]        = 0.0;
}


/*      INIT_CAND -  Sets up a new pool of candidates to be trained for
        insertion into the network.  Determines the type for each of these new
        units.
*/

void  init_cand  ( void )
{
  int i,j;      /*  Indexing variables */

  /*  Reset the candidate values      */

  for  ( i = 0 ; i < Ncand ; i++ )  {
    cand.values  [i] = 0.0;
    cand.sumVals [i] = 0.0;
    for  ( j = 0 ; j < Noutputs ; j++ )  {
      cand.corr [i][j]     = 0.0;
      cand.pCorr [i][j] = 0.0;
    }
    for  ( j = 0 ; j < Nunits ; j++ )  {
      cand.weights [i][j]     = RANDOM_WEIGHT( parm.weightRange );
      cand.deltas  [i][j]     = 0.0;
      cand.slopes  [i][j]     = 0.0;
      cand.pSlopes [i][j]     = 0.0;
    }

    /*  Select a type for the new unit.  This is either candNewType, or a  */
    /* mix, if candNewType is set to VARIED                                 */

    if  ( parm.candNewType == VARIED )
      switch ( i % 4 )  {
        case 0:  cand.types [i] = SIGMOID;
                 break;
        case 1:  cand.types [i] = ASIGMOID;
                 break;
        case 2:  cand.types [i] = GAUSSIAN;
                 break;
        case 3:  cand.types [i] = VARSIGMOID;
                 break;
```

```
        }
      else
        cand.types [i] = parm.candNewType;
    }
}


/********************** Output Training Functions ****************************/

/*      TRAIN_OUTPUTS -  Train the outputs for a number of epochs, until
        victory, the maximum epoch set in the parameters is reached, or error
        ceases to improve measureably.  Return the final result in *status.
        Victory (WIN) is achieved when either error bits goes to zero (if
        measure = BITS), or error index falls below indexThresh (if measure =
        INDEX).  Stagnation (STAGNANT) occurs when we go 'patience' epochs
        without significant improvement.  Time Out (TIMEOUT) occurs when we
        go through 'out.epochs' epochs without either victory or stagnation.
*/

void  train_outputs  ( status_t *status )
{
  int     quitEpoch = 0,        /*  Epoch that training is considered */
                                /* stagnant                          */
          i;                    /*  Indexing variable                 */
  float   lastError;            /*  This is the True Error to beat    */

  for  ( i = 0 ; i < parm.out.epochs ; i++ )  {
    init_error ( &error );      /*  Initialize the error                    */

    output_epoch ( );           /*  Perform a training epoch on the outputs  */

    check_interrupt ( );        /*  Check for user interrupt                */

    /*  Check to see if victory was achieved.  If so, set status and return  */

    if  ( (error.measure == BITS) && (error.bits == 0) )  {
      *status = WIN;
      return;
    }  else
    if  (error.measure == INDEX)  {
      error.index = ERROR_INDEX( error.trueErr, error.stdDev, NtrainOutVals );
      if  ( error.index <= error.indexThresh )  {
        *status = WIN;
        return;
      }
    }

    /*  If victory was not achieved, adjust the output weights and increment */
    /* the epoch coounter                                                    */

    adjust_weights  ( );
    epoch++;

    /*  Check for appreciable change in error.  If no change is detected,    */
    /* check for stagnation                                                  */

    if  ( i == 0 )
      lastError = error.trueErr;
    else
    if  ( fabs( error.trueErr - lastError ) >
              ( lastError * parm.out.changeThresh) )  {
      lastError = error.trueErr;
      quitEpoch = epoch + parm.out.patience;
```

```
    }  else
    if   ( epoch == quitEpoch )   {
      *status = STAGNANT;
      return;
    }
  }

  *status = TIMEOUT;   /*  If haven't left by now, we must have timed out  */
}


/*      OUTPUT_EPOCH -  Perform forward propogation and error computation on
        each of the training points in the training set.
*/

void output_epoch   ( void )
{
  int  offsetLeft,     /*  Number of points until an output is expected  */
       i;              /*  Indexing variable                             */

  offsetLeft = netConfig.offset;      /*  Reset the offset  */

  for   ( i = 0 ; i < netConfig.train_pts ; i++ )

    /*  Check for a segment marker  */

    if   ( netConfig.train [i].inputs != NULL )   {
      if   ( offsetLeft == 0 )   {

        /*  Use cached values if the cache is on  */

        if   ( parm.useCache )   {
          net.values    = cache.values [i];
          error.errors  = cache.errors [i];
          output_pass ( );
        }  else
          forward_pass ( i, netConfig.train_pts, netConfig.train );

        /*  Compute error for this presentation  */

        compute_error ( netConfig.train [i].outputs, &error, TRUE, TRUE );

      }  else
        offsetLeft--;
    }  else
      offsetLeft = netConfig.offset;
}


/*      ADJUST_WEIGHTS -  Go through and use quickprop to adjust each of the
        output weights.  Uses slopes, previous slopes, and delta values
        computed in compute_error and in the last weight update.
*/

void  adjust_weights   ( void )
{
  float *ow,    /*  Output weights  */
        *od,    /*  Output deltas   */
        *os,    /*  Output slopes   */
        *op;    /*  Output pSlopes  */
  int   i,j;    /*  Indexing variables */

  /*  Update each of the output weights      */
```

```
  for  ( i = 0 ; i < Noutputs ; i++ )  {
    ow = out.weights [i];      /*  Simple speed hack to keep from  */
    od = out.deltas [i];       /* recomputing array locations      */
    os = out.slopes [i];
    op = out.pSlopes [i];
    for  ( j = 0 ; j < Nunits ; j++ )
      quickprop  ( j, ow, od, os, op, out.scaledEpsilon, parm.out.decay,
                   parm.out.mu, out.shrinkFactor );
  }
}


/********************** Candidate Training Code ******************************/

/*      TRAIN_CAND -  Train a new pool of candidates.  Training contiues until
        either the maximum number of training epochs for a candidate pool has
        been reached (TIMEOUT), or a specified number of epochs passes without
        noticeable improvement (STAGNANT).  Victory (WIN) is not possible at
        this juncture.  Training results are returned in *status, so that they
        may be reported to the user.

        Note:  Ideally, after each adjustment of the candidate weights, we
        would run two epochs.  The first would just determine the correlations
        between the candidate unit outputs and the residual error.  Then in a
        second pass, we would adjust each candidate's input weights so as to
        maximize the absolute value of the correlation.  We need to know the
        direction to tune the input weights.

        Since this ideal method doubles the number of epochs required for
        training the candidate, we cheat slightly and use the correlation
        values computed BEFORE the most recent weight update.  This combines
        the two epochs, saving us almost a factor of two.  To bootstrap the
        process, we begin with a single epoch that computes only the
        correlation.

        Since we look only at the sign of the correlation after the first ideal
        epoch and since that sign should change very infrequently, this
        probably is OK.   But keep a lookout for pathelogical situations in
        which this might cause oscillation.
*/

void  train_cand  ( status_t *status )
{
  float     lastScore = 0.0;  /*  This is the correlation score to beat   */
  int       quitEpoch = 0,    /*  This is the epoch that we should quit   */
                              /* if we don't beat lastScore             */
            i, j;             /*  Indexing variables                     */

  for  ( i = 0 ; i < Noutputs ; i++ ) /*  Average the sum errors over all  */
    error.sumErr [i]  /= NtrainPts;   /* the training patterns           */

  correlation_epoch  ( );     /*  Calculate initial correlation values  */

  check_interrupt  ( );             /*  Check for user interrupt          */

  for  ( i = 0 ; i < parm.cand.epochs ; i++ )  {
    cand_epoch  ( );          /*  Train the candidates for an epoch,  */
                              /* recomputing correlations as you go   */

    check_interrupt  ( );     /*  Check for user interrupt          */

    adjust_cand_weights  ( ); /*  Adjust the connections leading into the  */
```

```
                              /* candidates using quickprop              */
    adjust_correlations  ( ); /*  Adjust the correlation values for the    */
                              /* candidates.                              */

    epoch++;                  /*  Increment the epoch counter                 */

    /*  This is the stagnation machinery.  Basically the same as            */
    /* train_outputs, except that it keys off of correlation values instead */
    /* of True Error.  The first 'if' statement runs on the first pass      */
    /* through the loop and serves to bootstrap the rest of the process        */
    /* The second conditional checks for improvement in correlation values  */
    /* while the third checks for the actual stagnation                        */

    if  ( i == 0 )
      lastScore = cand.bestScore;
    else
    if  ( fabs( cand.bestScore - lastScore ) >
          ( lastScore * parm.cand.changeThresh ) )  {
      quitEpoch = epoch + parm.cand.patience;
      lastScore = cand.bestScore;
    } else
    if  ( epoch == quitEpoch )  {
      *status = STAGNANT;
      return;
    }
  }

  *status = TIMEOUT;   /*  If we haven't left by now, we've timed out  */
}


/*      CORRELATION_EPOCH -  Run a single epoch to compute correlation values
        only.  After this epoch, we will compute correlation values during
        training.  This is a slight deviation from the 'ideal' candidate
        training cycle that saves almost half the epochs.  For more
        information, see the notes at the end of the notes for TRAIN_CAND.
*/

void correlation_epoch  ( void )
{
  int offsetLeft,      /*  Number of pattern presentations left until an  */
                       /* output is expected                              */
      i;               /*  Indexing variable                               */

  offsetLeft = netConfig.offset;      /*  Reset offsetLeft                */

  for  ( i = 0 ; i < netConfig.train_pts ; i++ )
    if  ( netConfig.train [i].inputs != NULL )  {
      if  ( offsetLeft == 0 )  {

        /*  Either pull the activation values from the cache, if its on, or  */
        /* recompute them from scratch if it is off                          */

        if  ( parm.useCache )  {
          net.values   = cache.values [i];
          error.errors = cache.errors [i];
        }  else  {
          forward_pass  ( i, netConfig.train_pts, netConfig.train );
          compute_error ( netConfig.train [i].outputs, &error, FALSE, FALSE );
        }

        compute_correlations  ( );    /*  Compute correlation values for  */
                                      /* this training point              */
```

```
      } else
        offsetLeft--;
    } else
      offsetLeft = netConfig.offset;

  adjust_correlations ( );     /* Normalize the correlations and then */
  epoch++;                     /* update the epoch counter           */
}


/*      CAND_EPOCH - Train the candidates for an epoch. If the cache is not
        on, run a forward pass and compute error. Otherwise, retrieve
        activation values from the cache. Use these values to calculate the
        slopes and correlation values of each of the candidates.
*/

void cand_epoch  ( void )
{
  int offsetLeft,      /* Number of inputs to read without output */
      i;               /* Indexing variable                       */


  offsetLeft = netConfig.offset;      /* Reset the offset */

  for  ( i = 0 ; i < netConfig.train_pts ; i++ )
    if  ( netConfig.train [i].inputs != NULL )  {
      if  ( offsetLeft == 0 )  {

        /* Get the activation values and errors     */

        if  ( parm.useCache )  {
          net.values    = cache.values [i];
          error.errors  = cache.errors [i];
        } else  {
          forward_pass  ( i, netConfig.train_pts, netConfig.train );
          compute_error ( netConfig.train [i].outputs, &error, FALSE, FALSE );
        }

        compute_slopes  ( );  /* Compute slope and correlation values */

      } else
        offsetLeft--;
    } else
      offsetLeft = netConfig.offset;
}


/*      COMPUTE_CORRELATIONS - For the current training pattern, compute the
        activation of each candidate unit. Then begin to compute the
        correlation value for that unit. Activation values and error from the
        rest of the network have already been computed elsewhere.
*/

void compute_correlations  ( void )
{
  float sum,           /* Sum of input stimulus to this unit          */
        val,           /* Activation value of this unit               */
        *cWghts,       /* Pointer to this candidate's weights         */
        *cCorr;        /* Pointer to this candidate's correlation values */
  int   i, j;          /* Indexing variables                          */

  for  ( i = 0 ; i < Ncand ; i++ )  {
    sum = 0.0;                /* Reset Sum */
```

```
    cWghts      = cand.weights [i];    /*  Used in a simple speed hack  */
    cCorr       = cand.corr [i];

    for  ( j = 0 ; j < Nunits ; j++ )        /*  Compute sum of this cand */
      sum += cWghts [j] * net.values [j];

    /*  Compute the activation value of this unit    */

    val              =  activation( cand.types [i], sum );
    cand.values [i]  =  val;
    cand.sumVals [i] += val;

    /*  Compute the correlation for this unit        */

    for  ( j = 0 ; j < Noutputs ; j++ )
      cCorr[j] += val * error.errors [j];
  }
}


/*      ADJUST_CORRELATIONS -  Normalize each candidate's correlation score
        and then stuff that value into the previous correlation structure.
        Zero out the correlation score to prepare for the next round.  Note
        which unit has the best total score to date.
*/

void  adjust_correlations  ( void )
{
  float avgValue,      /*  Average value of this candidate over all    */
                       /* training points                              */
        score,         /*  Sum of the absolute values of the corelations  */
        cor,           /*  Correlation of this candidate to this output   */
        *curCor,       /*  Pointer to help improve speed                */
        *prevCor;
  int   i,j;           /*  Indexing variable                            */

  cand.best      = 0;  /*  Reset pointers to the best units             */
  cand.bestScore = 0.0;

  for  ( i = 0 ; i < Ncand ; i++ )  {
    avgValue =  cand.sumVals [i] / NtrainPts;       /*  Calculate  avgValue       */

    score    = 0.0;           /*  Reset score counter       */
    curCor   = cand.corr  [i];       /*  Set pointers to array index being worked */
    prevCor  = cand.pCorr [i];       /* on, this saves costly array operations    */

    /*  Calculate the correlation value for each of the outputs and then     */
    /* update the candidate's data structures.  Finally update this      */
    /* candidate's score                                                  */

    for  ( j = 0 ; j < Noutputs ; j++ )   {
      cor          = ( curCor [j] - avgValue * error.sumErr [j] )
                     / error.sumSqErr;
      prevCor [j]  = cor;
      curCor  [j]  = 0.0;
      score        += fabs( cor );
    }

    cand.sumVals [i] = 0.0;

    /*  See if this is the best candidate so far  */

    if  ( score > cand.bestScore )  {
```

```
      cand.bestScore = score;
      cand.best      = i;
    }
  }
}


/*      COMPUTE_SLOPES -  Use the precomputed correlation values to compute
        the slopes of the error for each individual candidate unit.  This will
        later be used to update the input weights to each of these candidates.

        Note:  This function is extremely compute-intensive.  If you want to
        spend some time optimizing, this is a good place to start.  I think
        I've done everything reasonable, but if you find something, please
        email me at 'mwhite+@cmu.edu', so that I can modify the release
        version.
*/

void  compute_slopes  ( void )
{
  float sum,            /*  Sum of stimulus coming into a candidate     */
        change,             /*  Used to compute both the slope and the corr    */
        value,          /*  The activation value of this candidate         */
        actPrime,       /*  Activation prime value for this candidate      */
        err,            /*  Raw error for this candidate for this pattern  */
        direction,      /*  Direction to adjust weights to improve corr    */
        *cWghts,        /*  These four pointers are used to help improve   */
        *cCorr,             /* speed                                              */
        *cPCorr,
        *cSlp;
  int   i, j;           /*  Indexing variables                             */

  for  ( i = 0 ; i < Ncand ; i++ )  {
    sum    = 0.0;           /*  Initialize to variables for this  */
    change = 0.0;           /* candidate                          */
    cWghts = cand.weights [i];
    cCorr  = cand.corr [i];
    cPCorr = cand.pCorr [i];
    cSlp   = cand.slopes [i];

    for  ( j = 0 ; j < Nunits ; j++ )       /*  Compute stimulus for  */
      sum += net.values [j] * cWghts [j];   /* this candidate       */

    /*  Compute activation and activation prime for this candidate  */

    value              =  activation( cand.types [i], sum );
    actPrime           =  activation_prime( cand.types [i], value, sum );
    cand.sumVals [i]   += value;

    /*  Normalize activation prime by the sum squared error  */

    actPrime           /= error.sumSqErr;

    /*  Compute the correlation to each of the outputs   */

    for  ( j = 0 ; j < Noutputs ; j++ ) {
      err            =  error.errors [j];
      direction      =  ( cPCorr [j] < 0.0 ) ? -1.0 : 1.0;
      change         -= direction * actPrime * ( err - error.sumErr [j] );
      cCorr [j]      += err * value;
    }

    /*  Use the 'change' value just computed to compute the slopes for all  */
```

```
    /* the weights feeding into this candidate                              */

    for  ( j = 0 ; j < Nunits ; j++ )
      cSlp [j] += change * net.values [j];
  }
}


/*      ADJUST_CAND_WEIGHTS -  Using the slopes and deltas previously computed,
        use quickprop to update each the weights feeding into each of the
        candidates.
*/

void adjust_cand_weights  ( void )
{
  float scaledEpsilon, /*  Epsilon scaled by the number of training points  */
                       /* times the current number of units in the network  */
       *cw,            /*  These four pointers are used to cut down on      */
       *cd,            /* array operations, thus speeding things up in here */
       *cs,
       *cp;
  int   i, j;          /*  Indexing variables                               */

  /*  We must scale the epsilon locally, since the scaling factor is going   */
  /* to change on every cycle through the network                            */

  scaledEpsilon = parm.cand.epsilon / (float) ( NtrainPts * Nunits );

  /*  Perform a quickprop update on each of the weights  */

  for  ( i = 0 ; i < Ncand ; i++ )  {
    cw = cand.weights [i];     /*  Set the pointers to their positions in  */
    cd = cand.deltas [i];      /* the arrays                                */
    cs = cand.slopes [i];
    cp = cand.pSlopes [i];
    for  ( j = 0 ; j < Nunits ; j++ )
      quickprop  ( j, cw, cd, cs, cp, scaledEpsilon, parm.cand.decay,
                   parm.cand.mu, cand.shrinkFactor );
  }
}


/*      INSTALL_CAND -  Take the best candidate unit and install it in the
        network.  This is accomplished by copying its weights and unit type
        over into the 'net' data structure.  If the cache is in use, it is
        now necessary to recompute cache values so that the new unit is taken
        into consideration.
*/

void install_cand  ( void )
{
  float *newWeights,          /*  Pointer to the actual weight array       */
        *candBestWeights,     /*  Pointer to the candidate's weight array  */
        weightModifier;             /*  Scaling factor for new output weights    */
  int   i;                    /*  Indexing variable                        */

  /*  Set up pointers to the appropriate weight arrays  */

  newWeights      = net.weights [Nunits];
  candBestWeights = cand.weights [cand.best];

  /*  Copy the weights over  */
```

```
  for  ( i = 0 ; i < Nunits ; i++ )
    newWeights [i] = candBestWeights [i];

  /*  Use the correlation score to each output as a starting point for the  */
  /* new output weights.  Modify this value by the weight multiplier.  If   */
  /* we are working with an error index, scale this value by the number of  */
  /* active units in the network.                                           */

  if  ( error.measure == BITS )
    weightModifier = parm.weightMult;
  else
    weightModifier = parm.weightMult / Nunits;

  for  ( i = 0 ; i < Noutputs ; i++ )
    out.weights [i][Nunits] = -cand.pCorr [cand.best][i] * weightModifier;

  net.unitTypes [Nunits] = cand.types [cand.best];

  if  ( parm.useCache )              /*  Recompute cache values for the new unit  */
    recompute_cache  ( Nunits );

  Nunits++;    /*  Let the rest of the network know about the new unit  */
}


/********************** Generalization Code *********************************/

/*      VALIDATION_EPOCH -  Run a complete validation epoch.  No training is
        performed, only a measurement of error.  If a certain number of epochs,
        measured by valPatience, passes without improvement, restore the net to
        the point of its best generalization.
*/

void  validation_epoch  ( float testThreshold, int maxUnits, status_t *status )
{
  static float         *vVals;        /*  Dummy activation values array  */
  float                *vErr,         /*  Dummy error array              */
       *vSumErr,       /*  Dummy sum of errors array    */
                       *oldVals;      /*  Pointer back to orginal vals   */
  int                   offsetLeft,   /*  Number of inputs left with no  */
                                      /* outputs                         */
                       i,j;           /*  Indexing variables             */
  static boolean       firstTime = TRUE;     /*  First val. epoch?      */
  char                 *fn = "Validation Epoch";     /*  Function ID          */
  static error_data    valErr;        /*  Dummy error structure for val  */

  /*  If this is the first validation epoch, allocate memory for the arrays  */
  /* that will be needed for these computations                                */

  if  ( firstTime )  {
    vErr    = (float *) alloc_mem  ( Noutputs, sizeof( float ), fn );
    vSumErr = (float *) alloc_mem  ( Noutputs, sizeof( float ), fn );
    vVals   = (float *) alloc_mem  ( maxUnits, sizeof( float ), fn );

    valErr.errors      = vErr;
    valErr.sumErr      = vSumErr;
  }

  oldVals              = net.values;  /*  Swap the values in net with the  */
  net.values           = vVals;       /* dummy activation values          */

  init_error  ( &valErr );            /*  Reset the error values          */
```

```
  offsetLeft = netConfig.offset;      /*  Reset the offset                */


  /*  Compute an epoch with the cache off.  Measure the error                  */

  for  ( i = 0 ; i < val.Npts ; i++ )
    if  ( val.data [i].inputs != NULL )  {
      if  ( offsetLeft == 0 )  {
        forward_pass  ( i, val.Npts, val.data );
        compute_error ( val.data [i].outputs, &valErr, TRUE, FALSE );
      }  else
        offsetLeft--;
    }  else
      offsetLeft = netConfig.offset;

  check_interrupt  ( );

  /*  Compute the error index for this validation epoch                        */

  if  ( error.measure == INDEX )
    valErr.index = ERROR_INDEX( valErr.trueErr, val.stdDev, val.NoutVals );

  /*  Keep track of the best error values to date and the connections that   */
  /* go along with them.  If generalization ceases to improve, restore the   */
  /* network to the state of its best performance.  This is accomplished by  */
  /* setting the 'Nunits' value back down to only count the helpful units    */
  /* and by copying the connections between these units and the outputs back */
  /* into the 'out' structure.       Setting a status flag informs the calling    */
  /* function that training has stagnated and that it should end.       */

  if  ( firstTime )  {
    firstTime     = FALSE;
    val.bestScore = valErr.trueErr;
    val.bestPass  = Nunits;
  }  else
  if  ( valErr.trueErr < val.bestScore )  {
    val.bestScore = valErr.trueErr;
    val.bestPass  = Nunits;
    for  ( i = 0 ; i < Noutputs ; i++ )
      for  ( j = 0 ; j < Nunits ; j++ )
        val.bestOutConn [i][j] = out.weights [i][j];
  }  else
  if  ( (Nunits - val.bestPass) > parm.valPatience )  {
    Nunits = val.bestPass;
    for  ( i = 0 ; i < Noutputs ; i++ )
      for  ( j = 0 ; j < Nunits ; j++ )
        out.weights [i][j] = val.bestOutConn [i][j];
    *status = STAGNANT;
    firstTime = TRUE;
  }

  net.values   = oldVals;             /*  Restore the internal activation  */
                                      /* array to its origonal state            */

  output_val_results ( valErr );      /*  Output validation results to the */
                                      /* user                                */
}


/*      TEST_EPOCH -  Run one final epoch over an alternate data set, to see if
        we have generalized this problem well.
*/
```

```
void  test_epoch  ( void )
{
  int  offsetLeft,     /*  Number of inputs until output  */
       i;              /*  Indexing variable              */
  FILE *fptr;            /*  Pointer to dump file for test outputs  */

  if  ( (fptr=fopen("test.results","w")) == NULL )  {
    fprintf (stderr,"ERROR: Unable to open file 'test.results' for writing");
    exit( 1 );
  }

  offsetLeft = netConfig.offset;

  init_error ( &error );

  for  ( i = 0 ; i < test.Npts ; i++ )
    if  ( test.data [i].inputs != NULL )  {
      if  ( offsetLeft == 0 )  {
        forward_pass  ( i, test.Npts, test.data );
        dump_results  ( fptr, test.data[i].inputs, test.data[i].outputs );
        compute_error ( test.data [i].outputs, &error, TRUE, FALSE );
      }  else
        offsetLeft--;
    }  else
      offsetLeft = netConfig.offset;

  fclose( fptr );

  check_interrupt  ( );
}

void dump_results  ( FILE *fptr, float *inputs, float *outputs )
{
  int i;

  for  ( i = 0; i < Ninputs-1; i++)
    fprintf (fptr,"%f, ",inputs[i]);
  fprintf (fptr,"%f  =>  ",inputs[Ninputs-1]);
  for  ( i = 0; i < Noutputs-1; i++)
    fprintf (fptr,"%f (%f), ",outputs[i],out.values[i]);
  fprintf (fptr,"%f (%f);\n",outputs[i],out.values[i]);
}

/************************ Cache Routines ************************************/

/*      BUILD_CACHE -  Allocate memory for the cache.  If, at any point, we run
        out of memory, dismantle the cache and return a FALSE value to the
        calling function, so that it knows to turn off the cache.
*/

boolean build_cache  ( int maxUnits )
{
  int i;        /*  Indexing variable  */

  if  ( ( cache.values = (float **) calloc ( netConfig.train_pts,
          sizeof( float * ) )) == NULL )
    NO_CACHE;
  if  ( ( cache.errors = (float **) calloc ( netConfig.train_pts,
          sizeof( float * ) )) == NULL )
    NO_CACHE;

  for  ( i = 0 ; i < netConfig.train_pts ; i++ )  {
    if  ( ( cache.values [i] = (float *) calloc ( maxUnits,
```

```
            sizeof( float ) )) == NULL )
      NO_CACHE;
    if  ( ( cache.errors [i] = (float *) calloc ( Noutputs,
            sizeof( float ) )) == NULL )
      NO_CACHE;
  }

  return TRUE;
}


/*      DESTROY_CACHE -  Go through and deallocate memory for the cache,
        first checking at each point that memory was allocated in the first
        place.
*/

void destroy_cache  ( void )
{
  int i;        /*  Indexing variable  */

  for  ( i = 0 ; i < netConfig.train_pts ; i++ )  {
    if  ( cache.values [i] != NULL )
      free( cache.values [i] );
    if  ( cache.errors [i] != NULL )
      free( cache.errors [i] );
  }

  if  ( cache.values != NULL )
    free( cache.values );
  if  ( cache.errors != NULL )
    free( cache.errors );

  cache.values = NULL;
  cache.errors = NULL;
}


/*      COMPUTE_CACHE -  Go through and setup all the inputs for the cache.
*/

void  compute_cache  ( void )
{
  int i;

  for  ( i = 0 ; i < netConfig.train_pts ; i++ )
    if  ( netConfig.train [i].inputs != NULL )  {
      net.values = cache.values [i];
      setup_inputs ( i, netConfig.train_pts, netConfig.train );
    }
}


/*      RECOMPUTE_CACHE -  After a new unit has been added to the network, it
        is necessary to compute cache activation values for it.  That's what
        this function does.
*/

void recompute_cache  ( int unit_no )
{
  float sum;    /*  Sum of stimulus for the unit  */
  int i, j;     /*  Indexing variables          */

  for  ( i = 0 ; i < netConfig.train_pts ; i++ )
```

```
    if  ( netConfig.train [i].inputs != NULL )  {
      sum = 0.0;

      /*  Compute stimulus for this new unit  */

      for  ( j = 0 ; j < unit_no ; j++ )
        sum += cache.values [i][j] * net.weights [unit_no][j];

      /*  Store activation value in the cache  */

      cache.values [i][unit_no] = activation( net.unitTypes [unit_no], sum );

#ifdef CONNX
      connx += unit_no;
#endif
    }
}


/****************** Miscellanious Network Functions ************************/

/*      FORWARD_PASS -  Forward propogate through the network for a single
        training pattern.  The data set is specifiable through the dataSet
        parameter.  Npts is the number of points in the data set and pattern
        indicates which training pattern to use.
*/

void  forward_pass  ( int pattern, int Npts, data_set dataSet )
{
  float sum,            /*  Sum of stimulus for the unit being calculated   */
        *wghts;               /*  Pointer to the weights being used.  Speed hack  */
  int   i, j;           /*  Indexing variables  */

  setup_inputs  ( pattern, Npts, dataSet );

  /*  Compute the values of the hidden units  */

  for  ( i = Ninputs + 1 ; i < Nunits ; i++ )  {
    sum   = 0.0;
    wghts = net.weights [i];

    /*  Sum the stimulus  */

    for ( j = 0 ; j < i ; j++ )
      sum += net.values [j] * wghts [j];

    /*  Get the activation value  */

    net.values [i] = activation( net.unitTypes [i], sum );

#ifdef CONNX
    connx += i - 1;
#endif
  }

  output_pass  ( );    /*  Compute the values of the outputs  */
}


/*      SETUP_INPUTS -  Setup the input units for forward propogation.  This is
        complicated by the possible presence of sequence data sets.  If this is
        a sequence data set, then a nettalk-like approach is used on it.  The
        size of the window used is user selectable in parm.winRadius.  A '0'
```

```
        would mean just look at this pattern, a '1' would mean look at this
        pattern and the ones immediately before and after, and so on.  If a
        segment marker is reached, then all further inputs in that direction
        receive no stimulus.
*/

void  setup_inputs  ( int pattern, int Npts, data_set dataSet )
{
  boolean  nullEnc;    /*  Has a marker been encountered in this direction? */
  int      i, j,       /*  General indexing variables                       */
           pat;              /*  Indexing variable pointing to location in the    */
                     /* the data set.                                        */

  net.values [0] = parm.bias; /*  Setup bias unit  */

  if  ( isSeq )  {
    i       = parm.winRadius; /*  Initialize local variables  */
    pat     = pattern;
    nullEnc = FALSE;

    /*  Setup the center inputs  */

    for  ( j = 0 ; j < netConfig.inputs ; j++ )
      net.values [(i*netConfig.inputs)+j+1] = dataSet[pat].inputs[j];

    i--;
    pat--;

    /*  Go through and setup the inputs before the center  */

    while ( i >= 0 )  {
      if  ( dataSet[pat].inputs == NULL )
        nullEnc = TRUE;
      if  ( nullEnc || ( pat < 0 ) )
        for  ( j = 0 ; j < netConfig.inputs ; j++ )
          net.values [(i*netConfig.inputs)+j+1] = 0.0;
      else
        for  ( j = 0 ; j < netConfig.inputs ; j++ )
          net.values [(i*netConfig.inputs)+j+1] = dataSet[pat].inputs[j];
      i--;
      pat--;
    }

    /*  Go through and setup the inputs after the center set  */

    i       = parm.winRadius  + 1;
    pat     = pattern + 1;
    nullEnc = FALSE;
    while  ( i <= (2*parm.winRadius + 1) )  {
      if  ( dataSet[pat].inputs == NULL )
        nullEnc = TRUE;
      if  ( nullEnc || ( pat >= Npts ) )
        for  ( j = 0 ; j < netConfig.inputs ; j++ )
          net.values [(i*netConfig.inputs)+j+1] = 0.0;
      else
        for  ( j = 0 ; j < netConfig.inputs ; j++ )
          net.values [(i*netConfig.inputs)+j+1] = dataSet[pat].inputs[j];
      i++;
      pat++;
    }
  } else {

    /*  If this is a standard IO data set, there isn't nearly so much to do  */
```

```
    for  ( i = 0 ; i < Ninputs ; i++ )
      net.values [i + 1] = dataSet [pattern].inputs [i];
  }

#ifdef CONNX
  connx += Ninputs + 1;
#endif
}


/*      OUTPUT_PASS -  After doing a forward pass, or retrieving those values
        from the cache, it is still necessary to compute the output values.
*/

void  output_pass  ( void )
{
  float sum,    /*  Sum of stimulus to a specific output  */
        *outW;  /*  Pointer to the output weights         */
  int   i, j;   /*  Indexing variables                    */

  for  ( i = 0 ; i < Noutputs ; i++ )  {
    sum  = 0.0;                        /*  Initialize the local variables  */
    outW = out.weights [i];

    /*  Compute the sum stimulus for this output  */

    for  ( j = 0 ; j < Nunits ; j++ )
      sum += net.values [j] * outW [j];

    /*  Compute the activation value for this output  */

    out.values [i] = output_function( out.types [i], sum );
  }

#ifdef CONNX
  connx += Noutputs * Nunits;
#endif
}


/*      COMPUTE_ERROR -  Computes the error between the actual activation
        values of the outputs and the goal vector.  This error information is
        stored in a data structure 'err', allowing this same function to be
        used in validation epochs as well as training epochs.  The boolean
        variable 'alter_stats' determines whether or not to collect error
        statistics.  'alterSlopes' indicates whether to compute slope values
        for this training point.
*/

void  compute_error  ( float *goal, error_data *err,
                       boolean alter_stats, boolean alterSlopes )
{
  float dif,           /*  The difference between the output being examined */
                       /* and it's goal                                     */
        error_prime,   /*  The error prime for this output                  */
        val;           /*  The output value being examined                  */
  int   i, j;          /*  Indexing variables                               */

  for  ( i = 0 ; i < Noutputs ; i++ )  {

    /*  Find the difference between output and goal, and then compute error  */
    /* error prime                                                           */
```

```
    val         = out.values [i];
    dif         = val - goal [i];
    error_prime = dif * output_prime( out.types [i], val );

    err -> errors [i]  = error_prime;

    /*  Compute True Error, Sum Squared Error, Sum Error and Error Bits for  */
    /* this output value                                                      */

    if  ( alter_stats )  {
      if  ( fabs( dif ) > error.scoreThresh )
        err -> bits++;
      err -> trueErr   += dif * dif;

      err -> sumErr [i]  += error_prime;
      err -> sumSqErr    += error_prime * error_prime;
    }

    /*  Compute the slopes for this output                                    */

    if  ( alterSlopes )
      for ( j = 0 ; j < Nunits ; j++ )
        out.slopes [i][j] += error_prime * net.values [j];
  }
}


/*      QUICKPROP -  Perform a weight update on the specified weight using
        Scott Fahlman's QuickProp algorithm.  Weights, deltas, slopes and
        previous slopes are passed to quickprop via pointers to arrays.  The
        specific weight to update is selected via the 'node' parameter.
        Epsilon, decay, mu, and shrink factor are just passed in raw.
*/

void  quickprop  ( int node, float *weights, float *deltas, float *slopes,
                   float *pSlopes, float epsilon, float decay, float mu,
                   float shrinkFactor )
{
  float w,              /*  Weight being updated  */
        d,              /*  Delta value for this weight  */
        s,              /*  Slope for this weight  */
        p,              /*  Previous slope for this weight  */
        nextStep = 0.0;       /*  Step to be taken  */

  w = weights [node];               /*  Initialize local variables  */
  s = slopes [node] + decay * w;    /*  Add decay to the slope    */
  d = deltas [node];
  p = pSlopes [node];

  if  ( d < 0.0 )  {
    if  ( s > 0.0 )
      nextStep -= epsilon * s;

    if  ( s >= ( shrinkFactor * p ) )
      nextStep += mu * d;

    else
      nextStep += d * s / (p - s);
  }
  else if  ( d > 0.0 )  {
    if  ( s < 0.0 )
      nextStep -= epsilon * s;
```

```
    if  ( s <= ( shrinkFactor * p ) )
      nextStep += mu * d;
    else
      nextStep += d * s / (p - s);
  }
  else
    nextStep -= epsilon * s;  /*  Last step was zero, so only use linear   */
                              /* term                                      */

  deltas [node]  =  nextStep; /*  Copy information back into the original  */
  weights [node] += nextStep; /* data structures, and modify the weight    */
  pSlopes [node] =  s;        /* value                                     */
  slopes [node]  =  0.0;
}


/*      ACTIVATION -  Compute the activation value of a unit of type 'unitType'
        and stimulus 'sum'.  This is basically just a big case statement.  New
        unit types can easily be added by inserting them here, output_function,
        the two activation prime functions, and in the unit_t data structure.
*/

float activation  ( int unitType, float sum )
{
  float  temp; /*  Temporary variable to compute gaussian units  */

  switch  ( unitType )  {
    case SIGMOID    :  if  ( sum < -15.0 )
                         return -0.5;
                       if  ( sum > 15.0 )
                         return 0.5;
                       return  ( 1.0 / (1.0 + exp( -sum )) - 0.5 );
    case ASIGMOID   :  if  ( sum < -15.0 )
                         return 0.0;
                       if  ( sum > 15.0 )
                         return 1.0;
                       return  ( 1.0 / (1.0 + exp( -sum  )) );
    case VARSIGMOID :  if  ( sum < -15.0 )
                         return parm.cand.sigMin;
                       if  ( sum > 15.0 )
                         return parm.cand.sigMax;
                       return  ( (parm.cand.sigMax - parm.cand.sigMin) /
                                 (1.0 + exp( -sum )) +
                                 parm.cand.sigMin );
    case GAUSSIAN   :  temp = -0.5 * sum * sum;
                       if  ( temp < -75.0 )
                         return 0.0;
                       else
                         return ( exp( temp ) );
    default         :  fprintf ( stderr, "ERROR: Illegal Unit Type in ");
                       fprintf ( stderr, "activation function (%d)\n",
                                 unitType );
                       exit( 1 );
  }
}


/*      ACTIVATION_PRIME -  Compute the activation prime for the unit type and
        activation specified.  Note that no offset is added into the value
        because this confuses the correlation machinery.
*/

float activation_prime  ( int unitType, float value, float sum )
```

```
{
  switch  ( unitType )  {
    case SIGMOID     :  return  ( 0.25 - value * value );
    case ASIGMOID    :  return  ( value * ( 1.0 - value ) );
    case VARSIGMOID  :  return  ( ( value - parm.cand.sigMin ) *
                                  ( 1.0 - ( value - parm.cand.sigMin ) /
                                  ( parm.cand.sigMax - parm.cand.sigMin ) ));
    case GAUSSIAN    :  return  ( sum * (-value) );
    default          :  fprintf ( stderr, "ERROR: Illegal Unit Type in ");
                        fprintf ( stderr, "activation prime (%d)\n",
                                  unitType );
                        exit( 1 );
    }
}


/*      OUTPUT_FUNCTION -  This is essentially the same function as the
        other activation function, except that it uses the out.sigMax and
        out.sigMin instead of the candidate values.  Also, there is a LINEAR
        unit for outputs.  This is often used for continuous floating point
        outputs.
*/

float output_function  ( int unitType, float sum )
{
  switch  ( unitType )  {
    case SIGMOID     :  if  ( sum < -15.0 )
                          return -0.5;
                        if  ( sum > 15.0 )
                          return 0.5;
                        return  ( 1.0 / ( 1.0 + exp( -sum ) ) - 0.5 );
    case LINEAR      :  return( sum );
    case ASIGMOID    :  if  ( sum < -15.0 )
                          return 0.0;
                        if  ( sum > 15.0 )
                          return 1.0;
                        return  ( 1.0 / ( 1.0 + exp( -sum ) ) );
    case VARSIGMOID  :  if  ( sum < -15.0 )
                          return parm.out.sigMin;
                        if  ( sum > 15.0 )
                          return parm.out.sigMax;
                        return  ( (parm.out.sigMax - parm.out.sigMin) /
                                  (1.0 + exp( -sum ))
                                  + parm.out.sigMin );
    default          :  fprintf ( stderr, "ERROR: Illegal Unit Type in ");
                        fprintf ( stderr, "output function (%d)\n",
                                  unitType );
                        exit( 1 );
  }
}


/*      OUTPUT_PRIME -  Compute the activation prime of an output, this time
        adding in the offset term, to eliminate flat spot.  This can
        dramatically speed up training.
*/

float output_prime  ( int unitType, float value )
{
  switch  ( unitType )  {
    case SIGMOID     :  return ( parm.sigPrimeOffset + 0.25 - value * value );
    case LINEAR      :  return 1.0;
    case ASIGMOID    :  return ( parm.sigPrimeOffset + value * (1.0 - value) );
```

```
    case VARSIGMOID  :  return ( parm.sigPrimeOffset +
                               (value - parm.out.sigMin) *
                               (1.0 - (value - parm.out.sigMin)
                               / (parm.out.sigMax - parm.out.sigMin) ) );
    default          :  fprintf ( stderr, "ERROR: Illegal Unit Type in ");
                        fprintf ( stderr, "output prime (%d)\n", unitType );
                        exit( 1 );
  }
}


/*      STD_DEV -  Compute the standard deviation of a data set.  This is used
        later to compute error index.
*/

float std_dev  ( data_set train, int Npoints, int Nvals )
{
  float cur,           /*  Current output value  *
        sum,           /*  Sum of output values  */
        sumSq;         /*  Sum of the squared output values  */
  int   offsetLeft,    /*  Offset counter  */
        i, j;          /*  Indexing variables  */

  sum          = 0.0;
  sumSq = 0.0;
  offsetLeft   = netConfig.offset;

  for  ( i = 0 ; i < Npoints ; i++ )
    if  ( train [i].inputs != NULL )  {
      if  ( offsetLeft == 0 )  {
        for  ( j = 0 ; j < Noutputs ; j++ )  {       /*  Compute values for this  */
          cur   =  train [i].outputs [j];    /* output                    */
          sum   += cur;
          sumSq += cur * cur;
        }
      }  else
        offsetLeft--;
    }  else
      offsetLeft = netConfig.offset;

  /*  Return the standard deviation of this data set  */

  return  ( sqrt( (Nvals * sumSq - sum * sum) / (Nvals * (Nvals - 1.0)) ) );
}
```

The vita has been removed from the scanned document