Chapter 10

Linear Least Squares Methods

Chapter Contents

In this chapter, we introduce batch and recursive least squares methods for tuning approximator structures where the parameters that will be tuned enter linearly. In particular, we study the tuning of the $p \times 1$ vector θ for the linear in the parameters approximator

$$F_{lip}(x, \theta) = \theta^\top \phi(x)$$

where $\phi(x)$ is a known specified $p \times 1$ vector function. For the tuning, we use the given set of training data $G = \{(x(i), y(i)) : i = 1, 2, \ldots, M\}$.

Section 9.3 outlined several approximator structures that fit this form, including the linear approximator $F_l(x, \theta)$; the polynomial approximator $F_{poly}(x, \theta)$ where the coefficients are tuned; the multilayer perceptron $F_{mlp}(x, \theta)$ with one hidden layer, a linear activation function at the output, and known activation functions in the hidden layer; and the Takagi-Sugeno fuzzy system $F_{ts}(x, \theta)$ with known premise membership functions. In each case, the function $\phi(x)$ is known once x is specified, and the form of θ depends on which approximator structure you use.

The batch least squares method can be used to find approximator parameters that enter linearly when all training data is given a priori.

In this chapter, we simply focus on tuning of θ and will not concern ourselves (except in the examples) with which of the approximator structures is used to implement the approximator (i.e., we will not focus on the construction of $\phi(x)$).

10.1 Batch Least Squares

First, we derive the least squares solution to the approximation problem. Then we provide a simple example where we fit a line to data, and a more interesting example where we train a multilayer perceptron and Takagi-Sugeno fuzzy system to match the function in Figure 9.10.

10.1.1 Batch Least Squares Derivation

In the batch least squares method, we define

$$Y(M) = [y(1), y(2), \ldots, y(M)]^\top$$

to be an $M \times 1$ vector of output data where the $y(i)$, $i = 1, 2, \ldots, M$ come from G (i.e., $y(i)$ such that $(x(i), y(i)) \in G$). We let

$$\Phi(M) = \begin{bmatrix} \phi^\top(x(1)) \\ \phi^\top(x(2)) \\ \vdots \\ \phi^\top(x(M)) \end{bmatrix}$$

be an $M \times p$ matrix that is constructed by stacking the $1 \times p$ $\phi^\top(x(i))$ vectors into a matrix (i.e., the $x(i)$ are such that $(x(i), y(i)) \in G$). Let $\epsilon(i) = y(i) - F_{lip}(x(i), \theta) = y(i) - \theta^\top \phi(x(i))$, which is the same as

$$\epsilon(i) = y(i) - \phi^\top(x(i))\theta$$

be the error in approximating the data pair $(x(i), y(i)) \in G$ where θ is used in the approximation structure. Define

$$E(M) = [\epsilon(1), \epsilon(2), \ldots, \epsilon(M)]^\top$$

so that

$$E = Y - \Phi\theta$$

Choose

$$J(\theta, G) = \frac{1}{2} E^\top E$$

to be a measure of how good the approximation is for all the data in G for a given θ. $J(\theta, G)$ is the sum of the squares of the errors in approximation for each of the training data pairs. We want to pick θ to minimize $J(\theta, G)$ and that is why we use the term "least squares." It is "linear" least squares since our approximator is linear in the parameters.

Notice that $J(\theta, G)$ is convex in θ so that a local minimum is a global minimum. Next, we seek to find the value of θ that will achieve the global minimum. Using basic ideas from calculus, if we take the partial derivative of J with respect to θ and set it equal to zero, we get an equation for θ, the best estimate (in the least squares sense) of the unknown θ^*. Leaving this approach to the derivation (which depends on the use of vector calculus) to a homework exercise, we take a simple (matrix) algebraic approach to the minimization by noting that

$$2J = E^\top E = Y^\top Y - Y^\top \Phi\theta - \theta^\top \Phi^\top Y + \theta^\top \Phi^\top \Phi\theta$$

Then, we "complete the square" by assuming that $\Phi^\top \Phi$ is invertible and letting

$$\begin{aligned} 2J &= Y^\top Y - Y^\top \Phi\theta - \theta^\top \Phi^\top Y + \theta^\top \Phi^\top \Phi\theta \\ &\quad + Y^\top \Phi(\Phi^\top\Phi)^{-1}\Phi^\top Y - Y^\top \Phi(\Phi^\top\Phi)^{-1}\Phi^\top Y \end{aligned}$$

(where we are simply adding and subtracting the same terms at the end of the equation). Hence,

$$\begin{aligned} 2J &= Y^\top(I - \Phi(\Phi^\top\Phi)^{-1}\Phi^\top)Y \\ &\quad + (\theta - (\Phi^\top\Phi)^{-1}\Phi^\top Y)^\top \Phi^\top\Phi(\theta - (\Phi^\top\Phi)^{-1}\Phi^\top Y) \end{aligned} \tag{10.1}$$

The parameters computed via batch least squares minimize the sum of the squared error between the approximator output and the training data outputs; however, it only adjusts the parameters that enter linearly to achieve this minimization.

The first term in this equation is independent of θ, so we cannot reduce $J(\theta, G)$ via this term, so it can be ignored. Hence, to get the smallest value of $J(\theta, G)$, we choose θ so that the second term is zero. We will denote the value of the parameters that achieves the minimization of J by θ, and we notice that

$$\theta = (\Phi^\top\Phi)^{-1}\Phi^\top Y \tag{10.2}$$

since the smallest we can make the last term in the above equation is zero (since it is positive). This is the equation for batch least squares that shows we can directly compute the least squares estimate θ from the "batch" of data that are taken from G and loaded into Φ and Y. If we pick the inputs to the system so that it is "sufficiently excited" [331], then we will be guaranteed that $\Phi^\top \Phi$ is invertible; if the data come from a linear mapping with p (the number of parameters in the linear in the parameters approximator) as the number of underlying linear terms in the nonlinear function, then for sufficiently large M we will achieve perfect estimation of the plant parameters.

In "weighted" batch least squares, we use

$$J(\theta, G) = \frac{1}{2} E^\top W E \tag{10.3}$$

where, for example, W is an $M \times M$ diagonal matrix with its diagonal elements $w_i > 0$ for $i = 1, 2, \ldots, M$ and its off-diagonal elements equal to zero. These w_i can be used to weight the importance of certain elements of G more than others. For example, we may choose to have it put less emphasis on older data by choosing $w_1 < w_2 < \cdots < w_M$ when $x(2)$ is collected after $x(1)$, $x(3)$ is collected after $x(2)$, and so on. One way to select the weights in this case is to suppose that $0 < \lambda \leq 1$, then let $w_i = \lambda^{M-i}$, $i = 1, 2, \ldots, M$. In any case, the resulting parameter estimates can be shown to be given by

$$\theta_{wbls} = (\Phi^\top W \Phi)^{-1} \Phi^\top W Y \tag{10.4}$$

To show this, simply use Equation (10.3) and proceed with the derivation in the same manner as above.

10.1.2 Numerical Issues in Computing the Estimate

In practical problems, numerical issues often arise in computing the inverse

$$(\Phi^\top \Phi)^{-1}$$

needed to compute the batch least squares solution due to $\Phi^\top \Phi$ being "ill-conditioned." Such issues can arise even for relatively simple "academic" problems. For example, these issues arise in the examples to be considered in this book where we typically use the Matlab "backslash" operation to compute the least squares estimate as

```
theta = Phi \ Y
```

where `theta` is θ, `Phi` is Φ, and Y is `Y`. Basically, most view the inverse in Equation (10.2) as a statement of how the least squares estimate is found theoretically. In practice, direct computation of the inverse is generally not used.

In practical applications, numerical issues in computing the least squares estimate must be confronted.

To avoid numerical issues you have several options. First, if you can select $x(i)$ explicitly (which you often cannot, either due to physical limitations of the mapping you are trying to learn, or because you cannot pick $x(i)$ because it it provided by another system), then you can avoid the problems. To do this, basically you want to choose the $x(i)$ so that the $\phi(x(i))$ that are loaded into Φ have values that are aligned in such a way that the inverse can be computed (i.e., so that $\Phi^\top \Phi$ is positive definite, with a good "condition number"). Without getting into details, one way to get "rich" enough data so that the inverse is computable is to use noise as the components of $x(i)$. Of course, this is not always possible, so we often have to turn to other methods.

For instance, a common approach to solve numerical problems with computing the least squares solution is to use a "square root" method. The details of the variety of possible methods and their advantages and disadvantages are beyond the scope of this discussion; however, if you run into numerical problems, you can basically proceed in four ways. First, you can rely on an existing software package to provide a numerically sound solution (i.e., perhaps you should not just employ a direct method to computing the inverse but use more sophisticated methods). Second, you can see the "For Further Study" section at the end of this part to find references to learn more about how to overcome numerical problems. Third, you could turn to an RLS (or gradient) approach to process the data sequentially (e.g., by cycling several times through the data set G) as we explain in the next sections. Fourth, you could use the singular value decomposition approach that we discuss next, whose solution has interesting and useful properties.

It is possible to provide the least squares solution whether or not $\Phi^\top \Phi$ is invertible. The common approach to doing this is to use the singular value decomposition (SVD) method. If Φ is an $M \times p$ matrix and U and V are $M \times M$ and $p \times p$ "unitary" matrices, respectively (i.e., $U^\top U = I$ and $V^\top V = I$ so $U^{-1} = U^\top$ and $V^{-1} = V^\top$), then the SVD of Φ is

$$U^\top \Phi V = \begin{bmatrix} \Sigma & 0 \\ 0 & 0 \end{bmatrix} = S$$

where S is $M \times p$, the "0" elements in S are, in general, matrices (what are their dimensions?),

$$\Sigma = diag(\sigma_1, \sigma_2, \ldots, \sigma_r)$$

where

$$\sigma_1 \geq \sigma_2 \geq, \cdots, \geq \sigma_r > 0$$

are the "singular values" and $r = rank(\Phi)$.

The least squares estimate is then

$$\theta = (\Phi^\top \Phi)^{-1} \Phi^\top Y = V \begin{bmatrix} \Sigma^{-1} & 0 \\ 0 & 0 \end{bmatrix} U^\top Y$$

Note that the matrix

$$\begin{bmatrix} \Sigma^{-1} & 0 \\ 0 & 0 \end{bmatrix}$$

in this computation is a $p \times M$ matrix. Also, note that the SVD computes $(\Phi^\top \Phi)^{-1} \Phi^\top$, which is the "pseudoinverse" of matrix Φ. To see that this is a valid computation for the least squares estimate, recall that

$$J(\theta, G) = \frac{1}{2} E^\top E = \frac{1}{2} (Y - \Phi\theta)^\top (Y - \Phi\theta)$$

and since U and V are unitary,

$$J(\theta, G) = \frac{1}{2}\left[U^\top(Y - \Phi V^\top V\theta)\right]^\top \left[U^\top(Y - \Phi V^\top V\theta)\right]$$

Now, let

$$V^\top\theta = \bar{v} = \begin{bmatrix} \bar{v}_1 \\ \bar{v}_2 \end{bmatrix}$$

where $\bar{v}_1$ is $r \times 1$ and $\bar{v}_2$ is $(n-r) \times 1$, and

$$U^\top Y = \bar{u} = \begin{bmatrix} \bar{u}_1 \\ \bar{u}_2 \end{bmatrix}$$

where $\bar{u}_1$ is $r \times 1$ and $\bar{u}_2$ is $(M-r) \times 1$. Note that since we can choose θ to minimize $J(\theta, G)$, we can choose $\bar{v}$. Since

$$U^\top \Phi V = \begin{bmatrix} \Sigma & 0 \\ 0 & 0 \end{bmatrix}$$

we know that

$$\begin{aligned} J(\theta, G) &= \left[\begin{bmatrix} \bar{u}_1 \\ \bar{u}_2 \end{bmatrix} - \begin{bmatrix} \Sigma & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} \bar{v}_1 \\ \bar{v}_2 \end{bmatrix}\right]^\top \left[\begin{bmatrix} \bar{u}_1 \\ \bar{u}_2 \end{bmatrix} - \begin{bmatrix} \Sigma & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} \bar{v}_1 \\ \bar{v}_2 \end{bmatrix}\right] \\ &= \begin{bmatrix} \bar{u}_1 - \Sigma\bar{v}_1 \\ \bar{u}_2 \end{bmatrix}^\top \begin{bmatrix} \bar{u}_1 - \Sigma\bar{v}_1 \\ \bar{u}_2 \end{bmatrix} \end{aligned}$$

To get $J(\theta, G)$ as small as possible, choose

$$\bar{v}_1 = \Sigma^{-1}\bar{u}_1$$

and also choose $\bar{v}_2 = 0$ (since we can choose it to be anything we would like). We have $V^\top\theta = \bar{v}$ so

$$\theta = V\bar{v} = V\begin{bmatrix} \Sigma^{-1}\bar{u}_1 \\ 0 \end{bmatrix} = V\begin{bmatrix} \Sigma^{-1} & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} \bar{u}_1 \\ \bar{u}_2 \end{bmatrix} = V\begin{bmatrix} \Sigma^{-1} & 0 \\ 0 & 0 \end{bmatrix}U^\top Y$$

In the weighted batch least squares case, with W a diagonal matrix with all positive numbers on its diagonal, if you let

$$W = \sqrt{W}\sqrt{W}$$

we know that $\sqrt{W} = \sqrt{W}^\top$. Hence, $\theta_{wbls} = (\Phi^\top\sqrt{W}\sqrt{W}\Phi)^{-1}\Phi^\top\sqrt{W}\sqrt{W}Y$, and if we let $\bar{\Phi} = \sqrt{W}\Phi$ and $\bar{Y} = \sqrt{W}Y$, we have $\theta_{wbls} = (\bar{\Phi}^\top\bar{\Phi})^{-1}\bar{\Phi}^\top\bar{Y}$ and so you can use the same approach as above.

Finally, it is interesting to note that even in the case where $M < p$, where we have the "underdetermined case," the singular value decomposition will provide a solution even though in this case, there are an infinite number of θ solutions. Actually, out of the infinite number of possible solutions to the linear least squares problem in this case, the θ computed via the singular value decomposition is the one solution such that $\theta^\top\theta$ has the smallest possible size (so sometimes it is a reasonable choice).

10.1.3 Example: Fitting a Line to Data

As an example of how batch least squares can be used, suppose that we would like to use this method to fit a line to a set of data. Suppose that $n = 1$. In this case, our parameterized linear (polynomial) approximator is

$$y = F_{lip}(x, \theta) = \theta^\top \phi(x) = \theta^\top [\phi_1(x), 1]^\top = \theta_1 x_1 + \theta_2 \tag{10.5}$$

which is an equation for a line (note that the 1 in the second row of $\phi(x) = [\phi_1(x), 1]^\top$ is used to include the affine term θ_2). Suppose that the data that we would like to fit the line to is given by

$$G = \{(1,1), (2,1), (3,3)\}$$

and that these data were generated from an unknown function $G(x, z)$ (we assume that they are numbered from left to right, so that $(x(1), y(1)) = (1, 1)$). Notice that $M = 3$.

We will use Equation (10.2) to compute the parameters for the line that best fits the data (in the sense that it will minimize the sum of the squared distances between the line and the data). First, let

$$Y = \begin{bmatrix} 1 \\ 1 \\ 3 \end{bmatrix}$$

Next, form the $\phi(x(i))$, $i = 1, 2, 3$, and let

$$\Phi = \begin{bmatrix} 1 & 1 \\ 2 & 1 \\ 3 & 1 \end{bmatrix}$$

With this,

$$\theta = (\Phi^\top \Phi)^{-1} \Phi^\top Y = \left(\begin{bmatrix} 14 & 6 \\ 6 & 3 \end{bmatrix} \right)^{-1} \begin{bmatrix} 12 \\ 5 \end{bmatrix} = \begin{bmatrix} 1 \\ -\frac{1}{3} \end{bmatrix}$$

Hence, the line

$$y = x_1 - \frac{1}{3}$$

best fits the data in the least squares sense.

To see that the line fits the data, consider Figure 10.1, where we plot both the data in G and the line $F_{lip}(x, \theta)$. Clearly, the data were not generated by a linear mapping. The least squares method tries to overcome this problem, and results in a good fit to the data, the best in the least squares sense that is possible for a linear approximator. Notice that the line is raised up toward the two points above it, balancing out the error that is created in the approximation, considering that there is only one point below it.

The same general approach works for larger data sets. The reader may want to experiment with weighted batch least squares to see how the weights w_i affect

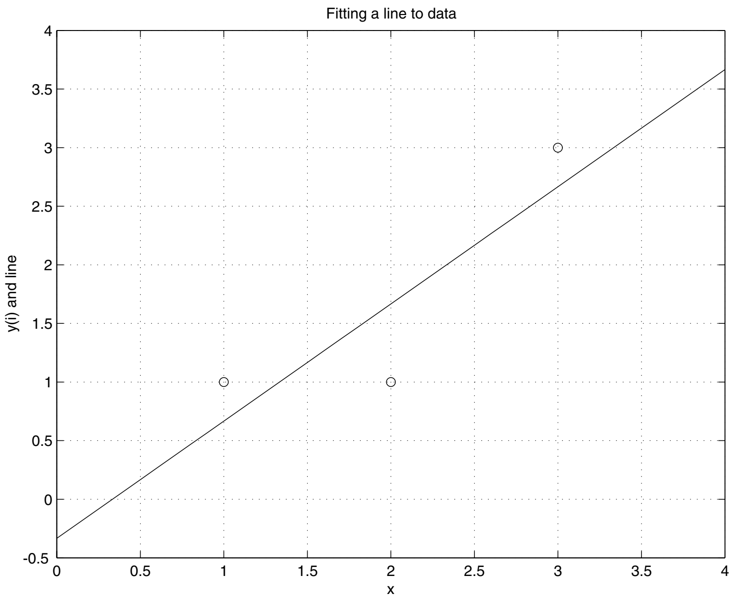

Figure 10.1: Training data and line that is the best fit to the data.

the way that the line will fit the data (making it more or less important that the data fit at certain points). In doing this, you will see that you can, by various choices of the weighting factors, move the line so that it more closely matches any point that you put a relatively high weight value on.

10.2 Example: Offline Tuning of Approximators

In this section we will show how to use batch least squares to tune a multilayer perceptron and Takagi-Sugeno fuzzy system to match the training data shown in Figure 9.10 (this defines G and in our case, we have $M = 121$). In particular, we will first use the same multilayer perceptron and Takagi-Sugeno fuzzy system as studied in Section 9.3 and compare the approximation accuracy when a least squares approach is used to tune the parameters that enter linearly to the approximation accuracy that we obtained via manual tuning.

It is good practice to first try a linear in the parameter approximator, or even one that is linear in its inputs.

10.2.1 Multilayer Perceptrons

Improved Accuracy Over the Manually Tuned Neural Network

Recall that we were using the perceptron with a single hidden layer shown in Figure 9.13 with $n_1 = 2$ neurons in the hidden layer. This is represented by

$$y = F_{mlp}(x, \theta) = \theta^\top \phi(x) = [w_1, w_2, b][\phi_1(x), \phi_2(x), 1]^\top$$

where via our heuristic approach, we used $f(\bar{x}) = \frac{1}{1+\exp(-\bar{x})}$ and had chosen

$$\phi_1(x) = f(b_1 + w_{1,1}x)$$

with $b_1 = 0$ and $w_{1,1} = 1.5$, and

$$\phi_2(x) = f(b_2 + w_{1,2}x)$$

with $b_2 = -6$ and $w_{1,2} = 1.25$. We had chosen $\theta = [3, 1, 0.6]^\top$. We will use the batch least squares approach to see how it can pick a better θ.

Batch least squares is often a very effective method for computing the parameters that enter linearly; however, it relies on your choice of the parameters that enter nonlinearly.

To do this, we simply form the matrices Y and Φ and use the batch least squares formula to find

$$\theta = [2.5747, 1.6101, 0.7071]^\top$$

which, when we use these values for the approximator parameters, results in the approximator shown with the training data in Figure 10.2. The approximation accuracy is clearly better than what we obtained via manual tuning (see Figure 9.15) and the batch least squares method provided an automatic method to pick some of the parameters, in particular, w_1, w_2, and b. For the other parameters we relied on our heuristic tuning discussed earlier.

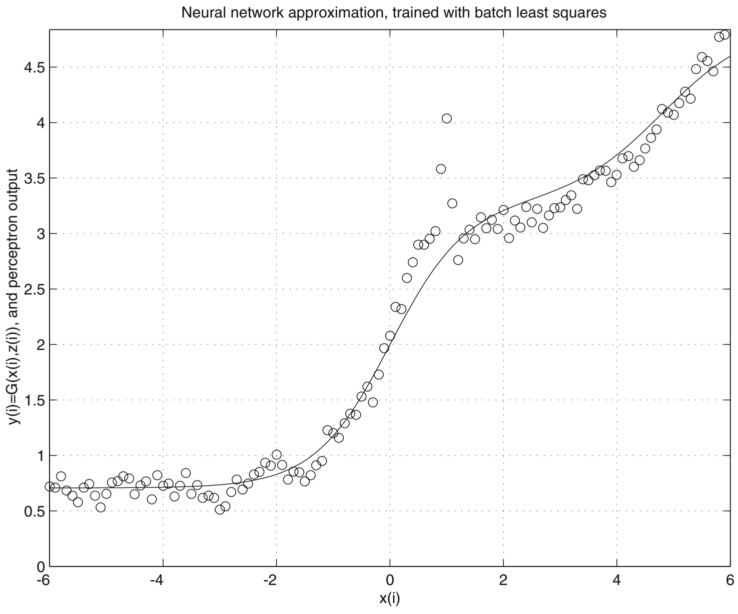

Figure 10.2: Multilayer perceptron approximator trained with batch least squares, 2 neurons.

Seeking More Approximation Accuracy: Increasing the Number of Hidden Neurons

The main reason for not considering more neurons in the hidden layer when we were considering manual tuning of the perceptron for this example was that the tuning can become complicated due to interactions between the tuning parameters. With the assistance of batch least squares, however, we can easily tune approximators with more parameters. Generally, you want to use much more training data than parameters (to avoid what is called "overfitting" below) so since we use $M = 121$, we will now consider $n_1 = 11$ neurons in the hidden layer (for a total of $11(2) + 11 + 1 = 34$ parameters).

Notice, however, that we need a scheme to pick the weights and biases of the hidden layer. To do this, we will use a simple heuristic approach (others are possible, some suggested by the application at hand). To pick the biases, we choose them to be evenly spaced over -5 to 5, so that $b_1 = -5$, $b_2 = -4$, all the way to $b_{11} = 5$. This should help spread the points where the activation functions turn on across the input space. The choice for the w^j, $j = 1, 2, \ldots, 11$, is more difficult if you take the view that we did in the manual tuning of the perceptron. Notice that there we assumed that we could examine the training data and pick off slopes to set these values. This is often unrealistic for complex real world problems. Here, we will exploit the fact that the scaling factors in w are used to modify the slopes to what we will need, so we simply pick $w^j = 1$, $j = 1, 2, \ldots, 11$ (for applications where $n > 1$, this scheme may not be as effective; in those cases, you will want the weights to take on values that will allow for a range of slopes). This completes the specification of the hidden layer.

Next, we use batch least squares to tune the 12 parameters in $\theta = [w^\top, b]^\top$. We get

$$\begin{aligned}\theta \;=\; & [2.7480, 2.0120, -11.9865, 34.7556, -69.6968, 93.4042,\\ & -80.8496, 57.0819, -34.6710, 15.8048, -3.9398, 0.8087]\end{aligned}$$

For this case we get the approximation shown in Figure 10.3, which is a significant improvement over Figure 10.2, where we used $n_1 = 2$ neurons in the hidden layer and Figure 9.15, where we tuned the approximator manually. Notice that in the vector θ, we have both positive and negative values. The negative ones help to implement the parts of the nonlinearity where the slope goes negative. Clearly, it would be quite difficult to tune the approximator manually to get this kind of accuracy.

Fine-Tuning to Capture High Frequency Behavior

Next, to illustrate what can happen if you use even more parameters in your approximator, we use $n_1 = 25$ neurons (to get a total of $25(2) + 25 + 1 = 76$ parameters). We choose the biases in a similar fashion to the above, but spread them over the whole range -6 to 6 to get $b_1 = -6$, $b_2 = -5.5$, all the way to $b_{25} = 6$. As above, we pick all the weights in the hidden layer to be unity. We use batch

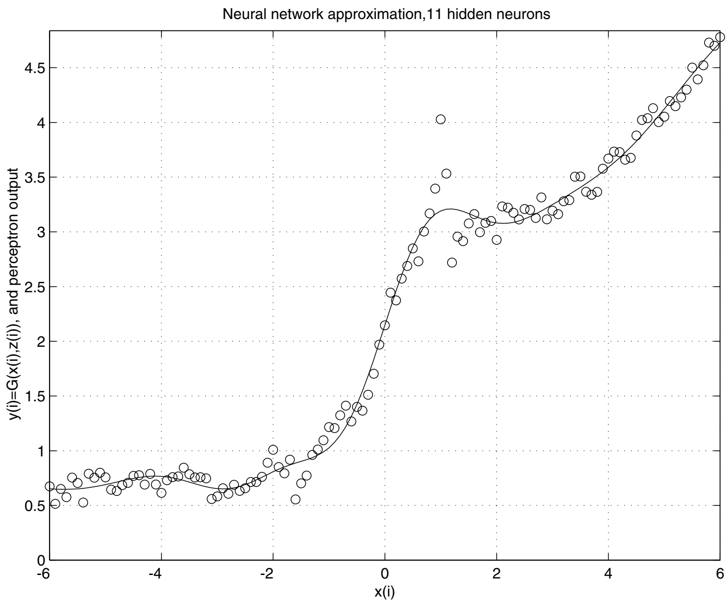

Figure 10.3: Multilayer perceptron approximator trained with batch least squares, 11 neurons.

least squares to tune the 26 parameters in $\theta = [w^\top, b]^\top$. For this case, we get the approximation shown in Figure 10.4, which is an improvement over Figure 10.3, where we used $n_1 = 11$ neurons (notice that the approximator is starting to find some of the structure of the underlying function that is illustrated in Figure 9.9; least squares is particularly good at finding this structure, *in this case*, due to how the noise on z enters). Also, notice that with more neurons we are able to approximate more and more of the "high frequency" behavior in the function (with even more neurons, perhaps concentrated in the region around 1, we can get an even more accurate approximation of the peaking behavior found in that region).

Generally, using a larger approximator structure can improve approximation accuracy; however, if you use a structure that is too complex, it can be too aggressive in trying to represent the noise (overfitting) rather than seeking to achieve a good interpolation.

Overfitting Where the Approximator Seeks to Model Noise

Next, we show that this approach of continually increasing n_1 can be taken too far. Suppose that we choose $n_1 = 121$ (for a total of $121(2) + 121 + 1 = 364$ parameters), $b_1 = -6$, $b_2 = -5.9$, all the way to $b_{121} = 6$, and the weights in the hidden layer as all unity. In this case, we have θ as a 122×1 vector so that we have more parameters to tune than data pairs. We use batch least squares to train the network and the result is shown in Figure 10.5. Notice that in this plot, we have also plotted approximator nonlinearity on top of the function $G(x)$ (i.e., where we have removed the effects of the noise z). This illustrates a very important fact: if you use too many parameters, you may start trying

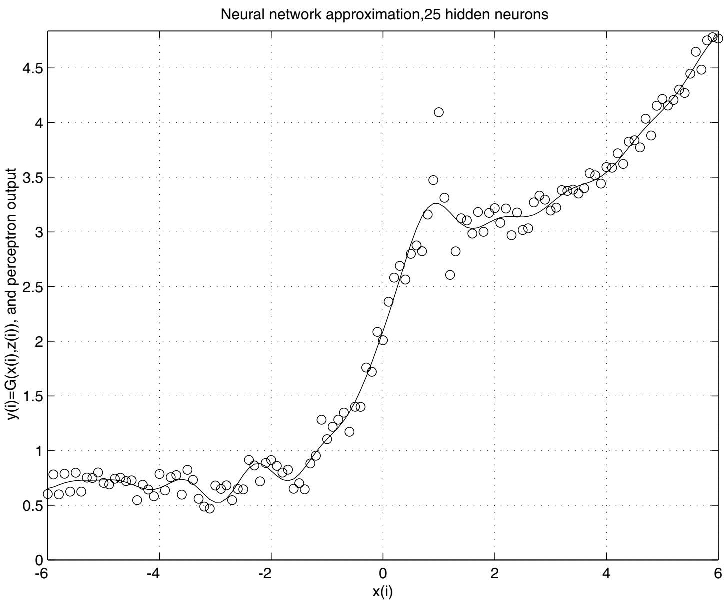

Figure 10.4: Multilayer perceptron approximator trained with batch least squares, 25 neurons.

to approximate characteristics of the noise, and not the underlying function. Even without the presence of z, it is possible to get similar "overfitting" where in between the training data, the approximator moves far away from where it should be (but for this example, if you train without the influence of z in the data, the approximator will do a very good job at approximating the function and does not exhibit this problem). Basically, this highlights the fact that there are often situations where it is desirable to capture some of the higher frequency behavior, but not behavior that is too high a frequency since this may represent uncertainty (noise) in the system.

10.2.2 Takagi-Sugeno Fuzzy Systems

In this section, we study how to tune the Takagi-Sugeno fuzzy system to match the function in Figure 9.10. Here, however, we will not consider the many different cases as we did for the neural network in the last section since the same basic ideas apply (least squares offers a nice automated method for tuning, additional parameters can be used to achieve improved accuracy, and if you use too many parameters, you can get a type of overfitting). Instead, our focus will simply be on how to construct the premise membership functions.

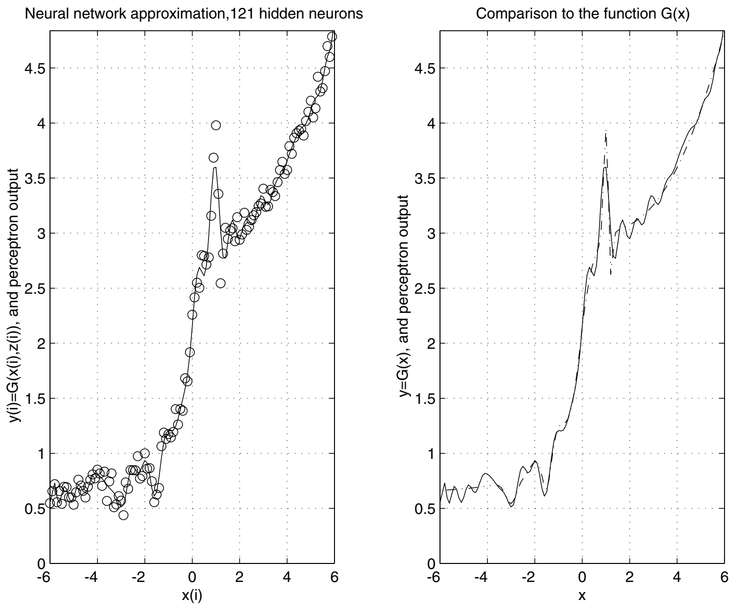

Figure 10.5: Multilayer perceptron approximator trained with batch least squares, 121 neurons, plus comparison to $G(x)$.

Getting Similar Accuracy to the Neural Network

Suppose we use $R = 20$ rules so that we will have $4R = 80$ parameters to tune, a number close to the 76 parameters used for the neural network above, with 25 neurons in the hidden layer. It is interesting to note that we will tune 40 values of the Takagi-Sugeno fuzzy system compared to 26 for the perceptron with $n_1 = 25$ neurons in the hidden layer. For this reason, we will have fewer parameters to tune manually (i.e., the function ϕ for the Takagi-Sugeno fuzzy system takes fewer parameters to specify than the one for the neural network).

Comparisons between approximator structure types must include complexity of the structure, ease of training, and approximation accuracy.

For the Takagi-Sugeno fuzzy system, we have to pick the parameters for

$$\mu_i(x) = \exp\left(-\frac{1}{2}\left(\frac{x_j - c_j^i}{\sigma_j^i}\right)^2\right)$$

where $j = 1$ (since $n = 1$) and $i = 1, 2, \ldots, R$. A logical strategy is to space the c_1^i points on a uniform grid across the x axis (especially in cases where you do not know the form of the underlying function; for this example, since we can easily examine the data, it would make more sense to use a nonuniform distribution of the c_1^i points, with more concentrated where there is more high frequency behavior). To do this, for convenience, we choose to spread the 20 c_1^i points across the range $[-5.4, 6]$ in increments of 0.6. Next, we pick all the $\sigma_1^i = 0.1$. This gives us the ξ_i functions shown in Figure 10.6 and the approximator shown in Figure 10.7. Notice that with our choice of $\sigma_1^i = 0.1$, we get very steep slopes between the basis functions so that they switch somewhat abruptly from one line for the approximator to the next. This results in the somewhat erratic behavior in the plot.

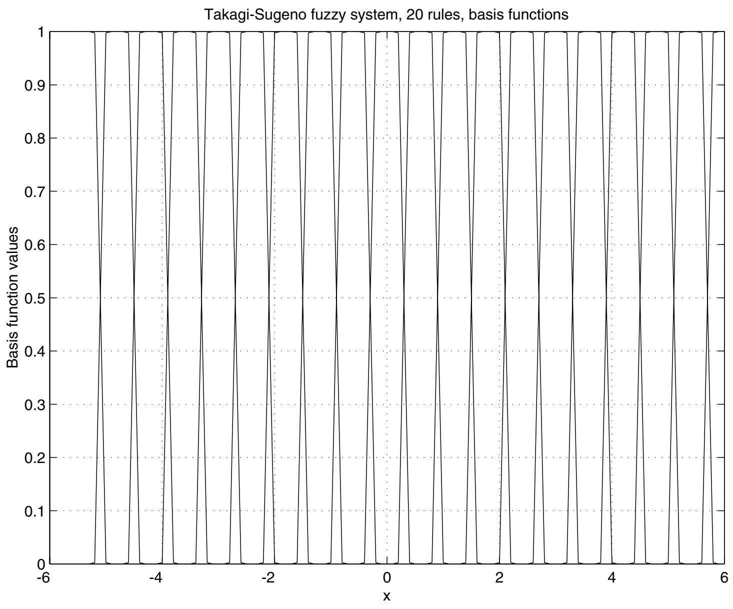

Figure 10.6: Takagi-Sugeno basis functions, $R = 20$, $\sigma_1^i = 0.1$ case.

Manually Tuning the Nonlinear Part of the Approximator

If we pick $\sigma_1^i = 1$, we get the ξ_i functions shown in Figure 10.8 (why are they not perfectly symmetric?) and the approximator shown in Figure 10.9. This shows that the value of $\sigma_1^i = 1$ provides for a much smoother transition between basis functions, which results in smoother transitions between the lines used for approximation. Overall, in terms of approximator accuracy, we obtain results similar to those obtained for the perceptron with $n_1 = 25$ neurons in the hidden layer; however, this may not always be the case. Sometimes, one approximator will be able to achieve better accuracy with fewer parameters.

There exist good intuitive ideas on how to manually tune the nonlinear part of the approximator structure.

Overall, this shows some ideas on how to tune the premise membership functions (that extend to the more general case where $n > 1$). As a final note, we caution against using this discussion—and that given in the last section—to draw general conclusions about which approximator structure to use. In general, different applications will dictate the need for different approximator structures, numbers of parameters to tune, and methods to tune them.

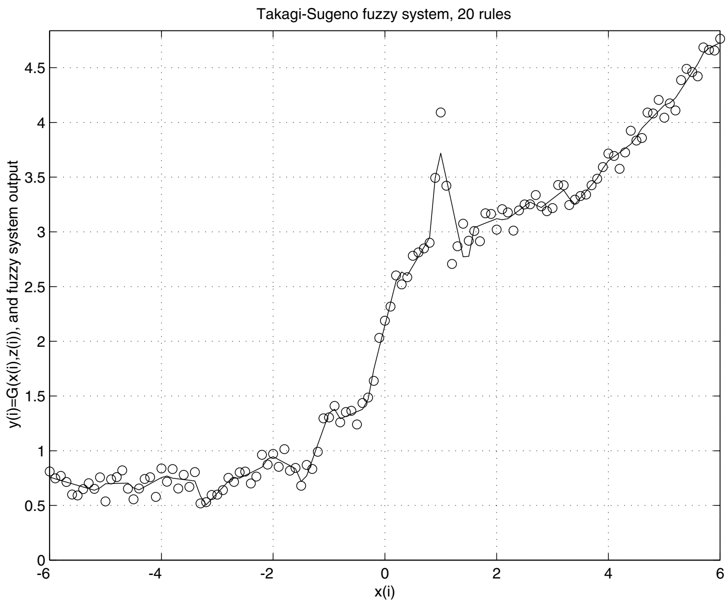

Figure 10.7: Takagi-Sugeno approximator, $R = 20$, $\sigma_1^i = 0.1$ case.

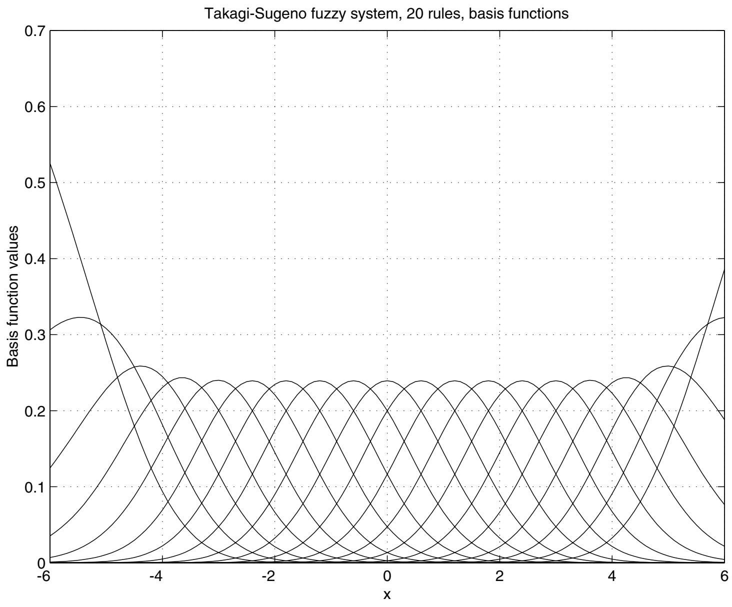

Figure 10.8: Takagi-Sugeno basis functions, $R = 20$, $\sigma_1^i = 1$ case.

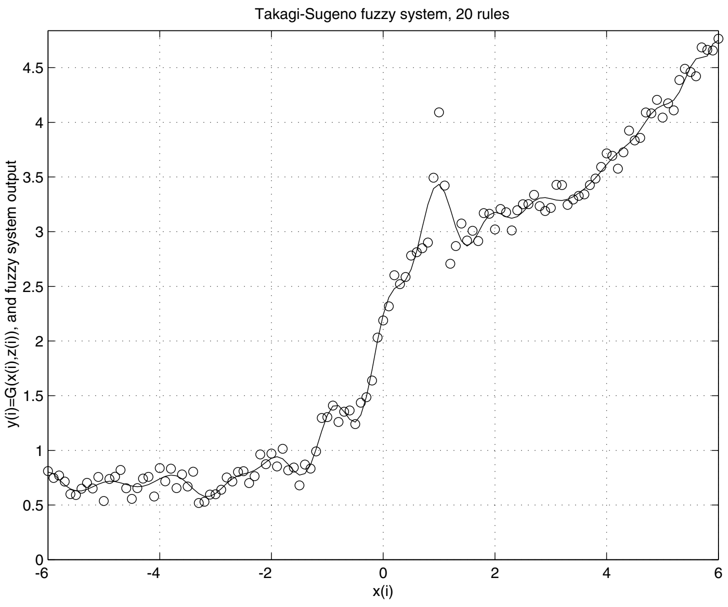

Figure 10.9: Takagi-Sugeno approximator, $R = 20$, $\sigma_1^i = 1$ case.

10.3 Design Example: Rule Synthesis Using Operator Data

In this problem, taken directly from [498] (where other estimation methods are studied), suppose you are given data from how a human operator controls a chemical plant (see the Web site for this book to get the data set). See Figure 10.10, where we suppose that the operator has measurements of monomer concentration (u_1), change in monomer concentration (u_2), monomer flow rate (u_3), some local temperatures in the plant (u_4, u_5), and with these makes decisions on how to select the set point for the monomer flow rate (y). The actual value of the monomer flow rate to be put into the plant is controlled by a PID controller and the value of y is the set point for that controller.

It is possible to construct a fuzzy (or neural) controller from a set of numeric examples of how an expert human would solve the problem. This offers another nonmodel-based strategy to construct a nonlinear controller.

In Chapter 5 we studied how to construct a fuzzy controller using heuristic ideas about how the plant behaves. Here, we take a different approach where we gather plant data (that actually represents the heuristic control ideas of the operator about how to control the plant) and create an interpolator for these data using a fuzzy controller. After appropriate testing, this controller could then be put into operation either to provide advice to novice operators or to completely replace the expert operator.

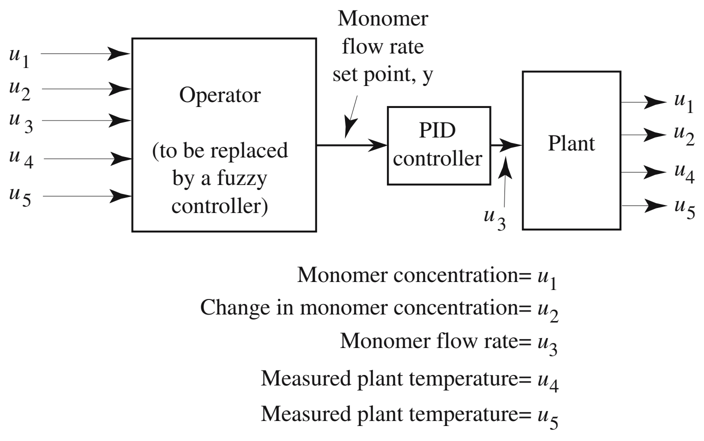
Operator
(to be replaced by a fuzzy controller)
Monomer flow rate set point, y
PID controller
Plant
u_1
u_2
u_3
u_4
u_5
Monomer concentration= u_1
Change in monomer concentration= u_2
Monomer flow rate= u_3
Measured plant temperature= u_4
Measured plant temperature= u_5

Figure 10.10: Operator for controlling a plant.

10.3.1 Data Analysis, Correlation Analysis, and Controller Input Selection

There are $M = 70$ data pairs that were obtained by monitoring how the operator performs this task. In the data set, the first 5 columns hold $u_i(k)$, $i = 1, 2, 3, 4, 5$, and each row corresponds to a different time k. The last column holds the corresponding set point values $y(k)$ that are determined by the operator. The data are shown in Figure 10.11. In practical applications, it is often good to plot the data and examine them. Here, it is interesting to note that we clearly may not have enough data to perform a good approximation over a wide range of values of the inputs since we do not have output settings for a very wide range of input combinations. Moreover, by examining the plots more carefully you may suspect that the operator is not using all five data values to make decisions (the operator is the expert, so while the data might be available, the operator may not use it to make decisions since the operator may have found a few key variables are the important ones to consider).

From our examination of the data, we begin by performing some data analysis to study how the operator makes decisions. In particular, using the approach in [343], we calculate the correlation coefficients between each input and the output (and in fact, between all the different variables) and we show this in Figure 10.12. Now, while this is a linear analysis, it does give an indication of which inputs are important to the operator in making decisions. Notice that $u_4(k)$ and $u_5(k)$ do not have a high correlation with the output $y(k)$ (the magnitudes of the correlation coefficients are less than about 0.2 for both cases), so this leads us to suspect that the operator is ignoring these inputs in his decision-making process (perhaps the operator could be asked if this is the case). Moreover, $u_2(k)$ has a correlation coefficient of only about 0.33 so it does not seem to be a key variable for decision-making either. Notice, however, that $u_1(k)$ and

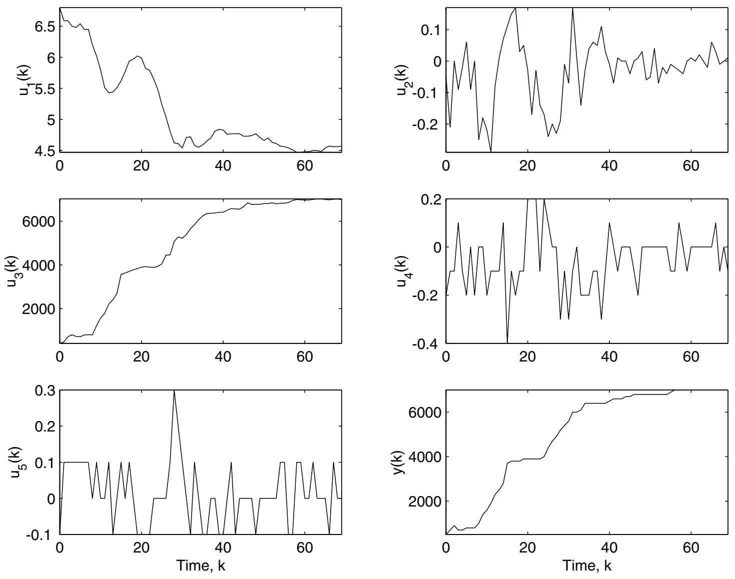

Figure 10.11: Data that indicate how the operator selects the set point for the monomer flow rate.

$u_3(k)$ have correlation coefficients that are close to 1 in magnitude and this indicates that each of these variables seems to be important in the decision-making process. Recall that $u_3(k)$ is the monomer flow rate (the output of the PID controller) so we expect a correlation with $y(k)$, the set point for the controller (the correlation indicates that the PID controller is successful in forcing the actual monomer flow rate to be equal to the one that is commanded by the operator). The input $u_1(k)$ is the monomer concentration and seems to be a key variable for decision-making.

While the above analysis is instructive, it is also important to consider the cross-correlation between the inputs that we decide to keep as inputs to the controller. If one input is significantly correlated to another one that you want to keep, then it may be that they are carrying basically the same information so it might be possible to remove one of them. For instance, the correlation coefficient between $u_1(k)$ and $u_3(k)$ is -0.9381 so since its magnitude is near 1, it seems that removing one of these inputs is possible. From the physics of the problem, it does not make sense to *only* use the input $u_3(k)$ as an input to the controller since it is the actual value of the monomer flow rate that is input to the plant, and its value is directly dictated by the monomer flow rate set point that is set by the controller to be constructed. Hence, when we only want to use one input variable, we will consider the case where we remove $u_3(k)$ and hence, only use $u_1(k)$ as the input to the controller. We will, however, also consider the use of other inputs as you will see below.

Before proceeding, however, note that there are other methods for selecting

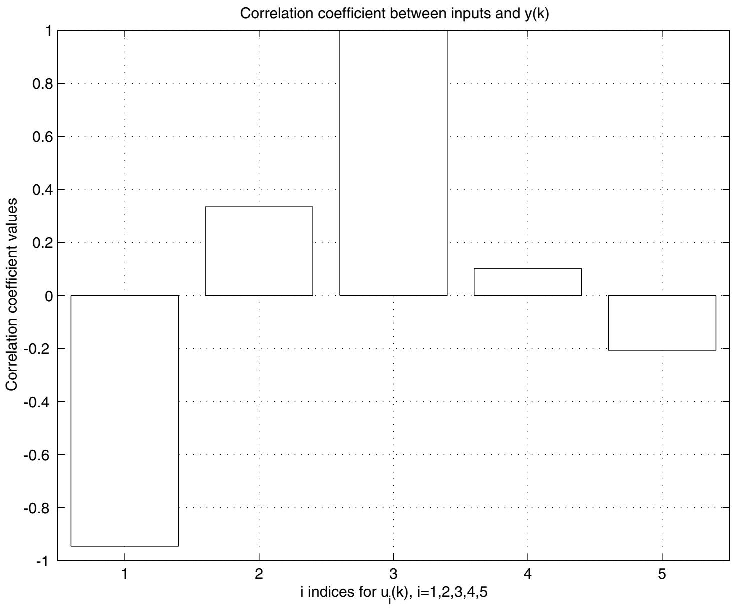

Figure 10.12: Correlation coefficients between each input, $u_i(k)$, $i = 1, 2, 3, 4, 5$, and the output $y(k)$.

inputs to the controller and that these of course also apply to general function approximation problems (note that we are essentially trying here to pick the regressor vector length and composition). While here we use the approach in [343], you could, in some situations (e.g., when you have plenty of training data and not too many inputs), simply use an exhaustive approach where you train approximators for all possible combinations of input variables. Another approach is to normalize the data so that they all lie between -1 and 1, and then construct a linear least squares estimator between the inputs and the output and consider the magnitude of the regressor coefficients. Then you can discard regressor components that have coefficients that are small in magnitude. Note that even though this is also a linear analysis approach, it can lead to different conclusions from the correlation analysis above. Moreover, all this analysis is complicated by the fact that the conclusions that you reach can depend on the controller that you end up constructing (e.g., you may have two sets of inputs to choose between, and your analysis may say that one is better than the other, but you may not be able to construct a nonlinear approximator that performs better for that set of inputs).

10.3.2 Determine if a Linear Controller Is Sufficient

We start by trying to use a linear (actually affine) mapping to fit the operator data so that we get a linear (affine) controller. We do this first for two reasons.

First, if the linear controller performs reasonably well, then we will guess that the linear correlation analysis of the last subsection is valid. Second, if the linear controller works well, then we will want to use it since it is simpler to implement than a fuzzy controller.

Due to the lack of a significant amount of training data, we will train the approximator using all $M = 70$ data pairs (this specifies G). This approach, however, creates problems with validating the accuracy of the approximation since we can only test at the data that the approximator was trained at. Here, since we cannot access the plant to generate more data, we will artificially generate a test data set. To do this, we simply create data points in between each of the given data points by taking the average value of two adjacent points (i.e., average value of each component), and associating it with the average value between two output data points. This will give us $M_\Gamma = 69$ test data pairs in our test set Γ. We will treat these values as if they were actually generated in an experimental setting.

Using a linear least squares method to train an affine approximator structure, we get the results shown in Figure 10.13. For this, if $F(x, \theta)$ is the affine approximator mapping with θ chosen using batch least squares and we use all the inputs so

$$x(k) = [u_1(k), u_2(k), u_3(k), u_4(k), u_5(k)]^\top$$

we get a mean squared error at the training data of

$$\frac{1}{M} \sum_{(x,y)\in G} ((y - F(x,\theta))^2 = 1.1142 \times 10^4$$

and we get a mean squared error at the test data of

$$\frac{1}{M_\Gamma} \sum_{(x,y)\in \Gamma} ((y - F(x,\theta))^2 = 8.6598 \times 10^3$$

Note that the mean squared error values at the training and testing data are similar, but in this case the training error is higher (this is a bit atypical; normally the test error is slightly higher).

Notice that we achieve reasonable approximation accuracy, but there are several points at which there are significant deviations between what the operator did and what the linear controller does (suppose that the operator feels that the errors are "significant"). We could conclude from this, however, that a linear estimator does reasonably well, so we place more confidence in our earlier correlation analysis. From this, we suspect that we may be able to remove input variables and achieve similar approximation accuracy.

10.3.3 Study the Effects of Removing Input Variables

We could study the performance of the approximator by successively removing more input variables. Here, we will trust the earlier correlation analysis and first consider a two-input linear controller that only uses $u_1(k)$ and $u_3(k)$ as

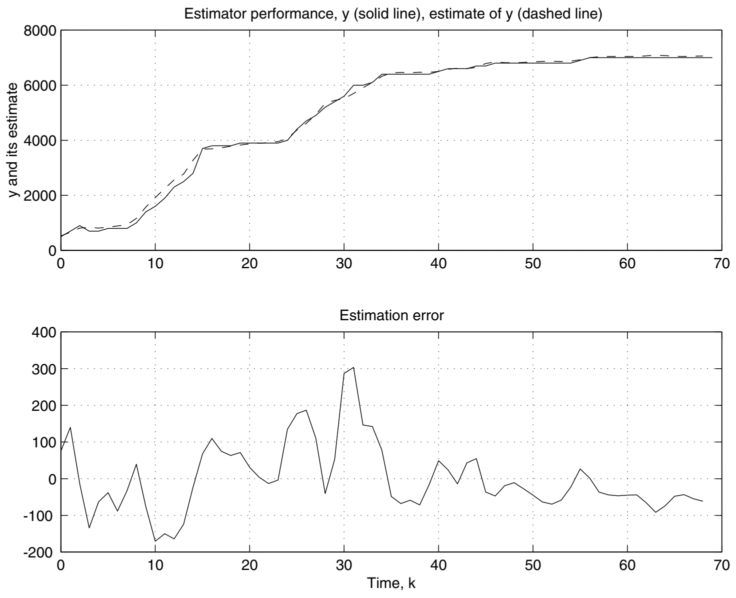

Figure 10.13: Operator settings, linear controller settings, and error between these.

inputs (the case for using inputs $u_1(k)$, $u_2(k)$, and $u_3(k)$ is similar, with just a slightly worse approximation error than the case where we use all the inputs). After that we will consider the case where we only use the input $u_1(k)$.

Using a linear least squares method to train an affine approximator structure with only two inputs, we get the results shown in Figure 10.14. For this, if $F(x, \theta)$ is the affine approximator mapping with θ chosen using batch least squares and

$$x(k) = [u_1(k), u_3(k)]^\top$$

we get a mean squared error at the training data of

$$\frac{1}{M} \sum_{(x,y)\in G} ((y - F(x, \theta))^2 = 1.1669 \times 10^4$$

and we get a mean squared error at the test data of

$$\frac{1}{M_\Gamma} \sum_{(x,y)\in \Gamma} ((y - F(x, \theta))^2 = 9.1087 \times 10^3$$

Notice that our mean squared error did not increase drastically even though we removed three inputs.

Notice, however, that if we only use $u_1(k)$ as an input, then using a linear least squares method to train an affine approximator structure with only one

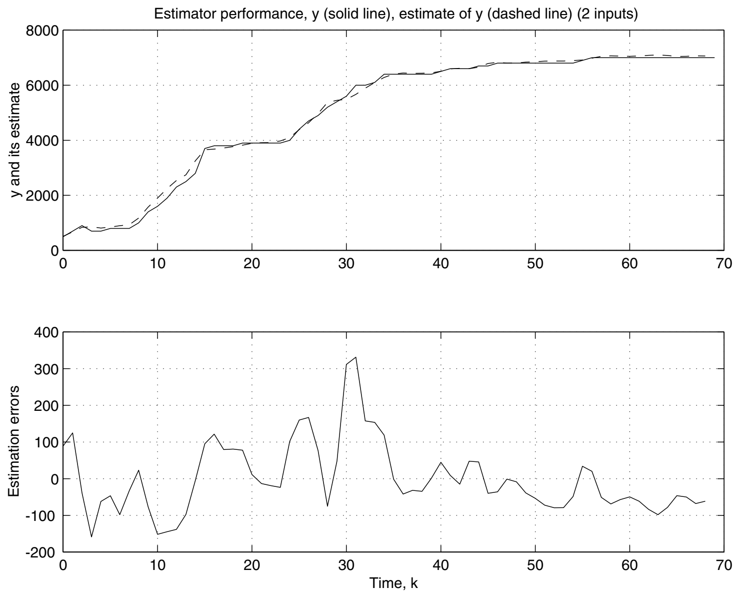

Figure 10.14: Operator settings, linear controller settings, and error between these, $u_1(k)$ and $u_3(k)$ as inputs.

input, we get the results shown in Figure 10.15. For this, if $F(x, \theta)$ is the affine approximator mapping with θ chosen using batch least squares and

$$x(k) = [u_1(k)]^\top$$

we get a mean squared error at the training data of

$$\frac{1}{M} \sum_{(x,y) \in G} ((y - F(x, \theta))^2 = 5.2090 \times 10^5$$

and we get a mean squared error at the test data of

$$\frac{1}{M_\Gamma} \sum_{(x,y) \in \Gamma} ((y - F(x, \theta))^2 = 5.0309 \times 10^5$$

Notice that in this case, we get a significant degradation in performance. You could be led to several different conclusions. First, you may think, via the earlier correlation analysis, that even though the cross-correlation between $u_1(k)$ and $u_3(k)$ was high, there was still some important information in the $u_3(k)$ input that we are now ignoring. In that case it would seem that the operator is primarily looking at two inputs to make decisions. However, there is a second possibility that is important to consider. It is possible that the linear approach is failing. In particular, it could be that the errors for the single-input

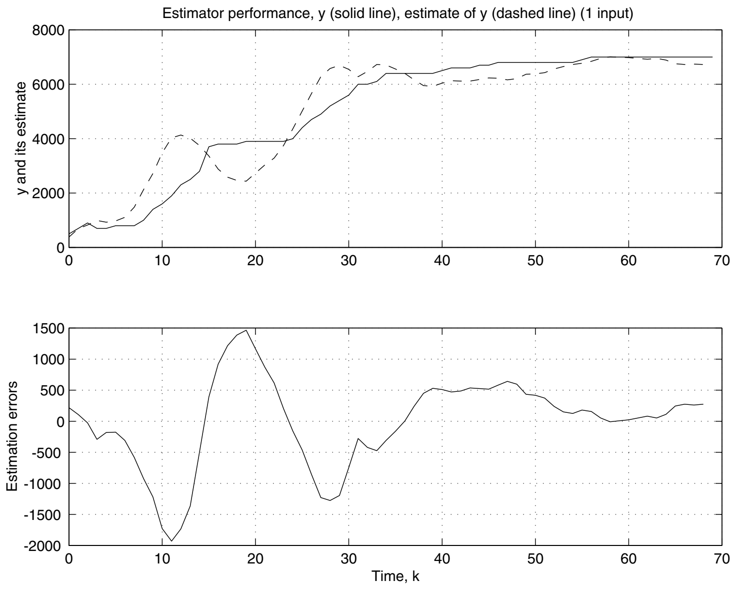

Figure 10.15: Operator settings, linear controller settings, and error between these, $u_1(k)$ as an input.

case are arising due to the fact that there is a nonlinearity in the underlying operator decision-making that the linear mapping is not suited to represent. It is for this reason that we turn to a nonlinear approximator, the fuzzy system, and try to use only one input (of course you could also construct a neural network in an analogous manner). We keep in mind, however, that if we do not succeed in this approach, we will try to add a second input, $u_3(k)$.

10.3.4 Construct a Fuzzy Controller from Operator Data

Next, we attempt to reduce the approximation errors so that the decisions made are closer to those of the operator than what we were able to obtain with a linear approximator, no matter how many inputs were used. We first construct a single-input fuzzy controller. We will, in fact, construct a Takagi-Sugeno fuzzy system with Gaussian input membership functions and affine consequent functions, both with only one input. When this is done, using $R = 9$ rules and one choice for the membership function parameters, we do reduce the approximation error (we get a mean squared testing error of 3.0702×10^5), but not significantly, and it is *worse* than the cases in the last section, where we also used $u_2(k)$ and $u_3(k)$ as inputs. Now, you could increase the number of membership functions on the input universe of discourse to try to improve accuracy; however, we will take a different approach here since the development of the linear approximators indicates that the inputs $u_2(k)$ and $u_3(k)$ do carry some information.

First Attempt: Problems with Overfitting

Since we have found via the correlation analysis that $u_1(k)$ seems to be the most informative input variable, we will use it as an input to the premise membership functions; however, for the consequent membership functions we will use either $u_1(k)$, $u_2(k)$, and $u_3(k)$, or all the inputs to try to get as much information from the inputs as possible. Notice that this will increase the complexity of the approximator, but for R rules there are only $R(n+1)$ consequent parameters (n is the number of inputs to the consequent membership functions). For R rules, with only one input to the premise membership functions, there are only $2R$ parameters needed to define the membership functions. Notice that if you used all the inputs to the premise membership functions, and all possible combinations of rules (that results from gridding the input space with N membership functions on each input dimension), then we need $2R$ parameters but in this case, $R = N^n$ so that the approximator can easily become very complex. (If we used $n = 5$ inputs with $N = 3$ membership functions on each input universe of discourse, then there would be $R = 3^5 = 243$ rules which would be defined with 486 parameters, which is far greater than M, not even considering the additional parameters needed for the consequent functions.)

Here, we will consider two cases. In both cases we will use one input to the premise membership function and $R = 9$ rules, so we will need 18 parameters to define the input membership functions. We will, however, consider different numbers of inputs to the consequent functions. First, we will consider using 3 inputs to the consequent functions ($u_1(k)$, $u_2(k)$, and $u_3(k)$); hence, with $n = 3$ we will need $R(n+1)$ parameters for a total of $18 + 9(4) = 54$ parameters in the approximator. In the second case, we will consider using all the inputs to the consequent functions so that there will be $18 + 9(6) = 72$, which is greater than $M = 70$; hence, in the second case we must be especially concerned that the approximator will "overfit" the data and hence not generalize well in between the data.

We use a grid on the input space of 9 input membership functions so we get $R = 9$ (we omit the actual values of the centers and spreads of the Gaussian input membership functions and invite the reader to solve this problem in a design problem at the end of the chapter). We use a linear least squares method to train the Takagi-Sugeno fuzzy system approximator. In the first case, we use $u_1(k)$, $u_2(k)$,and $u_3(k)$ as inputs to the consequent functions and get the results shown in Figure 10.16. For this, if $F(x, \theta)$ is the Takagi-Sugeno fuzzy system approximator mapping with θ chosen using batch least squares, we get a mean squared error at the training data of

$$\frac{1}{M} \sum_{(x,y)\in G} ((y - F(x, \theta))^2 = 2.2077 \times 10^3$$

and we get a mean squared error at the test data of

$$\frac{1}{M_\Gamma} \sum_{(x,y)\in \Gamma} ((y - F(x, \theta))^2 = 1.1329 \times 10^4$$

Note that the testing error is significantly higher than the training error. This result shows that there is some overfitting occurring (in between the training data there are some excursions where the interpolation is not performing very well). If we proceeded according to our plan and used all the inputs in the consequent functions, we find that this overfitting problem gets significantly worse so we do not present those results (we get a mean squared training error of 594.1489 and a mean squared testing error of 6.3452×10^9). Clearly, the fact that we have more parameters to tune than training data is causing a significant problem in this last case. This example clearly shows the importance of using both training and testing sets; if you only used the training data set you would think that you had significantly improved approximation accuracy when in fact all you have done is match the training data very well. While in operation, when data different from the training data are encountered, the controller could provide very unreasonable inputs.

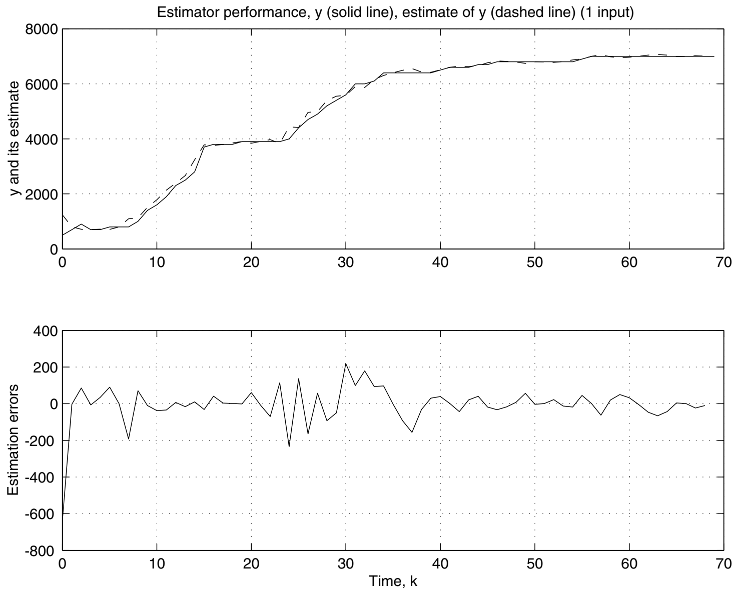

Figure 10.16: Operator settings, fuzzy controller settings, and error between these (one input to the premise membership functions, three to the consequent functions).

Before we continue with the design process for the approximator, consider Figure 10.17 where, using ideas from [343], we see that by including the $u_1(k)$, $u_2(k)$, and $u_3(k)$ inputs in producing the result in Figure 10.16, they are uncorrelated with the estimation error (notice that while this correlation analysis is again linear, it does take into account the nonlinear mapping implemented by the Takagi-Sugeno fuzzy system). Notice also that the $u_4(k)$ and $u_5(k)$ inputs are only a bit correlated with the approximation error so that we expect that if we add these inputs, they probably would not help much with approximation accuracy (but this is just a guess based on the linear analysis).

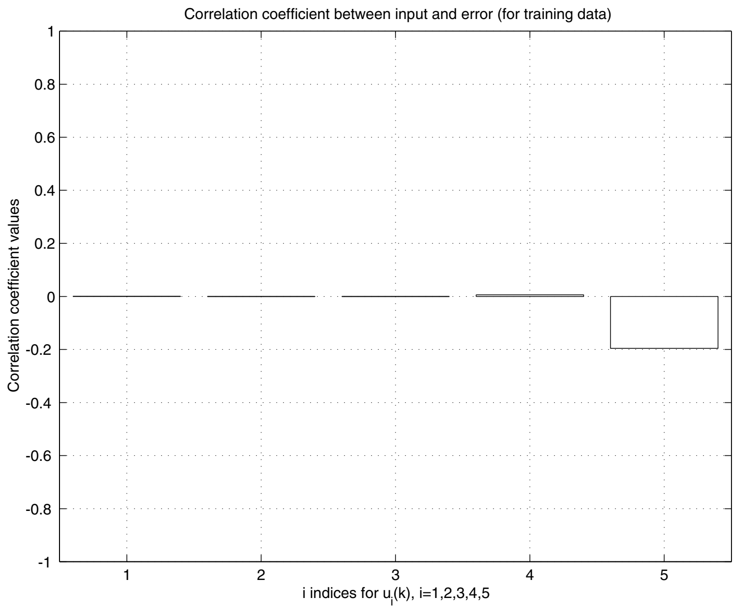

Figure 10.17: Correlation coefficients between each input, $u_i(k)$, $i = 1, 2, 3, 4, 5$, and the output approximation error, using the training data (one input to the premise membership functions and three to the consequent functions).

Second Attempt: A Good Controller

Notice that we have not improved the approximation accuracy over the previous cases by using a nonlinear approximator. How do we improve the accuracy? We could certainly try to improve the accuracy by tuning the premise membership function parameters or adding more rules (but we are limited in this last approach by the small amount of training data). Another approach would be to use more inputs to the premise membership functions. Recall from the past section that if we add such inputs, we can quickly increase the number of rules and hence the number of parameters in the approximator; therefore, we only add one more input, $u_3(k)$. This approach may make sense since then it will provide for a nonlinear map between two variables that the operator seems to be using in decision-making.

In this case, we grid the membership functions on the input space and get the results shown in Figure 10.18. For this, if $F(x, \theta)$ is the Takagi-Sugeno fuzzy system approximator mapping with θ chosen using batch least squares, we get a mean squared error at the training data of

$$\frac{1}{M}\sum_{(x,y)\in G}((y-F(x,\theta))^2 = 4.0401\times 10^3$$

and we get a mean squared error at the test data of

$$\frac{1}{M_\Gamma}\sum_{(x,y)\in \Gamma}((y-F(x,\theta))^2 = 3.1657\times 10^3$$

which is significantly better than any of the controllers that we have constructed so far.

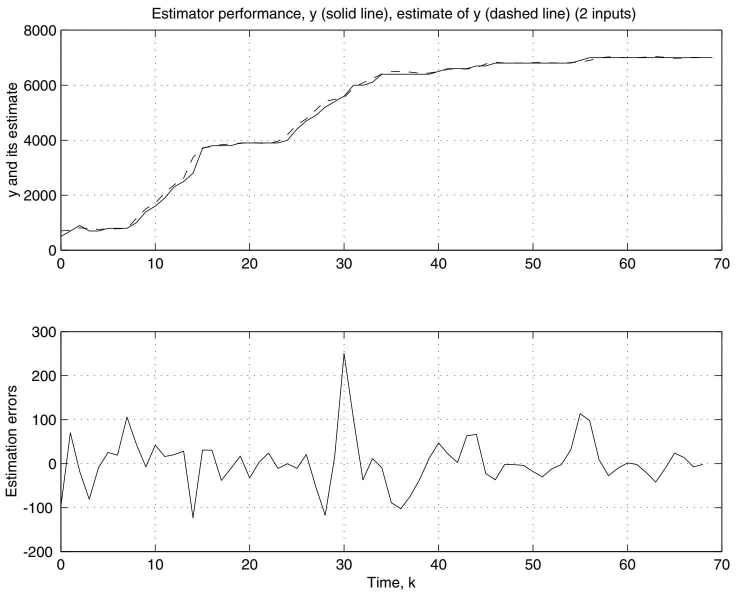

Figure 10.18: Operator settings, fuzzy controller settings, and error between these (u_1 and u_3 inputs to the premise membership functions and consequent functions).

Before we continue with the design process for the approximator, consider Figure 10.19 where, using ideas from [343], we see that by including the $u_1(k)$ and $u_3(k)$ inputs in producing the result in Figure 10.18, they are uncorrelated with the estimation error. Notice also that the $u_2(k)$, $u_4(k)$ and $u_5(k)$ inputs are only a bit correlated with the approximation error so that we guess that if we add these inputs, they probably would not help much with approximation accuracy.

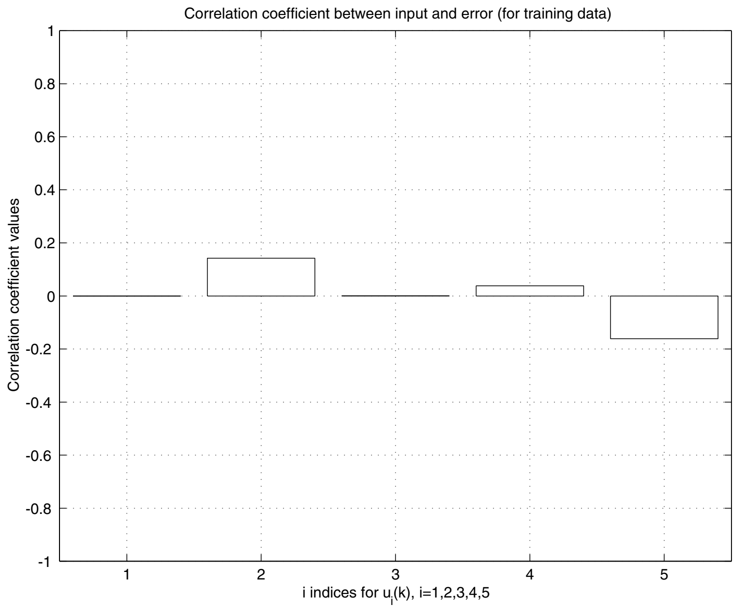

Figure 10.19: Correlation coefficients between each input, $u_i(k)$, $i = 1, 2, 3, 4, 5$, and the output approximation error, using the training data (one input to the premise membership functions and three to the consequent functions).

Third Attempt: No, Nothing Better

At this point, we return to our original correlation analysis and evaluate if there are other possibilities for improving on the performance. Recall that the analysis indicated that if we kept u_1 as an input, then we may not need to keep u_3 since these are correlated quite strongly. Earlier, in the construction of the linear estimators, we found that we could eliminate all the variables but these and we would still do pretty well, and that if we eliminated u_3, and hence only used u_1, there was a significant decrease in performance. This led us to conclude that there must be some useful information in the u_3 variable.

There is, however, a different line of reasoning that can be used. Note that u_1 is certainly a useful variable and so suppose that we use it as an input. Then, based on constraints due to approximator complexity in relation to the size of the training data set, and the problems we encountered in overfitting, we could try to pick a second variable that has the highest correlation with the output, but the lowest correlation with u_1. Which variable is this? Recall that u_2 had the third highest correlation with the output (a value of 0.3343), but it has correlation with u_1 of -0.1906. Note that u_4 has a relatively low correlation with the output of only 0.1012 (the u_5 variable had a correlation of -0.2068) but a correlation with u_1 of only -0.0283 (and the u_5 variable had a correlation with u_1 of 0.0837). To take a different approach, suppose that due to the low correlation between u_4 and u_1, we try to use u_4 as an input (u_5 may also work).

We will simply take the same approach as in our "second attempt," but we will use u_4 as an input, rather than u_3. We try different numbers of inputs, but do not find better results than earlier. Hence, while the line of reasoning above made sense, it did not end up helping to improve approximator accuracy for this approximator, and this training strategy. It may have been a good approach if we had used a different approximator or different training method. Why even discuss a case that does not work well? It helps to illustrate the normal process that you encounter in a real-world problem. Generally, you need to establish a logical approach to construction or improvement of an approximator, and try out the approach. It may or may not work better than your previous approach. Sometimes you win, sometimes you lose!

10.3.5 Methods to Test Generalization/Extrapolation and Controller Validity

While the best Takagi-Sugeno fuzzy system (as measured by the mean squared error) that we constructed in the last subsection gave a good approximation error, it could be that in between the training and testing data there are large excursions in the fuzzy controller mapping that intuitively may not be reasonable interpolations. Similarly, there could be large excursions in certain regions where there is a need to extrapolate since there is not good data in those regions. For low dimensional cases, it is possible to test the validity of the approximator by visually inspecting the approximator mapping (or perhaps a few dimensions can be studied at a time). Another alternative is to analytically determine the maximum slope of the approximator mapping. Or, as an approximation, you could numerically determine the maximum slope of the function on a fine grid of input data. In particular, in this approach, you would numerically compute an approximation to

$$\frac{\partial F(x,\theta)}{\partial x}$$

for the value of θ that was used in the approximator. You may want to study the data and analyze two different cases, one where the x is in a region where we had training data (to test generalization), and the other where x is in a region where there were no training data (to test extrapolation).

It is important to note that often an integral part of the validation of the controller will be to consult the operator and ask if it is reasonable. To do this, it may be convenient to convert the $R = 9$ rules that were trained into a type of linguistic equivalent. To do this, you could first assign linguistic values to the input membership functions that were specified for the Takagi-Sugeno fuzzy system. Now, if you used a standard fuzzy system you could assign linguistics to the output membership functions; then the rules would be simple to explain to the operator to get their "approval" of the rules. When we use a Takagi-Sugeno fuzzy system, you need to discuss this with the operator as being a "smooth switching" between the use of different linear (affine) functions of the inputs. See more details in [498].

10.4 Recursive Least Squares

While the batch least squares approach has proven to be very successful for a variety of applications, it is by its very nature a "batch" approach (i.e., all the data are gathered, then processing is done). For small M, we could clearly repeat the batch calculation for increasingly more data as they are gathered, but the computations can become prohibitive due to the computation of the inverse of $\Phi^\top\Phi$ and due to the fact that the dimensions of Φ and Y depend on M. Next, we derive a recursive version of the batch least squares method that will allow us to update our θ estimate each time we get a new data pair, without using all the old data in the computation and without having to compute the inverse of $\Phi^\top\Phi$. This "recursive least squares" approach allows us to implement an online function approximator, as we will illustrate in our examples in the next section.

The recursive least squares method can be used to tune the approximator parameters that enter linearly, with adjustments to the approximator mapping made online as each new training data pair is obtained.

10.4.1 Recursive Least Squares Derivation

Since we will be considering successively increasing the size of G, and we will assume that we increase the size by one at each time step, we assume that $(x(i), y(i))$ as gathered at time $k = i$. At time $k = 0$ we have no data. Suppose that $k = M$ so that you have gathered M pieces of training data and for $k \geq 1$, let the $p \times p$ matrix

$$P(k) = (\Phi^\top\Phi)^{-1} = \left(\sum_{i=1}^{k} \phi(x(i))\phi^\top(x(i))\right)^{-1} \tag{10.6}$$

($P(k)$ is called the "covariance matrix"). We will define $P(0)$ when we explain how to initialize the recursive least squares algorithm. Assume that $\Phi^\top\Phi$ is nonsingular for all k. We have

$$P^{-1}(k) = \Phi^\top\Phi = \sum_{i=1}^{k} \phi(x(i))\phi^\top(x(i))$$

so we can pull the last term from the summation to get

$$P^{-1}(k) = \sum_{i=1}^{k-1} \phi(x(i))\phi^\top(x(i)) + \phi(x(k))\phi^\top(x(k))$$

and hence

$$P^{-1}(k) = P^{-1}(k-1) + \phi(x(k))\phi^\top(x(k)) \tag{10.7}$$

Now, the least squares estimate for k pieces of training data is $\theta(k)$, which using Equation (10.2), is

$$\theta(k) \quad = \quad (\Phi^\top\Phi)^{-1}\Phi^\top Y$$

$$\begin{aligned}
&= \left(\sum_{i=1}^{k} \phi(x(i))\phi^\top(x(i))\right)^{-1} \left(\sum_{i=1}^{k} \phi(x(i))y(i)\right) \\
&= P(k)\left(\sum_{i=1}^{k} \phi(x(i))y(i)\right) \\
&= P(k)\left(\sum_{i=1}^{k-1} \phi(x(i))y(i) + \phi(x(k))y(k)\right) \qquad (10.8)
\end{aligned}$$

Hence, from the second to last equation, if we shift the time index back one, we have

$$\theta(k-1) = P(k-1)\sum_{i=1}^{k-1} \phi(x(i))y(i)$$

If we multiply both sides of this equation by $P^{-1}(k-1)$, we get

$$P^{-1}(k-1)\theta(k-1) = \sum_{i=1}^{k-1} \phi(x(i))y(i)$$

Now, replacing $P^{-1}(k-1)$ in this equation with the result in Equation (10.7), we get

$$(P^{-1}(k) - \phi(x(k))\phi^\top(x(k)))\theta(k-1) = \sum_{i=1}^{k-1} \phi(x(i))y(i)$$

Using the result from Equation (10.8), this gives us

$$\begin{aligned}
\theta(k) &= P(k)(P^{-1}(k) - \phi(x(k))\phi^\top(x(k)))\theta(k-1) + P(k)\phi(x(k))y(k) \\
&= \theta(k-1) - P(k)\phi(x(k))\phi^\top(x(k))\theta(k-1) + P(k)\phi(x(k))y(k) \\
&= \theta(k-1) + P(k)\phi(x(k))(y(k) - \phi^\top(x(k))\theta(k-1)). \qquad (10.9)
\end{aligned}$$

This provides a method to compute an estimate of the parameters $\theta(k)$ at each time step k from the past estimate $\theta(k-1)$ and the latest data pair that we received, $(x(k), y(k))$. Notice that $(y(k) - \phi^\top(x(k))\theta(k-1))$ is the error in predicting $y(k)$ using $\theta(k-1)$.

To update θ in Equation (10.9), we need $P(k)$, so we could use

$$P^{-1}(k) = P^{-1}(k-1) + \phi(x(k))\phi^\top(x(k)) \qquad (10.10)$$

But then we will have to compute an inverse of a matrix at each time step (i.e., each time we get another data pair $(x(k), y(k))$). Clearly, this is not desirable for real time implementation, so we would like to avoid this. To do so, recall that the "matrix inversion lemma" indicates that if A, C, and $(C^{-1} + DA^{-1}B)$ are nonsingular square matrices, then $A + BCD$ is invertible and

$$(A + BCD)^{-1} = A^{-1} - A^{-1}B(C^{-1} + DA^{-1}B)^{-1}DA^{-1}$$

We will use this fact to remove the need to compute the inverse of $P^{-1}(k)$ that comes from Equation (10.10) so that it can be used in Equation (10.9) to update θ. Notice that

$$\begin{aligned}
P(k) &= (\Phi^\top(k)\Phi(k))^{-1} \\
&= (\Phi^\top(k-1)\Phi(k-1) + \phi(x(k))\phi^\top(x(k)))^{-1} \\
&= (P^{-1}(k-1) + \phi(x(k))\phi^\top(x(k)))^{-1}
\end{aligned}$$

and that if we use the matrix inversion lemma with $A = P^{-1}(k-1)$, $B = \phi(x(k))$, $C = I$, and $D = \phi^\top(x(k))$, we get

$$\begin{aligned}
&P(k) = P(k-1) - \\
&P(k-1)\phi(x(k))(1 + \phi^\top(x(k))P(k-1)\phi(x(k)))^{-1}\phi^\top(x(k))P(k-1)
\end{aligned} \tag{10.11}$$

Now, using the fact that

$$P(k)\phi(x(k)) = \frac{P(k-1)\phi(x(k))}{1 + \phi^\top(x(k))P(k-1)\phi(x(k))}$$

(to see this, substitute $P(k)$ from Equation (10.11) into $P(k)\phi(x(k))$ on the left side of this equation). This gives us

$$\begin{aligned}
\theta(k) &= \theta(k-1) + K(k)\left(y(k) - \phi^\top(x(k))\theta(k-1)\right) \\
K(k) &= \frac{P(k-1)\phi(x(k))}{1 + \phi^\top(x(k))P(k-1)\phi(x(k))} \\
P(k) &= \left(I - K(k)\phi^\top(x(k))\right)P(k-1)
\end{aligned} \tag{10.12}$$

which is called the "recursive least squares (RLS) algorithm." Basically, the matrix inversion lemma turns a matrix inversion into the inversion of a scalar (i.e., the term $(1 + \phi^\top(x(k))P(k-1)\phi(x(k)))^{-1}$ is a scalar). Note that $K(k)$ is sometimes viewed as a time-varying gain on the prediction error $y(k) - \phi^\top(x(k))\theta(k-1)$ that dictates how $\theta(k-1)$ is adjusted to get $\theta(k)$.

We need to initialize the RLS algorithm (i.e., choose $\theta(0)$ and $P(0)$). One approach to do this is to use $\theta(0) = 0$ and $P(0) = P_0$ where $P_0 = \alpha I$ for some large $\alpha > 0$. This is the choice that is often used in practice. Other times, you may pick $P(0) = P_0$, but choose $\theta(0)$ to be the best guess that you have at what the parameter values are.

10.4.2 Weighted Recursive Least Squares: Using a Forgetting Factor

There is a "weighted recursive least squares" (WRLS) algorithm also. Suppose that the parameters of the physical system vary slowly. In this case, it may be advantageous to minimize

$$J(\theta, G)|_{M=k} = \frac{1}{2}\sum_{i=1}^{k} \lambda^{k-i}\left(y(i) - \phi^\top(x(i))\theta\right)^2$$

where $0 < \lambda \leq 1$ is called a "forgetting factor" since it gives the more recent data higher weight in the optimization (to see this, consider the effect of the term λ^{k-i} in the above summation). See the discussion on weighted batch least squares in the previous section.

Using a similar approach to the RLS case, you can show that the equations for WRLS are given by

$$\begin{aligned}
\theta(k) &= \theta(k-1) + K(k)\big(y(k) - \phi^\top(x(k))\theta(k-1)\big) \qquad (10.13)\\
K(k) &= \frac{P(k-1)\phi(x(k))}{\lambda + \phi^\top(x(k))P(k-1)\phi(x(k))}\\
P(k) &= \frac{1}{\lambda}\left(I - K(k)\phi^\top(x(k))\right)P(k-1)
\end{aligned}$$

(where when $\lambda = 1$, we get the equation for standard RLS given above).

10.4.3 Numerical Issues and Covariance Modifications

Briefly, we note that for practical problems, you can have numerical problems with the computation of the (weighted) recursive least squares update algorithm. One solution to this problem is to employ the "factorization" methods that are highlighted in the "For Further Study" section at the end of this part. Here, we will discuss other issues that arise in the use of the recursive least squares method.

RLS with Covariance Resetting

One particular problem that has been encountered in implementations of the online recursive least squares algorithm (e.g., in adaptive control) is when the matrix $P(k)$ has elements that become too small so that $P^{-1}(k)$ is difficult to compute. This can occur, for instance, when you do not get data that gives sufficient information about the underlying unknown mapping. To be more concrete, notice that for RLS ($\lambda = 1$), we had derived in Equation (10.7) that

$$P^{-1}(k) = P^{-1}(k-1) + \phi(x(k))\phi^\top(x(k))$$

Notice that $\phi(x(k))\phi^\top(x(k))$ is a $p \times p$ matrix with squared terms so that they are always positive. From this it is easy to see that it is possible that the elements of $P^{-1}(k)$ can grow unbounded.

In this situation, sometimes a "covariance resetting method" is used where at each time instant k, you check to see if

$$\lambda_{min}(P(k)) < \delta_1$$

(where $\lambda_{min}(P(k))$ is the minimum eigenvalue of $P(k)$) for some fixed $\delta_1 > 0$ and if it is, then you let

$$P(k+1) = \delta_2 I$$

where I is the identity matrix and $\delta_2 \geq \delta_1$ (e.g., you could choose $\delta_2 = \alpha > \delta_1$ where α was used to initialize $P(0)$). This ensures that we keep the minimum eigenvalue of $P(k)$ above some fixed value so that $P(k)$ is positive definite so the inverse $P^{-1}(k)$ exists (i.e., it is bounded). Now, this modification results in a method that is not a pure least squares method (of course, between the resets, it is); however, in adaptive control it is often found to be adequate to maintain certain desirable closed-loop properties.

WRLS with Covariance Modification

It is interesting to note that for the WRLS method where $0 < \lambda < 1$, we must proceed differently. In this case, you can show that

$$P^{-1}(k) = \lambda P^{-1}(k-1) + \phi(x(k))\phi^\top(x(k))$$

(indeed, this is one step in the derivation of the WRLS formula in Equation (10.13)) so that $P^{-1}(k)$ will stay bounded (you can think of the above equation as a stable discrete time system with a bounded input so long as $\phi(x(k))$ is bounded, which it often is simply by the choice of the structure of the approximator). However, in this case, $P(k)$ may have elements that grow without bound. To see this, recall that we had derived

$$P(k) = \frac{1}{\lambda}P(k-1) - \frac{1}{\lambda}\frac{P(k-1)\phi(x(k))\phi^\top(x(k))P(k-1)}{\lambda + \phi^\top(x(k))P(k-1)\phi(x(k))}$$

Notice that while

$$\lambda + \phi^\top(x(k))P(k-1)\phi(x(k)) > 0$$

since $\lambda > 0$, it could be that

$$P(k-1)\phi(x(k))\phi^\top(x(k))P(k-1) = 0$$

Now, since $\frac{1}{\lambda} > 1$, it is possible that elements of $P(k)$ can grow without bound.

To avoid this, we can modify the WRLS algorithm. To do this, we use Equation (10.13) as an update formula for $P(k)$ so long as

$$||P(k)||_2 \leq \delta_1$$

for some $\delta_1 > 0$ such that

$$||P(0)||_2 < \delta_1$$

If, however, at some time k,

$$||P(k)||_2 > \delta_1$$

we simply update $P(k)$ by letting $P(k) = P(k-1)$ (i.e., rather than using Equation (10.13)). This will ensure that all the elements of $P(k)$ will stay bounded. Here, note that

$$||P(k)||_2 = \left[\lambda_{max}\left(P^\top(k)P(k)\right)\right]^{\frac{1}{2}}$$

where $\lambda_{max}\left(P^\top(k)P(k)\right)$ is the maximum eigenvalue of $P^\top(k)P(k)$. But since $P(k)$ is symmetric and $P(k) \geq 0$ (i.e., it is positive semidefinite), we know that

$$||P(k)||_2 = \lambda_{max}\left(P(k)\right)$$

so that all we need to implement the modification to WRLS is to test eigenvalues.

10.4.4 Example: Fitting a Line to Data

As an example of how recursive least squares can be used, suppose that we would like to use this method to fit a line to data that are generated by a time-varying function. Suppose that $n = 1$. In this case our parameterized linear approximator is

$$y = F_{lip}(x,\theta) = \theta^\top\phi(x) = \theta^\top[\phi_1(x), 1]^\top = \theta^\top[x_1, 1]^\top = \theta_1 x_1 + \theta_2 \qquad (10.14)$$

which is an equation for a line. Suppose that the unknown function (we need to explicitly provide it here for the sake of illustration)

$$y(k) = G(x(k), z(k)) = \sin(0.01k)x_1(k) + 1$$

where $x(k) = x_1(k)$ and $z(k)$ captures the time-varying nature of the function (e.g., we could say that $z(k) = k$). Notice that we can think of the $\sin(0.01k)$ as a time-varying slope and the 1 as the intercept of a time-varying line. We assume that we have examined the data and determined that there is some type of time-varying behavior, so we decide to try to use recursive least squares. We emphasize, however, that we assume that we do not know the explicit structure of the unknown function.

First, we must specify the input $x(k)$. Here, we simply choose $x(k)$ to be a random value that is uniformly distributed on $[-1, 1]$. Next, we pick $\theta(0) = [0, 0]^\top$ and $P(0) = \alpha I$ where $\alpha = 100$. Then we show the performance of the estimator in Figure 10.20 for the case where $\lambda = 1, 0.98, 0.95, 0.7$. First, notice that in the nonweighted case ($\lambda = 1$), the estimate θ_1 quickly converges to the true value of 1 but that the estimate θ_2 is quite poor. The reason for this is that even though the underlying system has a parameter that changes over time, the estimation algorithm does not forget what the old data told it about how to do a good estimate. On the other hand, in the case where $\lambda = 0.98$, which represents that we want to forget *some* old information, the estimator does much better. Continuing with the tuning in this manner, we see that we get successive improvements, until the case where $\lambda = 0.7$ and we get a very good estimate. In this case, we tuned λ so that we are forgetting enough old information about the underlying function so that we are listening enough to what new data are saying about how the underlying function is shaped. It is clear then that when you have slowly varying changes in the underlying function, you probably want a large λ (i.e., near 1), where if the underlying system changes quickly, you probably want a smaller value of λ so that the algorithm quickly forgets old information; however, generally you do not want to pick λ to be too small since then it will quickly forget old information and may not perform well.

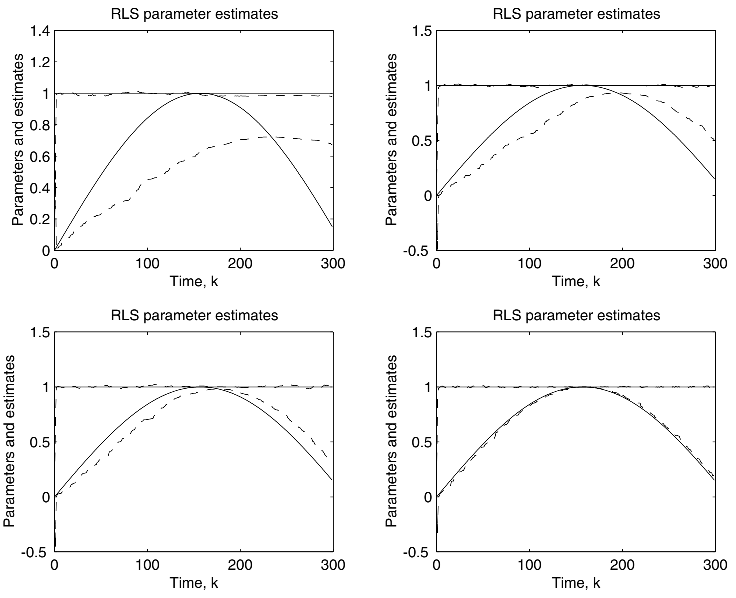

Figure 10.20: RLS parameter estimates, $\lambda = 1$ (upper left), $\lambda = 0.98$ (upper right), $\lambda = 0.95$ (lower left), $\lambda = 0.7$ (lower right).

10.5 Example: Online Tuning of Approximators

In this section, we continue with the examples considered in Section 10.2 where we considered the use of the batch least squares method to tune the approximator parameters.

10.5.1 Multilayer Perceptrons

Here, we consider the case where we had $n_1 = 25$ neurons and use all the same values of the biases and weights in the hidden layer that we developed in Section 10.2. Here, we use RLS to tune the neural network output layer weights and bias (these are stacked in the 26×1 vector θ). We will focus in particular on how the training data are presented to the algorithm and how this affects the accuracy of the approximator, how the forgetting factor affects the algorithm, and how initialization affects approximator accuracy.

Relatively Uniform Coverage of the Input Space

We will let the input x be uniformly distributed on $[-6, 6]$ and try to train the neural network to match the function in Figure 9.10. We let $\lambda = 1$ and initialize the algorithm with $\theta(0) = 0$ and $P(0) = \alpha I$ with $\alpha = 100$. To illustrate how the shape of the approximator nonlinearity evolves over time, we show the first 10 iterations of the RLS algorithm in Figure 10.21. Notice that for $k = 1$, *The performance of recursive least squares in constructing a mapping over the entire input space depends critically on the gathered data being properly spread over this space.*

we actually have two data points shown since we generated one data pair at $k = 0$ simply in case you wanted to also evaluate the estimation error of the approximator at time $k = 0$. At this time, however, only one data point has been used to tune the approximator. Notice that with only one data point, the estimator mapping is far from providing a good approximation of the mapping in Figure 9.10 (not surprising since with only one data pair, it knows little about its shape). Notice, however, that as k increases, we get more and more training data and our representation becomes more and more accurate. Here, $x(k)$ provides a relatively uniform coverage of the input space so even after only 10 iterations, we get a relatively good approximation.

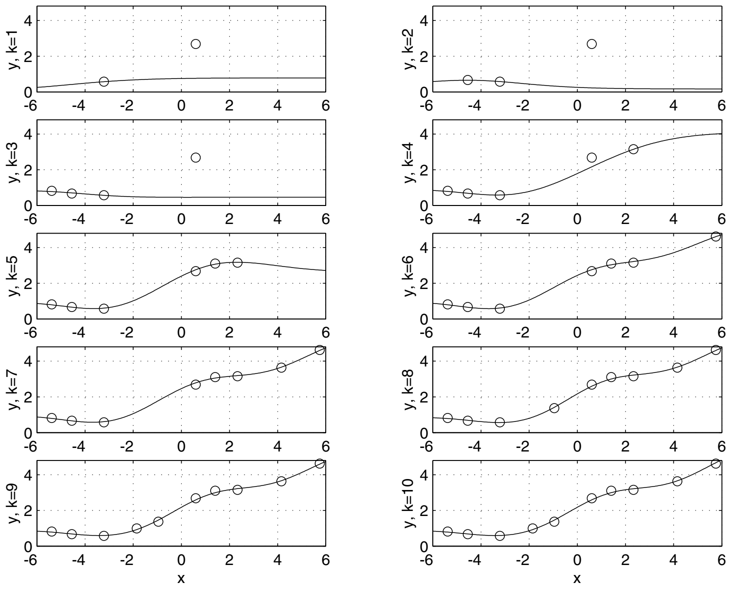

Figure 10.21: Neural network mappings generated using increasing amounts of training data ($k = 1$ to $k = 10$).

Nonuniform Coverage of the Input Space

Next, consider what happens when we do not get a good coverage of the input space. To do this, we simply run the program used to generate Figure 10.21 a few times, until by random chance it does not place any x data in one region of the input space. When this happens, we get the sequence of neural network mappings shown in Figure 10.22, where there are no input data in the region near $x = -6$ so we see that the approximation is poor in that region (until at $k = 10$, where it gets one more point and it improves the approximation). This is, of course, not surprising since, in this case the approximator is *extrapolating* for $k \leq 9$ so we cannot expect it to perform very well. It is also the case that if there are "holes" in the input space where there are no data, we will generally get poor approximation accuracy in that region. The general principle is that if you want to get good accuracy in any region, you need data to tell you what the shape of the function in that region is.

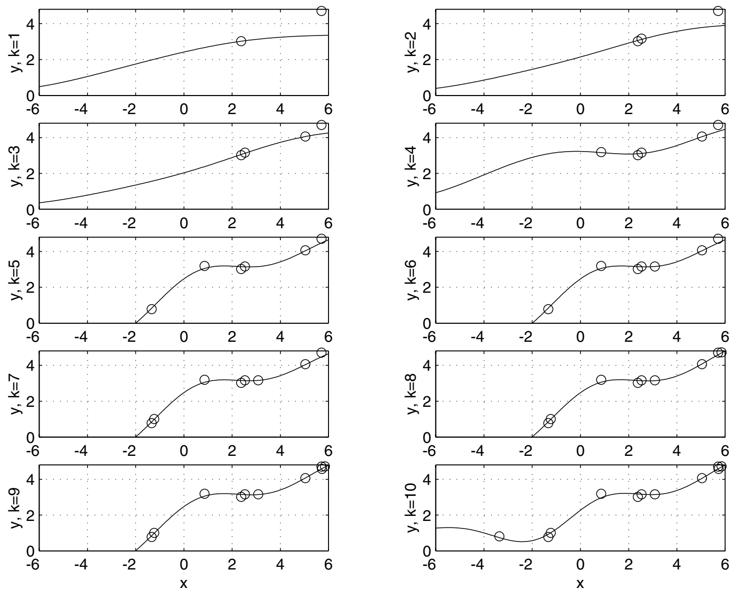

Figure 10.22: Neural network mappings generated using increasing amounts of training data (no data near $x = -6$ for $k \leq 9$).

Effects of Tuning the Forgetting Factor

In this case we will use a data set that has many input-output data pairs that are uniformly distributed across the input space. To do this, we will simply run the RLS algorithm for 300 steps, generating data at each of these steps (qualitatively, we obtain similar results if we use 1000 steps). To avoid showing 300 plots, we simply show how the last approximator performs compared to the training data. In particular, see Figure 10.23, and notice that this is a reasonably good approximation (we rely on random chance to get the uniform distribution of training data shown). Here, the approximator misses some of the structure inherent in the function, especially the "high frequency content" of the function (e.g., the peak at about $x = 1$ is considered to be an outlier since it is not encountered often). It seems to smooth out the approximator shape and does avoid being distracted by noise.

It is interesting to compare the accuracy of this approximator to that which was trained with batch least squares (see Figure 10.4). The one trained with batch least squares appears to be more accurate even though it was trained with

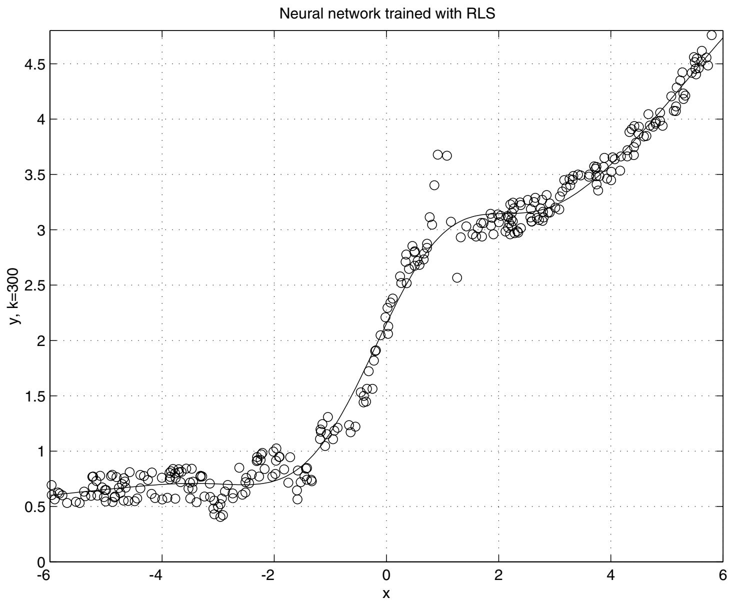

Figure 10.23: Neural network tuned with RLS for 300 steps.

fewer data (i.e., using less information about the unknown function). Notice, however, that the critical issue here is *how* the data were used. For batch least squares, all 121 data pairs were used at the same time to minimize the approximation error. For RLS, we provide the 300 data pairs in a sequence, and update the approximator each time we get a new data pair, placing equal importance on each piece of data obtained (i.e., we have $\lambda = 1$ here). Actually, by training with fewer steps (e.g., 121), you may get better overall accuracy (similar to that of the batch least squares).

Note, however, that by tuning λ, it is sometimes possible to get the approximator to do a better job at approximating the higher frequency content of the function. (Of course, another option is to increase the number of neurons in the hidden layer as we did in the batch least squares case.) For instance, if we let $\lambda = 0.95$, we get the plot shown in Figure 10.24, where we see that the approximator is trying to model the higher frequency behavior. Why? Well, the effect of $\lambda = 0.95$ is to place less significance on old data (i.e., data encountered for low values of k when we are at a higher value of k) so that different regions tend to become somewhat independent of each other so we can shape based on local data. However, this example is not to be overgeneralized. Sometimes you can pick λ to take on certain values and get disastrous results (where the approximator shape diverges from the shape it should have). The parameter λ offers the *potential* for performance improvements but cannot be guaranteed to provide these every time.

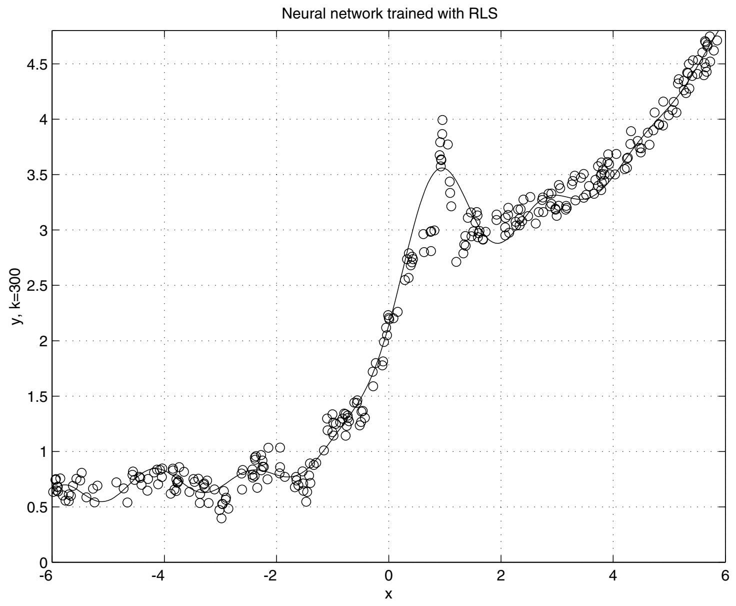

Figure 10.24: Neural network tuned with RLS for 300 steps, $\lambda = 0.95$.

Effects of Good Initialization of the Approximator

It should be intuitively clear that if we initialize, via choice of $\theta(0)$, the approximator close to the shape that it should ultimately be, that the RLS method should perform better. How do we initialize θ? One approach is to collect a set of training data and train with batch least squares first. If you do this for our example using the result of the batch least squares training studied in Section 10.2, then RLS ends up tuning the shape very little, even after 300 iterations (this is partly because the $(y(k) - \phi(x(k))^\top \theta(k-1))$ term in Equation (10.12), which is the error in predicting $y(k)$ using $\theta(k-1)$, is small for a good initialization, so the updates to $\theta(k)$ are small).

Generally, better initialization of the mapping results in higher quality approximators after training; however, training could decrease the quality of the initial guess.

We would like to show, however, that if you have a reasonably good initialization this can help the approximator accuracy, but that RLS also can improve on this accuracy by using more data to tune the approximator. So, how do we obtain a "reasonably good" initialization? One approach is to simply guess, but this can be very difficult when there are many parameters. Another approach is to use fewer training data in the batch least squares approach (in practical applications, this is typically the approach used). Here, however, simply for the sake of illustration, we will perturb the θ we had obtained from the batch least squares training by letting $\theta(0) = \theta + 0.25\theta$. In this case, we obtain the results shown in Figures 10.25 and 10.26. Notice that in Figure 10.25, the initial approximator shape is better than in, for instance, Figure 10.21, but that near $x = 6$, the approximation is not very good. As data are gathered, however, the RLS further improves the accuracy of the approximator and to see this, the final approximator is illustrated in Figure 10.26.

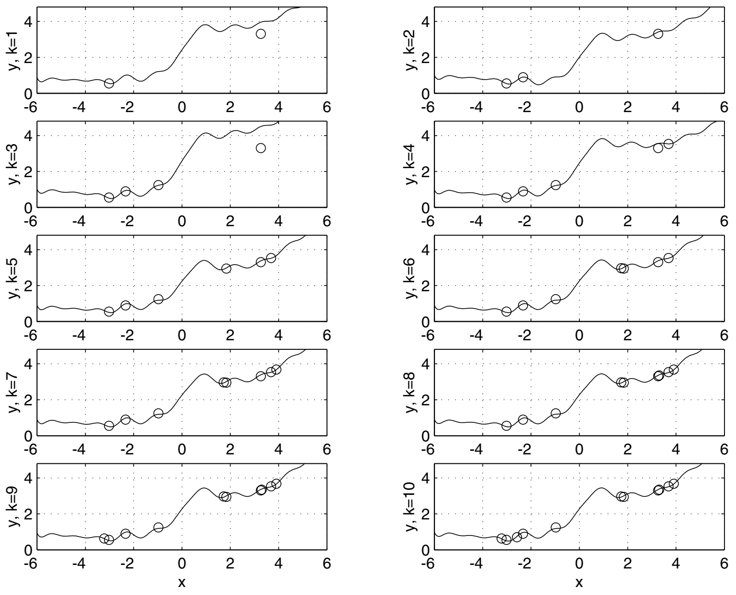

Figure 10.25: Neural network mappings generated using increasing amounts of training data (reasonably good initialization).

10.5.2 Takagi-Sugeno Fuzzy Systems

Here, we consider the case where we had $R = 20$ rules and use all the same values of the centers and spreads for the premise membership functions that we developed in Section 10.2. Here, we use RLS to tune the Takagi-Sugeno fuzzy system output function parameters (these are stacked in the 40×1 vector θ).

Input Space Coverage

Due to locality properties of the Takagi-Sugeno fuzzy system, it tends to adjust the mapping only where it gets data and tends not to destroy what it has learned in one area when making adjustments in another area.

In Figure 10.27 we show what happens in the first 10 steps of the algorithm for training the Takagi-Sugeno fuzzy system. Notice that in this case, we are initializing the parameter vector to be all zeros so that the mapping is zero initially. As it begins to get data, it shapes the mapping, but apparently in a very different way from what was done for the neural network (compare to Figure 10.21). Each time it gets another data point, it tries to pass the approximator mapping through that data point. In a sense, it trusts the initialization and does not adjust parameters to match in places where it does not know how to adjust. Of course, while this does not appear to be a good property in this example, for early steps of the approximator construction it can be quite beneficial since the approximator shape only changes locally.

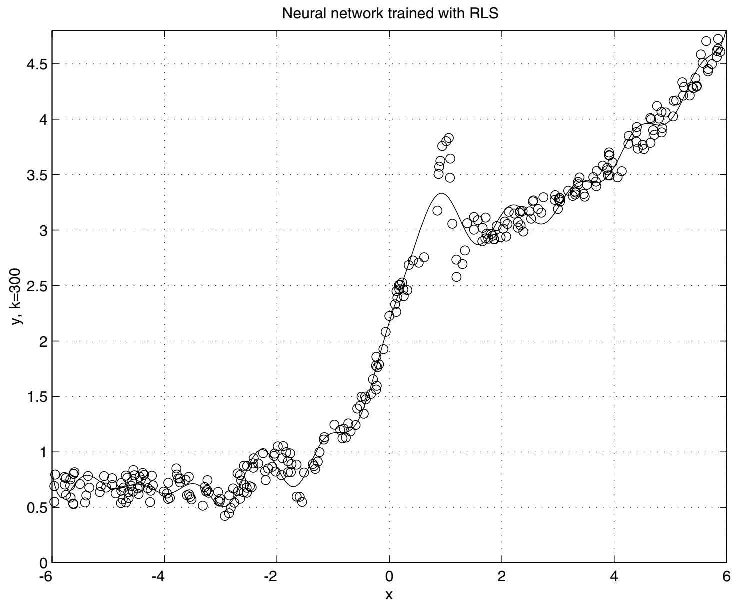

Figure 10.26: Neural network mappings generated using increasing amounts of training data, after 300 iterations.

Figure 10.27 dramatically illustrates issues of input space coverage. Bad coverage can clearly result in bad mappings, just as in the neural network case. Good coverage is obtained with more data. For example, if we use 300 pieces of training data, we get the mapping shown in Figure 10.28. The approximation error is somewhat lower compared to the neural network as you can see by comparing to Figure 10.23 (but do not draw any general conclusions from that). The tuning of the forgetting factor acts in a similar way (qualitatively) to that which was illustrated for the neural network.

Effects of Good Initialization of the Approximator

Here, we briefly illustrate the effects of using a good initialization for the parameters of the Takagi-Sugeno fuzzy system. We will follow the same approach as for the neural network in the last section. First, we show in Figure 10.29 what happens if you use the exact value of θ found by batch least squares. Notice that, as compared to Figure 10.27, the initial shapes are quite good due to the initialization (and if you run this for 300 steps, then it gets the result shown in Figure 10.30).

Next, we will perturb the good initialization by letting $\theta(0) = \theta + 0.25\theta$ where θ was found using the batch least squares method. The results for this case are shown in Figure 10.31. By studying the sequence of plots, you can see that as data are obtained in new regions, the approximator updates the shape

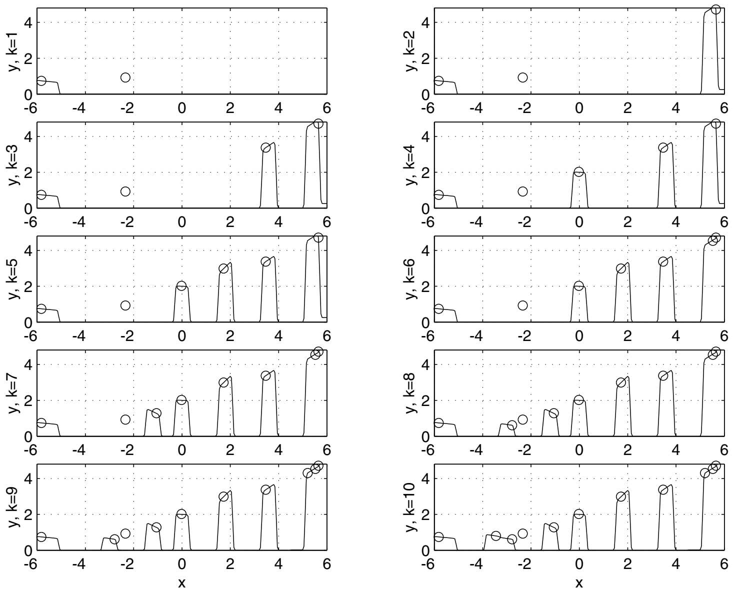

Figure 10.27: Takagi-Sugeno fuzzy system mappings generated using increasing amounts of training data ($k = 1$ to $k = 10$).

to more closely approximate the unknown function. In fact, after 300 iterations, the shape shown in Figure 10.32 is quite good and even similar to the case where a very good initialization was used. This shows how RLS can improve upon an initialization to provide good approximation accuracy.

10.6 Exercises and Design Problems

Exercise 10.1 (Batch Least Squares Derivation): Recall that for batch least squares, we had

$$J(\theta) = \frac{1}{2}E^\top E$$

(a) Using basic ideas from calculus, take the partial of J with respect to θ and set it equal to zero. From this, derive an equation for how to pick the least squares estimate. Compare it to Equation (10.2). Hint: If m and b are two $n \times 1$ vectors and A is an $n \times n$ symmetric matrix (i.e., $A = A^\top$), then $\frac{d}{dm} b^\top m = b$, $\frac{d}{dm} m^\top b = b$, and $\frac{d}{dm} m^\top A m = 2Am$.

(b) Repeat (a) for the weighted batch least squares approach and compare it to Equation (10.4).

Exercise 10.2 (Recursive Least Squares Derivation): In this problem you will derive the RLS method for two different cases.

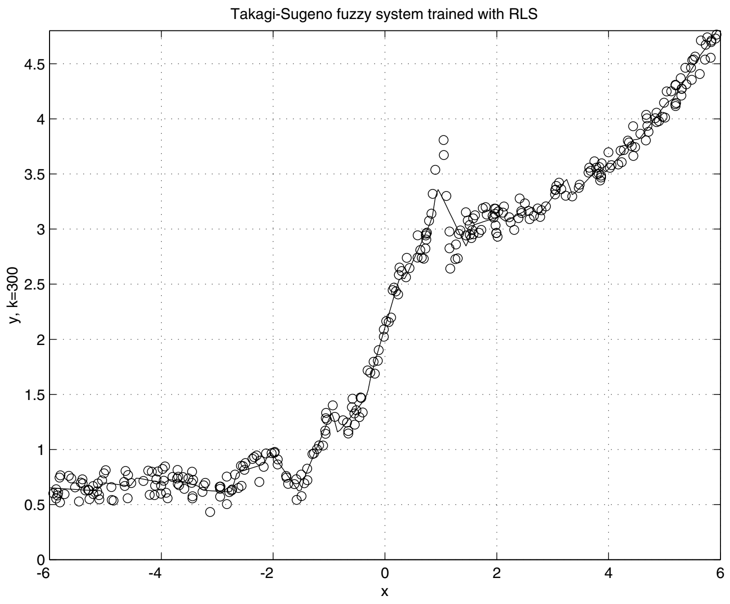

Figure 10.28: Takagi-Sugeno fuzzy system approximator mapping after 300 steps.

(a) Derive the update Equations (10.13) for the weighted recursive least squares approach (show every step of the derivations).

(b) In some applications we have a vector of measurements at each time instant (i.e., multiple measurements), rather than a single measurement. Derive the RLS equations for this case, and clearly identify the dimensions of all the matrices and vectors that you use.

Design Problem 10.1 (Gas Furnace CO_2 One-Step Ahead Prediction): See the Web page for the book and download the Box-Jenkins data for the gas furnace (it is in the form of a Matlab `.dat` file, but will download as a text file that you will remove the .txt extension from, before using in Matlab; of course, you can examine the file and easily use it in other programs).

(a) Develop an estimator using a batch least squares approach for $y(k)$, assuming that you use all the inputs to the estimator. You can think of this as a one-step ahead predictor since the estimate will depend on past inputs and outputs, in particular, $y(k-1)$, $y(k-2)$, $y(k-3)$, $y(k-4)$, $u(k-1)$, $u(k-2)$, $u(k-3)$, $u(k-4)$, $u(k-5)$, and $u(k-6)$. Use an affine approximator mapping (i.e. linear with bias). There are 290 data pairs in the "boxjenkins.dat" file. Use only the first 145 for training (i.e., $M = 145$). Use the last 145 for testing (i.e., $M_\Gamma = 145$). Provide values of the mean squared error for the approximator that

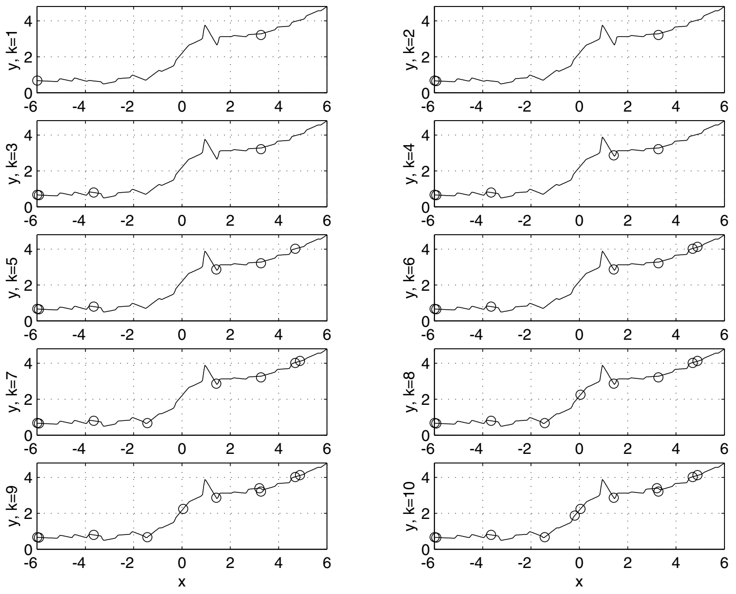

Figure 10.29: Takagi-Sugeno fuzzy system approximator mapping for the first 10 steps, good initialization.

you trained, both at the training data and the testing data. Plot $y(k)$ vs. k and the estimate of $y(k)$ vs. k on the same plot (use different line types for each) for $k = 1, 2, ..., 145, 146, ..., 290$, so that this plot will show how the estimator will perform at both the training and testing data (with the plots concatenated). Plot the error between the approximator estimate and the training data and testing data on the same type of plot (but on a different graph). Discuss the results.

(b) Suppose that you are constrained to only be able to use two inputs to your estimator. Pick the best two (with "best" defined by minimizing the mean squared error for the testing data). Moreover, specify a methodology for picking these. Use this methodology and repeat the process for three inputs. Compare, using the same plotting methodology and mean squared errors, to part (a).

(c) For (a), switch the training and test data. Repeat the design and testing of the estimator. Comment on the values of the mean squared error relative to the ones found in (a).

Design Problem 10.2 (Controller Construction from Process Operator Data):

In this problem you will follow the development in Section 10.3 to construct a controller using process operator data. (You can get the data at the Web site for the book.)

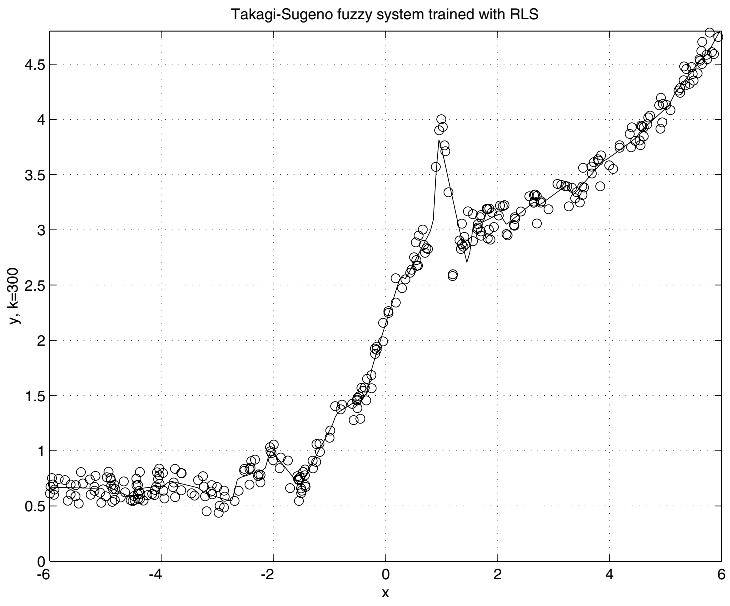

Figure 10.30: Takagi-Sugeno fuzzy system approximator mapping obtained after 300 iterations, good initialization.

(a) Verify the correlation analysis that was used to try to select inputs to the controller by reproducing Figure 10.12.

(b) Develop using a batch least squares approach linear (affine) controllers for the cases where you use all the inputs, 3 inputs, 2 inputs, and 1 input, as was done in the chapter. Verify the results there for the values obtained for the mean square errors in each case. In each case, produce the same types of plots to illustrate the performance of the estimator. Also, for each case, compute the correlation coefficients between the inputs and the estimation error and explain the resulting values using ideas from Section 10.3.

(c) Next, using ideas presented in Section 10.3, develop a fuzzy controller that obtains a lower mean squared error than you obtained for any case in (b). Produce the same types of plots as you did in (b) to illustrate the performance of the estimator and provide the value of the mean square error. You may use the same type of approach as given in the chapter (including the correlation analysis), or you may want to try to tune a neural network, or try a different tuning method. No matter which approach you take, be certain to pay attention to the number of parameters that you use in the approximator you are tuning. In fact, for each approximator structure you study, provide the number of parameters in the approximator and keep in mind that it is typically best to use the simplest approximator (where "simple"

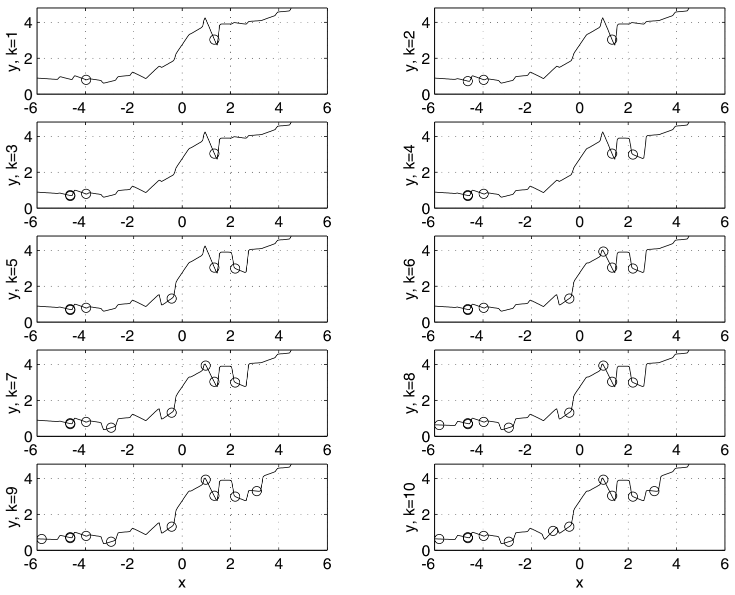

Figure 10.31: Takagi-Sugeno fuzzy system approximator mapping for the first 10 steps, reasonably good initialization.

may be measured by the size of the approximator, i.e., the number of parameters that are used in its definition).

(e) Optional: If you used a standard fuzzy controller in (c), develop linguistic rules from the ones that were constructed with data. You can pick linguistic values. Use these rules to explain some operating characteristics of the plant.

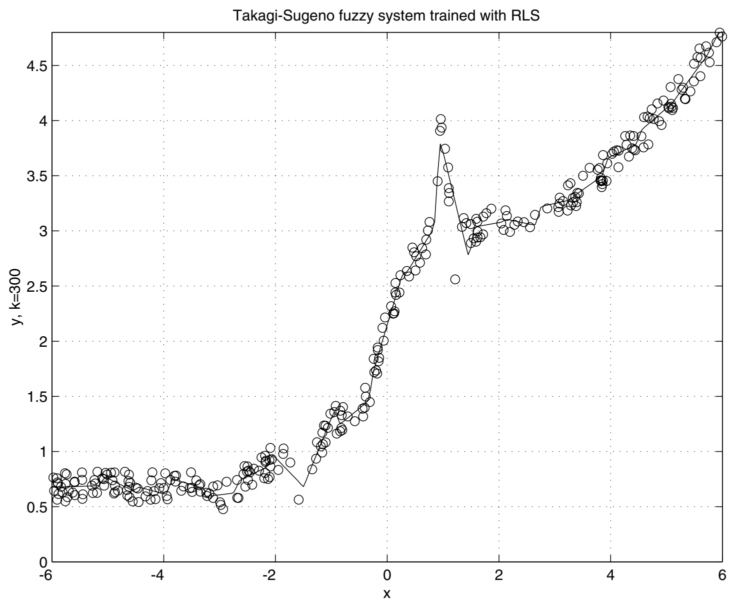

Figure 10.32: Takagi-Sugeno fuzzy system approximator mapping after 300 iterations, reasonably good initialization.