INTERNATIONAL JOURNAL OF ROBUST AND NONLINEAR CONTROL
Int. J. Robust Nonlinear Control 2002; **12**:959–985 (DOI: 10.1002/rnc.727)

An introduction to the use of neural networks in control systems

Martin T. Hagan[1,*,†], Howard B. Demuth[2] and Orlando De Jesús[1]

[1] *School of Electrical & Computer Engineering, 202 Engineering South, Oklahoma State University, Stillwater, OK 74075, U.S.A.*

[2] *Electrical & Computer Engineering Department, University of Colorado, Boulder, CO 80309, U.S.A.*

SUMMARY

The purpose of this paper is to provide a quick overview of neural networks and to explain how they can be used in control systems. We introduce the multilayer perceptron neural network and describe how it can be used for function approximation. The backpropagation algorithm (including its variations) is the principal procedure for training multilayer perceptrons; it is briefly described here. Care must be taken, when training perceptron networks, to ensure that they do not overfit the training data and then fail to generalize well in new situations. Several techniques for improving generalization are discussed. The paper also presents three control architectures: model reference adaptive control, model predictive control, and feedback linearization control. These controllers demonstrate the variety of ways in which multilayer perceptron neural networks can be used as basic building blocks. We demonstrate the practical implementation of these controllers on three applications: a continuous stirred tank reactor, a robot arm, and a magnetic levitation system. Copyright © 2002 John Wiley & Sons, Ltd.

KEY WORDS: neurocontrol; model reference control; model predictive control; feedback linearization

1. INTRODUCTION

In this tutorial paper we want to give a brief introduction to neural networks and their application in control systems. The paper is written for readers who are not familiar with neural networks but are curious about how they can be applied to practical control problems. The field of neural networks covers a very broad area. It is not possible in this paper to discuss all types of neural networks. Instead, we will concentrate on the most common neural network architecture—the multilayer perceptron. We will describe the basics of this architecture, discuss its capabilities and show how it has been used in several different control system configurations. (For introductions to other types of networks, the reader is referred to References [1–3].)

*Correspondence to: Martin T. Hagan, Oklahoma State University, College of Engineering, Architecture and Technology, School of Electrical and Computer Engineering, 202 Engineering South, Stillwater, Oklahoma 74078-5032 U.S.A.
†E-mail: mhagan@okstate.edu

 Accepted 17 May 2002

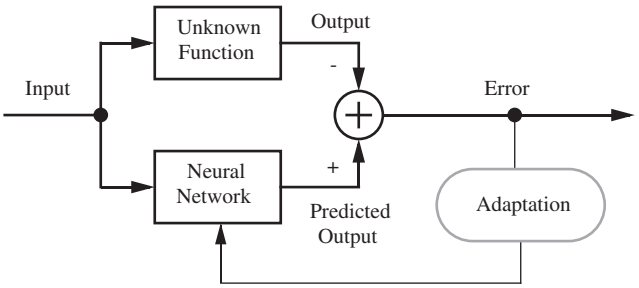

Figure 1. Neural network as function approximator.

For the purposes of this paper we will look at neural networks as function approximators. As shown in Figure 1, we have some unknown function that we wish to approximate. We want to adjust the parameters of the network so that it will produce the same response as the unknown function, if the same input is applied to both systems.

For our applications, the unknown function may correspond to a system we are trying to control, in which case the neural network will be the identified plant model. The unknown function could also represent the inverse of a system we are trying to control, in which case the neural network can be used to implement the controller. At the end of this paper we will present several control architectures demonstrating a variety of uses for function approximator neural networks.

In the next section we will present the multilayer perceptron neural network, and will demonstrate how it can be used as a function approximator.

2. MULTILAYER PERCEPTRON ARCHITECTURE

2.1. Neuron model

The multilayer perceptron neural network is built up of simple components. We will begin with a single-input neuron, which we will then extend to multiple inputs. We will next stack these neurons together to produce layers. Finally, we will cascade the layers together to form the network.

A single-input neuron is shown in Figure 2. The scalar input p is multiplied by the scalar *weight* w to form wp, one of the terms that is sent to the summer. The other input, 1, is multiplied by a bias b and then passed to the summer. The summer output n, often referred to as the *net input*, goes into a *transfer function* f, which produces the scalar neuron output a.

The neuron output is calculated as

$$a = f(wp + b)$$

Note that w and b are both *adjustable* scalar parameters of the neuron. Typically, the transfer function is chosen by the designer, and then the parameters w and b are adjusted by some learning rule so that the neuron input/output relationship meets some specific goal.

The transfer function in Figure 2 may be a linear or a nonlinear function of n. One of the most commonly used functions is the *log-sigmoid transfer function*, which is shown in Figure 3.

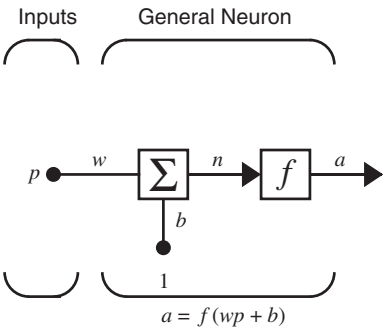

Figure 2. Single-input neuron.

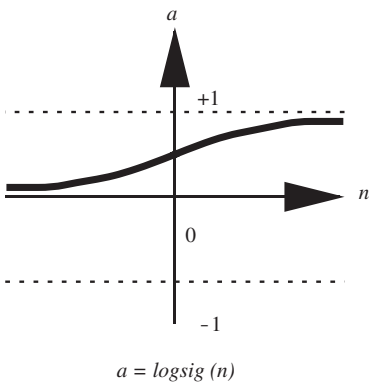

Figure 3. Log-sigmoid transfer function.

This transfer function takes the input (which may have any value between plus and minus infinity) and squashes the output into the range 0–1, according to the expression

$$a = \frac{1}{1 + e^{-n}} \tag{1}$$

The log-sigmoid transfer function is commonly used in multilayer networks that are trained using the backpropagation algorithm, in part because this function is differentiable.

Typically, a neuron has more than one input. A neuron with R inputs is shown in Figure 4. The individual inputs $p_1, p_2, \ldots, p_R$ are each weighted by corresponding elements $w_{1,1}, w_{1,2}, \ldots, w_{1,R}$ of the *weight matrix* $\mathbf{W}$.

The neuron has a bias b, which is summed with the weighted inputs to form the net input n:

$$n = w_{1,1}\,p_1 + w_{1,2}\,p_2 + \cdots + w_{1,R}\,p_R + b \tag{2}$$

This expression can be written in matrix form

$$n = \mathbf{W}\mathbf{p} + b \tag{3}$$

where the matrix $\mathbf{W}$ for the single neuron case has only one row.

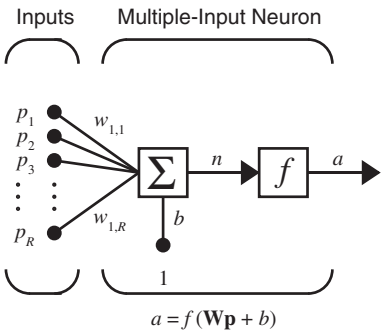

Figure 4. Multiple-input neuron.

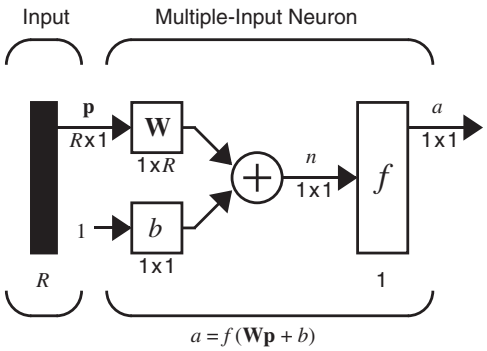

Figure 5. Neuron with R inputs, matrix notation.

Now the neuron output can be written as

$$a = f(\mathbf{W}\mathbf{p} + b) \tag{4}$$

Figure 5 represents the neuron in matrix form.

2.2. *Network architectures*

Commonly one neuron, even with many inputs, is not sufficient. We might need 5 or 10, operating in parallel, in what is called a *layer*. A single-layer network of S neurons is shown in Figure 6. Note that each of the R inputs is connected to each of the neurons and that the weight matrix now has S rows. The layer includes the weight matrix **W**, the summers, the bias vector **b**, the transfer function boxes and the output vector **a**. Some authors refer to the inputs as another layer, but we will not do that here. It is common for the number of inputs to a layer to be different from the number of neurons (i.e. $R \neq S$).

The S-neuron, R-input, one-layer network also can be drawn in matrix notation, as shown in Figure 7.

2.2.1. *Multiple layers of neurons*

Now consider a network with several layers. Each layer has its own weight matrix **W**, its own bias vector **b**, a net input vector **n** and an output vector **a**. We need to introduce some additional

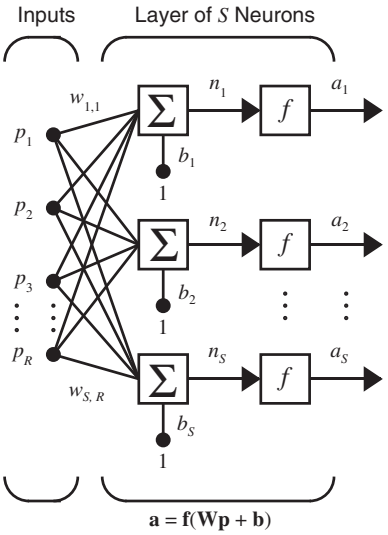

Figure 6. Layer of S neurons.

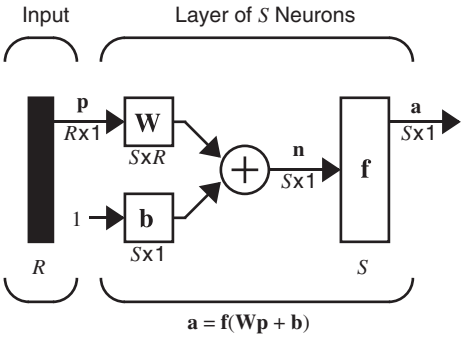

Figure 7. Layer of S neurons, matrix notation.

notation to distinguish between these layers. We will use superscripts to identify the layers. Thus, the weight matrix for the first layer is written as $\mathbf{W}^1$, and the weight matrix for the second layer is written as $\mathbf{W}^2$. This notation is used in the three-layer network shown in Figure 8. As shown, there are R inputs, S^1 neurons in the first layer, S^2 neurons in the second layer, etc. As noted, different layers can have different numbers of neurons.

The outputs of layers one and two are the inputs for layers two and three. Thus layer 2 can be viewed as a one-layer network with $R = S^1$ inputs, $S = S^2$ neurons, and an $S^2 \times S^1$ weight matrix $\mathbf{W}^2$. The input to layer 2 is $\mathbf{a}^1$, and the output is $\mathbf{a}^2$. A layer whose output is the network output is called an *output layer*. The other layers are called *hidden layers*. The network shown in Figure 8 has an output layer (layer 3) and two hidden layers (layers 1 and 2).

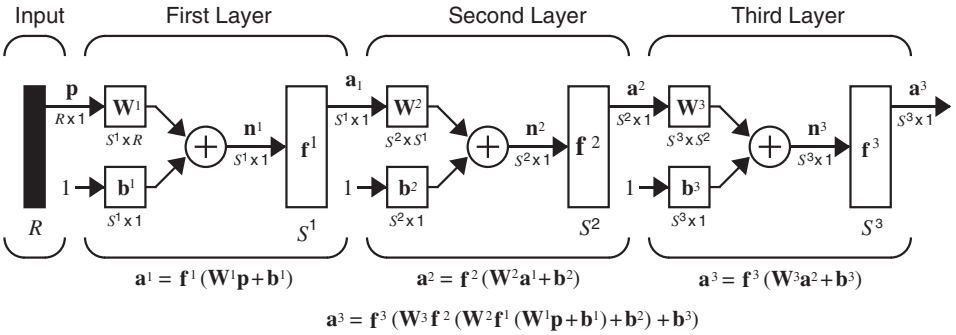

Figure 8. Three-layer network.

3. APPROXIMATION CAPABILITIES OF MULTILAYER NETWORKS

Two-layer networks, with sigmoid transfer functions in the hidden layer and linear transfer functions in the output layer, are universal approximators [4]. A simple example can demonstrate the power of this network for approximation.

Consider the two-layer, 1–2–1 network shown in Figure 9. For this example the transfer function for the first layer is log-sigmoid and the transfer function for the second layer is linear. In other words,

$$f^1(n) = \frac{1}{1+\mathrm{e}^{-n}} \quad \text{and} \quad f^2(n) = n \tag{5}$$

Suppose that the nominal values of the weights and biases for this network are

$$w^1_{1,1} = 10, \quad w^1_{2,1} = 10, \quad b^1_1 = -10, \quad b^1_2 = 10$$

$$w^2_{1,1} = 1, \quad w^2_{1,2} = 1, \quad b^2 = 0$$

The network response for these parameters is shown in Figure 10, which plots the network output a^2 as the input p is varied over the range $[-2, 2]$. Notice that the response consists of two steps, one for each of the log-sigmoid neurons in the first layer. By adjusting the network parameters we can change the shape and location of each step, as we will see in the following discussion. The centres of the steps occur where the net input to a neuron in the first layer is zero:

$$n^1_1 = w^1_{1,1} p + b^1_1 = 0 \quad \Rightarrow \quad p = -\frac{b^1_1}{w^1_{1,1}} = -\frac{-10}{10} = 1 \tag{6}$$

$$n^1_2 = w^1_{2,1} p + b^1_2 = 0 \quad \Rightarrow \quad p = -\frac{b^1_2}{w^1_{2,1}} = -\frac{10}{10} = -1 \tag{7}$$

The steepness of each step can be adjusted by changing the network weights.

Figure 11 illustrates the effects of parameter changes on the network response. The nominal response is repeated from Figure 10. The other curves correspond to the network response when one parameter at a time is varied over the following ranges:

$$-1 \leqslant w^2_{1,1} \leqslant 1, \quad -1 \leqslant w^2_{1,2} \leqslant 1, \quad 0 \leqslant b^1_2 \leqslant 20, \quad -1 \leqslant b^2 \leqslant 1 \tag{8}$$

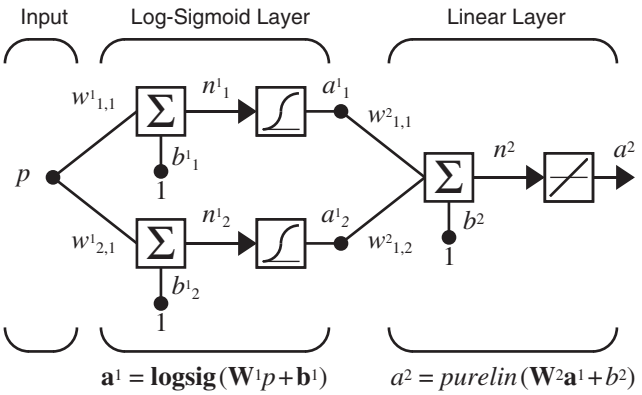

Figure 9. Example function approximation network.

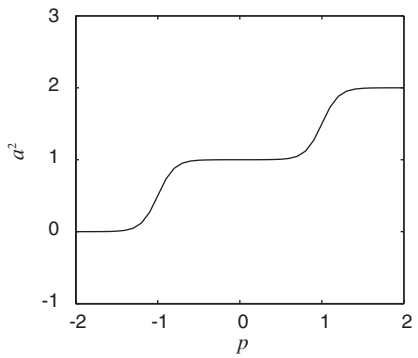

Figure 10. Nominal response of network of Figure 9.

Figure 11(a) shows how the network biases in the first (hidden) layer can be used to locate the position of the steps. Figures 11(b) and (c) illustrate how the weights determine the slope of the steps. The bias in the second (output) layer shifts the entire network response up or down, as can be seen in Figure 11(d).

From this example we can see how flexible the multilayer network is. It would appear that we could use such networks to approximate almost any function, if we had a sufficient number of neurons in the hidden layer. In fact, it has been shown that two-layer networks, with sigmoid transfer functions in the hidden layer and linear transfer functions in the output layer, can approximate virtually any function of interest to any degree of accuracy, provided sufficiently many hidden units are available. It is beyond the scope of this paper to provide detailed discussions of approximation theory, but there are many papers in the literature that can provide a deeper discussion of this field. In Reference [4], Hornik, Stinchcombe and White present a proof that multilayer perceptron networks are universal approximators. Pinkus gives a more recent review of the approximation capabilities of neural networks in Reference [5]. Niyogi and Girosi [6] develop bounds on function approximation error when the network is trained on noisy data.

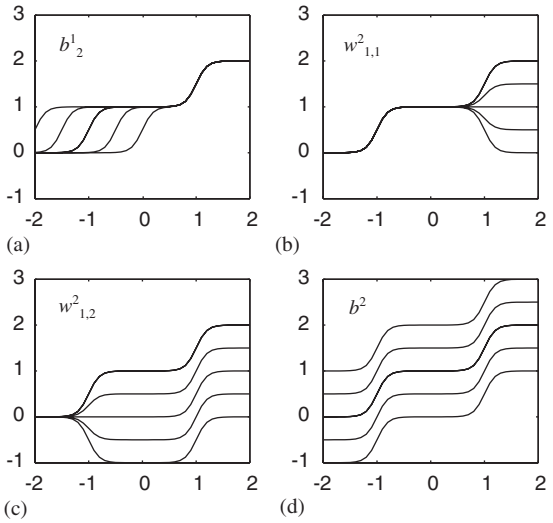

Figure 11. Effect of parameter changes on network response.

4. TRAINING MULTILAYER NETWORKS

Now that we know multilayer networks are universal approximators, the next step is to determine a procedure for selecting the network parameters (weights and biases) that will best approximate a given function. The procedure for selecting the parameters for a given problem is called *training* the network. In this section we will outline a training procedure called *backpropagation* [7,8], which is based on gradient descent. (More efficient algorithms than gradient descent are often used in neural network training. [1].)

As we discussed earlier, for multilayer networks the output of one layer becomes the input to the following layer (see Figure 8). The equations that describe this operation are

$$\mathbf{a}^{m+1} = \mathbf{f}^{m+1}(\mathbf{W}^{m+1}\mathbf{a}^m + \mathbf{b}^{m+1}) \quad \text{for } m = 0, 1, \ldots, M-1 \tag{9}$$

where M is the number of layers in the network. The neurons in the first layer receive external inputs:

$$\mathbf{a}^0 = \mathbf{p} \tag{10}$$

which provides the starting point for Equation (9). The outputs of the neurons in the last layer are considered the network outputs:

$$\mathbf{a} = \mathbf{a}^M \tag{11}$$

4.1. Performance index

The backpropagation algorithm for multilayer networks is a gradient descent optimization procedure in which we minimize a mean square error performance index. The algorithm is

provided with a set of examples of proper network behaviour:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\} \tag{12}$$

where $\mathbf{p}_q$ is an input to the network and $\mathbf{t}_q$ is the corresponding target output. As each input is applied to the network, the network output is compared to the target. The algorithm should adjust the network parameters in order to minimize the sum squared error:

$$F(\mathbf{x}) = \sum_{q=1}^{Q} e_q^2 = \sum_{q=1}^{Q} (t_q - a_q)^2 \tag{13}$$

where $\mathbf{x}$ is a vector containing all network weights and biases. If the network has multiple outputs this generalizes to

$$F(\mathbf{x}) = \sum_{q=1}^{Q} \mathbf{e}_q^{\mathrm{T}} \mathbf{e}_q = \sum_{q=1}^{Q} (\mathbf{t}_q - \mathbf{a}_q)^{\mathrm{T}} (\mathbf{t}_q - \mathbf{a}_q) \tag{14}$$

Using a stochastic approximation, we will replace the sum squared error by the error on the latest target:

$$\hat{F}(\mathbf{x}) = (\mathbf{t}(k) - \mathbf{a}(k))^{\mathrm{T}} (\mathbf{t}(k) - \mathbf{a}(k)) = \mathbf{e}^{\mathrm{T}}(k)\mathbf{e}(k) \tag{15}$$

where the expectation of the squared error has been replaced by the squared error at iteration k.

The steepest descent algorithm for the approximate mean square error is

$$w_{i,j}^m(k+1) = w_{i,j}^m(k) - \alpha \frac{\partial \hat{F}}{\partial w_{i,j}^m} \tag{16}$$

$$b_i^m(k+1) = b_i^m(k) - \alpha \frac{\partial \hat{F}}{\partial b_i^m} \tag{17}$$

where α is the learning rate.

4.2. Chain rule

For a single-layer linear network, these partial derivatives in Equations (16) and (17) are conveniently computed, since the error can be written as an explicit linear function of the network weights. For the multilayer network, the error is not an explicit function of the weights in the hidden layers, therefore these derivatives are not computed so easily.

Because the error is an indirect function of the weights in the hidden layers, we will use the chain rule of calculus to calculate the derivatives in Equations (16) and (17):

$$\frac{\partial \hat{F}}{\partial w_{i,j}^m} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial w_{i,j}^m} \tag{18}$$

$$\frac{\partial \hat{F}}{\partial b_i^m} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial b_i^m} \tag{19}$$

The second term in each of these equations can be easily computed, since the net input to layer m is an explicit function of the weights and bias in that layer:

$$n_i^m = \sum_{j=1}^{s^{m-1}} w_{i,j}^m a_j^{m-1} + b_i^m \tag{20}$$

Therefore,

$$\frac{\partial n_i^m}{\partial w_{i,j}^m} = a_j^{m-1}, \quad \frac{\partial n_i^m}{\partial b_i^m} = 1 \tag{21}$$

If we now define

$$s_i^m \equiv \frac{\partial \hat{F}}{\partial n_i^m} \tag{22}$$

(the *sensitivity* of $\hat{F}$ to changes in the ith element of the net input at layer m), then Equations (18) and (19) can be simplified to

$$\frac{\partial \hat{F}}{\partial w_{i,j}^m} = s_i^m a_j^{m-1} \tag{23}$$

We can now express the approximate steepest descent algorithm as

$$w_{i,j}^m(k+1) = w_{i,j}^m(k) - \alpha s_i^m a_j^{m-1} \tag{25}$$

$$b_i^m(k+1) = b_i^m(k) - \alpha s_i^m \tag{26}$$

In matrix form this becomes

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha \mathbf{s}^m (\mathbf{a}^{m-1})^{\mathrm{T}} \tag{27}$$

$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \alpha \mathbf{s}^m \tag{28}$$

where the individual elements of $\mathbf{s}^m$ are given by Equation (22).

4.3. Backpropagating the sensitivities

It now remains for us to compute the sensitivities $\mathbf{s}^m$, which requires another application of the chain rule. It is this process that gives us the term *backpropagation*, because it describes a recurrence relationship in which the sensitivity at layer m is computed from the sensitivity at layer $m+1$:

$$\mathbf{s}^M = -2\dot{\mathbf{F}}^M(\mathbf{n}^M)(\mathbf{t} - \mathbf{a}) \tag{29}$$

$$\mathbf{s}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^{\mathrm{T}}\mathbf{s}^{m+1}, \quad m = M-1, \ldots, 2, 1 \tag{30}$$

where

$$\dot{\mathbf{F}}^m(\mathbf{n}^m) = \begin{bmatrix} \dot{f}^m(n_1^m) & 0 & \ldots & 0 \\ 0 & \dot{f}^m(n_2^m) & \ldots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \ldots & \dot{f}^m(n_{s^m}^m) \end{bmatrix} \tag{31}$$

(See Reference [1, Chapter 11] for a derivation of this result.)

4.4. Variations of backpropagation

In some ways it is unfortunate that the algorithm we usually refer to as backpropagation, given by Equations (27) and (28), is in fact simply a steepest descent algorithm. There are many other optimization algorithms that can use the backpropagation procedure, in which derivatives are processed from the last layer of the network to the first (as given in Equation (30)). For example, conjugate gradient and quasi-Newton algorithms [9–11] are generally more efficient than steepest descent algorithms, and yet they can use the same backpropagation procedure to compute the necessary derivatives. The Levenberg–Marquardt algorithm is very efficient for training small to medium-size networks, and it uses a backpropagation procedure that is very similar to the one given by Equation (30) [12].

4.5. Generalization (interpolation & extrapolation)

We now know that multilayer networks are universal approximators, but we have not discussed how to select the number of neurons and the number of layers necessary to achieve an accurate approximation in a given problem. We have also not discussed how the training data set should be selected. The trick is to use enough neurons to capture the complexity of the underlying function without having the network overfit the training data, in which case it will not *generalize* to new situations. We also need to have sufficient training data to adequately represent the underlying function.

To illustrate the problems we can have in network training, consider the following general example. Assume that the training data is generated by the following equation:

$$\mathbf{t}_q = \mathbf{g}(\mathbf{p}_q) + \mathbf{e}_q \tag{32}$$

where $\mathbf{p}_q$ is the system input, $\mathbf{g}(\cdot)$ is the underlying function we wish to approximate, $\mathbf{e}_q$ is the measurement noise, and $\mathbf{t}_q$ is the system output (network target). Figure 12 shows an example of the underlying function $\mathbf{g}(\cdot)$ (thick line), training data target values $\mathbf{t}_q$ (circles), and total trained network response (thin line). The two graphs of Figure 12 represent different training strategies.

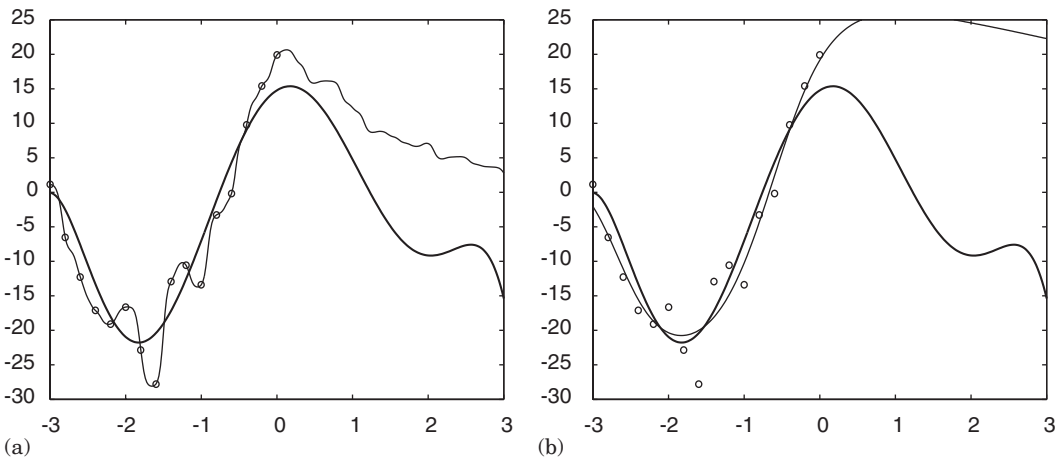

Figure 12. Example of (a) overfitting and (b) good fit.

In the example shown in Figure 12(a), a large network was trained to minimize squared error (Equation (13)) over the 15 points in the training set. We can see that the network response exactly matches the target values for each training point. However, the total network response has failed to capture the underlying function. There are two major problems. First, the network has overfit on the training data. The network response is too complex, because the network has more than enough independent parameters (61), and they have not been constrained in any way. The second problem is that there is no training data for values of ρ greater than 0. Neural networks (and other nonlinear black box techniques) cannot be expected to *extrapolate* accurately. If the network receives an input that is outside of the range covered in the training data, then the network response will always be suspect.

While there is little we can do to improve the network performance outside the range of the training data, we can improve its ability to *interpolate* between data points. Improved generalization can be obtained through a variety of techniques. In one method, called early stopping [13], we place a portion of the training data into a validation data set. The performance of the network on the validation set is monitored during training. During the early stages of training the validation error will come down. When overfitting begins, the validation error will begin to increase, and at this point the training is stopped.

Another technique to improve network generalization is called regularization. With this method the performance index is modified to include a term which penalizes network complexity. The most common penalty term is the sum of squares of the network weights:

$$F(\mathbf{x}) = \sum_{q=1}^{Q} \mathbf{e}_q^{\mathrm{T}} \mathbf{e}_q + \rho \sum (w_{i,j}^k)^2 \tag{33}$$

This performance index forces the weights to be small, which produces a smoother network response. The trick with this method is to choose the correct regularization parameter ρ. If the value is too large, then the network response will be too smooth and will not accurately approximate the underlying function. If the value is too small, then the network will overfit. There are a number of methods for selecting the optimal ρ. One of the most successful is Bayesian regularization [14,15]. Figure 12(b) shows the network response when the network is trained with Bayesian regularization. Notice that the network response no longer exactly matches the training data points, but the overall network response more closely matches the underlying function over the range of the training data.

A complete discussion of generalization and overfitting is beyond the scope of this paper. The interested reader is referred to References [1,3,6,14,15].

In the next section we will describe how multilayer networks can be used in neurocontrol applications.

5. CONTROL SYSTEM APPLICATIONS

Multilayer neural networks have been applied successfully in the identification and control of dynamic systems [16,17]. Rather than attempt to survey the many ways in which multilayer networks have been used in control systems, we will concentrate on three typical neural network controllers: model predictive control [18], NARMA-L2 control [19], and model reference control [20]. These controllers are representative of the variety of common ways in which

multilayer networks are used in control systems. As with most neural controllers, they are based on standard linear control architectures.

There are typically two steps involved when using neural networks for control: system identification and control design. In the system identification stage, we develop a neural network model of the plant that we want to control. In the control design stage, we use the neural network plant model to design (or train) the controller. In each of the three control architectures described in this paper, the system identification stage is identical. The control design stage, however, is different for each architecture. The next three subsections of this paper discuss model predictive control, NARMA-L2 control and model reference control and will describe how they can be applied in practice. Finally, we give a summary of the characteristics of the three controllers.

5.1. NN predictive control

There are a number of variations of the neural network predictive controller that are based on linear model predictive controllers [21]. The neural network predictive controller that is discussed in this paper (based in part on Reference [18]) uses a neural network model of a nonlinear plant to predict future plant performance. The controller then calculates the control input that will optimize plant performance over a specified future time horizon. The first step in model predictive control is to determine the neural network plant model (system identification). Next, the plant model is used by the controller to predict future performance.

The next section describes the system identification process. This is followed by a description of the optimization process and an application of predictive control to a magnetic levitation system.

5.1.1. System identification

The first stage of model predictive control (as well as the other two control architectures discussed in this paper) is to train a neural network to represent the forward dynamics of the plant. The prediction error between the plant output and the neural network output is used as the neural network training signal. The process is represented by Figure 13.

One standard model that has been used for nonlinear identification is the nonlinear autoregressive-moving average (NARMA) [19] model:

$$y(k+d) = h[y(k), y(k-1), \ldots, y(k-n+1), u(k), u(k-1), \ldots, u(k-m+1)] \tag{34}$$

where $u(k)$ is the system input $y(k)$ is the system output and d is the system delay (we will use a delay of 1 for the predictive controller). For the identification phase, we train a neural network to approximate the nonlinear function h. The structure of the neural network plant model is given in Figure 14, where the blocks labelled TDL are tapped delay lines that store previous values of the input signal. The equation for the plant model is given by

$$y_m(k+1) = \hat{h}[y_p(k), \ldots, y_p(k-n+1), u(k), \ldots, u(k-m+1); \mathbf{x}] \tag{35}$$

where $\hat{h}[., \mathbf{x}]$ is the function implemented by the neural network, and $\mathbf{x}$ is the vector containing all network weights and biases.

We have modified our previous notation here, to allow more than one input into the network. $\mathbf{IW}^{i,j}$ is a weight matrix from input number j to layer number i. $\mathbf{LW}^{i,j}$ is a weight matrix from layer number j to layer number i.

Although there are delays in this network, they occur only at the network input, and the network contains no feedback loops. For these reasons, the neural network plant model can be

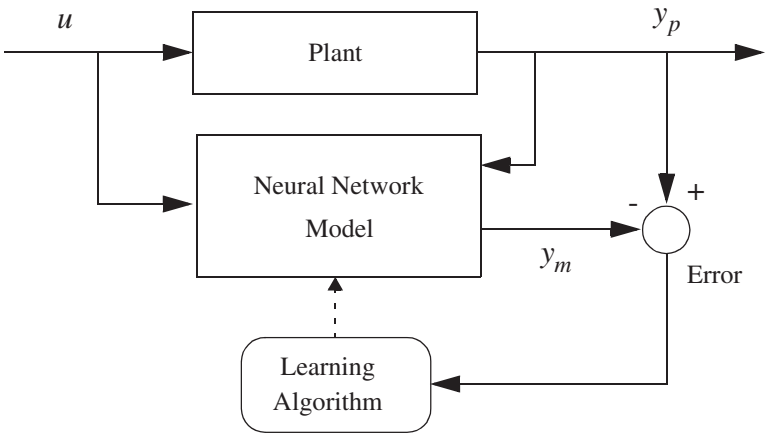

Figure 13. Plant identification.

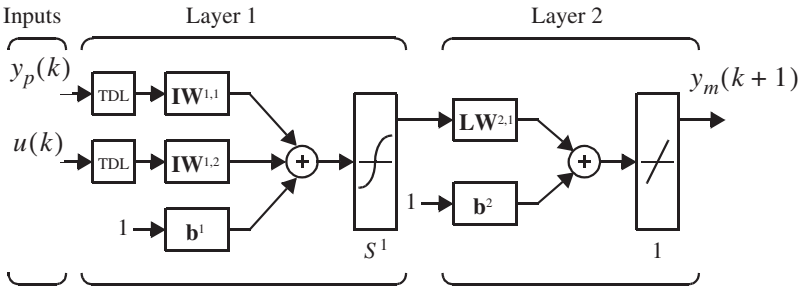

Figure 14. Neural network plant model.

trained using the backpropagation methods for feedforward networks described in the first part of this paper. (This is true for the system identification step of each of the controllers discussed in this paper.) It is important that the training data cover the entire range of plant operation, because we know from previous discussions that nonlinear neural networks do not extrapolate accurately. The input to this network is an $(n_y + n_u)$-dimensional vector of previous plant outputs and inputs. It is this space that must be covered adequately by the training data. The system identification process will be demonstrated in Section 5.1.3.

5.1.2. Predictive control

The model predictive control method is based on the receding horizon technique [18]. The neural network model predicts the plant response over a specified time horizon. The predictions are used by a numerical optimization program to determine the control signal that minimizes the following performance criterion over the specified horizon:

$$J = \sum_{j=N_1}^{N_2} (y_r(k+j) - y_m(k+j))^2 + \rho \sum_{j=1}^{N_u} (u'(k+j-1) - u'(k+j-2))^2 \tag{36}$$

where N_1, N_2 and N_u define the horizons over which the tracking error and the control increments are evaluated. The u' variable is the tentative control signal, y_r is the desired response

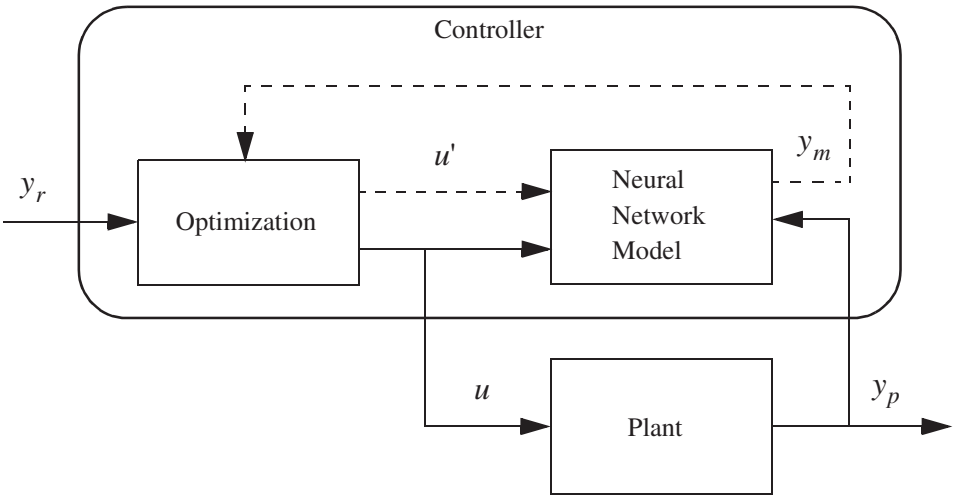

Figure 15. Neural network predictive control.

and y_m is the network model response. The ρ value determines the contribution that the sum of the squares of the control increments has on the performance index.

The following block diagram illustrates the model predictive control process. The controller consists of the neural network plant model and the optimization block. The optimization block determines the values of u' that minimize J, and then the optimal u is input to the plant. For the purposes of this paper, the BFGS quasi-Newton algorithm, with a backtracking line search [22], was used for the optimization block. (Figure 15)

5.1.3. Application—magnetic levitation system

Now we will demonstrate the predictive controller by applying it to a simple test problem. In this test problem, the objective is to control the position of a magnet suspended above an electromagnet, where the magnet is constrained so that it can only move in the vertical direction, as shown in Figure 16.

The equation of motion of the magnet is

$$\frac{d^2 y(t)}{dt^2} = -g + \frac{\alpha}{M}\frac{i^2(t)\mathrm{sgn}[i(t)]}{y(t)} - \frac{\beta}{M}\frac{dy(t)}{dt} \tag{37}$$

where $y(t)$ is the distance of the magnet above the electromagnet, $i(t)$ is the current flowing in the electromagnet, M is the mass of the magnet, and g is the gravitational constant. The parameter β is a viscous friction coefficient that is determined by the material in which the magnet moves, and α is a field strength constant that is determined by the number of turns of wire on the electromagnet and the strength of the magnet. For our simulations, the current is allowed to range from 0 to 4 As, and the sampling interval for the controller is 0.01 s. The parameter values are set to $\beta = 12$, $\alpha = 15$, $g = 9.8$ and $M = 3$.

The first step in the control design process is the development of the plant model. The performances of neural network controllers are highly dependent on the accuracy of the plant

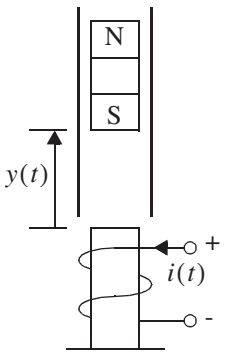

Figure 16. Magnetic levitation system.

identification. In this section we will begin with a discussion of some of the procedures that can be used to facilitate accurate plant identification.

As with linear system identification, we need to insure that the plant input is sufficiently exciting [23]. For nonlinear black box identification, we also need to be sure that the system inputs and outputs cover the operating range for which the controller will be applied. For our applications, we typically collect training data while applying random inputs which consist of a series of pulses of random amplitude and duration. The duration and amplitude of the pulses must be chosen carefully to produce accurate identification.

If the identification is poor, then the resulting control system may fail. Controller performance tends to fail in either steady-state operation, or transient operation, or both. When steady-state performance is poor, it is useful to increase the duration of the input pulses. Unfortunately, within a training data set, if we have too much data in steady-state conditions, the training data may not be representative of typical plant behaviour. This is due to the fact that the input and output signals do not adequately cover the region that is going to be controlled. This will result in poor transient performance. We need to choose the training data so that we produce adequate transient and steady-state performance. The following example will illustrate the performances of the predictive controllers when we use different ranges for the pulse widths of the input signal to generate the training data.

We found that it took about 4.5 s for the magnetic levitation system to reach steady-state in open-loop conditions. Therefore, we first specified a pulse width range of $0.01 < \tau < 5$. The neural network plant model used three delayed values of current ($m = 3$) and three delayed values of magnet position ($n = 3$) as input to the network, and 10 neurons were used in the hidden layer. After training the network with the data set shown in Figure 17, the resulting neural network predictive control system was unstable. (The network was trained with Bayesian regularization [15] to prevent overfitting.)

Based on the poor response of the initial controller, we determined that the training data did not provide significant coverage. Therefore, we changed the range of the input pulse widths to $0.01 < \tau < 1$, as shown in Figure 18. From this figure, we can see that the training data is more dense and provides wider coverage of the plant model input space than the first data set. After training the network using this data set, the resulting predictive control system was stable, although it resulted in large steady-state errors.

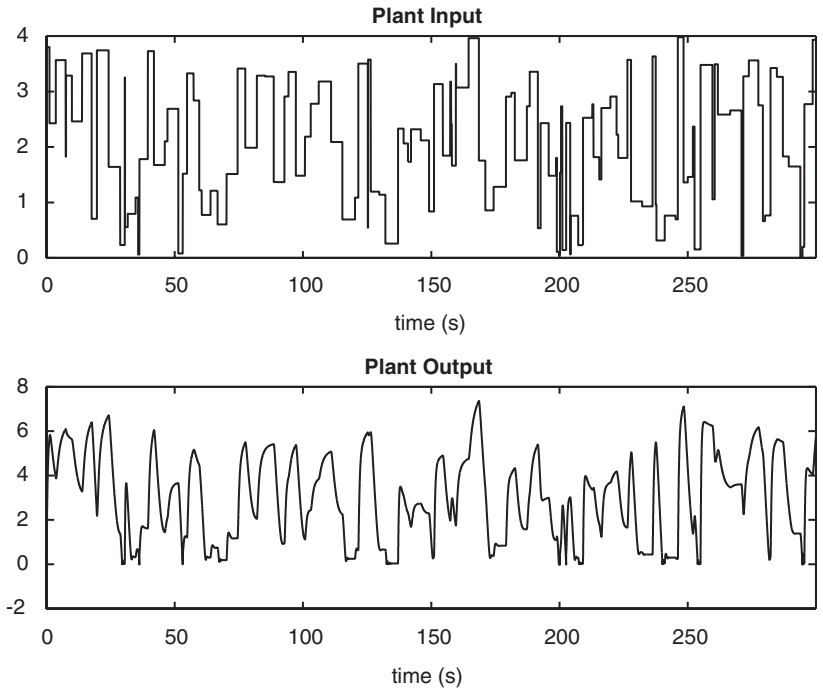

Figure 17. Training data with a long pulse width.

In the third test, we combined short pulse width and long pulse width (steady-state) data. The long pulses were concentrated only on some ranges of plant outputs. For example, we chose to have steady-state data over the ranges where the tracking errors from the previous case were large. The input and output signals are shown in Figure 19. The resulting controller performed well in both transient and steady-state conditions.

The left graph in Figure 20 shows the reference signal and the position of the magnet for the final predictive controller (using the neural network trained with the data shown in Figure 19 and controller parameters set to $N_2 = 15,\ N_u = 3,\ \rho = 0.01$). Steady-state errors were small, and the transient performance was adequate in all tested regions. We found that stability was strongly influenced by the selection of ρ. As we decrease ρ, the control signal tends to change more abruptly, generating a noisy plant output. As with linear predictive control, when we increase ρ too much, the control action is excessively smooth and the response is slow. The control action for this example is shown in the right graph of Figure 20.

5.2. *NARMA-L2 control*

The neurocontroller described in this section is referred to by two different names: feedback linearization control and NARMA-L2 control. It is referred to as feedback linearization when the plant model has a particular form (companion form) [24]. It is referred to as NARMA-L2 control when the plant model can be approximated by the same form. The central idea of this type of control is to transform nonlinear system dynamics into linear dynamics by cancelling the

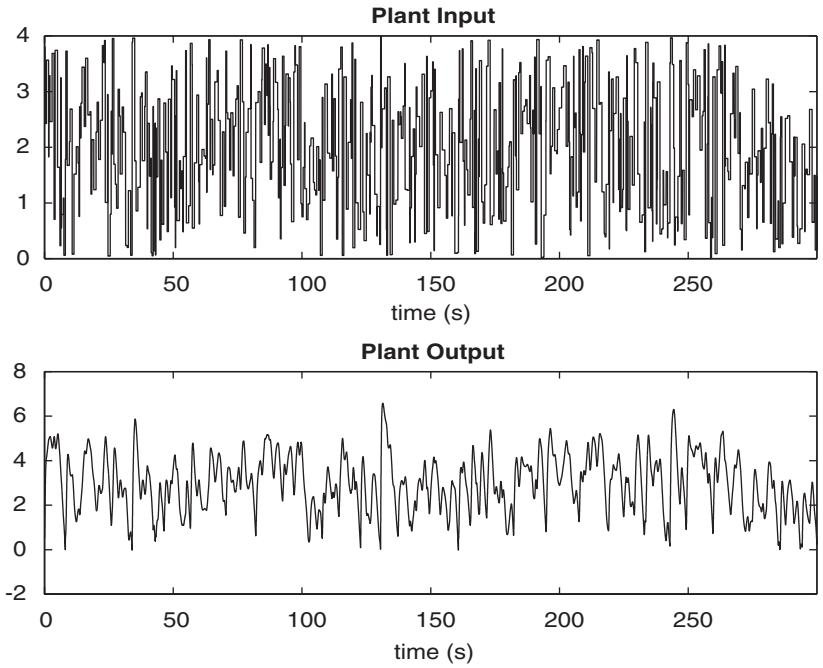

Figure 18. Training data with a short pulse width.

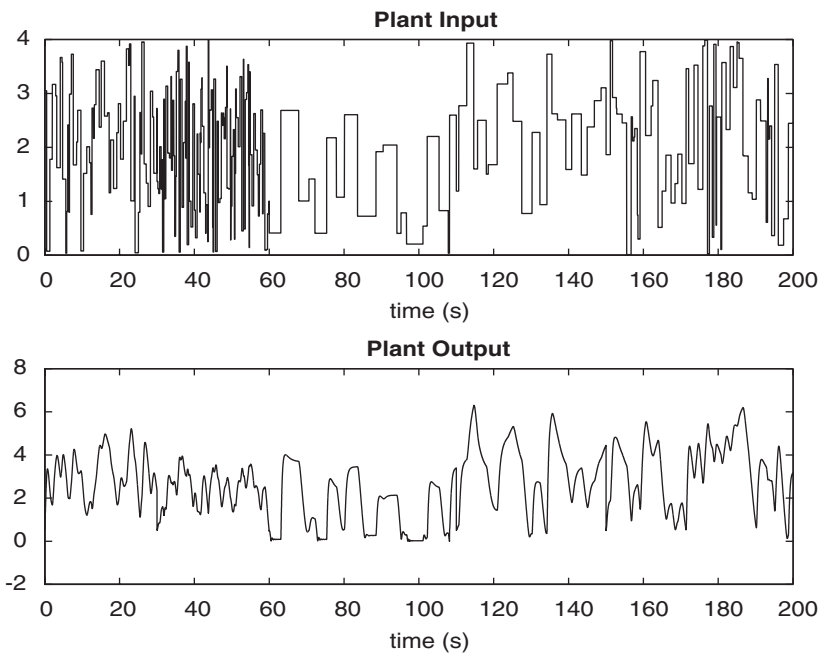

Figure 19. Training data with mixed pulse width.

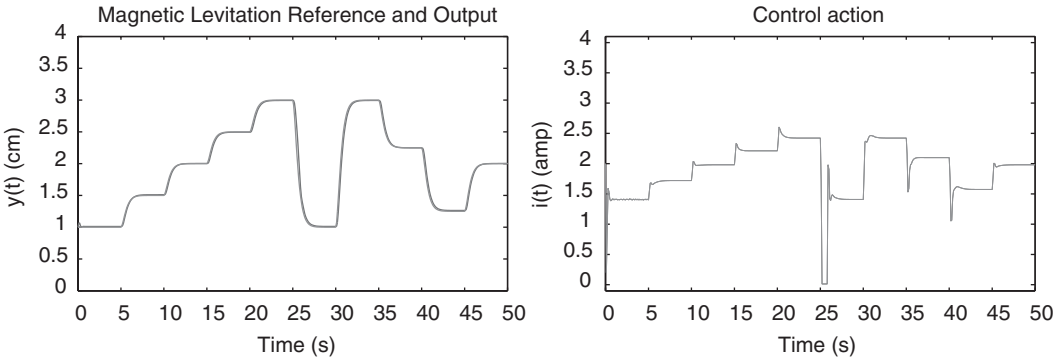

Figure 20. MagLev response and control action using the predictive controller.

nonlinearities. This section begins by presenting the companion form system model and demonstrating how a neural network can be used to identify this model. Then it describes how the identified neural network model can be used to develop a controller.

5.2.1. Identification of the NARMA-L2 model

As with model predictive control, the first step in using NARMA-L2 control is to identify the system to be controlled. The NARMA-L2 model [18] is an approximation of the NARMA model of Equation (34). The NARMA-L2 approximate model is given by

$$\hat{y}(k+d) = f[y(k), y(k-1), \ldots, y(k-n+1), u(k-1), \ldots, u(k-m+1)] + g[y(k), y(k-1), \ldots, y(k-n+1), u(k-1), \ldots, u(k-m+1)]u(k) \quad (38)$$

This model is in companion form, where the next controller input $u(k)$ is not contained inside the nonlinearity. Figure 21 shows the structure of a neural network NARMA-L2 representation for $d = 1$. Notice that we have separate subnetworks to represent the functions and $g(\ )$ and $f(\ )$.

5.2.2. NARMA-L2 controller

The advantage of the NARMA-L2 form is that you can solve for the control input that causes the system output to follow a reference signal: $y(k+d) = y_r(k+d)$. The resulting controller would have the form

$$u(k) = \frac{y_r(k+d) - f[y(k), y(k-1), \ldots, y(k-n+1), u(k-1), \ldots, u(k-m+1)]}{g[y(k), y(k-1), \ldots, y(k-n+1), u(k-1), \ldots, u(k-m+1)]} \quad (39)$$

which is realizable for $d \geqslant 1$. Figure 22 is a block diagram of the NARMA-L2 controller.

This controller can be implemented with the previously identified NARMA-L2 plant model, as shown in Figure 23.

5.2.3. Application—continuous stirred tank reactor

To demonstrate the NARMA-L2 controller, we use a catalytic continuous stirred tank reactor (CSTR) [25]. The dynamic model of the system is

$$\frac{\mathrm{d}h(t)}{\mathrm{d}t} = w_1(t) + w_2(t) - 0.2\sqrt{h(t)} \quad (40)$$

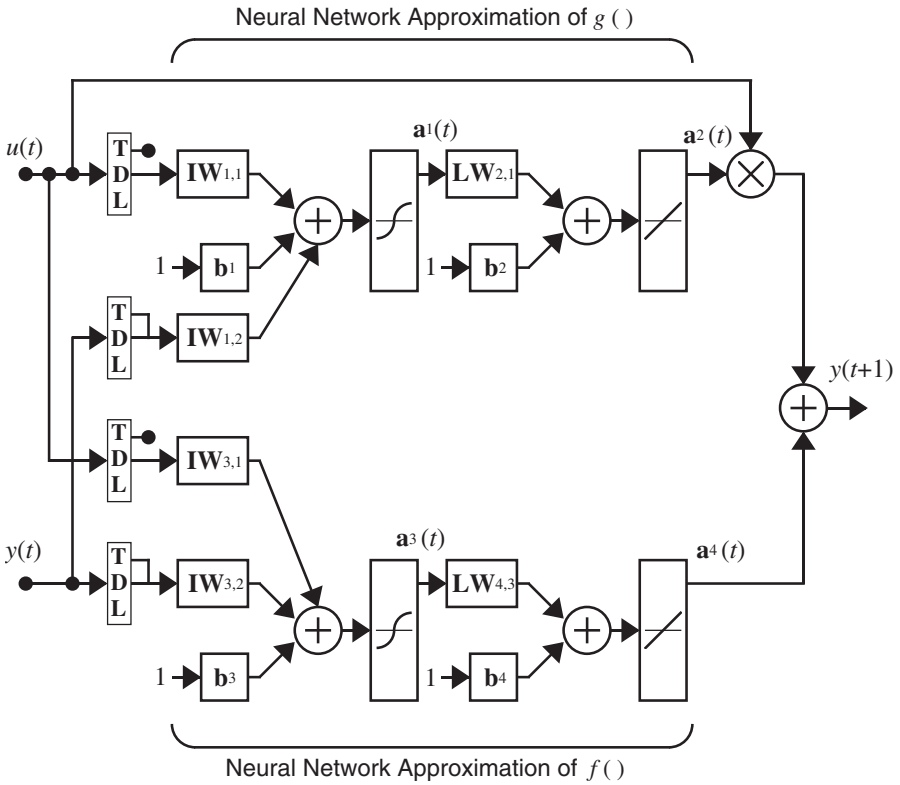

Figure 21. NARMA-L2 plant model.

$$\frac{\mathrm{d}C_b(t)}{\mathrm{d}t} = (C_{b1} - C_b(t))\frac{w_1(t)}{h(t)} + (C_{b2} - C_b(t))\frac{w_2(t)}{h(t)} - \frac{k_1 C_b(t)}{(1 + k_2 C_b(t))^2} \tag{41}$$

where $h(t)$ is the liquid level, $C_b(t)$ is the product concentration at the output of the process, $w_1(t)$ is the flow rate of the concentrated feed C_{b1}, and $w_2(t)$ is the flow rate of the diluted feed C_{b2}. The input concentrations are set to $C_{b1} = 24.9$ mol/cm^3 and $C_{b2} = 0.1$ mol/cm^3. The constants associated with the rate of consumption are $k_1 = 1$ and $k_2 = 1$. The objective of the controller is to maintain the product concentration by adjusting the flow $w_2(t)$. To simplify the demonstration, we set $w_1(t) = 0.1$ cm^3/s. The allowable range for $w_2(t)$ was assigned to be $[0, 4]$. The level of the tank $h(t)$ is not controlled for this experiment.

For the system identification phase, we used the same form of input signal as was used for the MagLev system. The pulse widths were allowed to range from 5 to 20 s, and the amplitude was varied from 0 to 4 cm^3/s. The neural network plant model used three delayed values of $w_2(t)$ ($n_u = 3$) and three delayed values of $C_b(t)$ ($n_y = 3$) as input to the network, and 3 neurons were used in the hidden layers. The sampling interval was set to 0.01 s.

The left graph of Figure 24 shows the reference signal and the system response for the NARMA-L2 controller. The output tracks the reference accurately, without significant overshoot. However, the NARMA-L2 controller generally produces more oscillatory control

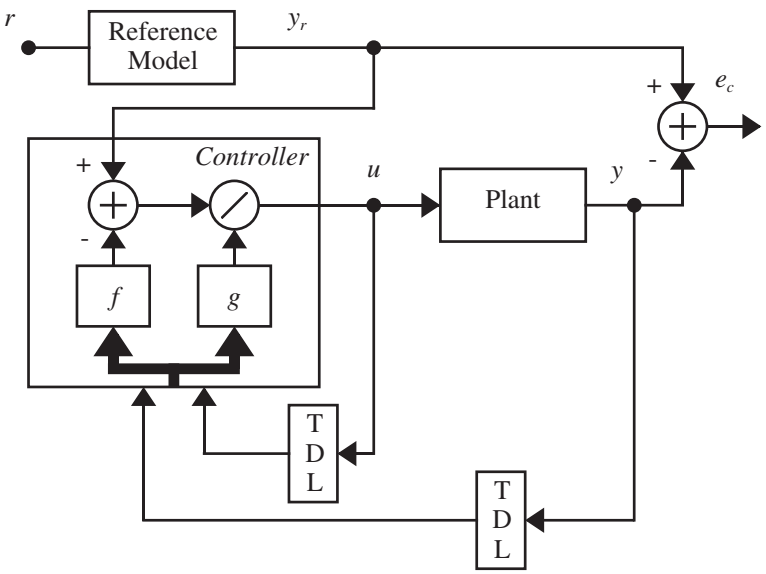

Figure 22. NARMA-L2 controller.

signals than the other controllers discussed here. This is illustrated in the control action plot shown in the right graph of Figure 24. This chattering can be reduced by filtering (as in sliding mode control), but it is typical of NARMA-L2 controllers.

5.3. *Model reference control*

The third neural control architecture we will discuss in this paper is model reference control [19]. This architecture uses two neural networks: a controller network and a plant model network, as shown in Figure 25. The plant model is identified first, and then the controller is trained so that the plant output follows the reference model output.

The online computation of the model reference controller, as with NARMA-L2, is minimal. However, unlike NARMA-L2, the model reference architecture requires that a separate neural network controller be trained, in addition to the neural network plant model. The controller training is computationally expensive, since it requires the use of dynamic backpropagation [26,19]. On the positive side, model reference control applies to a larger class of plant than does NARMA-L2 control, which requires that the plant be approximated by a companion form model.

Figure 26 shows the details of the neural network plant model and the neural network controller. There are three sets of controller inputs: delayed reference inputs, delayed controller outputs (plant inputs), and delayed plant outputs. For each of these inputs, we select the number of delayed values to use. Typically, the number of delays increases with the order of the plant. There are two sets of inputs to the neural network plant model: delayed controller outputs and delayed plant outputs.

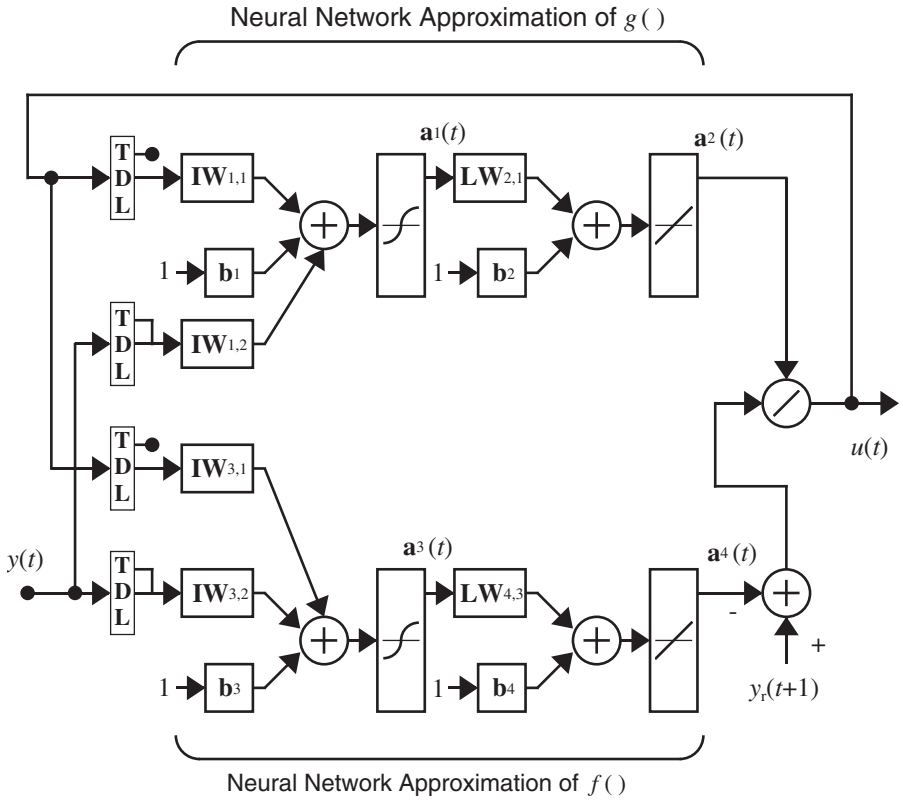

Figure 23. Implementation of NARMA-L2 controller.

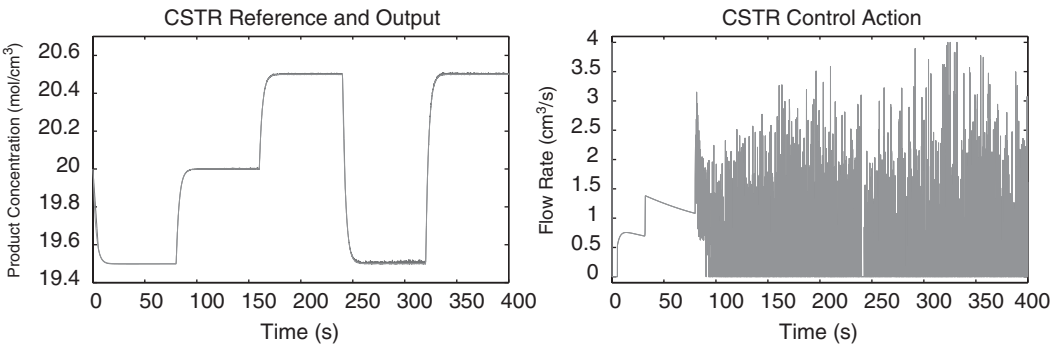

Figure 24. CSTR response and control action using the NARMA-L2 controller.

The plant identification process for model reference control is the same as that for the model predictive control, and uses the same NARMA model given by Equation (34). The training of the neural network controller, however, is more complex.

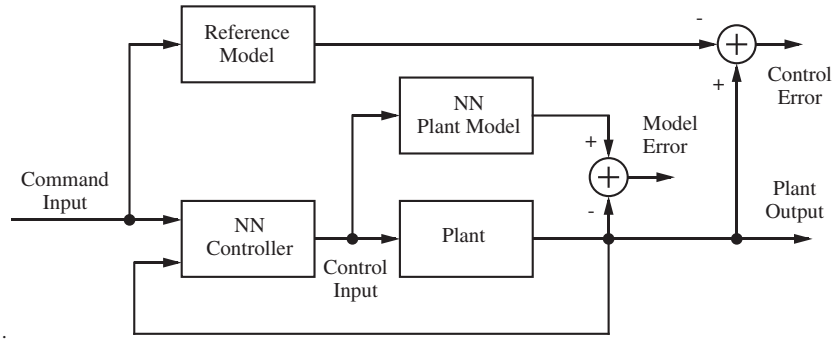

Figure 25. Model reference control architecture.

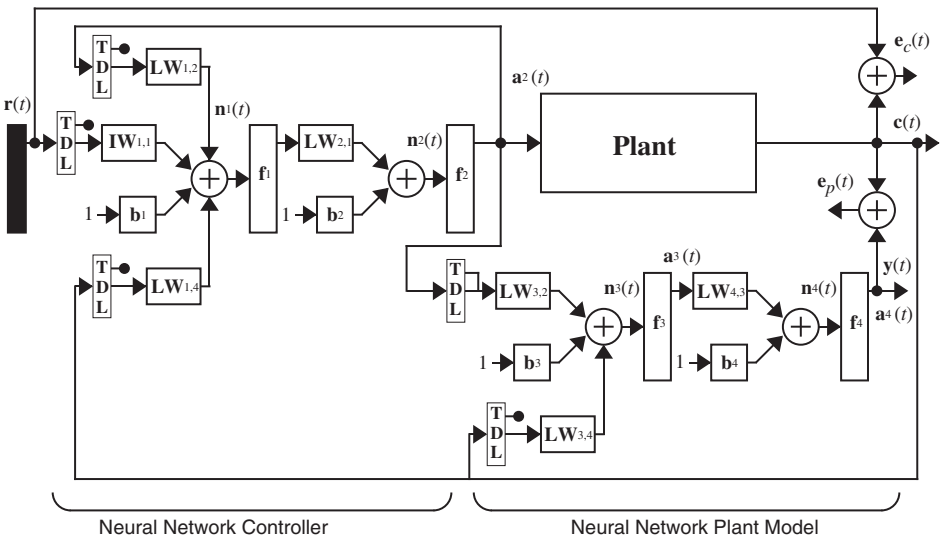

Figure 26. Detailed model reference control structure.

It is clear from Figure 26 that the model reference control structure is a recurrent (feedback) network. This type of network is more difficult to train than the feedforward networks that were discussed in the first half of this paper and that are used for plant identification. Suppose that we use the same gradient descent algorithm, Equation (16), that is used in the standard backpropagation algorithm. The problem with recurrent networks is that when we try to find the equivalent of Equation (23) (gradient calculation) we note that the weights and biases have two different effects on the network output. The first is the direct effect, which is accounted for by Equation (23). The second is an indirect effect, since some of the inputs to the network, such as $u(t-1)$, are also functions of the weights and biases. To account for this indirect effect we must use dynamic backpropagation to compute the gradients for recurrent networks. We do not have space in this paper to describe the various forms of dynamic backpropagation. The reader is referred to References [27,28] for a development of these algorithms for networks such as Figure 26.

In addition to the difficulty in computing the gradients for recurrent networks, the error surfaces for these networks pose special difficulties for gradient-based optimization algorithms. Gradient-based training procedures that are well suited to training recurrent networks are discussed in Reference [29].

The data used to train the model reference controller is generated while applying a random reference signal which consists of a series of pulses of random amplitude and duration. This data can be generated without running the plant, but using the neural network model output in place of the plant output.

5.3.1. Application—robot arm

We will demonstrate the model reference controller on the simple, single-link robot arm shown in Figure 27.

The equation of motion for the arm is

$$\frac{d^2\phi}{dt^2} = -10 \sin \phi - 2\frac{d\phi}{dt} + u \tag{42}$$

where ϕ is the angle of the arm, and u is the torque supplied by the DC motor. The system was sampled at intervals of 0.05 s. To identify the system, we used input pulses with intervals between 0.1 and 2 s, and amplitudes between -15 and $+15$ N m. The neural network plant model used two delayed values of torque ($m = 2$) and two delayed values of arm position ($n = 2$) as input to the network, and 10 neurons were used in the hidden layer (a 5–10–1 network).

The objective of the control system is to have the arm track the reference model

$$\frac{d^2 y_r}{dt^2} = -9y_r - 6\frac{dy_r}{dt} + 9r \tag{43}$$

where y_r is the output of the reference model, and r is the input reference signal. For the controller network, we used a 5–13–1 architecture. The inputs to the controller consisted of two delayed reference inputs, two delayed plant outputs, and one delayed controller output. The controller was trained using a BFGS quasi-Newton algorithm, with dynamic backpropagation used to calculate the gradients.

The left graph of Figure 28 shows the reference signal and the arm position using the trained model reference controller. The system is able to follow the reference, and the control actions (shown in the right graph) are smooth. At certain set points there is some steady-state error.

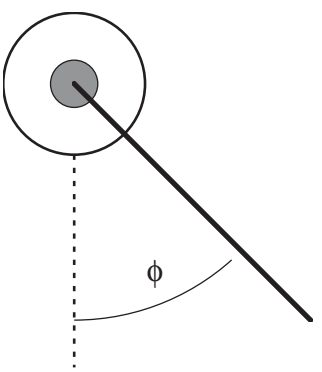

Figure 27. Single-link robot arm.

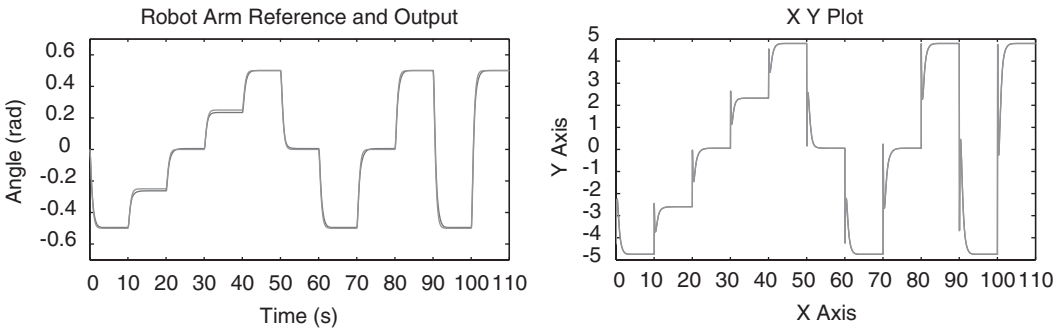

Figure 28. Robot arm response and control action for the model reference controller.

This error could be reduced by adding more training data at those steady-state conditions where the error is largest. The problem can occur in the plant model, or in the controller network.

5.4. Summary

We can summarize the characteristics of the three controllers as follows.

Model predictive control: Uses a neural network plant model to predict future plant behaviour. An optimization algorithm determines the control input that optimizes plant performance over a finite time horizon. The plant training requires only a batch algorithm for feedforward networks and is reasonably fast. The controller requires an online optimization algorithm, which requires more computation than the other two controllers.

NARMA-L2 control: This controller is a variation of the feedback linearization controller. An approximate companion form plant model is used. The next control input is computed to force the plant output to follow a reference signal. The neural network plant model is trained with static backpropagation. The controller is a rearrangement of the plant model, and requires minimal online computation.

Model reference control: A neural network plant model is first developed. The plant model is then used to train a neural network controller to force the plant output to follow the output of a reference model. This control architecture requires the use of dynamic backpropagation for training the controller. This generally takes more time than training static networks with the standard backpropagation algorithm. However, this approach applies to a more general class of plant than does the NARMA-L2 control architecture. The controller requires minimal online computation.

The simulations described in this section can be reproduced with files that can be downloaded from the following website: http://elec-engr.okstate.edu/mhagan. They are Simulink blocks that require MATLAB and the Neural Network Toolbox for MATLAB.

6. CONCLUSION

This paper has given a brief introduction to the use of neural networks in control systems. In the limited space it was not possible to discuss all possible ways in which neural networks have been applied to control system problems. We have selected one type of network, the multilayer

perceptron. We have demonstrated the capabilities of this network for function approximation, and have described how it can be trained to approximate specific functions. We then presented three different control architectures that use neural network function approximators as basic building blocks. The control architectures were demonstrated on three simple physical systems.

There have been many different approaches to using neural networks in control systems, and we have not attempted to provide a complete survey of all approaches in this paper. For those readers interested in finding out more about the application of neural networks to control problems, we recommend the survey papers [17,30–35]. There are also many books that describe the use of neural networks in control systems. None of these texts attempts to cover all neural controllers, but rather each text concentrates on a few selected types. Some neural control texts are References [36–46].

REFERENCES

1. Hagan MT, Demuth HB, Beale MH. *Neural Network Design*. PWS Publishing: Boston, 1996.
2. Bishop C. *Neural Networks for Pattern Recognition*. Oxford: New York, 1995.
3. Haykin S. *Neural Networks*: *A Comprehensive Foundation* (2nd edn.). Prentice-Hall: Englewood Cliffs, NJ, 1999.
4. Hornik KM, Stinchcombe M, White H. Multilayer feedforward networks are universal approximators. *Neural Networks* 1989; **2**(5):359–366.
5. Pinkus A. Approximation theory of the MLP model in neural networks. *Acta Numerica* 1999; 143–195.
6. Niyogi P, Girosi F. Generalization bounds for function approximation from scattered noisy data. *Advances in Computers and Mathematics* 1999; **10**:51–80.
7. Werbos PJ. Beyond regression: New tools for prediction and analysis in the behavioral sciences. *Ph.D. Thesis*, Harvard University, Cambridge, MA, 1974. Also published *as The Roots of Backpropagation*. John Wiley & Sons: New York, 1994.
8. Rumelhart DE, Hinton GE, Williams RJ. Learning representations by back-propagating errors. *Nature* 1986; **323**:533–536.
9. Shanno DF. Recent advances in numerical techniques for large-scale optimization. In *Neural Networks for Control*, Miller, Sutton and Werbos (eds). MIT Press: Cambridge, MA, 1990.
10. Scales LE. *Introduction to Non-Linear Optimization*. Springer-Verlag: New York, 1985.
11. Charalambous C. Conjugate gradient algorithm for efficient training of artificial neural networks. *IEEE Proceedings* 1992; **139**(3):301–310.
12. Hagan MT, Menhaj M. Training feedforward networks with the Marquardt algorithm. *IEEE Transactions on Neural Networks* 1994; **5**(6):989–993.
13. Sarle WS. Stopped training and other remedies for overfitting. *Proceedings of the 27th Symposium on the Interface*, 1995.
14. MacKay DJC. A practical framework for backpropagation networks. *Neural Computation* 1992; **4**:448–472.
15. Foresee FD, Hagan MT. Gauss–Newton approximation to Bayesian regularization. *Proceedings of the 1997 International Conference on Neural Networks*, Houston, TX, 1997.
16. Hagan MT, Demuth HB. Neural networks for control. *Proceedings of the 1999 American Control Conference*, San Diego, CA, 1999; 1642–1656.
17. Hunt KJ, Sbarbaro D, Zbikowski R, Gawthrop PJ. Neural networks for control system—a survey. *Automatica* 1992; **28**:1083–1112.
18. Narendra KS, Mukhopadhyay S. Adaptive control using neural networks and approximate models. *IEEE Transactions on Neural Networks* 1997; **8**:475–485.
19. Narendra KS, Parthasarathy K. Identification and control of dynamical systems using neural networks. *IEEE Transactions on Neural Networks* 1990; **1**:4–27.
20. Camacho EF, Bordons C. *Model Predictive Control*. Springer: London, 1998.
21. Soloway D, Haley PJ. Neural generalized predictive control. *Proceedings of the 1996 IEEE International Symposium on Intelligent Control*, 1996; 277–281.
22. Dennis JE, Schnabel RB. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall: Englewood Cliffs, NJ, 1983.
23. Ljung L. *System Identification*: *Theory for the User*. 2/e. Prentice-Hall: Englewood Cliffs, NJ, 1999.
24. Slotine J-JE, Li W. *Applied Nonlinear Control*. Prentice-Hall: Englewood Cliffs, NJ, 1991.
25. Brengel DD, Seider WD. Multistep nonlinear predictive controller. *Industrial Engineering and Chemical Research* 1998; **28**:1812–1822.

26. Hagan MT, De Jesus O, Schultz R. Training recurrent networks for filtering and control. *Recurrent Neural Networks*: *Design and Applications*, Medsker L, Jain LC (eds), Chapter 12, CRC Press: Boca Raton, FL, 1999; 311–340.
27. De Jesus O, Hagan MT. Backpropagation through time for a general class of recurrent network. *Proceedings of the International Joint Conference on Neural Networks*, vol. 4, 2001; 2638–2643.
28. De Jesus O, Hagan MT. Forward perturbation algorithm for a general class of recurrent network. *Proceedings of the International Joint Conference on Neural Networks*, vol. 4, 2001; 2626–2631.
29. De Jesus O, Horn JM, Hagan MT. Analysis of recurrent network training and suggestions for improvements. *Proceedings of the International Joint Conference on Neural Networks*, vol. 4, 2001; 2632–2637.
30. Widrow B, Rumelhart DE, Lehr MA. Neural networks: applications in industry, business and science. *Journal A* 1994; **35**(2):17–27.
31. Balakrishnan SN, Weil RD. Neurocontrol: a literature survey. *Mathematical Modeling and Computing* 1996; **23**: 101–117.
32. Agarwal M. A systematic classification of neural-network-based control. *IEEE Control Systems Magazine* 1997; **17**(2):75–93.
33. Suykens JAK, De Moor BLR, Vandewalle J. NLq theory: a neural control framework with global asymptotic stability criteria. *Neural Networks* 1997: **10**:615–637.
34. Kerr TH. Critique of some neural network architectures and claims for control and estimation. *IEEE Transactions on Aerospace and Electronic Systems* 1998; **34**(2):406–419.
35. Chowdhury FN, Wahi P, Raina R, Kaminedi S. A survey of neural networks applications in automatic control. *Proceedings of the 33rd Southeastern Symposium on System Theory* 2001; 349–353.
36. Miller WT, Sutton RS, Werbos PJ (eds). *Neural Networks for Control*. MIT Press: Cambridge, MA, 1990.
37. White DA, Sofge DA (eds). *The Handbook of Intelligent Control*. Van Nostrand Reinhold: New York, 1992.
38. Brown M, Harris C. *Neurofuzzy Adaptive Modeling and Control*. Prentice-Hall: Englewood Cliffs, NJ, 1994.
39. Pham DT, Liu X. *Neural Networks for Identification*, *Prediction*, *and Control*. Springer-Verlag: New York, 1995.
40. Widrow B, Walach E. *Adaptive Inverse Control*. Prentice-Hall: Englewood Cliffs, NJ, 1996.
41. Omatu S, Khalid MB, Yusof R. *Neuro-Control and its Applications*. Springer-Verlag: London, 1996.
42. Omidvar O, Elliott D. *Neural Systems for Control*. Academic Press: New York, 1997.
43. Hrycej T. *Neurocontrol*: *Towards an Industrial Control Methodology*. John Wiley & Sons: New York, 1997.
44. Rovithakis GA, Christodoulou MA. *Adaptive Control with Recurrent High-order Neural Networks*. Springer-Verlag: London, 2000.
45. Norgard M, Ravn O, Poulsen NK, Hansen LK. *Neural Networks for Modelling and Control of Dynamic Systems*. Springer-Verlag: London, 2000.
46. Liu GP. *Nonlinear Identification and Control*. Springer-Verlag: London, 2001.

