Proceedings of the American Control Conference
San Diego, California • June 1999

Neural Networks for Control

Martin T. Hagan
School of Electrical & Computer Engineering
Oklahoma State University
mhagan@ieee.org

Howard B. Demuth
Electrical Engineering Department
University of Idaho
hdemuth@uidaho.edu

Abstract

The purpose of this tutorial 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 discused. The tutorial also presents several control architectures, such as model reference adaptive control, model predictive control, and internal model control, in which multilayer perceptron neural networks can be used as basic building blocks.

1. Introduction

In this tutorial we want to give a brief introduction to neural networks and their application in control systems. The field of neural networks covers a very broad area. It would be impossible in a short time 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 [HBD96], [Bish95] and [Hayk99].)

For the purposes of this tutorial 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 tutorial we will present several control architectures demonstrating a variety of uses for function approximator neural networks.

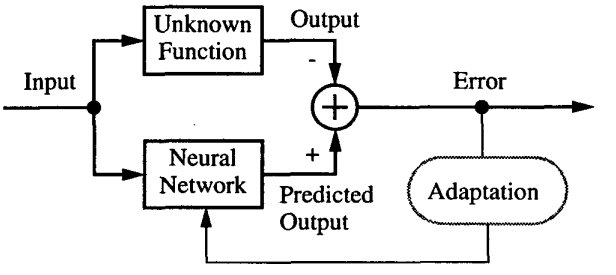


Figure 1 Neural Network as Function Approximator

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.

2.1.1 Single-Input Neuron

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. (Some authors use the term "activation function"

rather than *transfer function* and "offset" rather than *bias*.)

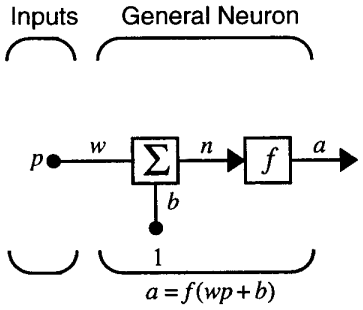


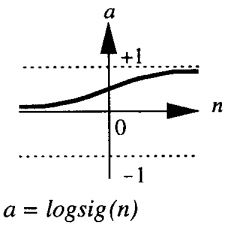
Figure 2 Single-Input Neuron

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 will be 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. A particular transfer function is chosen to satisfy some specification of the problem that the neuron is attempting to solve. One of the most commonly used functions is the *log-sigmoid transfer function*, which is shown in Figure 3.



Log-Sigmoid Transfer Function

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 to 1, according to the expression:

$$a = \frac{1}{1 + e^{-n}}. \qquad (1)$$

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.

2.1.2 Multiple-Input Neuron

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* $\mathbf{W}$.

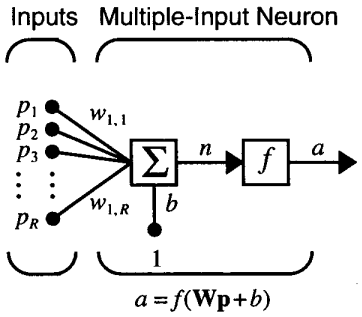


Figure 4 Multiple-Input Neuron

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

$$n = w_{1,1}p_1 + w_{1,2}p_2 + \cdots + w_{1,R}p_R + b. \qquad (2)$$

This expression can be written in matrix form:

$$n = \mathbf{W}\mathbf{p} + b, \qquad (3)$$

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

Now the neuron output can be written as

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

We have adopted a particular convention in assigning the indices of the elements of the weight matrix. The first index indicates the particular neuron destination for that weight. The second index indicates the source of the signal fed to the neuron. Thus, the indices in $w_{1,2}$ say that this weight represents the connection *to* the first (and only) neuron *from* the second source.

We would like to draw networks with several neurons, each having several inputs. Further, we would like to have more than one layer of neurons. You can imagine how complex such a network might appear if all the lines were drawn. It would take a lot of ink, could hardly be read, and the mass of detail might obscure the main features. Thus, we will use an *abbreviated notation*. A multiple-input neuron using this notation is shown in Figure 5.

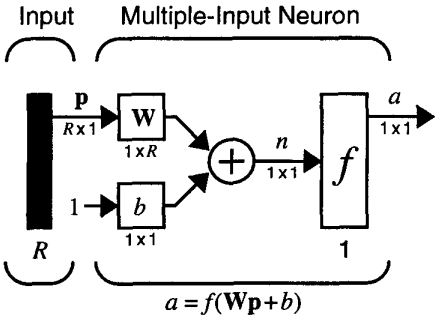


Figure 5 Neuron with R Inputs, Abbreviated Notation

As shown in Figure 5, the input vector **p** is represented by the solid vertical bar at the left. The dimensions of **p** are displayed below the variable as $R \times 1$, indicating that the input is a single vector of R elements. These inputs go to the weight matrix **W**, which has R columns but only one row in this single neuron case. A constant 1 enters the neuron as an input and is multiplied by a scalar bias b. The net input to the transfer function f is n, which is the sum of the bias b and the product **Wp**. The neuron's output a is a scalar in this case. If we had more than one neuron, the network output would be a vector.

Note that the number of inputs to a network is set by the external specifications of the problem. If, for instance, you want to design a neural network that is to predict kite-flying conditions and the inputs are air temperature, wind velocity and humidity, then there would be three inputs to the network.

2.2. Network Architectures

Commonly one neuron, even with many inputs, may not be sufficient. We might need five or ten, operating in parallel, in what we will call a "layer." This concept of a layer is discussed below.

2.2.1 A Layer of Neurons

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, 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.

Each element of the input vector **p** is connected to each neuron through the weight matrix **W**. Each neuron has a bias b_i, a summer, a transfer function f and an output a_i. Taken together, the outputs form the output vector **a**.

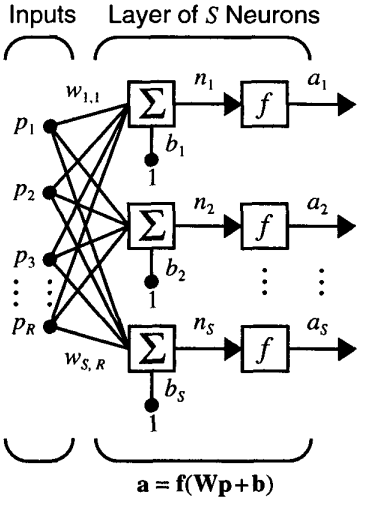


Figure 6 Layer of S Neurons

It is common for the number of inputs to a layer to be different from the number of neurons (i.e., $R \neq S$).

You might ask if all the neurons in a layer must have the same transfer function. The answer is no; you can define a single (composite) layer of neurons having different transfer functions by combining two of the networks shown above in parallel. Both networks would have the same inputs, and each network would create some of the outputs.

The input vector elements enter the network through the weight matrix **W**:

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ \vdots & \vdots & & \vdots \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}. \quad (5)$$

As noted previously, the row indices of the elements of matrix **W** indicate the destination neuron associated with that weight, while the column indices indicate the source of the input for that weight. Thus, the indices in $w_{3,2}$ say that this weight represents the connection *to* the third neuron *from* the second source.

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

Here again, the symbols below the variables tell you that for this layer, **p** is a vector of length R, **W** is an $S \times R$ matrix, and **a** and **b** are vectors of length S. As defined previously, the layer includes the weight matrix, the summation and multiplication operations, the bias vector **b**, the transfer function boxes and the output vector.

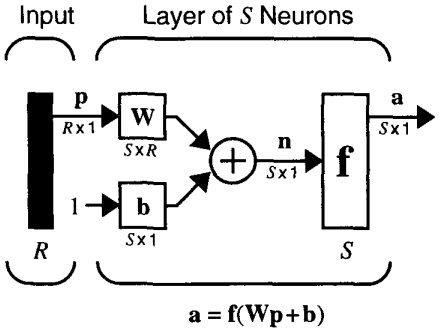


Figure 7 Layer of S Neurons, Abbreviated Notation

2.2.2 Multiple Layers of Neurons

Now consider a network with several layers. Each layer has its own weight matrix $\mathbf{W}$, its own bias vector $\mathbf{b}$, a net input vector $\mathbf{n}$ and an output vector $\mathbf{a}$. We need to introduce some additional notation to distinguish between these layers. We will use superscripts to identify the layers. Specifically, we append the number of the layer as a *superscript* to the names for each of these variables. 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 $S^1 \times S^2$ 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).

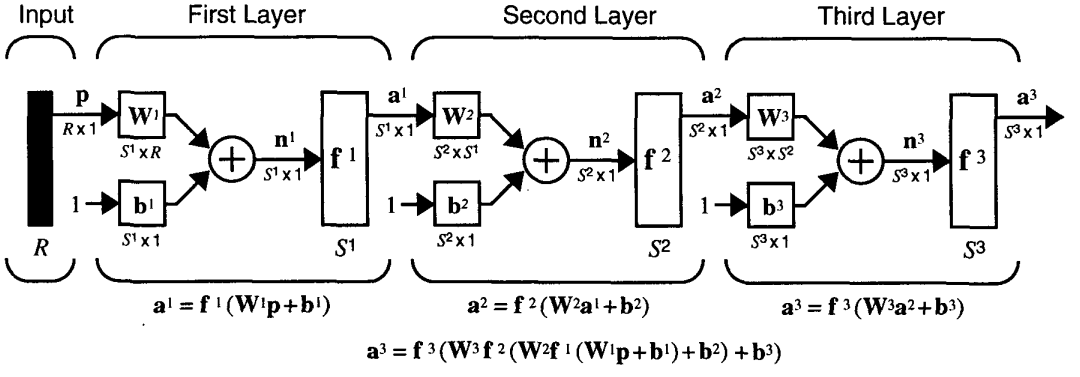


Figure 8 Three-Layer Network

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. 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,

$$f^1(n) = \frac{1}{1+e^{-n}} \quad \text{and} \quad f^2(n) = n . \tag{6}$$

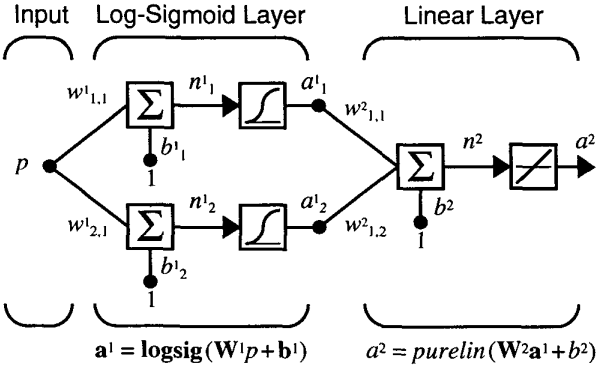


Figure 9 Example Function Approximation Network

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

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

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

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 centers of the steps occur where the net input to a neuron in the first layer is zero:

$$n^1_1 = w^1_{1,1}p + b^1_1 = 0 \quad \Rightarrow \quad p = -\frac{b^1_1}{w^1_{1,1}} = -\frac{-10}{10} = 1, \quad (7)$$

$$n^1_2 = w^1_{2,1}p + b^1_2 = 0 \quad \Rightarrow \quad p = -\frac{b^1_2}{w^1_{2,1}} = -\frac{10}{10} = -1. \quad (8)$$

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

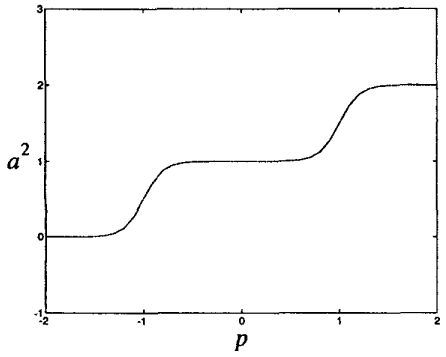


Figure 10 Nominal Response of Network of Figure 9

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:

$$-1 \le w^2_{1,1} \le 1,\ -1 \le w^2_{1,2} \le 1,\ 0 \le b^1_2 \le 20,\ -1 \le b^2 \le 1. \quad (9)$$

Figure 11 (a) shows how the network biases in the first (hidden) layer can be used to locate the position of the steps. Figure 11 (b) illustrates 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).

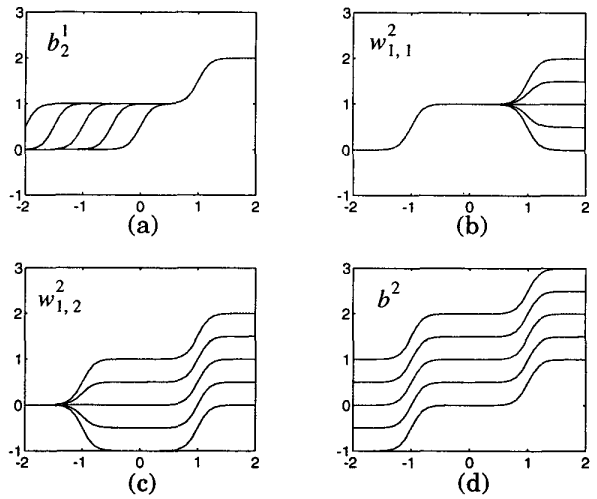


Figure 11 Effect of Parameter Changes on Network Response

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 (see [HoSt89]).

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) which 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*, which is based on gradient descent. (More efficient algorithms than gradient descent are often used in neural network training. The reader is referred to [HBD96] for discussions of these other algorithms.)

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

$$\mathbf{a}^{m+1} = \mathbf{f}^{m+1}(\mathbf{W}^{m+1}\mathbf{a}^m + \mathbf{b}^{m+1}) \text{ for } m = 0, 1, \ldots, M-1, \quad (10)$$

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

$$\mathbf{a}^0 = \mathbf{p}, \quad (11)$$

which provides the starting point for Eq. (10). The outputs of the neurons in the last layer are considered the network outputs:

$$\mathbf{a} = \mathbf{a}^M . \tag{12}$$

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 behavior:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}, \tag{13}$$

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:

$$F(\mathbf{x}) = \sum_{q=1}^{Q} e_q^2 = \sum_{q=1}^{Q} (t_q - a_q)^2 . \tag{14}$$

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

$$F(\mathbf{x}) = \sum_{q=1}^{Q} \mathbf{e}_q^T \mathbf{e}_q = \sum_{q=1}^{Q} (\mathbf{t}_q - \mathbf{a}_q)^T (\mathbf{t}_q - \mathbf{a}_q) . \tag{15}$$

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

$$\hat{F}(\mathbf{x}) = (\mathbf{t}(k) - \mathbf{a}(k))^T (\mathbf{t}(k) - \mathbf{a}(k)) = \mathbf{e}^T(k)\mathbf{e}(k), \tag{16}$$

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

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

$$b_i^m(k+1) = b_i^m(k) - \alpha \frac{\partial \hat{F}}{\partial b_i^m} , \tag{18}$$

where α is the learning rate.

4.2. Chain Rule

For a single-layer linear network these partial derivatives in Eq. (17) and Eq. (18) 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 Eq. (17) and Eq. (18):

$$\frac{\partial \hat{F}}{\partial w_{i,j}^m} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial w_{i,j}^m} , \tag{19}$$

$$\frac{\partial \hat{F}}{\partial b_i^m} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial b_i^m} . \tag{20}$$

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:

$$n_i^m = \sum_{j=1}^{S^{m-1}} w_{i,j}^m a_j^{m-1} + b_i^m . \tag{21}$$

Therefore

$$\frac{\partial n_i^m}{\partial w_{i,j}^m} = a_j^{m-1} , \frac{\partial n_i^m}{\partial b_i^m} = 1 . \tag{22}$$

If we now define

$$s_i^m \equiv \frac{\partial \hat{F}}{\partial n_i^m} , \tag{23}$$

(the *sensitivity* of $\hat{F}$ to changes in the ith element of the net input at layer m), then Eq. (19) and Eq. (20) can be simplified to

$$\frac{\partial \hat{F}}{\partial w_{i,j}^m} = s_i^m a_j^{m-1} , \tag{24}$$

$$\frac{\partial \hat{F}}{\partial b_i^m} = s_i^m . \tag{25}$$

We can now express the approximate steepest descent algorithm as

$$w_{i,j}^m(k+1) = w_{i,j}^m(k) - \alpha s_i^m a_j^{m-1} , \tag{26}$$

$$b_i^m(k+1) = b_i^m(k) - \alpha s_i^m . \tag{27}$$

In matrix form this becomes:

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha \mathbf{s}^m (\mathbf{a}^{m-1})^T , \tag{28}$$

$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \alpha \mathbf{s}^m \,, \tag{29}$$

where the individual elements of $\mathbf{s}^m$ are given by Eq. (23).

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 $m+1$:

$$\mathbf{s}^M = -2\dot{\mathbf{F}}^M(\mathbf{n}^M)(\mathbf{t}-\mathbf{a}) \,, \tag{30}$$

$$\mathbf{s}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T \mathbf{s}^{m+1} \,, \; m = M-1, \ldots, 2, 1 \tag{31}$$

where

$$\dot{\mathbf{F}}^m(\mathbf{n}^m) = \begin{bmatrix} \dot{f}^m(n_1^m) & 0 & \cdots & 0 \\ 0 & \dot{f}^m(n_2^m) & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & \dot{f}^m(n_{S^m}^m) \end{bmatrix} . \tag{32}$$

(See [HDB96], 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 Eq. (28) and Eq. (29), 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 Eq. (31)). For example, conjugate gradient and quasi-Newton algorithms ([Shan90], [Scal85], [Char92]) 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 Eq. (31) (see [HaMe94]).

We should emphasize that all of the algorithms that we will describe in this chapter use the backpropagation procedure, in which derivatives are processed from the last layer of the network to the first. For this reason they could all be called "backpropagation" algorithms. The differences between the algorithms occur in the way in which the resulting derivatives are used to update the weights.

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:

$$\mathbf{t}_q = \mathbf{g}(\mathbf{p}_q) + \mathbf{e}_q \,, \tag{33}$$

where $\mathbf{p}_q$ is the system input, $\mathbf{g}(\;)$ is the underlying function we wish to approximate, $\mathbf{e}_q$ is measurement noise, and $\mathbf{t}_q$ is the system output (network target).

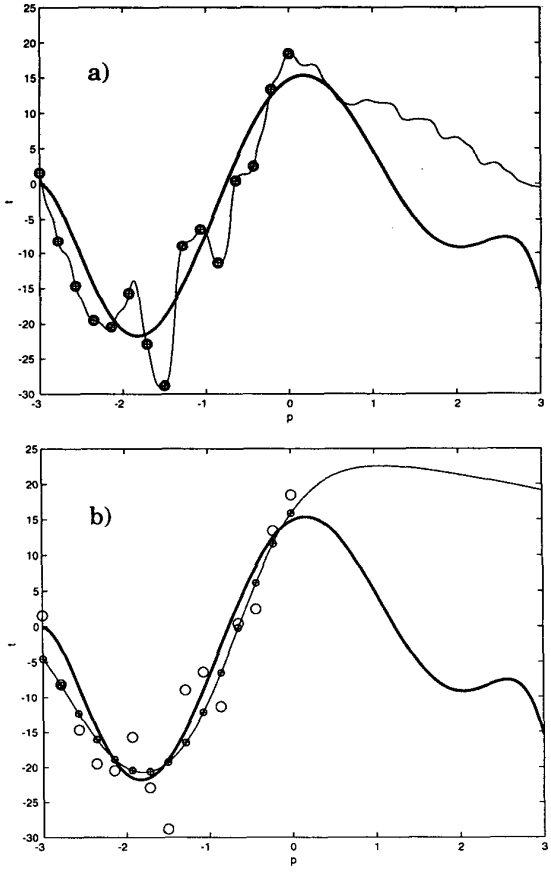


Figure 12 Example of Overfitting a) and Good Fit b)

Figure 12 shows an example of the underlying function $\mathbf{g}(\ )$ (thick line), training data target values $\mathbf{t}_q$ (large circles), network responses for the training inputs $\mathbf{a}_q$ (small circles with imbedded crosses), and total trained network response (thin line).

In the example shown in Figure 12 a), a large network was trained to minimize squared error (Eq. (14)) 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 too many 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 p greater than 0. Neural networks (and all other data-based approximation techniques) cannot be expected to *extrapolate* accurately. If the network receives an input which 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, 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:

$$F(\mathbf{x}) = \sum_{q=1}^{Q} \mathbf{e}_q^T \mathbf{e}_q + \rho \sum (w_{i,j}^k)^2 \qquad (34)$$

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 ([MacK92] and [FoHa97]). 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.

Even with Bayesian regularization, the network response is not accurate outside the range of the training data. As we mentioned earlier, we cannot expect the network to extrapolate accurately. If we want the network to respond accurately throughout the range [-3, 3], then we need to provide training data throughout this range. This can be more problematic in multi-input cases, as shown in Figure 13. On the top graph we have the underlying function. On the bottom graph we have the neural network approximation. The training inputs were provided over the entire range of each input, but only for cases where the first input was greater than the second input. We can see that the network approximation is good for cases within the training set, but is poor for all cases where the second input is larger than the first input.

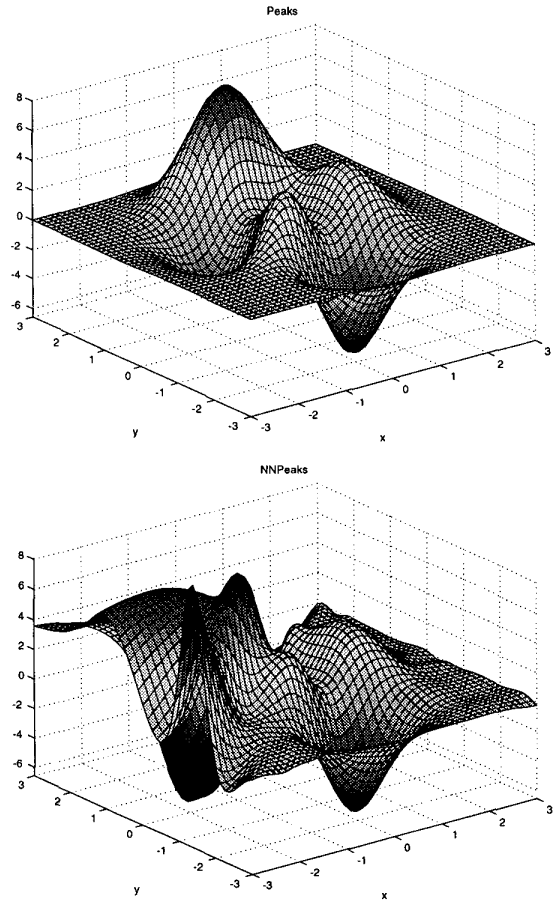


Figure 13 Two-Input Example of Poor Network Extrapolation

A complete discussion of generalization and overfitting is beyond the scope of this tutorial. The interest-

ed reader is referred to [HDB96], [Hayk99], [MacK92] or [FoHa97].

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

5. Control System Applications

Neural networks have been applied very successfully in the identification and control of dynamic systems. The universal approximation capabilities of the multilayer perceptron have made it a popular choice for modeling nonlinear systems and for implementing general-purpose nonlinear controllers. In the remainder of this tutorial we will introduce some of the more popular neural network architectures for system identification and control.

5.1. Fixed Stabilizing Controllers

Fixed stabilizing controllers (see Figure 14) have been proposed in [Kawa90], [KrCa90], and [Mill87]. This scheme has been applied to the control of robot arm trajectory, where a proportional controller with gain was used as the stabilizing feedback controller. From Figure 14 we can see that the total input that enters the plant is the sum of the feedback control signal and the feedforward control signal, which is calculated from the inverse dynamics model (neural network). That model uses the desired trajectory as the input and the feedback control as an error signal. As the NN training advances, that input will converge to zero. The neural network controller will learn to take over from the feedback controller.

The advantage of this architecture is that we can start with a stable system, even though the neural network has not been adequately trained. A similar (although more complex) control architecture, in which stabilizing controllers are used in parallel with neural network controllers, is described in [SaSl92].

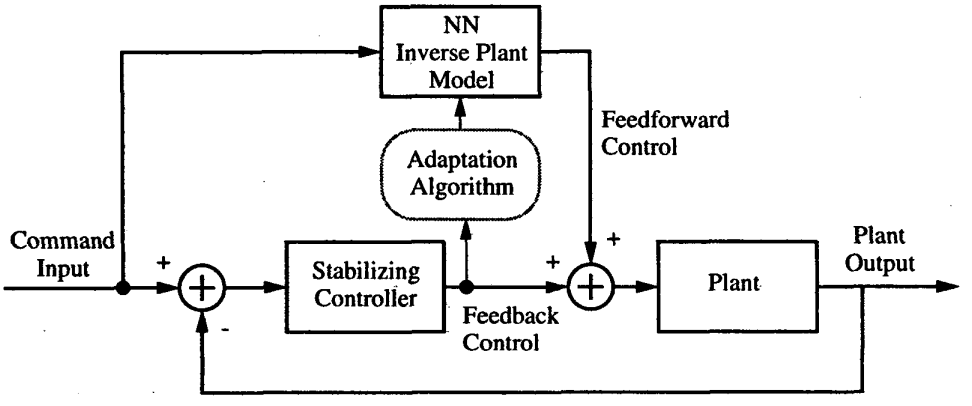


Figure 14 Stabilizing Controller

5.2. Adaptive Inverse Control

Figure 15 shows a structure for the Model Reference Adaptive Inverse Control proposed in [WiWa96]. The adaptive algorithm receives the error between the plant output and the reference model output. The controller parameters are updated to minimize that tracking error. The basic model reference adaptive control approach can be affected by sensor noise and plant disturbances. An alternative which allows cancellation of the noise and disturbances includes a neural network plant model in parallel with the plant. That model will be trained to receive the same inputs as the plant and to produce the same output. The difference between the outputs will be interpreted as the effect of the noise and disturbances at the plant output. That signal will enter an inverse plant model to generate a filtered noise and disturbance signal that is subtracted from the plant input. The idea is to cancel the disturbance and the noise present in the plant.

5.3. Nonlinear Internal Model Control

Nonlinear Internal Model Control (NIMC), shown in Figure 16, consists of a neural network controller, a neural network plant model, and a robustness filter with a single tuning parameter [NaHe92]. The neural network controller is generally trained to represent the inverse of the plant, if the inverse exists. The error between the output of the neural network plant model and the measurement of plant output is used as the feedback input to the robustness filter, which then feeds into the neural network controller.

The NN plant model and the NN controller (if it is an inverse plant model) can be trained off-line, using data collected from plant operations. The robustness filter is a first order filter whose time constant is selected to ensure closed loop stability.

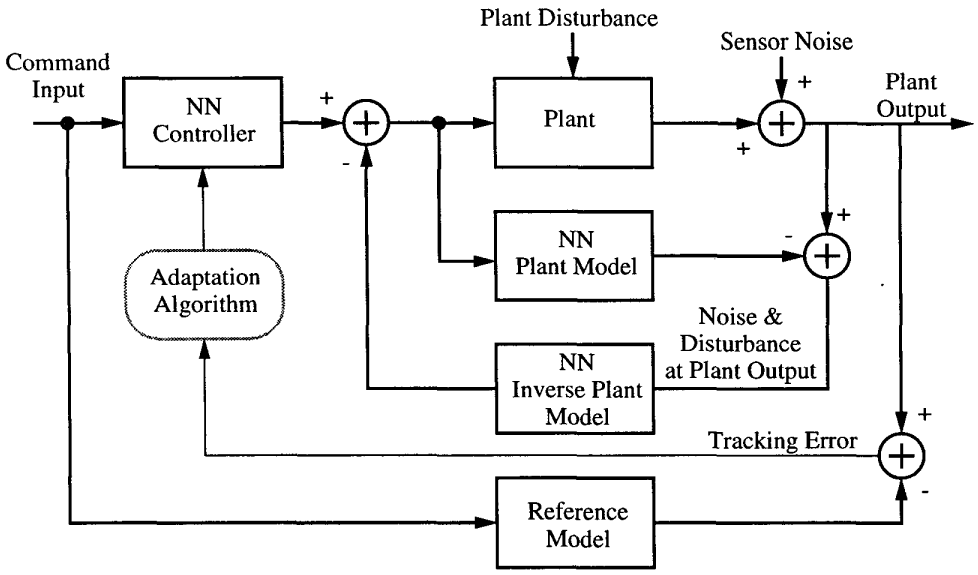


Figure 15 Adaptive Inverse Control System

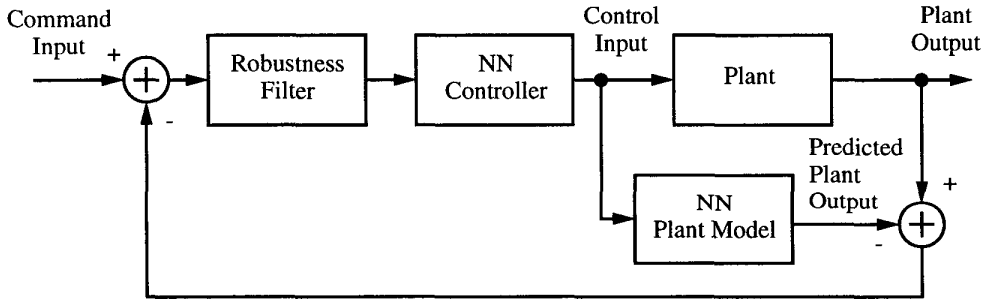


Figure 16 Nonlinear Internal Model Control

5.4. Model Predictive Control

Model Predictive Control (MPC), shown in Figure 18, optimizes the plant response over a specified time horizon [HuSb92]. This architecture requires a neural network plant model, a neural network controller, a performance function to evaluate system responses, and an optimization procedure to select the best control input.

The optimization procedure can be computationally expensive. It requires a multi-step ahead calculation, in which the neural network model is used to predict the plant response. The neural network controller learns to produce the input selected by the optimization process. When training is complete, the optimization step can be completely replaced by the neural network controller.

5.5. Model Reference Control or Neural Adaptive Control

As with other techniques, the Model Reference Adaptive Control (MRAC) configuration [NaPa90] uses two neural networks: a controller network and a model network. (See Figure 17.) The model network can be trained off-line using historical plant measurements. The controller is adaptively trained to force the plant output to track a reference model output. The model network is used to predict the effect of controller changes on plant output, which allows the updating of controller parameters.

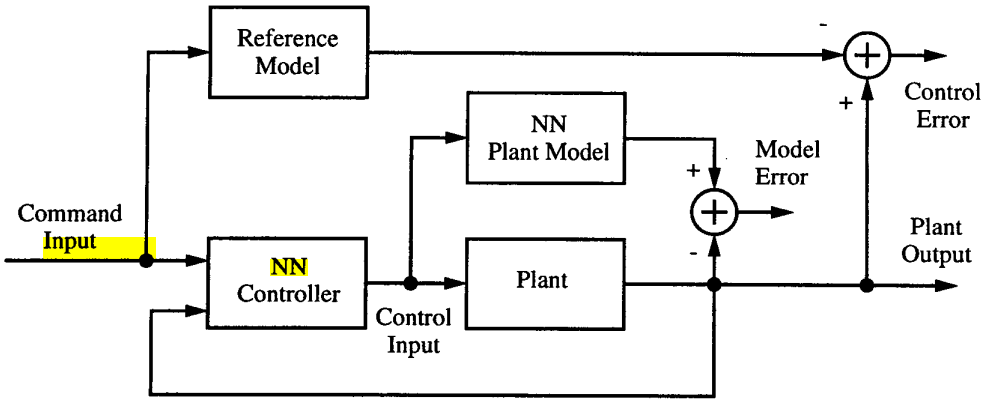


Figure 17 Model Reference Adaptive Control

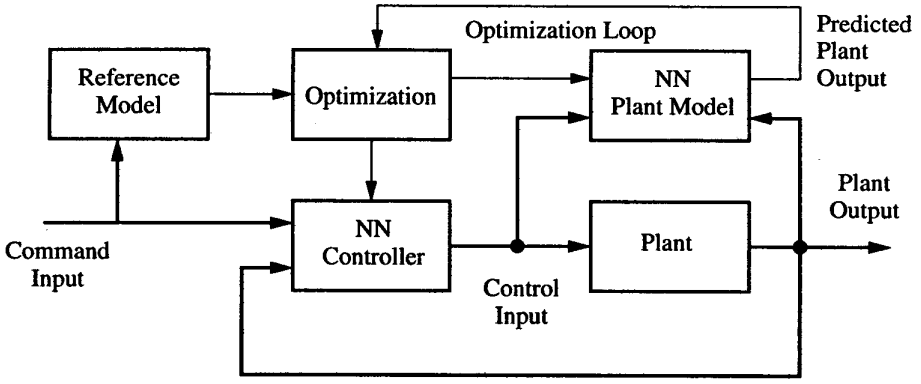


Figure 18 Model Predictive Control

5.6. Adaptive Critic

As shown in Figure 19, the Adaptive Critic controller consists of two neural networks [SuBa98]. The first network operates as an inverse controller and is called the Action or Actor network. The second network, called the Critic Network, predicts the future performance of the system. The Critic network is trained to optimize future performance. The training is performed using reinforcement learning, which is an approximation to dynamic programming. There have been many variations of the adaptive critic controller proposed in the last few years.

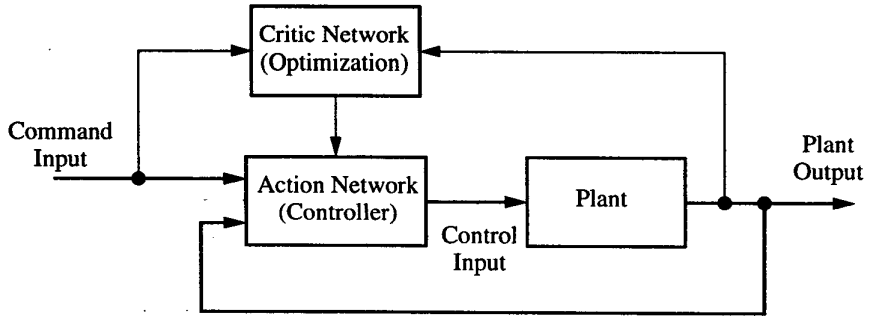


Figure 19 Adaptive Critic

5.7. Neural Adaptive Feedback Linearization

The neural adaptive feedback linearization technique is based on the standard feedback linearization controller [SlLi91]. An implementation is shown in Figure 20. The feedback linearization technique produces a control signal with two components. The first component cancels out the nonlinearities in the plant, and the second part is a linear state feedback controller. The class of nonlinear systems to which this technique can be applied is described by the relation [VaVe96]:

$$x_p^{(n)} = f(\mathbf{x}_p) + g(\mathbf{x}_p)u, \tag{35}$$

where

$$\mathbf{x}_p = \left[x_p \ \dot{x}_p \ \cdots \ x_p^{(n-1)}\right]^T \tag{36}$$

contains the system state variables and u is the control input. To obtain a linear system from the nonlinear system described by Eq. (35), we can use the input

$$u = \frac{1}{g(\mathbf{x}_p)}[-f(\mathbf{x}_p) - \mathbf{k}^T\mathbf{x}_p + r], \tag{37}$$

where $\mathbf{k}$ contains the feedback gains and r is the reference input.

Substitution of Eq. (37) into Eq. (35) results in the linear system

$$x_p^{(n)} = -\mathbf{k}^T\mathbf{x}_p + r, \tag{38}$$

whose behavior is completely controlled by the linear feedback gains.

We can use neural networks to implement the feedback linearization strategy. If we approximate the functions f and g using the neural networks NN_f and NN_g, we can rewrite the control signal as

$$u = \frac{1}{NN_g(\mathbf{x}_p)}[-NN_f(\mathbf{x}_p) - \mathbf{k}^T\mathbf{x}_p + r]. \tag{39}$$

We wish the system to follow the reference model given by

$$x_m^{(n)} = -\mathbf{k}^T\mathbf{x}_m + r. \tag{40}$$

By substituting Eq. (39) into Eq. (35) we obtain

$$x_p^{(n)} = f(\mathbf{x}_p) + \frac{g(\mathbf{x}_p)}{NN_g(\mathbf{x}_p)}[-NN_f(\mathbf{x}_p) - \mathbf{k}^T\mathbf{x}_p + r]. \tag{41}$$

The controller error is defined as

$$\mathbf{e} = \mathbf{x}_p - \mathbf{x}_m, \tag{42}$$

and the error differential equation is

$$e^{(n)} = -\mathbf{k}^T\mathbf{e} + \{f(\mathbf{x}_p) - NN_f(\mathbf{x}_p)\} + \{g(\mathbf{x}_p) - NN_g(\mathbf{x}_p)\}u \tag{43}$$

With an appropriate training algorithm, the error differential equation will be stable. The error will converge to zero "if structural error terms are sufficiently small." [VaVe96]

There are several variations on the neural adaptive feedback linearization controller, including the approximate models (in particular Model VI) of Narendra [NaBa94].

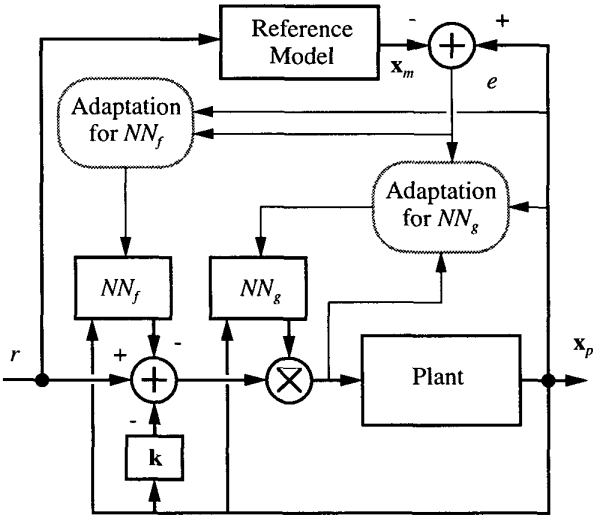


Figure 20 Neural Adaptive Feedback Linearization

5.8. Stable Direct Adaptive Control

There have been several recent direct adaptive control techniques which have been designed to guarantee overall system stability ([SaSl92], [Poly96], [SpCr98]). The method of [SaSl92] uses Lyapunov stability theory in the design of the network learning rule, rather than a gradient descent algorithm like backpropagation. The controller (see Figure 22) consists of three parts: linear feedback, a nonlinear sliding mode controller and an adaptive neural network controller. The total control signal is computed as follows:

$$u(t) = u_{pd}(t) + (1 - m(t))u_{ad}(t) + m(t)u_{sl}(t), \tag{44}$$

where $u_{pd}(t)$ is the linear feedback control, $u_{sl}(t)$ is the sliding mode control and $u_{ad}(t)$ is the adaptive neural control. The function $m(t)$ allows a smooth transition between the sliding and adaptive controllers, based on the location of the system state:

$$\begin{cases} m(t) = 0 & x(t) \in A_d \\ 0 < m(t) < 1 & otherwise \\ m(t) = 1 & x(t) \in A_c \end{cases} \quad (45)$$

where the regions might be defined as in Figure 21.

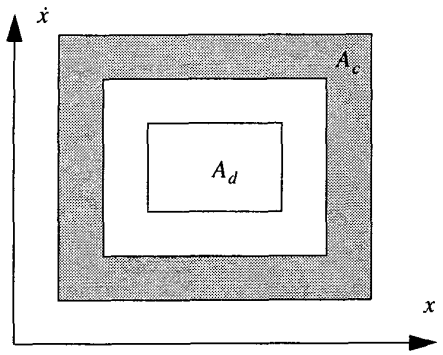


Figure 21 Controller Regions

The sliding mode controller is used to keep the system state in a region where the neural network can be accurately trained to achieve optimal control. The sliding mode controller is turned on (and the neural controller is turned off) whenever the system drifts outside this region. The combination of controllers produces a stable system which adapts to optimize performance.

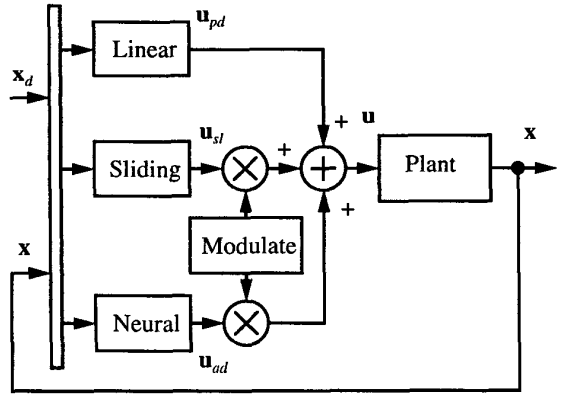


Figure 22 Stable Direct Adaptive Control

It should be noted that this neural controller uses the radial basis neural network. The radial basis output is a linear function of the network weights, which allows faster training and simpler analysis than is possible with multilayer networks. It has the disadvantage that it may require many neurons if the number of network inputs is large. It also requires that the centers and spread of the basis functions be selected before training.

5.9. Limitations and Cautions

Each of the neurocontrol architectures we have discussed has its own advantages and disadvantages. For example, the feedback linearization technique can only be applied to systems described by Eq. (35). The stable direct adaptive control technique requires that the unknown nonlinearities appear in the same equation as the control input in a state-space representation. The model reference adaptive control technique has no guarantee of stability. The adaptive inverse control technique requires the existence of a stable plant inverse.

Generally speaking, those techniques which guarantee stability apply to a restricted class of systems. As the field of neurocontrol continues to progress, stable neurocontrol methods will be developed for wider classes of systems.

One of the key practical problems for many of the neurocontrol systems is the generalization issue that we discussed earlier - the ability of a network to perform well in new situations. For example, the model predictive control architecture requires that a neural network model of the plant be identified. This plant model is a mapping from previous plant inputs and outputs to future plant outputs. In order to accurately model the plant, the network needs to be trained with data which covers the entire range of possible network inputs. It may be difficult to obtain this data, since we don't have direct control over previous plant outputs. We can sometimes have independent control over the plant inputs, but only indirect control over the plant outputs (which then become inputs to the network). For high-order systems it may be difficult to obtain data in which the plant response covers all usable portions of the state space. In these situations it will be important for the network to be able to detect situations in which the inputs fall outside the regions where the network received training data.

6. Conclusions

This tutorial has given a brief introduction to the use of neural networks in control systems. In the limited space it is 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 several different control architectures which use neural network function approximators as basic building blocks.

For those readers interested in finding out more about the application of neural networks to control problems, we recommend the following references: [BaWe96], [HuSb92], [BrHa94], [MiSu90],

[WhSo92], [SuDe97], [VaVe96], [WiWa96], [Agar97], [WiRu94], [Kerr98].

7. References

[Agar97] M. Agarwal, "A systematic classification of neural-network-based control," *IEEE Control Systems Magazine*, vol. 17, no. 2, pp. 75-93, 1997.

[BaWe96] S.N. Balakrishnan and R.D. Weil, "Neurocontrol: A Literature Survey," *Mathematical Modeling and Computing*, vol. 23, pp. 101-117, 1996.

[Bish95] C. Bishop, *Neural Networks for Pattern Recognition*, New York: Oxford, 1995.

[BrHa94] M. Brown and C. Harris, *Neurofuzzy Adaptive Modeling and Control*, New Jersey: Prentice-Hall, 1994.

[Char92] C. Charalambous, "Conjugate gradient algorithm for efficient training of artificial neural networks," *IEEE Proceedings*, vol. 139, no. 3, pp. 301–310, 1992.

[ChWe94] Q. Chen and W.A. Weigand, "Dynamic Optimization of Nonlinear Processes by Combining Neural Net Model with UDMC," *AIChE Journal*, vol. 40, pp. 1488-1497, 1994.

[FoHa97] F. D. Foresee and M. T. Hagan, "Gauss-Newton approximation to Bayesian regularization," *Proceedings of the 1997 International Conference on Neural Networks*, Houston, Texas, 1997.

[HBD96] M. Hagan, H. Demuth, and M. Beale, *Neural Network Design*, Boston: PWS, 1996.

[HaMe94] M. T. Hagan and M. Menhaj, "Training feedforward networks with the Marquardt algorithm," *IEEE Transactions on Neural Networks*, vol. 5, no. 6, pp. 989–993, 1994.

[Hayk99] S. Haykin, *Neural Networks: A Comprehensive Foundation, 2nd Ed.*, New Jersey: Prentice-Hall, 1999.

[HoSt89] K. M. Hornik, M. Stinchcombe and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989.

[HuSb92] K.J. Hunt, D. Sbarbaro, R. Zbikowski and P.J. Gawthrop, "Neural Networks for Control System - A Survey," *Automatica*, vol. 28, pp. 1083-1112, 1992.

[Kawa90] M. Kawato, "Computational Schemes and Neural Network Models for Formation and Control of Multijoint Arm Trajectory," *Neural Networks for Control*, W.T. Miller, R.S. Sutton, and P.J. Werbos, Eds., Boston: MIT Press, pp. 197-228, 1990.

[Kerr98] T.H. Kerr, "Critique of some neural network architectures and claims for control and estimation," IEEE Transactions on Aerospace and Electronic Systems, vol. 34, no. 2, pp. 406-419, 1998.

[KrCa90] L.G. Kraft and D.P. Campagna, "A Comparison between CMAC Neural Network Control and Two Traditional Control Systems," *IEEE Control Systems Magazine*, vol. 10, no. 2, pp. 36-43, 1990.

[MacK92] D. J. C. MacKay, "A Practical Framework for Backpropagation Networks," *Neural Computation*, vol. 4, pp. 448-472, 1992.

[Mill87] W.T. Miller, "Sensor-Based Control of Robotic Manipulators Using a General Learning Algorithm," *IEEE Journal of Robotics and Automation*, vol. 3, no. 2, pp. 157-165, 1987.

[MiSu90] W.T. Miller, R.S. Sutton, and P.J. Werbos, Eds., *Neural Networks for Control*, Cambridge, MA: MIT Press, 1990.

[MuNe92] R. Murray, D. Neumerkel and D. Sbarbaro, "Neural Networks for Modeling and Control of a Non-linear Dynamic System," *Proceedings of the 1992 IEEE International Symposium on Intelligent Control*, pp. 404-409, 1992.

[NaHe92] E.P. Nahas, M.A. Henso and D.E. Seborg, "Nonlinear Internal Model Control Strategy for Neural Models," *Computers and Chemical Engineering*, vol. 16, pp. 1039-1057, 1992.

[NaBa94] K.S. Narendra and B. Balakrishnan, "Improving Transient Response of Adaptive Control Systems Using Multiple Models and Switching," *IEEE Transactions on Automatic Control*, vol. 39, no. 9, pp. 1861-1866, 1994.

[NaPa90] K.S. Narendra, and K. Parthasarathy, "Identification and Control of Dynamical Systems Using Neural Networks," *IEEE Transactions on Neural Networks*, vol. 1, pp. 4-27, 1990.

[Poly96] M.M. Polycarpou, "Stable adaptive neural control scheme for nonlinear control," *IEEE Transactions on Automatic Control*, vol. 41, no. 3, pp. 447-451, 1996.

[RiBr93] M. Riedmiller and H. Braun, "A direct adaptive method for faster backpropagation learning: The RPROP algorithm," *Proceedings of the IEEE International Conference on Neural Networks*, San Francisco: IEEE, 1993.

[SaSl92] R.M. Sanner and J.J.E. Slotine, "Gaussian Networks for Direct Adaptive Control," *IEEE Transactions on Neural Networks*, vol. 3, pp. 837-863, 1992.

[Scal85] L. E. Scales, *Introduction to Non-Linear Optimization*, New York: Springer-Verlag, 1985.

[Shan90] D. F. Shanno, "Recent advances in numerical techniques for large-scale optimization," in *Neural Networks for Control*, Miller, Sutton and Werbos, eds., Cambridge, MA: MIT Press, 1990.

[SlLi91] J.-J. E. Slotine and W. Li, *Applied Nonlinear Control*, New Jersey: Prentice-Hall, 1991.

[SpCr98] J.C. Spall and J.A. Cristion, "Model-free control of nonlinear stochastic systems with discrete-time measurements," *IEEE Transactions on Automatic Control*, vol. 43, no. 9, pp. 1198-1210, 1998.

[SuBa98] R.S. Sutton, and A.G. Barto, *Introduction to Reinforcement Learning*, Cambridge, Mass.: MIT Press, 1998.

[SuDe97] J.A.K. Suykens, B.L.R. De Moor and J. Vandewalle, "NLq Theory: A Neural Control Framework with Global Asymptotic Stability Criteria," *Neural Networks*, vol. 10, pp. 615-637, 1997.

[VaVe96] A.J.N. Van Breemen and L.P.J. Veelenturf, "Neural Adaptive Feedback Linearization Control," *Journal A*, vol. 37, pp. 65-71, 1996.

[WhSo92] D.A. White and D.A. Sofge, Eds., *The Handbook of Intelligent Control*, New York: Van Nostrand Reinhold, 1992.

[WiRu94] B. Widrow, D.E. Rumelhart, and M.A. Lehr, "Neural networks: Applications in industry, business and science," *Journal A*, vol. 35, no. 2, pp. 17-27, 1994.

[WiWa96] B. Widrow and E. Walach, *Adaptive Inverse Control*, New Jersey: Prentice Hall, 1996.