Chapter 11

Gradient Methods

Chapter Contents

Gradient techniques offer practical and effective methods to perform optimization. When applied to either the offline or online function approximation problems, they seek to find θ to minimize the function approximation error. The methods operate in an iterative fashion by successively improving on "guesses" (estimates) of the ideal parameter vector.

Gradient methods can be used in a batch or online manner to tune all parameters of the approximator. The basic approach is to iteratively adjust the parameters to minimize the approximation error.

Consider minimizing

$$J(\theta, G) = \frac{1}{2}\sum_{i=1}^{M} |y(i) - F(x(i), \theta)|^2 \tag{11.1}$$

by the choice of $\theta = [\theta_1, \theta_2, \ldots, \theta_p]^\top$ for a given training data set

$$G = \{(x(i), y(i)) : i = 1, 2, \ldots, M\}$$

(note that in several cases below, we will develop the theory for the case where $F(x(i), \theta)$ and $y(i)$ are $\bar{N} \times 1$ vectors so that it will be clear how to update multi-input multi-output approximators if you need to do that). In Equation (11.1), $|\bar{x}| = \sqrt{\bar{x}^\top \bar{x}}$ if $\bar{x}$ is an $\bar{N} \times 1$ vector. Clearly, if we can pick θ to minimize $J(\theta, G)$, we will have done a good job at function approximation, at least at the training data pairs in G (perhaps not at the test set Γ where $G \subset \Gamma$).

At the outset it is important to highlight the fact that, in general, the optimal solution to the function approximation problem is difficult to find. Why? Basically, the answer lies in the fact that $J(\theta, G)$ can be a very complex "landscape" with many hills and valleys, such as the one shown in Figure 11.1, which is also shown in Figure 11.2 as a contour map.

The methods may search for the global minimum of such a function (which in this case, by inspection, is at $(15, 5)$), but it can get distracted by the multiple local minima at other positions (e.g., at $(20, 15)$, which is easier to see on the contour plot), or the very flat regions of the surface where, if you only have a local view (i.e., not the perspective you have by looking at the plots from a bird's-eye view where you can see the peaks and valleys), it is difficult to know which direction to head to find the minimum. In fact, on such low slope portions of the surface, relatively large changes in the parameters make very little progress in minimizing the function (and such low slope regions are often found in function approximation problems where the approximator structure is "overparameterized," i.e., more approximator structure is used than is needed to get a low representation error and hence, large changes in some parameters may have little effect on the shape of the function and hence, the quality of the approximation).

Local minima arise since, in general, the cost function that characterizes approximation error is not convex. Getting trapped at a local minimum corresponds to not tuning the approximator in a way that could further reduce the approximation error.

In general, we will not know that we have converged to a global or local minima for the gradient methods studied here (except in special cases). The most we will be able to hope for is to converge to a "stationary point;" that is, a zero slope region of the surface such as a local minimum or a flat region on the landscape.

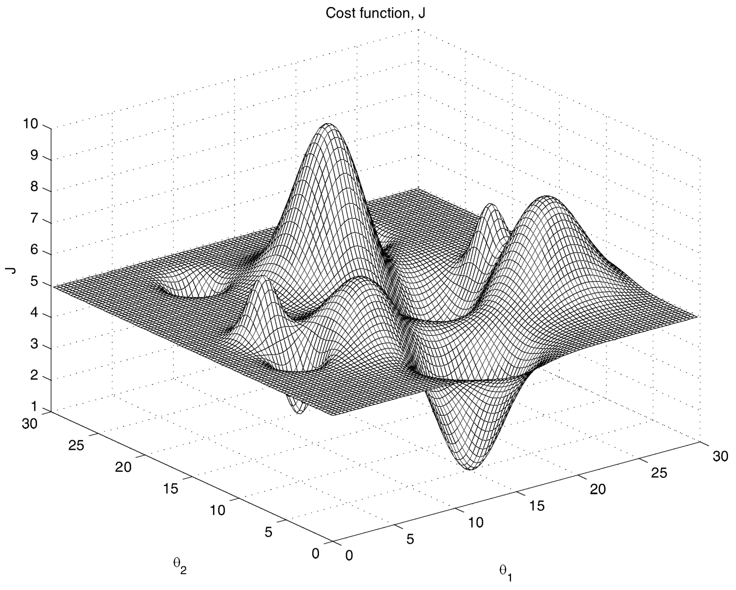

Figure 11.1: Candidate function for which we may seek to find the minimum.

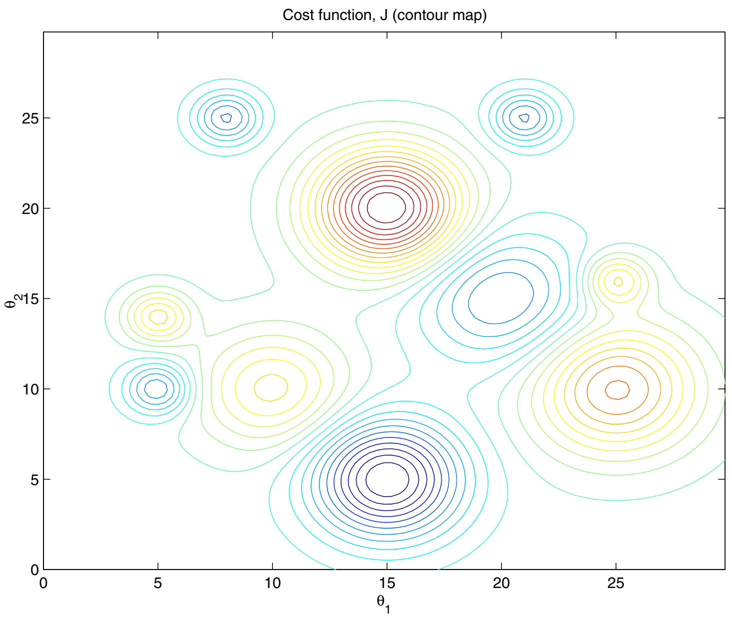

Figure 11.2: Candidate function for which we may seek to find the minimum (contour map).

11.1 The Steepest Descent Method

Let $\theta(j)$ be the current estimate of the parameter vector at iteration j (note that when we indexed θ with time we used k, but here j is used to emphasize that the index may simply be for the training data, not time).

11.1.1 Steepest Descent Parameter Updates

The basic form of the update using a gradient method to minimize the function $J(\theta, G)$ via the choice of $\theta(j)$, is

$$\theta(j+1) = \theta(j) + \lambda_j d(j) \tag{11.2}$$

where $d(j)$ is the $p \times 1$ "descent direction," and $\lambda_j > 0$ is a (scalar) positive "step size" that can depend on the iteration number j.

To see the rationale for the choice of this parameter update formula, consider the case where θ is a scalar and where we use the simple quadratic function

$$J(\theta, G) = \theta^2$$

in Figure 11.3, where we are searching for the point where the function reaches the minimum by picking the scalar θ. Name the point where the minimum is achieved θ^* and assume that it is unknown and that we want to find its value.

It is useful to think of gradient methods as "hill climbing" (here, climbing down hills).

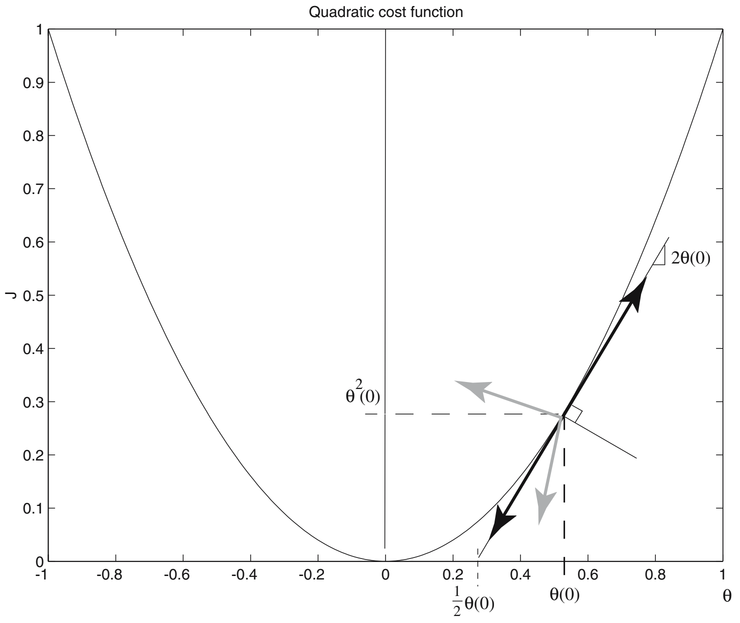

Figure 11.3: Scalar quadratic $J(\theta, G)$.

The update formula for this scalar case, where $\lambda_j = \lambda$ is a positive constant, is

$$\theta(j+1) = \theta(j) + \lambda d(j)$$

Notice that

$$d(j) = \frac{\theta(j+1) - \theta(j)}{\lambda} \tag{11.3}$$

With λ as a step size, we see that $d(j)$ is a descent direction in the sense that it is the direction in which the parameter is moving in order to try to minimize $J(\theta, G)$. What direction would we like this to be? We would like the parameter updates to always move in a direction that decreases $J(\theta, G)$ because, if it does this over a whole sequence of iterations (perhaps an infinite number of iterations), we may get $\theta(j) \to \theta^*$ as $j \to \infty$ (so $J(\theta^*, G) \leq J(\theta, G)$ for all other possible θ).

The above formula (Equation (11.3)) suggests that the direction should be the slope of the function $J(\theta, G)$ at $\theta = \theta(j)$. To see this, consider the example in Figure 11.3. Suppose that the initial (best) guess of θ^* is the $\theta(0)$ shown. Based on this guess, how would you next guess at θ? That is, how would you generate $\theta(1)$? Suppose that we can compute the slope (gradient) of $J(\theta, G)$ at $\theta(0)$. For our example, this gradient is

$$\left.\frac{\partial J(\theta, G)}{\partial \theta}\right|_{\theta=\theta(0)} = 2\theta|_{\theta=\theta(0)} = 2\theta(0) \tag{11.4}$$

and it is shown as the black arrow pointing up and to the right in Figure 11.3. The negative of this gradient is $-2\theta(0)$ and it is shown as the black arrow pointing down and to the left in Figure 11.3. These arrows indicate possible directions $d(j)$ to update the guess at $\theta(0)$. Clearly, to move *down* the function $J(\theta, G)$ to minimize it, one choice would be to use the direction

Steepest descent involves updating the parameters in a direction that appears to decrease the cost function the most.

$$d(j) = -\left.\frac{\partial J(\theta, G)}{\partial \theta}\right|_{\theta=\theta(j)} \tag{11.5}$$

(i.e., to move along the negative gradient). Intuitively, this choice is the "direction of steepest descent" (it corresponds to how a skier often moves down a snow-covered mountain) and hence, the parameter update formula for the steepest descent method, even for the p-dimensional case, is given by

$$\theta(j+1) = \theta(j) - \lambda_j \left.\frac{\partial J(\theta, G)}{\partial \theta}\right|_{\theta=\theta(j)} \tag{11.6}$$

11.1.2 Example: Convergence, Step Size, and Termination Issues

Using the hill climbing (or skiing) perspective, the step size indicates how big a step to move in the $d(j)$ direction. For instance, for $\lambda_j = \lambda = 0.1$ for all j, the update formula becomes

$$\begin{aligned}\theta(j+1) &= \theta(j) - 0.2\theta(j) \\ &= 0.8\theta(j)\end{aligned}$$

so clearly as $j \to \infty$, $\theta(j) \to \theta^* = 0$, the optimum point.

In optimization you do not know where the minimum point is to begin with, so you generally do not directly know if you are approaching it (or if you are at it), so it is difficult to know how to terminate the updates to $\theta(j)$. That is, we do not know when $\theta(j)$ has approached θ^* since θ^* is unknown. So, how do we terminate the gradient algorithm if we need a solution after a finite number of iterations? One way is to monitor

$$d(j) = -\left.\frac{\partial J(\theta, G)}{\partial \theta}\right|_{\theta=\theta(j)}$$

and if

$$d(j) = -\left.\frac{\partial J(\theta, G)}{\partial \theta}\right|_{\theta=\theta(j)} = 0$$

clearly Equation (11.6) will stop making changes to $\theta(j)$. In practice, we often simply terminate if

$$|d(j)| = \left| -\left.\frac{\partial J(\theta, G)}{\partial \theta}\right|_{\theta=\theta(j)} \right| \leq \epsilon$$

for some prechosen constant $\epsilon > 0$; however, there are other termination issues and methods that will be discussed later.

Next, it is important to further consider the effect of the choice of the step size, particularly on the convergence of the estimate to its true value. If you choose $\lambda_j = \lambda = 0.2$ for all j (i.e., double the step size compared to the choice above), the update formula becomes

$$\begin{aligned}\theta(j+1) &= \theta(j) - 0.4\theta(j) \\ &= 0.6\theta(j)\end{aligned}$$

so again as $j \to \infty$, $\theta(j) \to \theta^* = 0$, the optimum point. But notice that the "rate of convergence" is much faster in this case since this choice for λ corresponds to taking larger steps at each iteration. So this line of reasoning may lead one to believe that larger step sizes are generally better; this is not, however, the case. Notice that if you pick $\lambda_j = \lambda = 10$ for all j, then the update formula is

$$\begin{aligned}\theta(j+1) &= \theta(j) - 20\theta(j) \\ &= -19\theta(j)\end{aligned}$$

so that as j increases, the value of $\theta(j)$ oscillates between larger and larger values and does not converge (in terms of Figure 11.3 it climbs up the parabola). In this case, the step size is too big, so the algorithm is too aggressive and fails to converge to the minimum value of $J(\theta, G)$.

Generally, $\theta(j)$ may not have a limit point, it may diverge as in this example, or it may oscillate; however, gradient methods are generally able to find isolated stationary points (i.e., ones where you can draw a sphere around them and no other stationary points are in the sphere) if they start close to them (this is why initialization of the algorithms is so important). Sometimes, all we can hope to be able to show is that the parameters will remain bounded, and sometimes we can do this by appropriately choosing the step size so that at each iteration we are guaranteed to get descent. In this case, the parameter vector will belong to a bounded set and so it must have at least one limit point; however, it can be difficult to guarantee that the parameter vector will converge to a single limit point. Clearly, the direction of descent and step size are important parameters in the development of a gradient update formula, especially when $J(\theta, G)$ is very complex, as it often is in practical applications.

11.1.3 Descent Direction Possibilities and Momentum

The above simple example shows intuitively that the choice of

$$d(j) = -\left.\frac{\partial J(\theta, G)}{\partial \theta}\right|_{\theta=\theta(j)}$$

(i.e., the negative gradient) is the direction of steepest descent and that as long as an appropriate choice is made for the step size λ_j, the algorithm will converge for this example (since the function we are minimizing is convex and only has one minimum). There are, however, many other choices that can be made for the descent direction and these others can be useful in practical applications. (Indeed, in the case where $J(\theta, G)$ is quadratic, there are well-known methods for the solution to the optimization problem; in practical applications, it is most often not quadratic.) Notice that any direction $d(j)$ is a descent direction provided that the angle it makes with

Iterative update of the parameters in any direction that locally appears to decrease the cost results in viable gradient descent methods.

$$\left.\frac{\partial J(\theta, G)}{\partial \theta}\right|_{\theta=\theta(j)}$$

is more than 90°. Hence, the shaded arrows in Figure 11.3 are also descent directions for $\theta(0)$. The angle is greater than 90° if

$$\left(\frac{\partial J(\theta(j), G)}{\partial \theta(j)}\right)^{\top} d(j) < 0$$

As indicated, this formula also holds for the vector case. In the vector case, $d(j)$ is a $p \times 1$ vector and the gradient is a $p \times 1$ vector that is denoted by

$$\nabla J(\theta(j), G) = \frac{\partial J(\theta(j), G)}{\partial \theta(j)} = \begin{bmatrix} \frac{\partial J(\theta(j),G)}{\partial \theta_1(j)} \\ \frac{\partial J(\theta(j),G)}{\partial \theta_2(j)} \\ \vdots \\ \frac{\partial J(\theta(j),G)}{\partial \theta_p(j)} \end{bmatrix} \tag{11.7}$$

Clearly, the choice of

$$d(j) = -\left.\frac{\partial J(\theta, G)}{\partial \theta}\right|_{\theta=\theta(j)} = -\nabla J(\theta(j), G)$$

results in the satisfaction of this formula, but clearly many other choices do also.

One modification to the descent direction that has been found to be useful in some applications is to use a "momentum term." In this case the update formula is

$$\theta(j+1) = \theta(j) - \lambda_j \nabla J(\theta(j), G) + \beta\left(\theta(j) - \theta(j-1)\right) \qquad (11.8)$$

where $0 \leq \beta < 1$ is a fixed gain and $\beta\left(\theta(j) - \theta(j-1)\right)$ is the momentum term. Basically, momentum accelerates progress of the update where the gradients $\nabla J(\theta(j), G)$ are pointing in the same direction, but restricts update sizes when successive gradients are roughly opposite in direction. This can tend to damp oscillations in the parameter vector and keep the parameter vector moving in the proper direction.

In the following sections we will consider other choices for the descent direction, ones that can be particularly effective in practical applications (e.g., for tuning approximators that are not linear in the parameters). First, however, we will consider the application of the steepest descent method to the tuning of linear in the parameter approximators.

11.1.4 The Linear in the Parameter Case

For a linear in the parameter approximator $y = F(x, \theta) = \theta^\top \phi(x)$ we have, in the case where $\bar{N} = 1$,

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

and if we use the notation from Chapter 10, this is expressed as

$$d(j) = -\left.\frac{1}{2}\frac{\partial}{\partial \theta}E^\top E\right|_{\theta=\theta(j)}$$

where

$$E = [\epsilon_1, \epsilon_2, \ldots, \epsilon_M]^\top = Y - \Phi\theta$$

with $\epsilon(i) = y(i) - \theta^\top \phi(x(i))$. Now, if we follow the derivation of the batch least squares estimate, we find

$$\begin{aligned} d(j) &= -\left.\frac{1}{2}\frac{\partial}{\partial \theta}(Y - \Phi\theta)^\top (Y - \Phi\theta)\right|_{\theta=\theta(j)} \\ &= -\frac{1}{2}\frac{\partial}{\partial \theta}\left(Y^\top (I - \Phi(\Phi^\top \Phi)^{-1}\Phi^\top)Y + \right. \end{aligned}$$

$$\begin{aligned}
&\left.(\theta - (\Phi^\top\Phi)^{-1}\Phi^\top Y)^\top \Phi^\top\Phi(\theta - (\Phi^\top\Phi)^{-1}\Phi^\top Y))\right|_{\theta=\theta(j)} \\
= \; & -\frac{1}{2}\frac{\partial}{\partial\theta}\left.\left(\theta^\top\Phi^\top\Phi\theta - 2\theta^\top\Phi^\top\Phi(\Phi^\top\Phi)^{-1}\Phi^\top Y\right)\right|_{\theta=\theta(j)} \\
= \; & -\frac{1}{2}\frac{\partial}{\partial\theta}\left.\left(\theta^\top\Phi^\top\Phi\theta - 2\theta^\top\Phi^\top Y\right)\right|_{\theta=\theta(j)} \\
= \; & -\frac{1}{2}\left.\left(2\Phi^\top\Phi\theta - 2\Phi^\top Y\right)\right|_{\theta=\theta(j)} \\
= \; & \left.\Phi^\top(Y - \Phi\theta)\right|_{\theta=\theta(j)} \\
= \; & \left.\Phi^\top E\right|_{\theta=\theta(j)}
\end{aligned}$$

Suppose that $\lambda_k = \lambda > 0$ is a constant so that the steepest descent update formula for the linear in the parameters case is

$$\begin{aligned}
\theta(j+1) &= \theta(j) + \lambda\Phi^\top E = \theta(j) + \lambda\Phi^\top(Y - \Phi\theta(j)) \\
&= \theta(j) + \lambda\sum_{i=1}^{M}\phi(x(i))(y(i) - \theta^\top(j)\phi(x(i))) \qquad (11.9)\\
&= \lambda\left(I\frac{1}{\lambda} - \Phi^\top\Phi\right)\theta(j) + \lambda\Phi^\top Y
\end{aligned}$$

Now, if the steepest descent approach converges (and it will, assuming certain technical conditions are satisfied, for instance, having a diminishing step size), we get

$$\theta(j+1) \to \theta(j) = \theta_{sd}$$

as $j \to \infty$ (where we call θ_{sd} the value that the steepest descent converges to). In this case, we have

$$\theta_{sd} = \lambda\left(I\frac{1}{\lambda} - \Phi^\top\Phi\right)\theta_{sd} + \lambda\Phi^\top Y$$

Now, notice that

$$\begin{aligned}
\left(I - \lambda\left(I\frac{1}{\lambda} - \Phi^\top\Phi\right)\right)\theta_{sd} &= \lambda\Phi^\top Y \\
\lambda\Phi^\top\Phi\theta_{sd} &= \lambda\Phi^\top Y
\end{aligned}$$

so that if the inverse exists

$$\theta_{sd} = \left(\Phi^\top\Phi\right)^{-1}\Phi^\top Y$$

and we see that (if it converges) it converges to the least squares solution that we found in Equation (10.2). Notice, however, that this is an analysis of the *asymptotic* behavior of the estimate, and sometimes it is better simply to use an appropriate software package to compute the least squares estimate.

This provided a comparison to the batch least squares approach in the case where the batch of data is used in the steepest descent gradient method. What if, instead, we proceeded as in the recursive least squares case and processed the data sequentially? To do this, define

$$G_k = \{(x(k), y(k))\}$$

to be the data set at time k. In this case, since we make an iteration of the gradient method at each step, our update formula is

$$\theta(k+1) = \theta(k) + \lambda_k d(k)$$

(i.e., we replace j with k) and

$$d(k) = -\left.\frac{\partial J(\theta, G_k)}{\partial \theta}\right|_{\theta=\theta(k)}$$

so with $\lambda_k = \lambda = 1$ for all k, using Equation (11.9) (with $M = 1$ piece of data, the piece in G_k), we have

$$\theta(k+1) = \theta(k) + \phi(x(k))\left(y(k) - \theta^\top(k)\phi(x(k))\right)$$

Compare this to the update formula for recursive least squares given in Equation (10.12) that can be used for online parameter adjustment. Notice that the two are not the same. The update formula for recursive least squares has the extra $P(k+1)$ term that multiplies the second term in the above equation. Which is the better approach? In adaptive control problems, the recursive least squares approach tends to converge faster but you pay for this faster convergence by having to compute $P(k)$ and include it in the update formulas. Hence, sometimes for simplicity we may use a steepest descent gradient approach (sometimes, with certain modifications), even for the linear in the parameter case.

11.1.5 Step Size Choice

While in the last section (and in several subsequent ones) we focus on the selection of the descent direction $d(j)$, here we will consider the choice of the scalar step size λ_j. Clearly, while we will only discuss scalar step sizes, it is possible to have a diagonal matrix of step sizes, so that different parameters are updated at different rates.

Step size choice affects rate of convergence, how the algorithm copes with local minima, and asymptotic behavior of the algorithm.

Constant Step Size

For some applications (e.g., in adaptive control), a fixed step size $\lambda_j = \lambda$ for all j can be sufficient. Other times, it can be difficult to select λ. For instance, see the example in Section 11.1, where for a simple example, we showed that it is possible to pick it so that convergence is not achieved. Generally, as we saw in that example, if λ is too small, we get slow convergence; if it is too large, then we can get divergence. Indeed, in many problems a constant step size can result in “limit cycling” (oscillations in parameter values) near a local minimum since, when you are in a region near a local minimum, you must often take successively smaller steps to ensure that you do not “overshoot” the solution. Next, we discuss the case where a successively smaller step size is used.

Diminishing Step Size

In this case, the step size converges to zero as j goes to infinity, according to some formula. That is, we pick an algorithm for forcing

$$\lambda_j \to 0$$

as $j \to \infty$. For instance, we may choose

$$\lambda_j = \frac{\lambda}{j+1}$$

where $\lambda > 0$ is a constant or we could pick

$$\lambda_j = e^{-\alpha j}$$

for some $\alpha > 0$. While these rules can be simple to implement, in some cases λ_j may be chosen so that it goes to zero too fast so that the algorithm slows prematurely, before it gets near a solution. It is for this reason that often it is required that

$$\sum_{j=0}^{\infty} \lambda_j = \infty$$

which, in effect, forces the step size to persistently update the parameters (provided the gradient is sufficiently large).

Another way that a diminishing step size is sometimes implemented in practical applications is to let

$$\lambda_j = \max\left\{\lambda_{min}, \frac{\lambda_{max}}{1+\alpha j}\right\}$$

where at $j = 0$ we have $\lambda_0 = \lambda_{max} > 0$ and as j increases λ_j decreases, with rate governed by the choice of $\alpha > 0$, to $\lambda_{min} > 0$. All these parameters would need to be tuned for a particular application. Generally, this approach offers a big step size early in the processing and the step size diminishes as time goes on, but no lower than the value λ_{min}. This ensures that the step size will not get too small so that updates are persistently made (but of course in this case, we do not get $\lambda_j \to 0$ as $j \to \infty$).

Regardless, the general problem with a diminishing step size approach for practical applications is that it often slows convergence too much, or does not provide a fast enough update early in the processing. Due to this, tuning of the step size rule is normally needed. This is why in some practical problems, many have turned to line minimization approaches and the Armijo step size rule that we discuss next.

Line Minimization Approaches

For this, pick a scalar $\lambda^0 > 0$ that is the largest step size you think is reasonable for the problem at hand (sometimes information from the problem domain can suggest a choice for λ^0, while other times you must guess at it and experiment with the performance of the algorithm to get a good choice). Then, you pick λ_j so that

$$J(\theta(j) + \lambda_j d(j), G) = \min_{\lambda \in [0, \lambda^0]} J(\theta(j) + \lambda d(j), G)$$

This is simply a one-dimensional "line" minimization problem. The resulting value of λ_j yields the greatest reduction in the cost function over all step sizes such that $\lambda_j \geq 0$ (to keep the updates moving in the proper direction) and $\lambda_j \leq \lambda^0$ as it is specified above. "Line search," or what are sometimes called "line minimization" approaches, are used to solve this problem.

There are many line minimization methods. Some, like Newton's approach, require the use of second derivatives, while methods like the "secant method" only require the use of first derivatives. Other approaches use interpolation with candidate points generated in $[0, \lambda^0]$, or "golden section search," which is an interval reduction method. See the "For Further Study" section for more details.

Armijo Step Size Rule

In practice it is often the case that the line minimization approaches are computationally expensive, so methods that are based on "successive step size reduction" are often used. One popular method of this type is the Armijo step size rule.

In this rule, first pick the following scalars:

1. λ^0, an initial guess at the size of the step (often, for applications that are properly "scaled" you can pick $\lambda^0 = 1$).
2. γ, $0 < \gamma < 1$, a "reduction factor" (often, for applications, $\frac{1}{10} \leq \gamma \leq \frac{1}{2}$).
3. σ, $0 < \sigma < 1$, a scale factor (often, for applications, $0.00001 \leq \sigma \leq 0.1$).

While this provides guidelines for the choice of these parameters, they may need to be tuned. The Armijo step size rule is actually an iterative process for finding the step size λ_j that proceeds as follows for each iteration j:

1. Let $m = 0$.
2. Let $\lambda_j = \gamma^m \lambda^0$.
3. If
$$J(\theta(j), G) - J(\theta(j) + \lambda_j d(j), G) \geq -\sigma \lambda_j \nabla J(\theta(j), G)^\top d(j)$$
then return λ_j as the step size to be used at iteration j. Otherwise, let $m = m + 1$ (i.e., increase m by one) and go to Step 2.

Suppose that we apply the Armijo step size rule to the steepest descent method so that $d(j) = -\nabla J(\theta(j), G)$. The test in Step 3 then evaluates if the advance in reducing $J(\theta, G)$ (i.e., the left side of the inequality, $J(\theta(j), G) - J(\theta(j+1), G)$) is greater than a scaled version of the size of the gradient at $\theta(j)$. The Armijo step size rule decreases the step size from the initial guess λ^0 by a factor of γ until it is small enough to ensure a certain amount of decrease in $J(\theta(j+1), G)$ relative to $J(\theta(j), G)$. The amount of reduction is governed by the parameter σ, while the rate of decrease of λ_j is governed by the choice of γ (if γ is chosen too large, it can take too many iterations to find a solution, while if it is chosen too small, it may miss a larger valid step size that could have resulted in more decrease in the value of $J(\theta(j+1), G)$). Note that if $d(j)$ is a descent direction, $\nabla J(\theta(j), G)^\top d(j) < 0$ so that the Armijo step size rule algorithm will be guaranteed to converge in a finite number of iterations. Basically, the Armijo step size rule tries to combine the positive effects of a constant step size rule and a diminishing step size rule. Generally, when the gradient is large, it will try to take a big step (the size limited by the choice of λ^0) since it knows that it is probably not near a local minimum. When the gradient is small, it assumes that it is near a local minimum, and it takes smaller steps (clearly there is the possibility that it could reduce the steps too much so that convergence is slowed).

Step Size Choice Via Normalizing the Gradient

While there are many other ways to pick the step size, we will close this section with a brief discussion on one approach that has been found to be useful for online function approximation problems. In particular, we will focus on picking the step size for the steepest descent case for linear in the parameter approximators by "normalizing" the gradient.

For a linear in the parameter approximator $y = F(x, \theta) = \theta^\top \phi(x)$ and the case where we process one training data pair at a time (i.e., recursive processing as is often done in online approximation) we have, following the development in Section 11.1.4 in the case where $\bar{N} = 1$,

$$\theta(k+1) \quad = \quad \theta(k) + \lambda_k \phi(x(k))(y(k) - \theta^\top(k)\phi(x(k)))$$

where k is the time index. Consider the choice of

$$\lambda_k = \frac{\lambda}{1 + \gamma \phi^\top(x(k))\phi(x(k))} \tag{11.10}$$

where we consider $\lambda > 0$ to be a type of constant step size and γ to be a tuning parameter. To see how this works, first, view the term

$$\left(y(k) - \theta^\top(k)\phi(x(k))\right)$$

in the above update formula as simply a scalar (approximation error) that indicates how close the approximator is to doing a good job at approximation at time k. If it does well, then this term is small while if it does poorly then this term is larger. It does not, however, contribute to the direction of the update (except in its sign), and only to its size so we will not consider it to be a part of the "update direction" $d(k)$ (you could think of it as part of the step size that says that if you are not doing good approximation at that point, then make a big update, but that if you are doing a good job, then make a small update). Now, we see that the *direction* of the update is dictated by the vector $\phi(x(k))$ and of course, depending on its form, it can also affect the magnitude of the update. Clearly the size of the elements of $\phi(x(k))$ set how big the updates will be for the corresponding components of $\theta(k)$. In fact, $\phi^\top(x(k))\phi(x(k))$ is a measure of the overall size of the update suggested by $\phi(x(k))$ ("big" updates result from a large $\phi^\top(x(k))\phi(x(k))$.

Now, with this in mind, and with the step size in Equation (11.10), we see that if the gradient size, as characterized by $\phi^\top(x(k))\phi(x(k))$, is big, $1 + \gamma\phi^\top(x(k))\phi(x(k))$ will be particularly large, so that the step size λ_k will be small. If the gradient size, as characterized by $\phi^\top(x(k))\phi(x(k))$, is small, $1 + \gamma\phi^\top(x(k))\phi(x(k))$ will be close to 1, so that the step size λ_k is close to λ. In this case, we see that the parameter γ scales our characterization of size of the gradient so that if, for example, γ is very small, then even larger gradients will still result in a step size near λ.

To summarize, we see that the value of λ_k will vary between 0 (for a very big gradient) and λ (for a very small gradient), with γ governing the rate of variation between the two extremes. Hence, the step size will "diminish" if the gradient is large, but it does not generally "diminish" to zero as time progresses since if a local minimum is approached, then the gradient size goes to zero (not to mention the approximation error) and the step size approaches λ. Notice, however, that it does have one characteristic that is similar to some of the other rules. It fixes a maximum step size (λ) no matter how big the gradient is (compare to the Armijo step size rule).

11.1.6 Parameter Initialization, Constraints, and Update Termination

In this section we will discuss the practical issues of how to initialize the gradient methods (i.e., to pick $\theta(0)$), how to incorporate constraints on the parameter values into the dynamics of the algorithm, and if it is needed, how to terminate the algorithm.

Parameter Initialization

Initialization affects performance of the algorithm and ultimately the achieved approximation accuracy.

The choice of $\theta(0)$ can obviously have a significant effect on the quality of a solution provided by the gradient method. While in general you do not know where the optimal solution θ^* is, it is clearly best to pick $\theta(0)$ as close to this desired value as possible. The examples in Section 10.5 illustrated this for the recursive least squares algorithm (for the linear in the parameter case) and the same general concepts hold here. There are, however, other practical issues in choosing $\theta(0)$ for gradient methods.

First, note that since the functions $J(\theta, G)$ that we seek to minimize are in general not necessarily bowl-shaped and can have many local minima, it is often wise to execute the gradient algorithm for several choices of $\theta(0)$. This can help ensure that the value you found is indeed a global minimum. However, simple tests with multiple $\theta(0)$ cannot, in general, guarantee that you have found a global minimum.

Second, for a neural network with sigmoid nonlinearities, if you pick $\theta(0)$ (which will also have weights and biases of a hidden layer in it) to have certain components that are all too large, then it could be that all (or many of) the sigmoids are saturated so that the gradient will be small and updates will be slow (at least at the beginning). It is for this reason that for neural networks, a good choice for the weights and biases is often random small values. For a fuzzy system, there can be similar issues in specifying $\theta(0)$ (e.g., if Gaussian shaped membership functions are used and all the centers are specified to be too large, and the consequent function parameters are set to be zero, then the gradient will be small, at least initially).

In addition, when you generate the update formulas for a function approximator, you may find that the parameters cannot be allowed to take on certain values (both initially and for $j > 0$) or there will be a divide-by-zero error or other numerical problems. Hence, after you derive the update formulas, you should examine them for such problems and initialize appropriately. Moreover, you will then also need to make sure that during its operation, the algorithm never moves the parameters to inappropriate values. We discuss this next.

Constraints on Parameters

Generally, there are several reasons why there are constraints on the parameters in the optimization problems we study:

1. If the parameters $\theta(j)$ take on certain values, there are numerical problems as we discussed above (e.g., division by zero).

2. Since the parameters correspond to physical values (e.g., parameters for a neural network), there are practical limitations to their size (even in software there can be overflows if the values are too large).

3. Sometimes we know a "feasible region" for the optimal parameter values and hence, impose constraints on the set of values that $\theta(j)$ may take on because searching outside this set is fruitless.

4. Sometimes, we have extra information about the form of the underlying function that we seek to approximate (e.g., by physical insights or by simple inspection of the training data), and this can be used to constrain the choices of the parameters. For instance, in some problems you may know where on the input domain the unknown function has more nonlinear or oscillatory behavior, and hence, where you would like to "allocate" more of the approximator structure, since this is where it is needed to get good

approximation accuracy. For instance, if there are some oscillations in the function in a certain region, then you may want more sigmoids or membership functions there. Often, however you may want to allow the training method to perform the actual allocation of structure, rather than fixing it a priori. To do this, you would simply put appropriate constraints on the parameters to only allow them to move in the region where more accuracy is needed.

In any of these three cases, the constraints can be captured by requiring that

$$\theta(j) \in \Theta(j)$$

for all $j \geq 0$ where $\Theta(j)$ is the (known) parameter "constraint set" at iteration j. Often, we know that $\Theta(j) = \Theta$; that is, that the constraint set is the same at each iteration. For convenience, assume this in the discussion below since the case for a time-varying constraint set is similar.

There are effective ways to incorporate into the algorithm constraints on parameter values that are known a priori.

How do we ensure that $\theta(j) \in \Theta$ for all $j \geq 0$? First, we initialize so that $\theta(0) \in \Theta$ so that all we need to concern ourselves with is the case for $j > 0$. In general, the normal approach is to assume that Θ is a convex set. Then, if for some update the gradient method places $\theta(j+1)$ outside Θ, you simply require $\theta(j+1)$ to stay on the boundary of Θ. If it is on the boundary, and the gradient update says to put it outside the boundary, leave it on the boundary. But if it says to leave it on the boundary, or place it within Θ, you let it do that. While it is not difficult to characterize this (and "this" is called a gradient method with "projection" since we project the updates back into the convex feasible parameter set) precisely in mathematical terms, in practice the implementation is often easy and we will simply explain this.

The most common case in practice is when we know scalars θ_i^{min} and θ_i^{max}, $i = 1, 2, \ldots, p$, such that we want

$$\theta_i^{min} \leq \theta_i(j) \leq \theta_i^{max} \tag{11.11}$$

for all $j \geq 0$ (this specifies a convex set $\Theta = \{\theta : \theta_i^{min} \leq \theta_i \leq \theta_i^{max}, i = 1, 2, \ldots, p\}$). Then, each time you use a gradient update formula to generate $\theta(j+1)$, you test Equation (11.11) for each $i = 1, 2, \ldots, p$ and if it has chosen

$$\theta_i(j+1) > \theta_i^{max}$$

you let

$$\theta_i(j+1) = \theta_i^{max}$$

and if it has chosen

$$\theta_i(j+1) < \theta_i^{min}$$

you let

$$\theta_i(j+1) = \theta_i^{min}$$

If any generated $\theta_i(j+1)$ is within the range specified by Equation (11.11), then you accept the update $\theta_i(j+1)$ with no modification.

In this way, for all $j \geq 1$ we will never update the parameter vector $\theta(j)$ to lie outside Θ. You can see that this "projection method" is very easy to implement in practice, and hence, it is often easy to avoid the problems outlined at the beginning of this subsection.

Parameter Update Termination

In an offline function approximation problem, there is a need to specify a termination criterion so that when this criterion is met, the algorithm is stopped and the final computed value of the parameters is taken to be the best solution (which below, we will call $\theta^\star$, since the algorithm may not have found the optimal solution, i.e., it may be that $\theta^\star \neq \theta^*$). It is also possible in an online method, where you decide to do multiple iterations of the gradient method from one sampling instant to the next (an issue that is discussed more in the next section), to use a type of termination criterion as we will discuss below.

It is best to use "scale-free" termination criteria such as the following:

Choice of termination criteria is governed by the desire to terminate with the best possible approximation.

1. *Terminate if Parameter Change Rate is Low:* Terminate if

$$(\theta(j+1) - \theta(j))^\top (\theta(j+1) - \theta(j)) \leq \epsilon \theta(j)^\top \theta(j)$$

for some $\epsilon > 0$ and let $\theta^\star = \theta(j+1)$. This requires the relative amount of change in the parameter values to decrease enough before termination. In this case, it terminates since the parameters are not changing much anyway, so the algorithm is not making much progress. Other times, tests that check that the parameters have not changed much over the last fixed number of iterations are used.

2. *Terminate if the Gradient is Small:* Terminate if

$$\nabla J(\theta(j), G)^\top \nabla J(\theta(j), G) \leq \epsilon \nabla J(\theta(0), G)^\top \nabla J(\theta(0), G)$$

for some $\epsilon > 0$ and let $\theta^\star = \theta(j)$. In this case, when the size of the gradient is small relative to its size in the beginning, then you terminate since the algorithm is operating slowly at this point and will probably not make many further improvements.

While such methods are often used, in practice they are often augmented (i.e., used in conjunction with) other criteria such as the following:

1. *Terminate Based on a "Validation Set":* For many problems you have an idea of how much you would like to reduce the cost function and when you get to this value, you terminate. Along these lines, in the function approximation problem one common approach is to pick a "validation set"

$$V = \{(x(i), y(i)) : i = 1, 2, \ldots, M_v\}$$

on which the cost function value will be evaluated. While some training data from G may be used in V, it is best if V contains a significant amount of data that are not in G so that $J(\theta(j), V)$ is a measure of $J(\theta(j), \Gamma)$ (recall that Γ is the "test set") and hence also quantifies how well the function approximator "generalizes" (i.e., interpolates between training data points) and achieves its overall function approximation task. Now using V, terminate when $J(\theta(j), V)$ starts increasing since this will stop the algorithm before it starts generalizing poorly. Other times, you may terminate when $J(\theta(j), V) < \epsilon$ (i.e., when you have reduced a measure of the function approximation error to less than some value) or

$$J(\theta(j), V) \leq \epsilon J(\theta(0), V)$$

(i.e., you have reduced a measure of the function approximation error to some percentage of its initial value) and let $\theta^\star = \theta(j)$. Both of these approaches could make sense for particular applications.

2. *Terminate After a Maximum Number of Iterations:* In practical problems, you simply have to set a maximum number of iterations that you will allow. Otherwise, the above criteria may either never be met or take too long to reach. If N_{max} is the maximum number of iterations that you will allow, upon termination you will let $\theta^\star = \theta(N_{max})$.

Regardless of which termination method(s) you choose, it is important to view them as something that may also need to be tuned (i.e., modified for each application to get the best or at least an adequate approach).

11.1.7 Offline and Online, Serial and Parallel Data Processing

In this section we discuss how data can be processed by gradient algorithms. We focus on issues of order of data processing and parallel versus serial processing. We will, however, discuss such issues in the context of whether we are doing offline or online (i.e., real time) processing of training data.

The manner in which you process the given training data can significantly affect the performance of the approximator.

Offline Processing

In this case we know G a priori and hence its size M is fixed. The gradient methods discussed up till now (except some in the linear in the parameters case) process the data set G in "parallel" by repeated application of the gradient update formula to the entire data set. There are, however, ways to process the data in G serially and this can, in some cases, lead to savings in computational complexity, and improved convergence properties.

For instance, sometimes a sequential fixed-order processing of single data pairs from G can improve the rate of convergence over the case where the data in G is used in a batch fashion. In this case, you simply order the data in G and process it in that order many times (cycle through the finite data set G repeatedly). For each data pair you could execute one (or more) iterations of the gradient update formula. Normally, in this case you would use the parameter set last generated by the algorithm to initialize the algorithm when you start processing another piece of data.

Other times, when G is known a priori, you can process the data from G one at a time in a random order (perhaps with more than one iteration of the gradient update formula for each piece of data), and sometimes this has been found to provide better approximation accuracy.

In addition, whether you use fixed-order cycling through the data or a random order, you could process a *subset* of the data in G (i.e., more than one piece of training data) and perhaps execute more than one iteration for each subset (again, when you start the processing for one subset you normally would initialize the gradient update formula with the parameter set found at the last iteration for the last subset considered). Or, you could process subsets of different sizes at different times. For example, you could start by processing the data pairs one at a time and then increase the size of the subset processed at each subsequent step (the rate of the increase in subset size would be a design parameter) until all the data pairs in G were processed in a batch fashion. The algorithm could then continue in the normal batch processing mode until some termination criterion was met.

No matter which type of processing you choose for your application, it is important to keep in mind that step size choice and data processing choices are interrelated (e.g., sometimes you want to diminish your step size if you grow the size of the batch of data that you process at each step).

Online Processing

The data processing issues are different for the online case since we do not know G a priori since $M \to \infty$ (of course, the number of data considered is never really infinite, it is just convenient to think of it that way). The most common case in online (real time) processing is to use one training data pair per time step and take one iteration of the gradient formula; this aligns time steps, data acquisitions, and iterations of the gradient update formula.

But, of course, you could gather multiple pieces of data, and iterate the gradient update formula multiple times for each of these data sets (in this case, we acquire data at a higher rate than we initiate processing of the gradient update formula). Again, when you start the processing for one subset, you normally would initialize the gradient update formula with the parameter set, found at the last iteration for the last subset considered. Just as with the offline approach, you could process varying sizes of subsets of data at each step, and you will have to pay attention to the effects of data processing choice on selection of the step size.

11.1.8 Stochastic Gradient Optimization Basics

Suppose that we seek to minimize $J(\theta) \geq 0$ where $\nabla J(\theta(j))$ is Lipschitz in θ. Suppose that we use the gradient method

$$\theta(j+1) = \theta(j) + \lambda_j d(j)$$

with

$$d(j) = -\left(\nabla J(\theta(j)) + n(j)\right) \tag{11.12}$$

where

$$\sum_{j=0}^{\infty} \lambda_j = \infty, \sum_{j=0}^{\infty} \lambda_j^2 < \infty \tag{11.13}$$

(so the step size results in a persistent parameter update). Also, $n(j) \in \Re^p$ is a vector noise term with

$$E[n(j)|\mathcal{H}_j] = 0 \tag{11.14}$$

where $\mathcal{H}_j = \{\theta(i), d(i), \lambda_i : i = 0, 1, 2, \ldots, j\}$ holds the past information from the algorithm and $E[\cdot]$ denotes the expected value, and

$$E[n(j)^\top n(j)|\mathcal{H}_j] = c_1 + c_2 \nabla J(\theta(j))^\top \nabla J(\theta(j)) \tag{11.15}$$

where c_1 and c_2 are two positive constants. Under these conditions the sequence $J(\theta(j))$ converges, $\lim_{j\to\infty} \nabla J(\theta(j)) = 0$, and every limit point of $\theta(j)$ is a stationary point of J.

Equation (11.12) is a standard deterministic steepest descent approach modified at each step by perturbing the steepest descent direction with the direction $n(j)$. How could this help with the performance of the optimization process? On average, due to Equation (11.14), the algorithm will move in the steepest descent direction. Due to Equation (11.15), the noise perturbations will not be so large that they will destroy convergence properties. On a surface $J(\theta)$ that has many local minima, it may be that $n(j)$ will move the updates so as to avoid local minima (i.e., it could help "jump" out of local minima).

11.2 Levenberg-Marquardt and Conjugate Gradient Methods

In this section we introduce Newton's, conjugate gradient, and Levenberg-Marquardt methods, the latter two of which have been found to be quite effective in solving practical function approximation problems.

11.2.1 Newton's Method

For the nonlinear in the parameter approximator, the cost function $J(\theta, G)$ is not a quadratic function of θ as it is in the linear in the parameter case. For this reason, $J(\theta, G)$ can take on very complex shapes (with many local minima) with high slope regions in some areas of the parameter space and very low slope regions in other areas. In the low slope regions, the gradient is (very) small so if you use a constant step size, the changes to $\theta(j)$ will generally be small and convergence will generally be (very) slow. In the high slope regions, if a constant step size is used, the changes to $\theta(j)$ can be too large so that convergence is not achieved. While there are a variety of approaches to try to solve these problems with the steepest descent approach (e.g., adaptive step size methods, modifications to the descent direction, such as the "momentum term" approach), in practice such modifications are often found to be lacking. For example, the traditional "backpropagation" algorithm is a (stochastic) steepest descent method whose direct application has often been found to lead to slow convergence for practical problems. It is for these reasons that more sophisticated methods have been employed for tuning θ that rely on the more sophisticated use of information from $F(x, \theta)$. The methods we are referring to are the Newton, conjugate gradient and quasi-Newton, Gauss-Newton, and Levenberg-Marquardt methods.

Newton's Parameter Update Formula

Let

$$\nabla^2 J(\theta(j), G) = \left[\frac{\partial^2 J(\theta, G)}{\partial \theta_i \theta_j} \right] \Bigg|_{\theta = \theta(j)}$$

be the (symmetric) $p \times p$ "Hessian matrix," whose elements are the second partials of $J(\theta, G)$ at $\theta = \theta(j)$. In Newton's method we make a quadratic approximation of $J(\theta, G)$ at $\theta(j)$ at each iteration j and minimize this to generate $\theta(j+1)$. Let the quadratic approximation at $\theta(j)$ be the second order Taylor series expansion of $J(\theta, G)$ at $\theta(j)$ (i.e., a Taylor series expansion truncated after the second term), which we will denote by

Newton's method is based on producing a quadratic approximation to the cost function at the current parameter values and then choosing the next parameters to be those that minimize that quadratic cost.

$$\begin{aligned} J_q(\theta, G) &= J(\theta(j), G) + \nabla J(\theta(j), G)^\top (\theta - \theta(j)) + \\ &\quad \frac{1}{2} (\theta - \theta(j))^\top \nabla^2 J(\theta(j), G)(\theta - \theta(j)) \end{aligned}$$

Since this is quadratic in θ, if we take the derivative with respect to θ and set it equal to zero and solve for θ, its value will be $\theta(j+1)$, the parameter vector that minimizes $J_q(\theta(j), G)$. In particular,

$$\nabla J_q(\theta, G) = \nabla J(\theta(j), G) + \nabla^2 J(\theta(j), G)(\theta - \theta(j))$$

and if we let $\nabla J_q(\theta, G) = 0$, we get $\theta = \theta(j+1)$ and Newton's update formula is

$$\theta(j+1) = \theta(j) - \lambda_j \left(\nabla^2 J(\theta(j), G) \right)^{-1} \nabla J(\theta(j), G) \tag{11.16}$$

where we have added the step size $\lambda_j > 0$. In the case where $\lambda_j = 1$ for all j, the method is called the "pure form" of Newton's method. In this case, the Newton direction (which may not be a descent direction since it may be that $\nabla^2 J(\theta(j), G)$ is not positive definite, so it may be that J is not decreased at each iteration) is

Newton's method is not generally practical since it depends on computation of the inverse of the Hessian of the cost function.

$$d(j) = - \left(\nabla^2 J(\theta(j), G) \right)^{-1} \nabla J(\theta(j), G) \tag{11.17}$$

Notice here that we have to assume that $\nabla^2 J(\theta(j), G)$ is invertible (e.g., if it is positive definite, then it is invertible). Note also that this pure form for Newton's method can be attracted by both local minima and maxima. There are many methods that have been developed to try to solve these problems with Newton's method.

The Linear in the Parameter Case

For a function $J(\theta, G)$ that is quadratic in θ (the linear in the parameter case), Newton's method provides convergence in one step. To see this, first consider our simple scalar quadratic example in Figure 11.3, where $J(\theta, G) = \theta^2$. Recall that we had

$$\nabla J(\theta, G)|_{\theta=\theta(j)} = 2\theta(j)$$

so that

$$\nabla^2 J(\theta, G)\big|_{\theta=\theta(j)} = 2$$

Hence, Newton's method for this simple example is given by

$$\theta(j+1) = \theta(j) - \left(\frac{1}{2}\right)(2\theta(j))$$

if we pick $\lambda_j = \lambda = 1$ for all j (i.e., a "pure" Newton update). Clearly, if we guess any value of $\theta(0)$, we get

$$\theta(1) = 0$$

so we get convergence in one step (i.e., Newton's method modifies the descent direction so that it converges very fast). Of course, this one step convergence (to the least squares value) only works for quadratic functions and it works because the quadratic approximation $J_q(\theta, G)$ is exact.

In tuning only parameters that enter linearly, the quadratic approximation used in Newton's method is exact so its first update is a batch least squares solution and convergence occurs in one step.

In the general linear in the parameter case, using derivatives from the last section,

$$\nabla J(\theta, G)|_{\theta=\theta(j)} = -\Phi^\top E\big|_{\theta=\theta(j)}$$

and

$$\nabla^2 J(\theta, G)\big|_{\theta=\theta(j)} = \Phi^\top \Phi$$

so that the pure Newton method is

$$\begin{aligned}\theta(j+1) &= \theta(j) + (\Phi^\top\Phi)^{-1}\Phi^\top E\big|_{\theta=\theta(j)} \\ &= \theta(j) + (\Phi^\top\Phi)^{-1}\Phi^\top(Y - \Phi\theta(j)) \\ &= (\Phi^\top\Phi)^{-1}\Phi^\top Y\end{aligned}$$

and we see that we get one-step convergence no matter what the initial guess $\theta(0)$ is. Note that this simply shows that for the linear in the parameter case, Newton's method is equivalent to a batch least squares approach and hence it relies on the existence of the inverse shown in Equation (11.16). We pay the price for fast convergence by assuming the existence of the inverse and computing it.

11.2.2 Conjugate Gradient and Quasi-Newton Methods

In the nonlinear in the parameter case, it is very difficult to guarantee the existence of the inverse in the Newton update formula, and difficult to compute it. Hence, Newton's method is rarely used in practice for the tuning of function approximators. Newton's method does, however, set up a goal for us in terms of convergence rate and hence, many methods try to approximate it but still avoid the problems with the computation of the inverse. The Levenberg-Marquardt method is one approach to avoid the computations necessary for the Newton method but still try to achieve its nice rate of convergence properties. It is discussed later.

In this section we briefly study other methods to try to speed up the steepest descent method without the extra computations associated with Newton's method.

Conjugate Gradient Methods

Conjugate gradient methods were originally developed for quadratic optimization problems to try to keep the descent directions properly aligned to speed the convergence of the steepest descent approach. In fact, for a quadratic minimization problem with p variables, they can be shown to converge in p steps. For nonlinear optimization problems they cannot in general be shown to provide this fast convergence property; however, for many problems they do provide good convergence and rate of convergence properties. They achieve this without using the Hessian or any matrix inversion; hence, they have sometimes been found to be useful for problems with a large value of p.

There are many variations on the parameter update formula for conjugate gradient methods (many of these are equivalent for the quadratic case but different for the nonlinear case). Here, we pick just the most common one that is given by

$$\theta(j+1) = \theta(j) + \lambda_j d(j)$$

where λ_j is generated by a line minimization so that

$$J(\theta(j) + \lambda_j d(j), G) = \min_{\lambda \in [0, \lambda^0]} J(\theta(j) + \lambda d(j), G)$$

The accuracy of the line minimization can affect the performance of the algorithm and often you need to experiment with the choice of parameters of the line minimization method to get good performance. Here, we will assume that the Armijo step size rule is used to specify λ_j. The descent direction $d(j)$ is given by

$$d(j) = -\nabla J(\theta(j), G) + \beta(j) d(j-1) \tag{11.18}$$

where

$$\beta(j) = \frac{\nabla J(\theta(j), G)^\top \left(\nabla J(\theta(j), G) - \nabla J(\theta(j-1), G) \right)}{\nabla J(\theta(j-1), G)^\top \nabla J(\theta(j-1), G)}$$

is called the "Polak-Ribiere" formula.

In the practical application of the method, it has been found to be useful to make certain modifications to the method to improve its convergence properties. There are many ways to modify the algorithm. For instance, the method could be modified by using a steepest descent direction at the first step. Then, every N_{cg} steps, the algorithm is "restarted" by using a steepest descent update direction.

Quasi-Newton Methods

In "quasi-Newton methods" you try to avoid problems with existence and computation of the inverse in Equation (11.17) by choosing

$$d(j) = -\Lambda(j)\nabla J(\theta(j), G)$$

where $\Lambda(j)$ is a positive definite $p \times p$ matrix for all j and that is chosen to approximate $\left(\nabla^2 J(\theta(j), G)\right)^{-1}$. In this way, $d(j)$ may approximate a Newton direction and we may get the associated fast convergence without all the extra computations.

For example, in some cases the approximation is performed by letting $\Lambda(j)$ be a diagonal matrix with its elements set to the corresponding diagonal elements of $\left(\nabla^2 J(\theta(j), G)\right)^{-1}$ and in this case, the method is often referred to as the "diagonally scaled steepest descent method." In some practical applications this method can be quite effective.

Generally, if $\Lambda(j)$ is chosen properly, for some applications much of the convergence speed of Newton's method can be achieved. In other more sophisticated approaches, $\Lambda(j)$ is chosen to form an approximation to the inverse of the Hessian. Here, we outline just one, the "Broyden-Fletcher-Goldfarb-Shanno" (BFGS) method. For this, we have

$$\theta(j+1) = \theta(j) + \lambda_j d(j)$$

where λ_j is generated by a line minimization (e.g., the Armijo step size rule), and

$$d(j) = -\Lambda(j)\nabla J(\theta(j), G)$$

We define

$$c(j) = \theta(j+1) - \theta(j)$$

and

$$g(j) = \nabla J(\theta(j+1), G) - \nabla J(\theta(j), G)$$

Then, we let $\Lambda(0)$ be an arbitrary positive definite matrix, and

$$\begin{aligned}\Lambda(j+1) \quad = \quad & \Lambda(j) + \frac{c(j)c(j)^\top}{c(j)^\top g(j)} - \frac{\Lambda(j)g(j)g(j)^\top \Lambda(j)}{g(j)^\top \Lambda(j) g(j)} \\ & + g(j)^\top \Lambda(j) g(j) h(j) h(j)^\top\end{aligned}$$

where

$$h(j) = \frac{c(j)}{c(j)^\top g(j)} - \frac{\Lambda(j)g(j)}{g(j)^\top \Lambda(j) g(j)}$$

Even though storage of $\Lambda(j)$ and other computational requirements can be heavy for the BFGS method, there are situations where the BFGS method may be preferred to the conjugate gradient method. The BFGS method can provide fast convergence when it is near a solution, and generally seems to be less sensitive to line minimization accuracy. Depending on the complexity of computing $J(\theta(j), G)$ and its gradient, you may, however find one method preferred over the other.

11.2.3 Gauss-Newton and Levenberg-Marquardt Methods

Next, we consider the Gauss-Newton method that is used to solve a (nonlinear) least squares problem, such as finding θ to minimize $J(\theta, G)$ in Equation (11.1) when we do not use a linear in the parameter approximator. To develop the Gauss-Newton method, we have

$$J(\theta, G) = \frac{1}{2}\sum_{i=1}^{M} |y(i) - F(x(i), \theta)|^2$$

and let the $\bar{N} \times 1$ vectors

$$\epsilon(i) = y(i) - F(x(i), \theta)$$

(these are the function approximation errors arising at each piece of training data) and define the $\bar{N}M \times 1$ vector

$$\begin{aligned} \epsilon(\theta, G) &= [\epsilon(1)^\top, \epsilon(2)^\top, \ldots, \epsilon(M)^\top]^\top \\ &= [\epsilon_1, \epsilon_2, \ldots, \epsilon_{\bar{N}M}]^\top \end{aligned}$$

(where ϵ_j, $j = 1, 2, \ldots, \bar{N}M$, are scalars) to be a vector containing all of the approximation errors. Note that

$$J(\theta, G) = \frac{1}{2}\sum_{i=1}^{M} \epsilon(i)^\top \epsilon(i) = \frac{1}{2}\epsilon(\theta, G)^\top \epsilon(\theta, G)$$

In the Gauss-Newton method, the approximation error is linearized about the current parameter values and then least squares is used to minimize the linearized error value to provide the next parameter update.

For functions $J(\theta, G)$ that are quadratic in θ (scalar or vector case), Newton's method gave very fast convergence (in one step). For the function approximation problem, to get $J(\theta, G)$ quadratic in θ, we use a linear in the parameter approximator $F(x, \theta) = \theta^\top \phi(x)$ so that the approximation errors $\epsilon(i)$ and hence $\epsilon(\theta, G)$ are linear (affine) with respect to the parameters θ, and then $J(\theta, G)$ is quadratic in θ. If $F(x, \theta)$ is nonlinear in the parameters, then so are $\epsilon(i)$ and $\epsilon(\theta, G)$.

Gauss-Newton Parameter Update Formula

To tune nonlinear in the parameter approximators in the Gauss-Newton approach, at each iteration j we proceed according to the following steps:

1. Linearize the error $\epsilon(\theta, G)$ about the current value of $\theta(j)$.

2. Solve a least squares problem to minimize the linearized error value and provide the next guess at the parameter, $\theta(j+1)$.

Compared to Newton's method, in the Gauss-Newton method you create a quadratic approximation to the function you want to minimize at each iteration, but now it is done via linearization, rather than using second derivative information. We discuss these two steps in more detail next.

First, linearize $\epsilon(\theta, G)$ around $\theta(j)$ using a truncated Taylor series expansion to get

$$\hat{\epsilon}(\theta, \theta(j), G) = \epsilon(\theta(j), G) + \nabla\epsilon(\theta, G)^\top\big|_{\theta=\theta(j)} (\theta - \theta(j))$$

where $\hat{\epsilon}(\theta, \theta(j), G)$ is an approximation of $\epsilon(\theta, G)$ since we omitted the higher order terms (second order and higher) in the Taylor series expansion. We use the notation $\hat{\epsilon}(\theta, \theta(j), G)$ to emphasize the dependence on both θ and $\theta(j)$. Here,

$$\nabla\epsilon(\theta, G) = \begin{bmatrix} \frac{\partial\epsilon_1}{\partial\theta_1} & \cdots & \frac{\partial\epsilon_{\bar{N}M}}{\partial\theta_1} \\ & \ddots & \\ \vdots & & \vdots \\ & \ddots & \\ \frac{\partial\epsilon_1}{\partial\theta_p} & \cdots & \frac{\partial\epsilon_{\bar{N}M}}{\partial\theta_p} \end{bmatrix} = [\nabla\epsilon_1, \nabla\epsilon_2, \ldots, \nabla\epsilon_{\bar{N}M}] \quad (11.19)$$

is a $p \times \bar{N}M$ matrix and $\nabla\epsilon(\theta, G)^\top$ is the "Jacobian."

Second, minimize the (scaled) squared norm,

$$J_q(\theta, G) = \frac{1}{2}\hat{\epsilon}(\theta, \theta(j), G)^\top \hat{\epsilon}(\theta, \theta(j), G)$$

which is a quadratic approximation to $J(\theta, G)$ (at $\theta(j)$), which is nonlinear in θ, and different from the one used in Newton's method. Let

$$\begin{aligned} \theta(j+1) &= \arg\min_\theta J_q(\theta, G) \\ &= \arg\min_\theta \frac{1}{2}\hat{\epsilon}(\theta, \theta(j), G)^\top \hat{\epsilon}(\theta, \theta(j), G) \end{aligned}$$

(here, "$\arg\min_\theta$" is simply mathematical notation for the value of θ that minimizes the norm—it is the "argument" that provides the value that achieves the minimization).

We know how to solve this problem. It is the same as the batch least squares problem for the linear in the parameters case. To see this, note that

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

with $E = \hat{\epsilon}(\theta, \theta(j), G)$. Recall that we had

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

Here, we have

$$\hat{\epsilon}(\theta, \theta(j), G) = \left(\epsilon(\theta(j), G) - \nabla\epsilon(\theta, G)^\top\big|_{\theta=\theta(j)} \theta(j)\right) + \nabla\epsilon(\theta, G)^\top\big|_{\theta=\theta(j)} \theta$$

so if we let

$$Y = \left(\epsilon(\theta(j), G) - \nabla\epsilon(\theta, G)^\top\big|_{\theta=\theta(j)} \theta(j)\right)$$

and

$$\Phi = -\left.\nabla\epsilon(\theta, G)^\top\right|_{\theta=\theta(j)}$$

our least squares solution (the value of the parameter at the next iteration) is given by Equation (10.2) as

$$\theta(j+1) = (\Phi^\top\Phi)^{-1}\Phi^\top Y$$

so $\theta(j+1)$ is

$$-\left(\nabla\epsilon(\theta(j), G)\nabla\epsilon(\theta(j), G)^\top\right)^{-1}\nabla\epsilon(\theta(j), G)\left(\epsilon(\theta(j), G) - \nabla\epsilon(\theta(j), G)^\top\theta(j)\right)$$

where

$$\nabla\epsilon(\theta(j), G)^\top = \left.\nabla\epsilon(\theta, G)^\top\right|_{\theta=\theta(j)}$$

so that the resulting Gauss-Newton update formula is

$$\theta(j+1) = \theta(j) - \left(\nabla\epsilon(\theta(j), G)\nabla\epsilon(\theta(j), G)^\top\right)^{-1}\nabla\epsilon(\theta(j), G)\epsilon(\theta(j), G) \quad (11.20)$$

(If we had included a step size parameter, then the method is sometimes referred to as a "damped" Gauss-Newton approach.) Notice that compared with Newton's method, we do not need the Hessian, only the Jacobian. Essentially, a Gauss-Newton iteration is an approximation to a Newton iteration (in the sense that the quadratic approximation at each iteration tries to approximate the one in Newton's method that uses second derivative information in its quadratic approximation) so it can typically provide for faster convergence than, for instance, steepest descent, but generally not as fast as a pure Newton method. It is also interesting to note that the Gauss-Newton method is the same as the "extended Kalman filter" (EKF) except where the linearizations are performed. (In the EKF, where we process one data pair at a time, we perform the linearizations at each point; for the Gauss-Newton method where batch processing is used, we perform the linearizations for each batch.) To make the methods the same simply involves changing how the data are processed.

The Gauss-Newton method is equivalent to the extended Kalman filter if the data are processed in the same way.

Levenberg-Marquardt Parameter Update Formula

To avoid problems with computing the inverse in Equation (11.20), the method is often implemented as

$$\theta(j+1) = \theta(j) - \left(\nabla\epsilon(\theta(j), G)\nabla\epsilon(\theta(j), G)^\top + \Lambda(j)\right)^{-1}\nabla\epsilon(\theta(j), G)\epsilon(\theta(j), G) \quad (11.21)$$

where $\Lambda(j)$ is a $p \times p$ diagonal matrix such that

$$\nabla\epsilon(\theta(j), G)\nabla\epsilon(\theta(j), G)^\top + \Lambda(j)$$

is positive definite so that it is invertible. Sometimes, a "Cholesky factorization" is used to specify $\Lambda(j)$ at each iteration. In the Levenberg-Marquardt method, you choose $\Lambda(j) = \lambda I$ where $\lambda > 0$ and I is the $p \times p$ identity matrix. The parameter λ serves a role similar to the step size. When $\lambda = 0$, we get the standard Gauss-Newton method and as you increase λ, the descent direction moves towards the gradient. Hence, generally thinking of λ as a step size, we expect that for a small value of λ, we will get fast convergence; for a larger value, we should get slower convergence.

The Linear in the Parameter Case

As a simple example, notice that in the case where $J(\theta, G) = \theta^2$, $\epsilon(\theta, G) = \theta$, and if we pick $\lambda_j = \lambda = 1$, $\Lambda(j) = \Lambda = I$ for all j, then $\nabla\epsilon(\theta, G) = 1$ so the Gauss-Newton method is

$$\theta(j+1) = \theta(j) - \theta(j) = 0$$

(and the Levenberg-Marquardt method would be the same if you pick $\lambda_j = \lambda = 2$). Hence, no matter what the choice is for the initial guess $\theta(0)$, $\theta(1) = 0$, and we get convergence in one step (similar to Newton's method for this example). Generally, however, this only occurs in the case where $J(\theta, G)$ is quadratic and we have a linear in the parameters approximator.

11.3 Matlab for Training Neural Networks

There exist many software packages for solving optimization problems with gradient methods (you may want to search the Web to find some public-domain ones), and one that is particularly well-suited for the gradient training of neural networks is the Matlab Neural Networks Toolbox.

11.3.1 Motivation to Use Software Packages

This toolbox provides a variety of tools that facilitate the construction of neural network structures and the training of the parameters in these structures. The training methods include steepest descent, conjugate gradient methods, Levenberg-Marquardt, and others. Moreover, many numerical issues for these algorithms have been tested to help ensure their robustness and numerical accuracy for ease of use.

Excellent software packages exist for optimization and its application to training neural networks and fuzzy systems.

It is recommended that if you want to construct complex multilayer neural networks (e.g., a perceptron with two or more hidden layers) or regularly work with sophisticated practical applications, that you use some software package. In this part we have shown the basic concepts, but there are more issues to deal with when the networks get more complex, in addition to how sophisticated the gradient method is. For instance, for more complex multilayer neural networks, the recognition of the repeated calculations that are necessary in gradient update formulas (particularly steepest descent) led to the "backpropagation" method, which is a method for saving computations in the application of the gradient method (even though, most often, the term "backpropagation" is used to refer to the scheme for saving computations, *and* the gradient method employed).

The Matlab toolbox exploits the repeated calculations using a backpropagation method and frees you from these somewhat tedious details so that you can focus on the fundamental issues in training that are discussed here.

It is also important to point out that many software packages, including Matlab, provide functions for general nonlinear least squares minimization (e.g., using the Levenberg-Marquardt method) and all you have to do is find the gradients and provide the proper information to the software and it will provide a solution. Hence, with such packages it is not only possible to train neural networks but also Takagi-Sugeno fuzzy systems or other approximator structures.

An exercise at the end of the chapter asks you to solve a simple function approximation problem with software such as the Matlab toolbox. Next, we show how to train a multilayer perceptron with the Matlab Neural Networks Toolbox.

11.3.2 Example: Matlab Neural Networks Toolbox

In this section, we will show how to use the Matlab Neural Networks Toolbox to tune a multilayer perceptron 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 train a two layer multilayer perceptron with $n_1 = 11$ hidden layer neurons that have logistic activation functions and a linear activation function in the output layer (as shown in Figure 9.13). We will test different training methods, and will use 500 training "epochs" in each case. The code used is given at the Web site for the book listed in the Preface.

Gradient Descent Training

In this case we use the training option `traingd` which indicates that we want to use a gradient descent approach (this is the classical "backpropagation" approach). When you execute the program, it displays data indicating how the algorithm is performing (e.g., the mean squared error and size of the gradient) and a plot of the mean square error versus the epoch number as shown in Figure 11.4. Notice that as training progresses, the mean squared error decreases. The quality of the approximation for this case is shown in Figure 11.5, where we can see by inspection that a reasonably good approximation was achieved. Note that if you run the code at the Web site you will almost surely get a different plot, since the data are presented in a random order for the training.

Conjugate Gradient Training

In this case we use the training option `traincgp`, which indicates that we want to use a conjugate gradient approach (actually the Polak-Ribiere method). When you execute the program, it displays data indicating how the algorithm is performing (e.g., the mean squared error and size of the gradient) and a plot of the mean square error versus the epoch number as shown in Figure 11.6. Notice that as training progresses, the mean squared error decreases at a faster rate

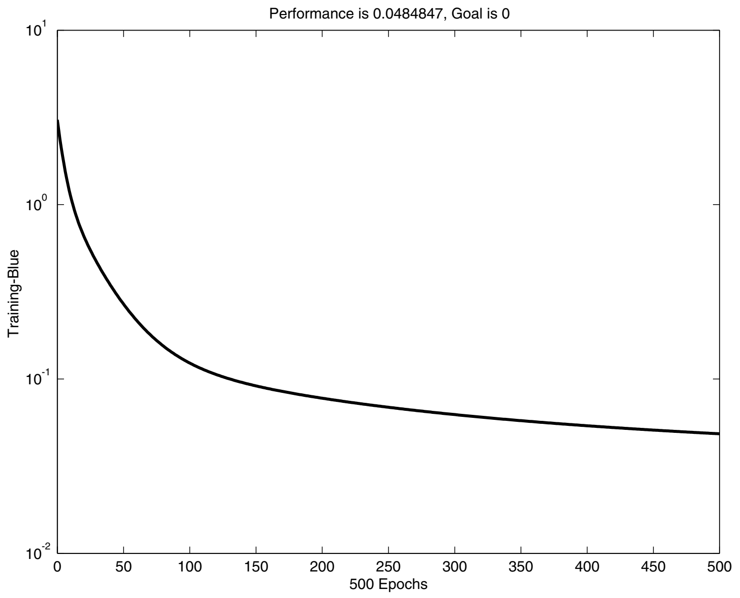

Figure 11.4: Mean squared error vs. epoch number for backpropagation training.

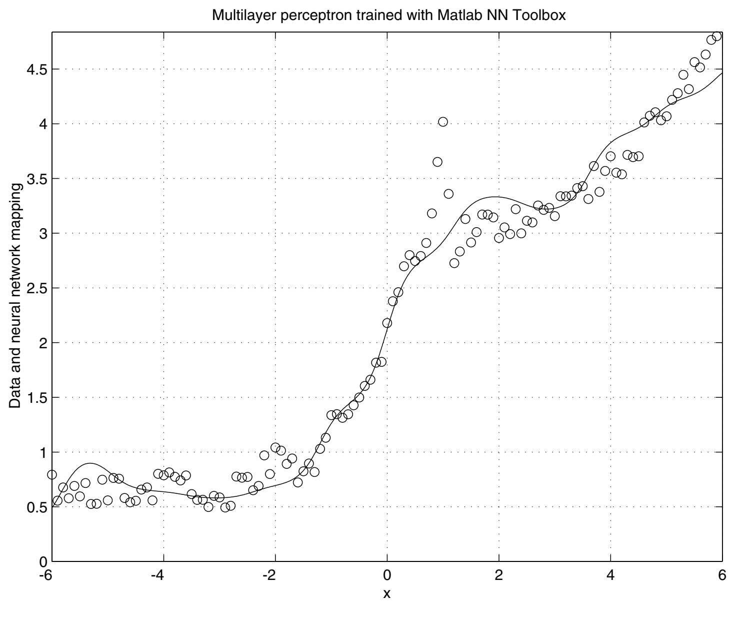

Figure 11.5: Approximator mapping and data for training with backpropagation.

and achieves a lower value than in the standard backpropagation method shown in Figure 11.4.

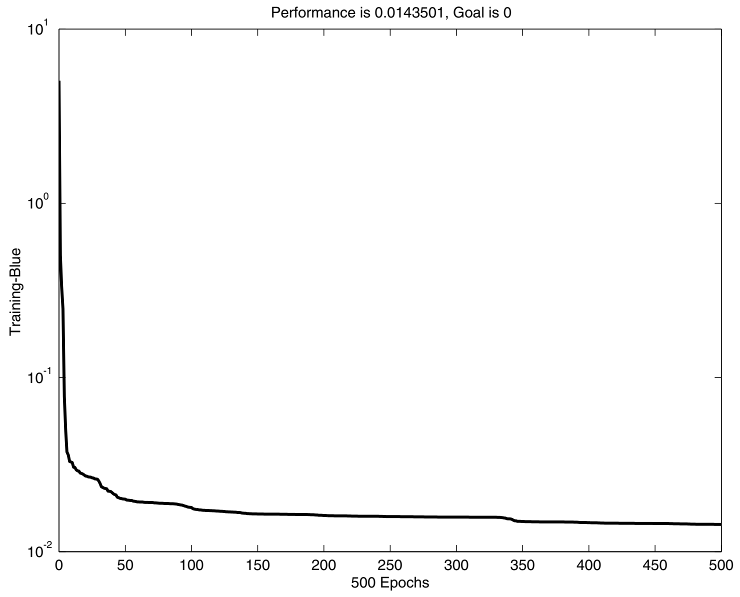

Figure 11.6: Mean squared error vs. epoch number for conjugate gradient training.

The quality of the approximation for this case is shown in Figure 11.7, where we can see by inspection that better approximation was achieved (for this number of training epochs and specific training run) than was achieved for backpropagation in Figure 11.5. It is typical to find slow training times for standard backpropagation and improvements on convergence rates and approximation errors if you compare to a conjugate gradient method.

Levenberg-Marquardt Training

In this case we use the training option `trainlm`, which indicates that we want to use a Levenberg-Marquardt training approach. When you execute the program, it displays data indicating how the algorithm is performing (e.g., the mean squared error and size of the gradient) and a plot of the mean square error versus the epoch number as shown in Figure 11.8. Notice that as training progresses, the mean squared error decreases at a relatively fast rate (even faster than what was obtained in the above training run for the conjugate gradient method in Figure 11.6), then levels off at about the same value as that which was obtained with the conjugate gradient method.

The quality of the approximation for this case is shown in Figure 11.9, where we can see by inspection that better approximation was achieved (for this num-

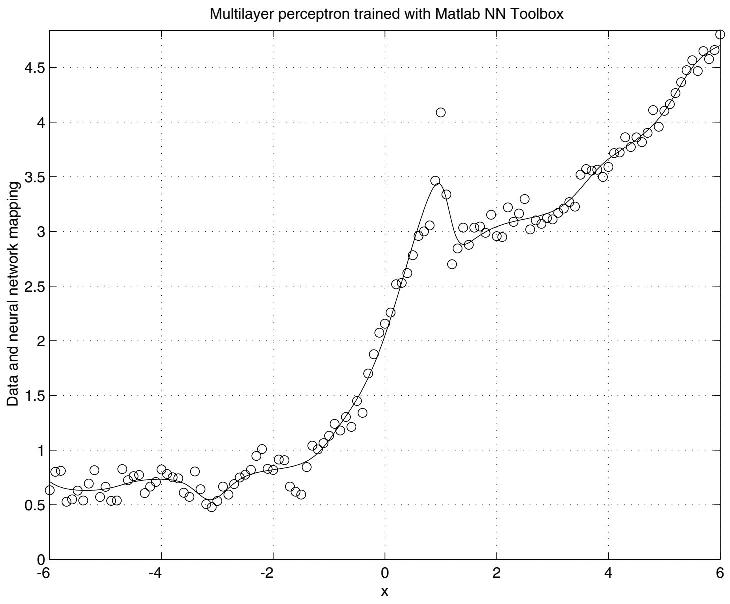

Figure 11.7: Approximator mapping and data for training with a conjugate gradient method.

ber of training epochs and specific training run) than was achieved for backpropagation in Figure 11.5. The result is, however, different from the one obtained with the conjugate gradient method in Figure 11.7, in that it does not adjust the map to the high frequency peak even though it achieves similar accuracy (of course, this is just for this training run; you should not reach any general conclusions by this).

To understand the "canned" software packages, it is useful to build a "homemade" optimization algorithm for approximator tuning.

11.4 Example: Levenberg-Marquardt Training of a Fuzzy System

In this section we study the use of the Levenberg-Marquardt method for training a Takagi-Sugeno fuzzy system with $R = 11$ rules. We will tune all 44 parameters of the approximator. Here, we consider offline batch processing of a data set $G = \{(x(i), y(i)) : i = 1, 2, \ldots, M\}$ from Figure 9.10 (where in this case $n = 1$).

In this case, our Takagi-Sugeno fuzzy system is given by

$$y = F_{ts}(x, \theta) = \frac{\sum_{i=1}^{R} g_i(x)\mu_i(x)}{\sum_{i=1}^{R} \mu_i(x)}$$

where $g_i(x) = a_{i,0} + a_{i,1}x_1$ and the $a_{i,j}$, $i = 1, 2, \ldots, R$, $j = 0, 1$ are constants.

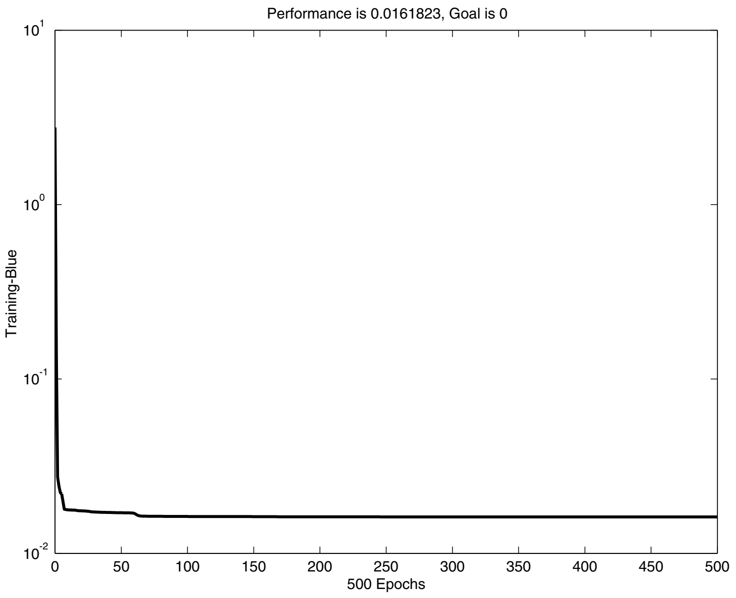

Figure 11.8: Mean squared error vs. epoch number for Levenberg-Marquardt training.

Also,

$$\mu_i(x) = \prod_{j=1}^{n} \exp\left(-\frac{1}{2}\left(\frac{x_j - c_j^i}{\sigma_j^i}\right)^2\right) = \exp\left(-\frac{1}{2}\left(\frac{x_1 - c_1^i}{\sigma_1^i}\right)^2\right)$$

where c_j^i is the point in the j^{th} input universe of discourse where the membership function for the i^{th} rule achieves a maximum, and $\sigma_j^i > 0$ is the relative width of the membership function for the j^{th} input and the i^{th} rule (since $n = 1$, the premise membership functions are the same as the input membership functions). Recall that we had defined

$$\xi_j = \frac{\mu_j(x)}{\sum_{i=1}^{R} \mu_i(x)}$$

$j = 1, 2, \ldots, R$. For our case, we have

$$\begin{aligned}\theta \quad = \quad & [c_1^1, \ldots, c_1^R, \sigma_1^1, \ldots, \sigma_1^R, \\ & a_{1,0}, a_{2,0}, \ldots, a_{R,0}, a_{1,1}, a_{2,1}, \ldots, a_{R,1}]^\top\end{aligned}$$

for a total of $p = 4R = 44$ parameters to tune.

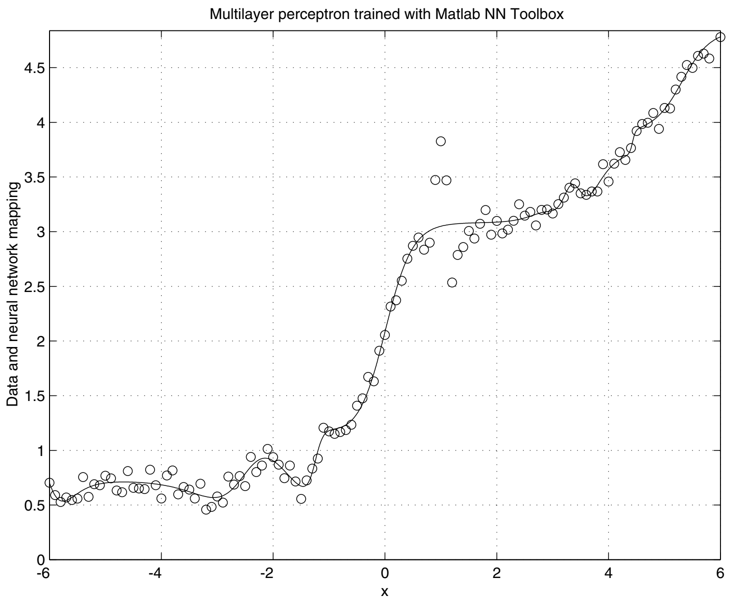

Figure 11.9: Approximator mapping and data for training with the Levenberg-Marquardt method.

11.4.1 Update Formulas

The update formula, given in Equation (11.21), is

$$\theta(j+1) = \theta(j) - \left(\nabla\epsilon(\theta(j),G)\nabla\epsilon(\theta(j),G)^\top + \Lambda(j)\right)^{-1} \nabla\epsilon(\theta(j),G)\epsilon(\theta(j),G) \tag{11.22}$$

where $\Lambda(j) = \lambda I$ where $\lambda > 0$ is a tuning parameter (where if λ is small, we can generally expect faster convergence, but we may need it to be larger to ensure the existence of the inverse) and I is the $p \times p$ identity matrix.

To make the computations for the update formula we need, for $\bar{N} = 1$, the $p \times M$ matrix $\nabla\epsilon(\theta(j), G)$ and the $M \times 1$ vector $\epsilon(\theta(j), G)$. With $\bar{N} = 1$, the scalars

$$\epsilon_i = \epsilon(i) = y(i) - F_{ts}(x(i), \theta)$$

for $i = 1, 2, \ldots, M$, and so $\epsilon(\theta, G) = [\epsilon_1, \epsilon_2, \ldots, \epsilon_M]^\top$. Here,

$$\nabla\epsilon(\theta, G) = \begin{bmatrix} \frac{\partial\epsilon_1}{\partial\theta_1} & \cdots & \frac{\partial\epsilon_M}{\partial\theta_1} \\ & \ddots & \\ \vdots & & \vdots \\ & \ddots & \\ \frac{\partial\epsilon_1}{\partial\theta_p} & \cdots & \frac{\partial\epsilon_M}{\partial\theta_p} \end{bmatrix}$$

Now, notice that for $i = 1, 2, \ldots, M$, $j = 1, 2, \ldots, p$,

$$\begin{aligned}\frac{\partial \epsilon_i}{\partial \theta_j} &= \frac{\partial}{\partial \theta_j}\left(y(i) - F_{ts}(x(i), \theta)\right) \\ &= -\frac{\partial}{\partial \theta_j} F_{ts}(x(i), \theta)\end{aligned}$$

It is convenient to compute this partial by considering various components of the vector in sequence (not forgetting about the minus sign in front of the partials).

Developing the update formulas simply requires the use of the partial of the approximator with respect to the parameters.

First, consider the update formulas for the centers of the premise membership functions. We will use indices i^* and j^* to help avoid confusion with the indices i and j. We find, for $j^* = 1, 2, \ldots, R$,

$$\frac{\partial}{\partial c_1^{j^*}} F_{ts}(x(i^*), \theta) = \frac{\partial}{\partial c_1^{j^*}} \left(\frac{\sum_{i=1}^{R} g_i(x(i^*)) \mu_i(x(i^*))}{\sum_{i=1}^{R} \mu_i(x(i^*))} \right)$$

where

$$\mu_i(x(i^*)) = \exp\left(-\frac{1}{2} \left(\frac{x(i^*) - c_1^i}{\sigma_1^i} \right)^2 \right)$$

(we replaced x_1 with x, since they are the same) and

$$\xi_{j^*}(x(i^*)) = \frac{\mu_{j^*}(x(i^*))}{\sum_{i=1}^{R} \mu_i(x(i^*))}$$

Hence, we have

$$\begin{aligned}\frac{\partial}{\partial c_1^{j^*}} F_{ts}(x(i^*), \theta) &= \frac{\left(\sum_{i=1}^{R} \mu_i(x(i^*))\right)\left(g_{j^*}(x(i^*)) \frac{\partial}{\partial c_1^{j^*}} \mu_{j^*}(x(i^*))\right)}{\left(\sum_{i=1}^{R} \mu_i(x(i^*))\right)^2} \\ &\quad - \frac{\left(\sum_{i=1}^{R} g_i(x(i^*)) \mu_i(x(i^*))\right)\left(\frac{\partial}{\partial c_1^{j^*}} \mu_{j^*}(x(i^*))\right)}{\left(\sum_{i=1}^{R} \mu_i(x(i^*))\right)^2} \\ &= \left(\frac{g_{j^*}(x(i^*)) - F_{ts}(x(i^*), \theta)}{\sum_{i=1}^{R} \mu_i(x(i^*))} \right) \frac{\partial}{\partial c_1^{j^*}} \mu_{j^*}(x(i^*))\end{aligned}$$

For this, let

$$\bar{x}^{j^*} = -\frac{1}{2} \left(\frac{x(i^*) - c_1^{j^*}}{\sigma_1^{j^*}} \right)^2$$

so that using the chain rule from calculus

$$\frac{\partial}{\partial c_1^{j^*}} \mu_{j^*}(x(i^*)) = \frac{\partial \mu_{j^*}(x(i^*))}{\partial \bar{x}^{j^*}} \frac{\partial \bar{x}^{j^*}}{\partial c_1^{j^*}}$$

We have

$$\frac{\partial \mu_{j^*}(x(i^*))}{\partial \bar{x}^{j^*}} = \mu_{j^*}(x(i^*))$$

and

$$\frac{\partial \bar{x}^{j^*}}{\partial c_1^{j^*}} = \frac{x(i^*) - c_1^{j^*}}{\left(\sigma_1^{j^*}\right)^2}$$

so

$$\frac{\partial}{\partial c_1^{j^*}} F_{ts}(x(i^*), \theta) = \left(\frac{g_{j^*}(x(i^*)) - F_{ts}(x(i^*), \theta)}{\sum_{i=1}^{R} \mu_i(x(i^*))} \right) \mu_{j^*}(x(i^*)) \frac{\left(x(i^*) - c_1^{j^*}\right)}{\left(\sigma_1^{j^*}\right)^2}$$

Next, for the spreads on the premise membership functions, we use the same development above to find

$$\frac{\partial}{\partial \sigma_1^{j^*}} F_{ts}(x(i^*), \theta) = \left(\frac{g_{j^*}(x(i^*)) - F_{ts}(x(i^*), \theta)}{\sum_{i=1}^{R} \mu_i(x(i^*))} \right) \mu_{j^*}(x(i^*)) \frac{\left(x(i^*) - c_1^{j^*}\right)^2}{\left(\sigma_1^{j^*}\right)^3}$$

since

$$\frac{\partial \bar{x}^{j^*}}{\partial \sigma_1^{j^*}} = \frac{\left(x(i^*) - c_1^{j^*}\right)^2}{\left(\sigma_1^{j^*}\right)^3}$$

Next, for the parameters of the consequent functions, notice that

$$\frac{\partial}{\partial a_{j^*,0}} F_{ts}(x(i^*), \theta) = \frac{\partial}{\partial a_{j^*,0}} \left(g_{j^*}(x(i^*)) \xi_{j^*}(x(i^*)) \right) = \xi_{j^*}(x(i^*))$$

and

$$\frac{\partial}{\partial a_{j^*,1}} F_{ts}(x(i^*), \theta) = x_1(i^*) \xi_{j^*}(x(i^*))$$

This gives us all the elements for the $\nabla \epsilon(\theta, G)$ matrix, and hence, we can implement the Levenberg-Marquardt update formula.

11.4.2 Parameter Constraint Set and Initialization

The chosen parameter constraint set simply forces the centers to lie between -6 and $+6$ (hence, we assume that we know the maximum variation on the input domain a priori) and spreads to all between 0.1 and 1 and uses projection to maintain this for each iteration. We place the constraints on the spreads for two reasons. First, we must keep the values of the spreads above some fixed value to ensure that we do not have a divide-by-zero error in computing the partials needed for the update formula. Second, it seems reasonable not to have spreads cover too much of the input domain, since then its corresponding consequent (a line) would have to produce an approximation over that large portion of the domain. We put no constraints on the parameters of the consequent functions.

The centers are initialized to be on a uniform grid across the input space, a reasonable choice if you do not know where high frequency behavior occurs; however, if you know that there is a region with higher frequency oscillations, then it may be advantageous to put more centers in that region. In particular, we choose $c_1^1 = -5$, $c_1^2 = -4$, up to $c_1^{11} = 5$. The spreads are all initialized to be 0.5 so that there is a reasonable amount of separation between them when one consequent function of one rule turns on and the other turns off. We will, however, experiment with the effects of the size of the initial spreads on the performance of the method. The parameters of the consequent functions are simply initialized to be all zero. It must be emphasized that while these choices make sense for this problem, and as you will see, work reasonably well for this problem, other initializations may work better (and others, much worse).

11.4.3 Approximator Tuning Results: Effects on the Nonlinear Part

Here, we first consider the $M = 121$ case for the function shown in Figure 9.10. We will simply show the mapping shape at various iterations and hence, will not implement a termination criterion. We choose $\lambda = 0.5$ (you can easily tune this parameter where, if you make it smaller, it tends to make bigger updates). Figure 11.10 shows the mapping after just one iteration. Clearly, even after one iteration, even though it has not tuned the centers and spreads much, the method has chosen reasonable values for the consequent functions and this is not surprising considering the performance of the batch least squares method for this approach and the similarities to that method.

Next, we will focus on how the method tunes the nonlinear part of the approximator (i.e., the μ_i, and hence ξ_i functions) but we must keep in mind that the linear part is also being tuned at the same time. Figure 11.11 shows that by the second iteration, there is already significant and successful tuning of the nonlinear part so that approximation errors are reduced, particularly in the region around $x = -2$.

Adjustments to the parameters that enter nonlinearly provide significant tuning flexibility for the shape of the mapping.

As the algorithm continues, it continues to tune the nonlinear part of the approximator. In particular, consider Figure 11.12 at $j = 5$, and we see that at this point, the training method has done quite a good job at shaping the nonlinear part to obtain good accuracy around $x = -2$. Note that here it is exploiting the parameters that enter in a nonlinear fashion to achieve interesting shapes for the nonlinearity (you could think of this as illustrating the inherent tuning flexibility associated with approximators, where we tune both the parameters that enter linearly and the ones that enter in a nonlinear fashion).

As the algorithm continues, it still continues to tune the nonlinear part of the approximator, both in the region around $x = -2$ and in the high frequency region around $x = 1$. In particular, consider Figure 11.13 at $j = 12$, and we see that while it has tuned the parameters some, it is not much different in the $x = -2$ region. It is, however, having difficulties in the $x = 1$ region due to

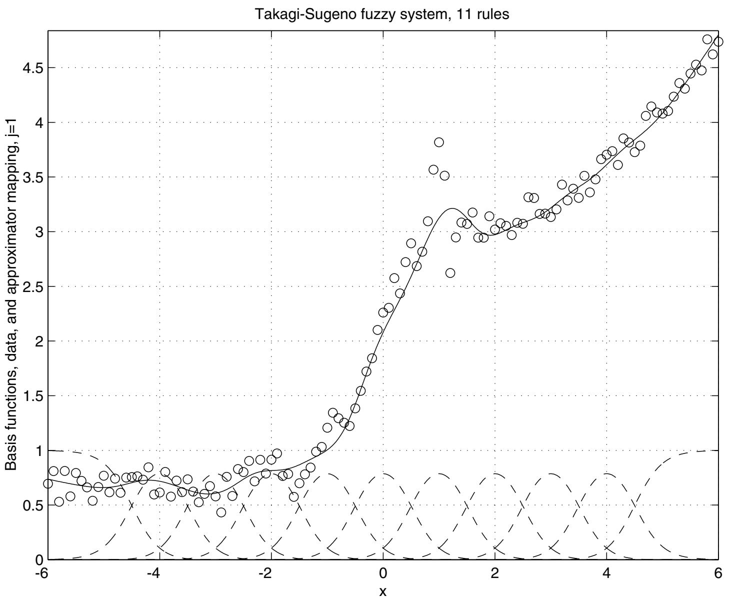

Figure 11.10: Levenberg-Marquardt training of a Takagi-Sugeno fuzzy system, mapping shape at iteration $j = 1$.

the high frequency behavior. It seems that for this example, for higher numbers of iterations, it tends to leave the approximator structure near $x = -2$ pretty much as it is and it tries to "fix" the part near $x = 1$. Consider the mapping at iteration $j = 15$, which is shown in Figure 11.14. Notice that in the $x = 1$ region, there is a significant change in the nonlinear shape. It tends to keep moving this shape around near $x = 1$ to try to improve accuracy.

Now, this is where the issue of termination arises. Do you terminate at $j = 12$ and declare success? Do you try to run the algorithm for many more iterations to see if it can "allocate" more approximator structure to the $x = 1$ high frequency region to improve the accuracy further? If you use more iterations will the overall approximation accuracy improve? Or, will it get even worse that it is here? These are all important issues, but they tend to be very application dependent. It is best if you are simply aware of all these issues and experiment with the particular application at hand to try to get the best possible results (where the definition of "best" certainly depends on the constraints of the particular application).

11.4.4 Approximator Tuning Results: Effects of Initialization

Next, consider the same initial parameters as above except let the spreads all be 0.2 instead of 0.5. Figure 11.15 shows the mapping shape at $j = 1$ and we see

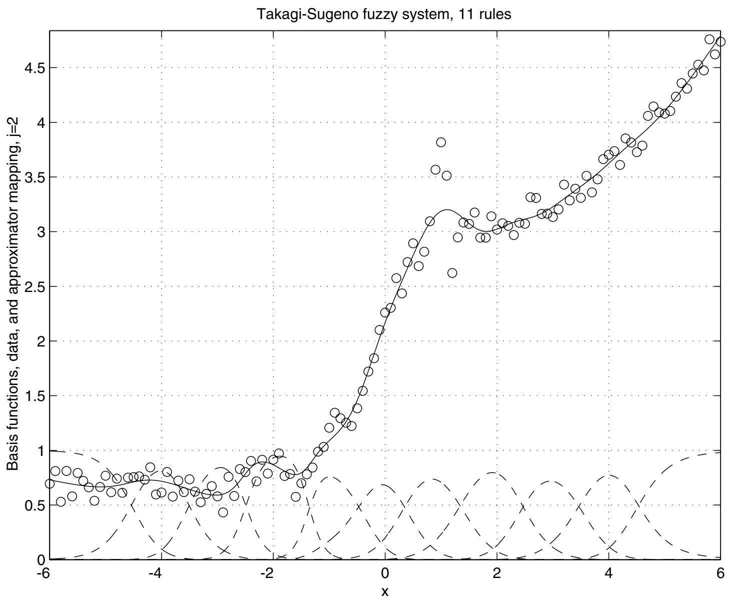

Figure 11.11: Levenberg-Marquardt training of a Takagi-Sugeno fuzzy system, mapping shape at iteration $j = 2$.

that the approximator is not performing too well. The small spread results in sharp transitions between the rules so that there is a sharp transition between the lines that are used in the consequents.

Figure 11.16 shows the mapping shape at $j = 2$. We see that the centers are updated to values that were similar to the 0.5 initialization case; the algorithm quickly recovers from what appeared to be a poor initialization (and then the behavior is qualitatively similar to the case where the spreads were initialized with 0.5 after $j = 2$).

Next, we will use the same initial parameters as above, except let the spreads all be 1 instead of 0.5. Figure 11.17 shows the mapping shape at $j = 1$ and this shows that as we smooth out the membership functions, we tend to get a smoothed out function. This time, however, the method does not recover from this initialization as fast as when the spreads were initialized at 0.2. For instance, notice that by $j = 15$ the mapping shape, which is shown in Figure 11.18, is not much better in the region around $x = -2$; it has, however, done something interesting: up to this point, the algorithm has focused on trying to allocate approximator structure to the high frequency region to try to improve approximation accuracy there.

Overall, these simulations show that the performance of the algorithm clearly depends on the initialization. We would like to start with the best possible initialization; however, for practical problems it can be particularly difficult to get a good one for a particular application without having significant insights

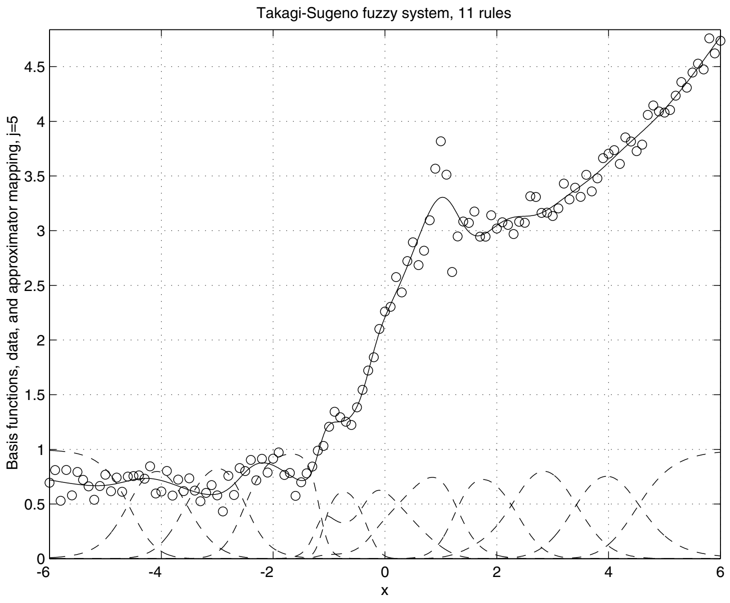

Figure 11.12: Levenberg-Marquardt training of a Takagi-Sugeno fuzzy system, mapping shape at iteration $j = 5$.

into the physics of the problem or by performing analysis on the data before training. Hence, even in practical problems, you may want to use the same basic approaches that we use here for this simple problem.

11.4.5 Overtraining, Overfitting, and Generalization

Next, we consider the case where $M = 13$, which is a much smaller data set than used above. We still use $R = 11$ rules and tune 44 parameters, so our number of parameters is greater than the number of data points. We use our earlier choice of initial parameters as $c_1^1 = -5$, $c_1^2 = -4$, up to $c_1^{11} = 5$ with all the spreads as 0.5. Also, we use $\lambda = 0.5$ as earlier. In Figure 11.19, we show the mapping shape at $j = 1$, and we see that it picks a reasonable shape considering how little information it has been given. There is, however, a problem when we train with so few data and so many parameters, that becomes even clearer if we allow a few more iterations to occur. In particular, consider Figure 11.20, where the mapping is shown at $j = 12$. We see that the algorithm, in one sense, does a very good job. It matches the training data almost exactly at every point. However, this causes a problem since at points outside the training data, the matching to the unknown function is poor (consider, e.g., the large peak near $x = 1$, where even though the mapping goes through one point in that region, we know its shape is not appropriate for the problem at hand). This is called poor "generalization." If the approximator generalizes well, then it will produce

Poor generalization, which is bad interpolation between training data, can occur if the approximator is too complex relative to the amount of information in the training data. You want your approximator simple to help avoid poor generalization, yet complex enough to provide flexibility to match the unknown function.

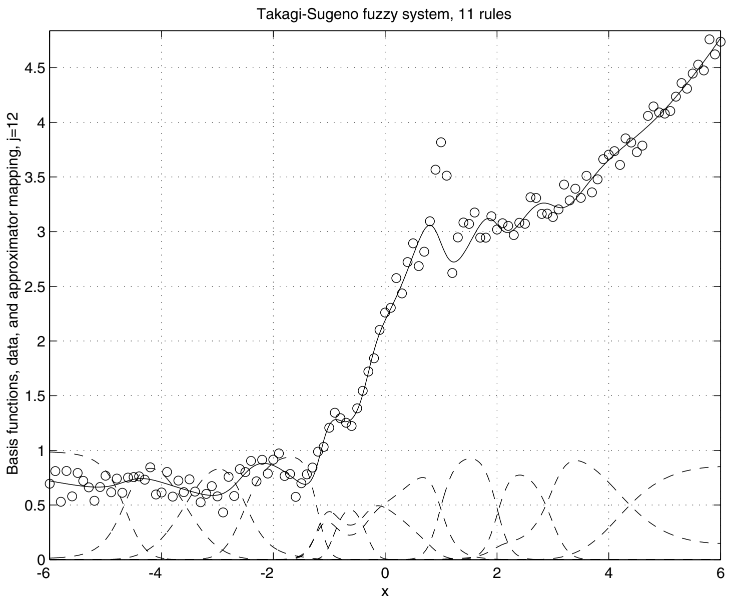

Figure 11.13: Levenberg-Marquardt training of a Takagi-Sugeno fuzzy system, mapping shape at iteration $j = 12$.

a good interpolation between the training data, not one that provides large oscillations between the data. Moreover, if you study Figure 11.20 carefully (and compare it to Figure 9.9 when noise is not added to the function), as we have seen in the least squares case, the approximator is failing also in the sense that it is trying to match the noise in the function (i.e., it is exhibiting overfitting).

How do we avoid these problems? First, you would normally never pick $p > M$; that is, you will normally have fewer parameters than training data pairs. Next, in some applications you need to make sure that you do not "overtrain;" that is, use too many iterations of the gradient update method. Sometimes this can result in forcing the approximator to match exactly at the data pairs at the expense of performing poor generalization (i.e., poor interpolation between the training data). Sometimes, the use of a "validation set" can help to detect when poor generalization is occurring and the updating can be terminated.

11.4.6 Approximator Reparameterization for Flexibility and Complexity Reduction

Sometimes an approximator has too much flexibility, in the sense that there are many ways to tune the parameters to get good approximation accuracy. One way to reduce this flexibility, and thereby simplify the parameter update method, is to make some of the parameters of the approximator that enter

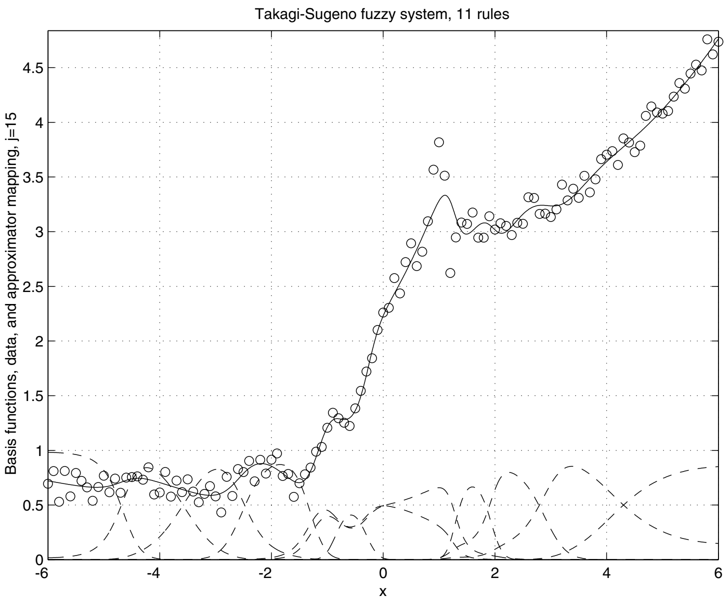

Figure 11.14: Levenberg-Marquardt training of a Takagi-Sugeno fuzzy system, mapping shape at iteration $j = 15$.

in a nonlinear fashion a function of some of the other parameters. In this way we reduce tuning flexibility, but not so much as to reduce it to the case where we only tune the parameters that enter in a linear fashion. For example, one way to do this for the Takagi-Sugeno fuzzy system is to simply make the spreads a function of the centers. One way to do this is to pick the spreads so that neighboring premise membership functions always cross over each other at 0.5. This way, when many centers are allocated to a region to try to improve approximation accuracy, the choice of the spreads will allow for the "turning on" and "off" of the appropriate consequent functions for a high density of membership functions. This approach could be good for some applications, but it should be emphasized that it does *reduce* approximator flexibility and so for some applications, it may not be a good choice. Moreover, the exact methods to specify the function specifying how the spreads change based on the centers will depend on the particular application.

11.4.7 Approximation Error Measures: Using a Test Set

To focus on other issues, we have been glossing over the issues of the use of a "test set" Γ for evaluating the approximation quality of our approximators. Instead we have been relying on visual inspection of the plots to comment on approximation accuracy. Generally, for more complex multidimensional applications, this is not a good approach and you will want to use some type of

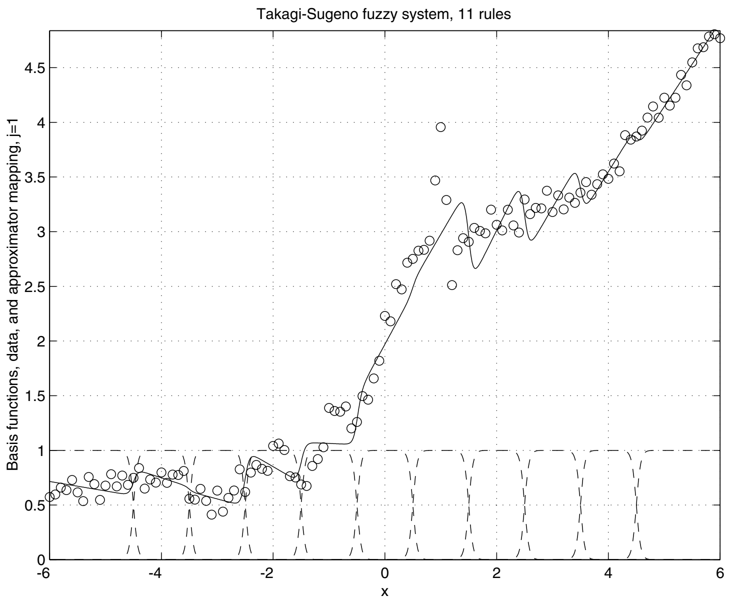

Figure 11.15: Levenberg-Marquardt training of a Takagi-Sugeno fuzzy system, mapping shape at iteration $j = 1$, different initialization.

numerical measure of approximation accuracy where you measure the accuracy *both* at the training data points and at points in between these. In addition, you will often want to evaluate the approximator for points where it is "extrapolating" from the data (e.g., at the end points of the input domains).

Such approximation error measures, based on, for example, a sum of squares or the maximum error over the domain, provide a way to quantify accuracy, and hence to compare different training methods and approximation structures.

11.5 Example: Online Steepest Descent Training of a Neural Network

For online function approximation, we must choose how we will process the data that we gather online. Here, we simply use $G_k = \{(x(k), y(k))\}$ so that we acquire and process one data pair at each time step. We will assume that $\bar{N} = 1$ so that there is only one output and hence $y(k)$ is a scalar (the development is similar for many outputs). Once again we will train the neural network to match the function in Figure 9.10.

We will use a single hidden layer neural network. Recall that ϕ_j, $j = 1, 2, \ldots, n_1$ denotes the output of the j^{th} neuron in the hidden layer, and b_j is its bias. We defined

$$w^j = [w_{1,j}, w_{2,j}, \ldots, w_{n,j}]^\top$$

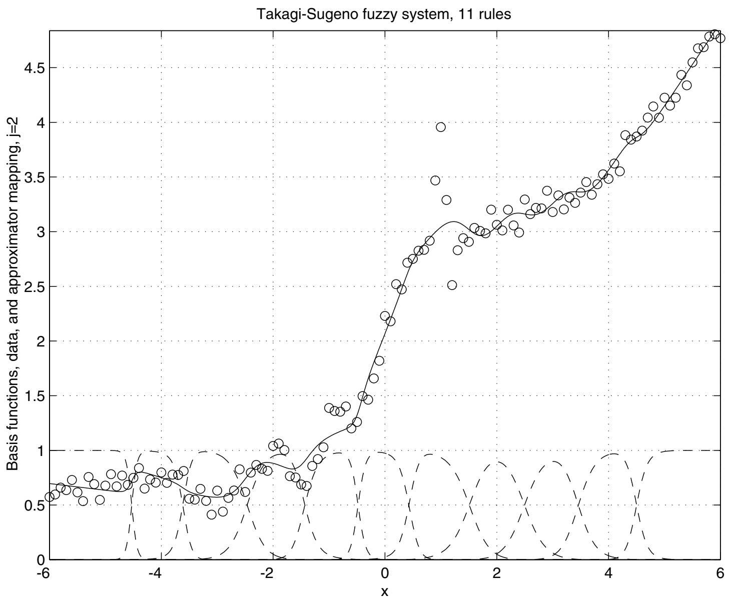

Figure 11.16: Levenberg-Marquardt training of a Takagi-Sugeno fuzzy system, mapping shape at iteration $j = 2$, different initialization.

so $\phi_j = f(b_j + (w^j)^\top x)$. Here, for every neuron in the hidden layer, we use the activation function

$$f(\bar{x}) = \frac{1}{1 + \exp(-\bar{x})}$$

Recall that w_j, $j = 1, 2, \ldots, n_1$ denotes a weight in the output layer and b is the bias for the output layer neuron. We have $w = [w_1, w_2, \ldots, w_{n_1}]^\top$. With a linear activation function in the output layer, the approximator is

$$y = F_{mlp}(x, \theta) = b + \sum_{j=1}^{n_1} w_j \left(f(b_j + (w^j)^\top x) \right)$$

If we tune all the parameters of this approximator, both the ones that enter linearly and in a nonlinear fashion, we let

$$\theta = [(w^1)^\top, b_1, (w^2)^\top, b_2, \ldots, (w^{n_1})^\top, b_{n_1}, w^\top, b]^\top$$

In this case, if n is the number of inputs to the approximator, the number of parameters to be tuned is $p = nn_1 + n_1 + n_1 + 1 = n_1(n + 2) + 1$.

Here, we use the steepest descent training method to update the parameter vector $\theta = [\theta_1, \theta_2, \ldots, \theta_p]^\top$ and use a constant step size. We will only execute one iteration of the gradient update formula for each piece of data gathered; hence, we are aligning gradient iterations with time steps. In particular, for our

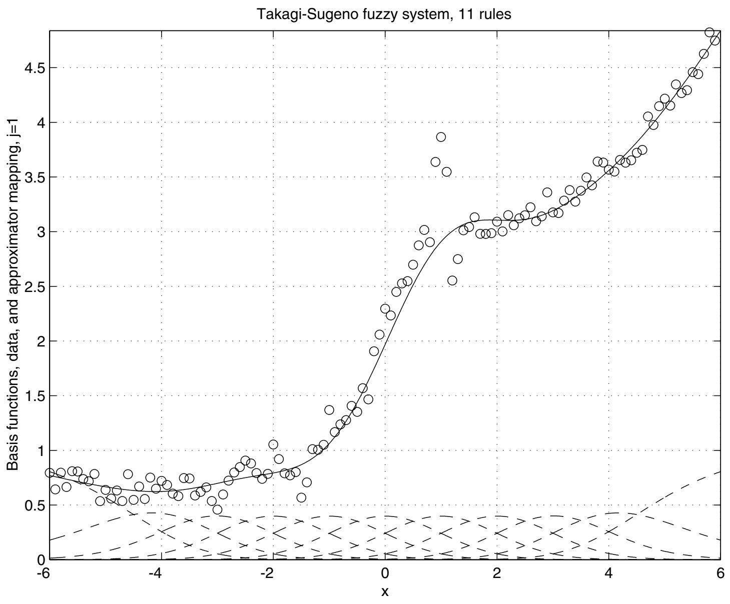

Figure 11.17: Levenberg-Marquardt training of a Takagi-Sugeno fuzzy system, mapping shape at iteration $j = 1$, different initialization.

online case, our update formula is given in Equation (11.6), which we repeat here as

$$\theta(k+1) = \theta(k) - \lambda \left. \frac{\partial J(\theta, G_k)}{\partial \theta} \right|_{\theta=\theta(k)}$$

where $\lambda > 0$ is the constant step size. Recall that we have a cost function given by Equation (11.1), which in our case is

$$J(\theta, G_k) = \frac{1}{2} \left(y(k) - F_{mlp}(x(k), \theta)\right)^2$$

From this, in order to fully specify the parameter update law, it is clear that we must provide

$$\frac{\partial J(\theta, G_k)}{\partial \theta}$$

for this case. This is what we do next.

11.5.1 Update Formulas

Clearly, we have

$$\begin{aligned}\frac{\partial J(\theta, G_k)}{\partial \theta} &= \frac{1}{2} \frac{\partial}{\partial \theta} \left(y(k) - F_{mlp}(x(k), \theta)\right)^2 \\ &= -\epsilon \frac{\partial F_{mlp}(x(k), \theta)}{\partial \theta}\end{aligned}$$

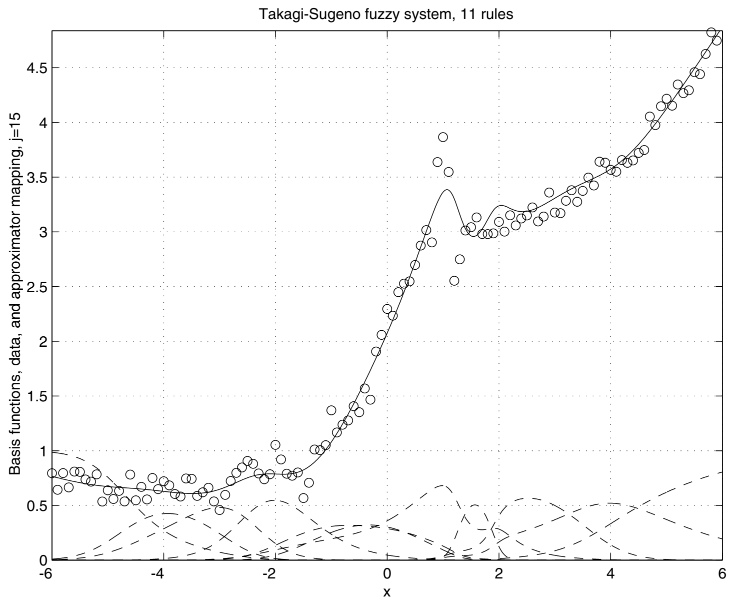

Figure 11.18: Levenberg-Marquardt training of a Takagi-Sugeno fuzzy system, mapping shape at iteration $j = 15$, different initialization.

where we let the scalar $\epsilon(k) = y(k) - F_{mlp}(x(k), \theta)$. Now, using the definition of the approximator structure

$$\frac{\partial F_{mlp}(x(k),\theta)}{\partial \theta} = \frac{\partial}{\partial \theta}\left(b + \sum_{j=1}^{n_1} w_j f\left(b_j + (w^j)^\top x\right)\right)$$

At this point, it is convenient to develop the update formula for different components of the θ vector individually, since there will be special cancellations in each case. First, we derive the case for the weights of the hidden layer, then its biases. Then we will proceed to the case for the parameters that enter linearly, the weights and bias of the output layer.

Development of update formulas simply requires the chain rule from calculus and some algebra.

To help avoid confusion with the use of the indices, we will use j^* and i^* to denote the particular parameter value that we seek to derive the update formula for. Hence, we seek to find, for $j^* = 1, 2, \ldots, n_1$, and $i^* = 1, 2, \ldots, n$,

$$\frac{\partial F_{mlp}(x(k),\theta)}{\partial w_{i^*,j^*}} = \frac{\partial}{\partial w_{i^*,j^*}}\left(b + \sum_{j=1}^{n_1} w_j f\left(b_j + (w^j)^\top x\right)\right)$$

Now, taking the partial we find, using the chain rule from calculus,

$$\frac{\partial F_{mlp}(x(k),\theta)}{\partial w_{i^*,j^*}} \quad = \quad w_{j^*}\frac{\partial}{\partial w_{i^*,j^*}} f\left(b_{j^*} + (w^{j^*})^\top x\right)$$

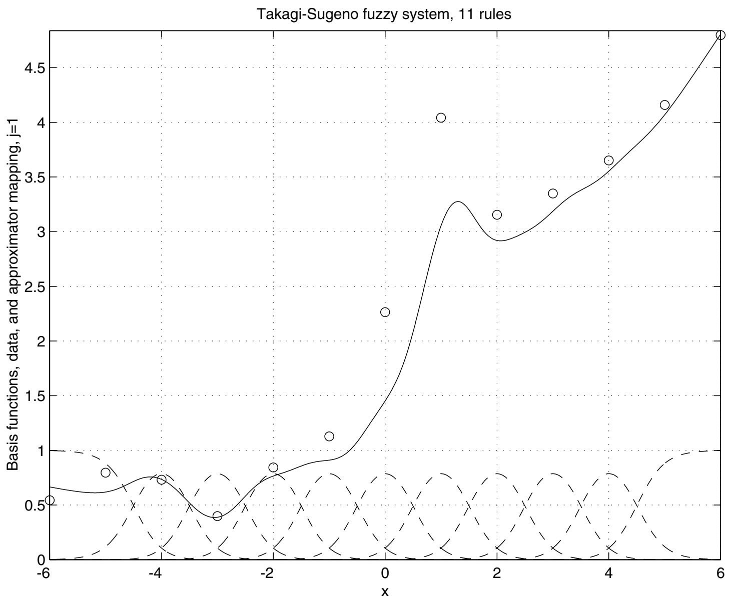

Figure 11.19: Levenberg-Marquardt training of a Takagi-Sugeno fuzzy system, mapping shape at iteration $j = 1$, $M = 13$.

$$= \quad w_{j^*} \frac{\partial f}{\partial \bar{x}_{j^*}} \frac{\partial \bar{x}_{j^*}}{\partial w_{i^*,j^*}}$$

Here, $\bar{x}_{j^*} = b_{j^*} + (w^{j^*})^\top x$ and note that using simple rules from calculus, with the above definition for the logistic function,

$$\frac{\partial f}{\partial \bar{x}_{j^*}} = f(\bar{x}_{j^*})(1 - f(\bar{x}_{j^*}))$$

If we had used the hyperbolic tangent for the activation functions f, then

$$\frac{\partial f}{\partial \bar{x}_{j^*}} = 1 - (f(\bar{x}_{j^*}))^2$$

Returning to the logistic function case, notice that

$$\frac{\partial \bar{x}_{j^*}}{\partial w_{i^*,j^*}} = x_{i^*}$$

so

$$\frac{\partial F_{mlp}(x(k), \theta)}{\partial w_{i^*,j^*}} = w_{j^*} f(\bar{x}_{j^*})(1 - f(\bar{x}_{j^*}))x_{i^*}$$

Hence, the update formula for the weights in the hidden layer is

$$w_{i,j}(k+1) = w_{i,j}(k) + \tag{11.23}$$
$$\lambda\epsilon(k) w_j f\left(b_j(k) + (w^j)^\top(k)x(k)\right)\left(1 - f\left(b_j(k) + (w^j)^\top(k)x(k)\right)\right) x_i(k)$$

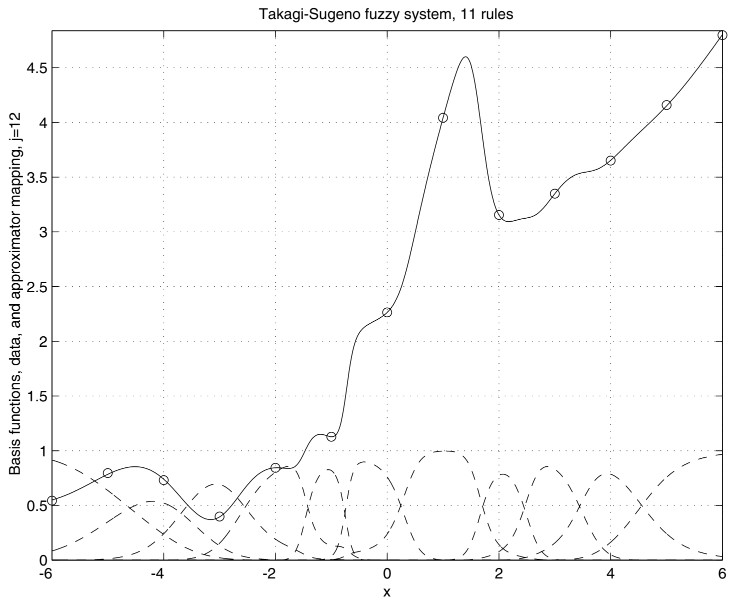

Figure 11.20: Levenberg-Marquardt training of a Takagi-Sugeno fuzzy system, mapping shape at iteration $j = 12$, $M = 13$.

for $j = 1, 2, \ldots, n_1$, and $i = 1, 2, \ldots, n$, where $\epsilon(k) = y(k) - F_{mlp}(x(k), \theta(k))$.

Next, we will derive the update formula for the biases that enter the n_1 neurons in the hidden layer. For this, for $j^* = 1, 2, \ldots, n_1$,

$$\begin{aligned}\frac{\partial F_{mlp}(x(k),\theta)}{\partial b_{j^*}} &= w_{j^*}\frac{\partial}{\partial b_{j^*}} f\left(b_{j^*} + (w^{j^*})^\top x\right)\\ &= w_{j^*}\frac{\partial f}{\partial \bar{x}_{j^*}}\frac{\partial \bar{x}_{j^*}}{\partial b_{j^*}}\\ &= w_{j^*} f(\bar{x}_{j^*})(1 - f(\bar{x}_{j^*}))\end{aligned}$$

Note that

$$\frac{\partial \bar{x}_{j^*}}{\partial b_{j^*}} = 1$$

Hence, we get the update formula

$$\begin{aligned}&b_j(k+1) = b_j(k) + \\ &\lambda\epsilon(k) w_j(k) f\left(b_j(k) + (w^j)^\top(k)x(k)\right)\left(1 - f\left(b_j(k) + (w^j)^\top(k)x(k)\right)\right)\end{aligned} \tag{11.24}$$

for $j = 1, 2, \ldots, n_1$, where $\epsilon(k) = y(k) - F_{mlp}(x(k), \theta(k))$.

Next, we derive the update formula for the n_1 weights in the output layer. For this, for $j^* = 1, 2, \ldots, n_1$,

$$\frac{\partial F_{mlp}(x(k),\theta)}{\partial w_{j^*}} = f\left(b_{j^*} + (w^{j^*})^\top x\right)$$

Hence, we get the update formula

$$w_j(k+1) = w_j(k) + \lambda\epsilon(k) f\left(b_j(k) + (w^j)^\top(k)x(k)\right) \tag{11.25}$$

for $j = 1, 2, \ldots, n_1$, where $\epsilon(k) = y(k) - F_{mlp}(x(k), \theta(k))$. Notice that this is the update formula for parameters that enter linearly, and you will generally find such a relationship for this case.

Finally, we derive the scalar update formula for the bias b in the output layer. For this

$$\frac{\partial F_{mlp}(x(k), \theta)}{\partial b} = 1$$

Hence, we get the update formula

$$b(k+1) = b(k) + \lambda\epsilon(k) \tag{11.26}$$

where $\epsilon(k) = y(k) - F_{mlp}(x(k), \theta(k))$.

To summarize, the update formulas for $\theta(k)$ are given by Equations (11.23), (11.24), (11.25), and (11.26). Clearly, while we use only one constant step size, you could use different ones for the different update formulas.

Notice that, as a practical computational issue, there are many shared calculations that are used in the update formulas. It is for this reason that it is probably best to first update the output layer bias, the output layer weights, the hidden layer biases, then finally the hidden layer weights (and then each update can use some of the calculations needed for the previous update).

11.5.2 Parameter Constraints and Initialization

Notice that for the update formulas we derived, there are no particular values of parameters that will cause, for instance, the functions on the right side of the update formulas to be undefined (which could cause, e.g., a divide-by-zero error). Hence, we will not have to constrain the parameters to avoid such situations. Moreover, in this simple example, we will not assume that, due to implementation concerns, the parameters must lie in certain bounded regions. For this reason, we will not put any constraints on the parameter update laws from a parameter constraint set. We emphasize, however, that generally the more information you have about the underlying function, the more you tend to know about how to initialize the approximator. Here, for the sake of illustration, we assume that we know nothing useful for the initialization (even though we could certainly analyze the data to learn some useful ideas for initialization, just as we have done in Section 10.5).

How do we initialize the algorithm? That is, how do we specify $\theta(0)$? There are many ways to choose this, but often in practice the parameters are simply chosen to be random small values (here in one case we choose $\theta_i(0)$ to be uniformly distributed on $[-0.1, 0.1]$). This often tends to be a good choice for several reasons. First, we get some initial random distribution of the biases that place the sigmoid functions across at least some small region of the space (sometimes, if you know the range of possible values on some input space a priori, then you can spread the sigmoids randomly across this range). Next, by choosing the weights to be small but random, we start with "steps" that are going both up and down with small slopes and this tends to make sure that the gradient is not too small initially. Finally, the small values for the output layer provide something close to picking the values at zero, which as we saw in the recursive least squares case, can be a good choice.

There are many cases for practical applications where you can determine what may be a better initialization than simply using small random values. For instance, in Section 10.5, we chose initial values for the parameters of the approximator that enter in a nonlinear fashion (i.e., the hidden layer weights and biases) in a way that when it was tuned with the recursive least squares method, it determined a good approximation to the function after 300 iterations. It did this whether we chose the initial values for the parameters that enter linearly (the output layer weights and bias) as all zero, or if we used values perturbed off the ones that batch least squares finds. Such initialization by some educated guessing at the nonlinear part and using batch least squares to specify the linear part is generally a good approach and one that we will study here. We must keep in mind, however, that in practical applications you are sometimes limited by how many data are available a priori so that initialization with batch least squares is not always possible. It is for this reason that we will also study the case where we simply pick the parameters that enter linearly to be zero.

11.5.3 Approximator Tuning Results: Effects of Step Size

For our example, we pick $n_1 = 25$, the same as we have studied in the recursive least squares case in Section 10.5. Notice that now, however, we will tune all 76 parameters of the neural network. Tuning this many parameters is probably not necessary for this problem to get a reasonable level of accuracy (e.g., consider the similar effects on the shape of the nonlinearity for the weights in the hidden and output layers), but we will use this example simply for illustration. Without much tuning, we picked $\lambda = 0.1$ or $\lambda = 0.01$ to illustrate the differences in the algorithm's behavior (and we note that if you pick it too much larger, the algorithm will diverge in some cases, as it did for the simple scalar quadratic example considered earlier).

When you use the batch least squares initialization, with $\lambda = 0.1$, you get the results shown in Figure 11.21 for the first 10 steps, in Figure 11.22 for the last 10 steps, and in Figure 11.23 at $k = 1000$ steps. Notice that the algorithm quickly tunes the shape to be a reasonable approximation, but that it does not ultimately achieve the kind of approximation accuracy that was achieved with the recursive least squares method in Section 10.5, even though it has what is most likely a better initialization (the actual values found from batch least squares, rather than the perturbed ones used there).

For a fixed step size, under very general conditions, asymptotically the map will "oscillate" by persistently trying to match the most recent data. Smaller step sizes result in smaller asymptotic oscillations, but slower convergence.

If you examine the shapes in Figure 11.22 for the last 10 steps, you find that the accuracy found at $k = 1000$ is also found at earlier steps and the shape changes at each iteration try to accommodate the new piece of training data

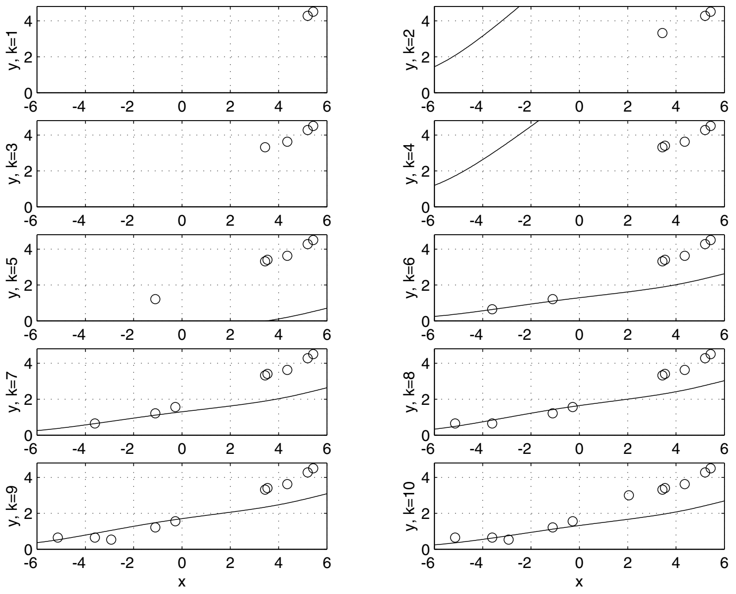

Figure 11.21: Steepest descent training of a neural network, mapping shapes for first 10 steps, batch least squares initialization, step size $\lambda = 0.1$.

(hence, the result in Figure 11.23 should only be taken as representative of the shapes found). The shape is still moving around at $k = 1000$ (and will for higher numbers of iterations also); it is not fixed at that point.

Next, if you use the batch least squares initialization, with $\lambda = 0.01$, you get the results shown in Figure 11.24 for the first 10 steps, in Figure 11.25 for the last 10 steps, and in Figure 11.26 after 1000 steps. Notice that with a smaller value for λ, the shape initially changes slowly and also near the end. Basically, the algorithm is less aggressive in trying to match each new piece of training data. This may be a desirable characteristic of an algorithm for online operation in some applications. Generally, larger step sizes will tend to force the method to pay more significant attention to each new piece of data, while smaller ones allow for it to partially ignore new data. There is generally a good choice that will allow the algorithm to slowly shape the nonlinear mapping as new information is gathered, allowing new information to partially reshape the nonlinearity, but not too much so that the information encountered in the past is not forgotten (some think of the algorithm as being "greedy" in seeking to achieve the minimization, which in this case means that it tries to approximate the information provided by the new piece of training data, with the amount of greed proportional to the step size). Sometimes, to keep the shape from moving around too much at each step, you have to use a very small step size, and then generally you need more steps in the algorithm to get convergence.

Next, recall that we are presenting data to the algorithm where x is uniformly

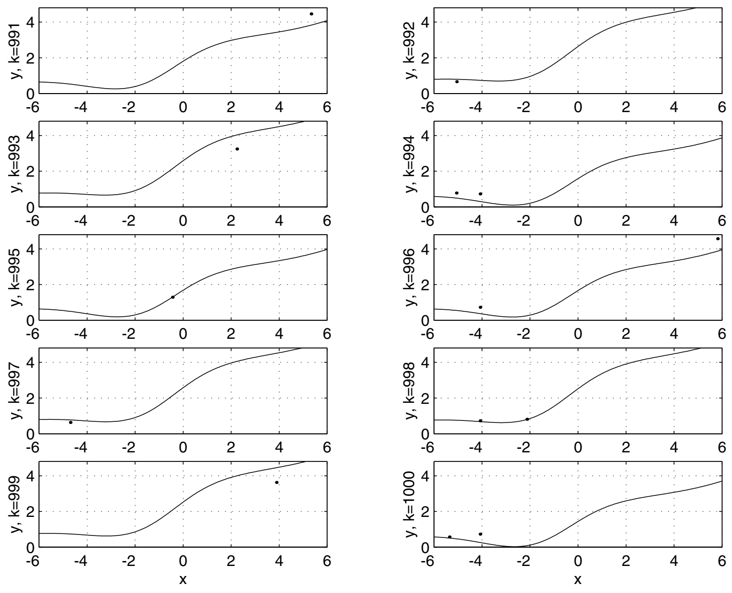

Figure 11.22: Steepest descent training of a neural network, mapping shapes for last 10 steps, batch least squares initialization, step size $\lambda = 0.1$.

distributed on $[-6, 6]$. Now, if we are unlucky and we only get data in one region of the x domain over several initial steps, then we generally will not get the kind of initial accuracy that you see in Figure 11.21. Clearly if it does not have data in certain regions, then it generally will do poor approximation in that region (of course you may get lucky and it may do a good extrapolation). Generally, the performance and convergence properties of the algorithm depend on the order of presentation of the training data. Finally, note that while we have run the algorithm for many iterations and the parameters did not diverge, we must emphasize that this does not *prove* that they will not; it could be that after only a few more iterations they will diverge. Generally, you must be very careful to ensure boundedness for parameters that you adjust online and one way to do this is to use a parameter constraint set (which we did not do here just to keep things simple).

11.5.4 Approximator Tuning Results: Effects of Initialization

In this subsection, we will assume that $\lambda = 0.01$. First, we initialize the parameters that enter linearly to be all zero, as we did for the recursive least squares method in Section 10.5. Using this initialization, we get the approximator mapping shown in Figure 11.27 after 1000 iterations (the plots for the first and last ten steps are omitted as they are similar to the case above where we initialized

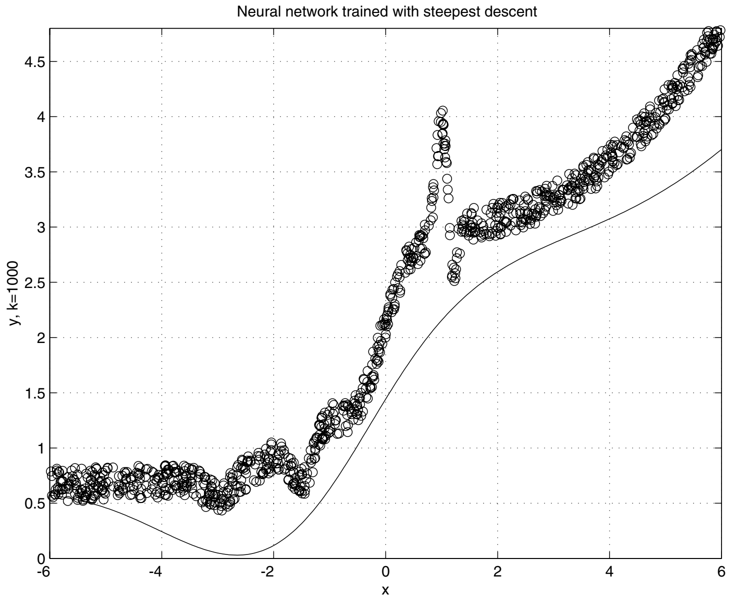

Figure 11.23: Steepest descent training of a neural network, mapping shape after 1000 steps, batch least squares initialization, step size $\lambda = 0.1$.

with batch least squares). At 1000 iterations, this approximator shape provides an approximation accuracy that is clearly close to that shown in Figure 11.23. It seems that in this case for this choice of training data (which is random), the steepest descent method ultimately picked the parameters just as well as when it had (what was probably) a better initialization. We can say that it seemed to overcome the poor initialization in this case (of course, we cannot always expect this).

When we initialize with all small random values, the results are shown in Figure 11.28 after 1000 steps. The mapping shapes for the first 10 iterations are not shown, but basically, it is as you would guess: little progress is seen in coming up with a good approximation since λ is small and the initialization is not very good. The mapping shapes for the last 10 iterations are close to the one shown in $k = 1000$ in Figure 11.28, showing that it appears that the mapping shape has converged. Hence, it seems that we have found that this initialization, which is often used when you know nothing better about how to initialize the mapping, results in poorer approximation accuracy as compared to the others.

11.5.5 Can We Improve Approximation Accuracy?

Well, there are many things that you can try, but the choices depend on the particular application. For the simple example we have been studying, there is

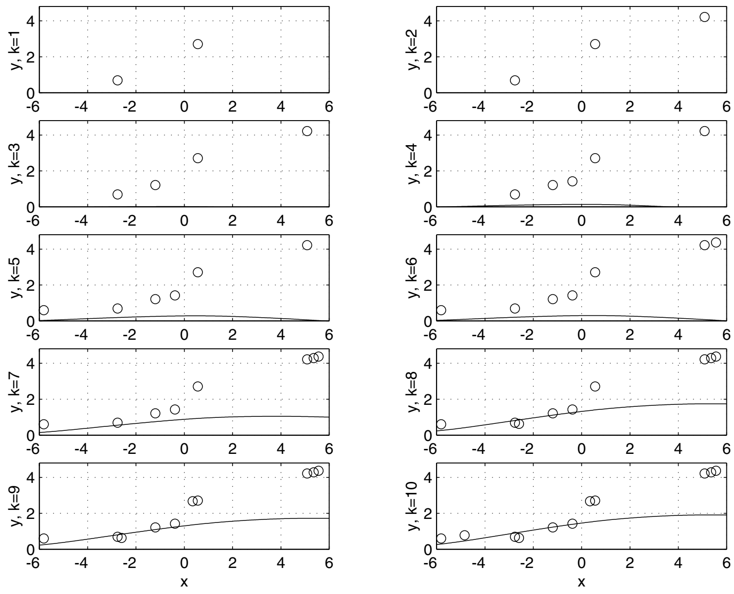

Figure 11.24: Steepest descent training of a neural network, mapping shapes for first 10 steps, batch least squares initialization, step size $\lambda = 0.01$.

clearly room for improvement of approximation accuracy, and there are several options that become apparent after completion of the above investigations.

First, you could try to use a diminishing step size rule, such as the one that starts with a certain step size and then decreases it to some minimum value. For some applications, this can ensure that the initial data are quickly used to tune the approximator to get a reasonable accuracy, but then the step size decreases so that the oscillations in the shape of the mapping do not occur at later iterations after it has learned more about the shape.

Second, you could try processing more than one data pair per step, for instance, by "windowing" the data. Then, at each iteration, you would execute several iterations of the gradient method to try to get the approximator to match the function (often you would simply terminate the iterations after some fixed number, since you will often be constrained by processor resources; however, other times you could use a termination criterion at each step). This can help alleviate the problems with the algorithm being too aggressive in seeking to match the data pair just encountered. In such an approach, you could weight the old data as being less important than the new data, just as we did in the least squares approach with a forgetting factor. To do this, you would need to add weighting factors to the cost function you are trying to minimize. Overall, such an approach can offer improved accuracy but you are certainly paying for it in computational complexity.

Third, you could try to use a different gradient method such as the Levenberg-

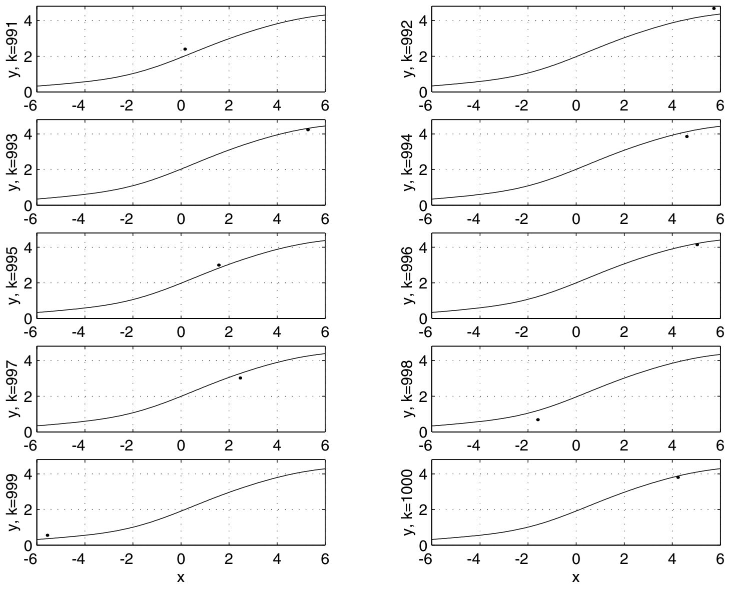

Figure 11.25: Steepest descent training of a neural network, mapping shapes for last 10 steps, batch least squares initialization, step size $\lambda = 0.01$.

Marquardt method, and again you may want to consider serially processing batches of data as we discussed above (with the computational complexity generally increasing with an increase in the batch size). Why might this have a chance at improving approximation accuracy? First, it should try to approximate a Newton method so that it should get fast convergence, but even with tuning, you may get the type of behavior seen with the steepest descent method where the mapping shape oscillates. Second, experience has shown that the Levenberg-Marquardt approach is generally better than the steepest descent algorithm for offline training, so we might find the same or similar benefits for online training. At the same time, using a more sophisticated method can raise other problems, such as ensuring that the inverse for the Levenberg-Marquardt update formulas can be computed.

11.5.6 Local Vs. Global Tuning/Learning

It is interesting to consider how the mapping shape changes over time as we have done in the recursive least squares case. To do this, we will return to the first case where we had initialized with the batch least squares and had $\lambda = 0.1$ (see Figures 11.21, 11.22, and 11.23), since this will most dramatically illustrate the ideas here. Figure 11.21 shows the approximator nonlinearity for the first 10 steps, and notice that for the first 5 steps, the approximator does not have much data and hence the quality of approximation is quite poor. Next,

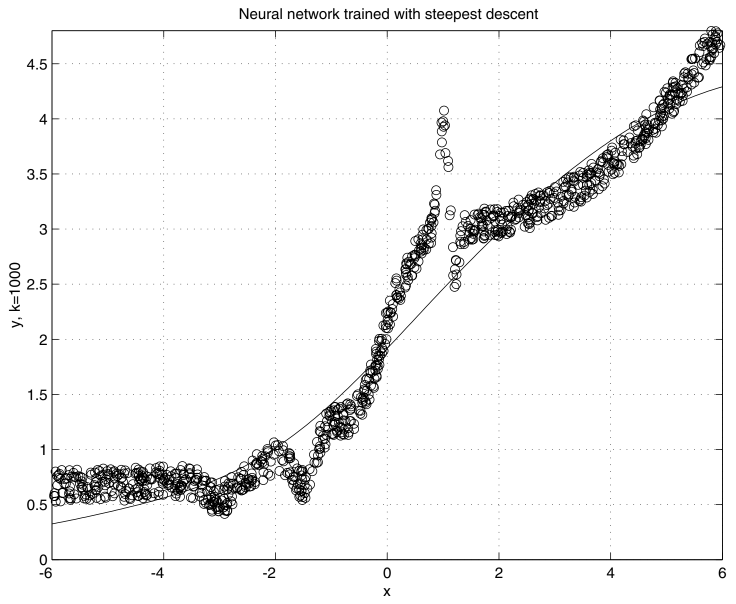

Figure 11.26: Steepest descent training of a neural network, mapping shape after 1000 steps, batch least squares initialization, step size $\lambda = 0.01$.

however, notice that at $k = 6$ a data pair is obtained, and the approximator is tuned to provide a reasonable approximation to the given data (although it does not match the data near $x = 5$ very well). At times $k = 7, 8, 9$, data are obtained on the left side and the approximator shape changes very little. Now, at $k = 10$, a data point is obtained on the right, but it does not modify the approximator shape much to try to improve the accuracy, because the step size is relatively small.

For some approximator structures trained with some methods, learning in the present can destroy what has been learned in the past (the stability-plasticity dilemma).

What would we have liked to see in this initial sequence? Well, by $k = 4$, we had data on the right side that the approximator did not match very well, and we would have liked to see it do better. Then, when at steps $k = 5, 6, 7, 8, 9$, it got data on the left side we would have liked to see it let the approximator pass through the data gathered earlier, but also force the approximator to pass through these new data. Then, when the data pair is gathered at $k = 10$, we would like to have seen it adjust the approximator on the right, without disturbing (forgetting) what it had already done on the left. In summary, we would have liked it to have made “local” adjustments to the approximator nonlinearity, depending on where it gathered data, so that it incrementally learns the proper shape.

Such problems arise for a variety of reasons, such as step size choice, the choice of using a gradient method, and only processing one data point at each iteration; however, one other significant contributing factor can be the choice of the approximator structure. For neural networks with sigmoid nonlineari-

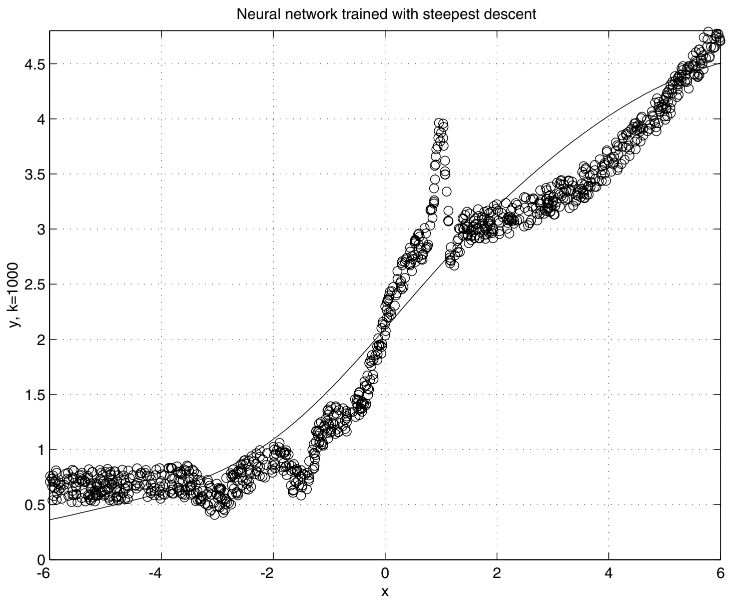

Figure 11.27: Steepest descent training of a neural network, mapping shape after 1000 steps, parameters that enter linearly initialized to zero.

ties, and other approximators, a change in one parameter can change the whole mapping shape (like the bias on the output layer, which shifts the whole plot vertically up and down) so that when it should only be shaping the mapping locally, where it got the training data it does so "globally." At times, this is not a problem as the method can sometimes be designed so that it shapes the nonlinearity appropriately, or since the neural network is a universal approximator, it can provide for local learning too if it picks the parameters properly. Sometimes, however, it is difficult to achieve this with the neural network or with other approximator structures. At times, it can be beneficial to *force* a type of local learning to help overcome this problem by picking the nonlinear part of the approximator to have functions that approximately have "local support" (i.e., they are only positive in a certain domain of the input space) so that only local adjustments are made. Radial basis function neural networks can achieve local support as well as the Takagi-Sugeno fuzzy system with Gaussian input membership functions.

11.6 Clustering for Classifiers and Approximators

It is important to realize that gradient methods are very general and applicable to many optimization problems you can encounter in engineering. In particular,

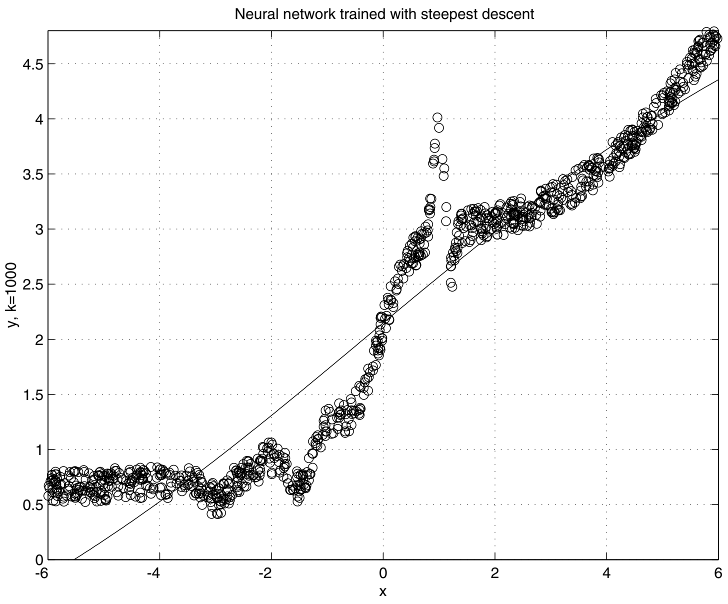

Figure 11.28: Steepest descent training of a neural network, mapping shape after 1000 steps, parameters initialized to small random values.

they have been found to be useful in many roles in the development of intelligent control systems. One area that they can be particularly useful is in the tuning of nonlinearities to partition data vectors into different "classes" (the data vectors can be numeric representations of the parameters of many different kinds of objects, from computer vision and image processing data, to speech signals and plant input-output data). These can then be used in "classifiers" that can take a given input vector and indicate which of a finite set of classes that input corresponds to. There is a wide variety of "pattern recognition" problems that can use classification methods. In some approaches, the data vectors are grouped into "clusters" that partition the data. Then, when an input is given, the membership in each cluster is determined and the one it best matches is declared to be the cluster that the data vector belongs to. In this way, even if we get an input vector that is somewhat different from the center of the cluster (e.g., due to noise), it can still correctly classify the object to the proper class.

There are also times when it is useful to use clustering to tune a portion of an approximator to solve a function approximation problem. For instance, using the "input portion" x of the training data set G, we can form clusters around similar x vectors. Then we can use these clusters in the nonlinear portion of the approximator (i.e., in the ϕ function) and train the remaining linear portion of the approximator to solve a function approximation problem. Indeed, the classification problem discussed above can be thought of as a type of function approximation problem where the output portion of the training data y simply indicates which class x belongs to, where $(x, y) \in G$. It is for this reason that many different approximators and training methods of the previous sections can be used for the classification task.

In this section, after we explain how to use approximators as classifiers, we show how to form clusters around data. The resulting methods will be shown to provide either classifiers or function approximators. You can think of the methods of this chapter as a different approach to tune nonlinear in the parameter approximators. Here, you first use a cost function that characterizes quality of clustering to get the clusters, and hence the nonlinear portion of the approximator. Then you can use a linear least squares criterion that characterizes approximation accuracy to specify a least squares method to find the parameters that enter linearly.

Classifiers indicate which group of data (cluster) an input vector belongs to (is associated with).

11.6.1 Using Approximators to Solve Classification Problems

We must emphasize that *any* of the methods developed in the previous sections can be used as function approximators to solve a classification problem. To explain how this is done in a bit more detail, note that the key to formulating the classification problem as a function approximation problem is to start by picking the data set and this will suggest whether you use a single- or multiple-output approximator.

Single-Output Classifiers

One way is to assume that we have n_c different classes that objects can belong to. Here, our objects are characterized by (parameterized by) a vector of n numbers. Suppose that these classes are simply labeled with numbers $1, 2, \ldots, n_c$. Suppose that we have M examples that pair objects with their classes, such as $(x(i), y(i))$ where $x(i)$ is a specific data vector and $y(i) \in \{1, 2, \ldots, n_c\}$ is its class. Clearly, we can use these data to specify the training data set G and the resulting approximator (which has n inputs and one output) can be trained to classify the data. In such an approach, the output will be a scalar and you will have to specify a method to determine which integer $1, 2, \ldots, n_c$ the output is closest to in order to classify it into one of the finite number of possible classes.

Note that the issue of approximator structure choice can be very important in the design of a classifier. For instance, suppose that $n_c = n = 2$ and that the input space is simply split by a line where vectors on one side of the line belong to the first class and the ones on the other side belong to the second class. In this case, it may be good to use a neural network with a logistic function since it can then be tuned to provide for the splitting of the space along the line mentioned above. If the two classes were defined by being in or out of a circular region, then a different nonlinearity might work better. Which one? Usually the choice is very application-dependent and requires significant insight into the problem; however, in this case you may consider a normalized Gaussian function (like ξ_i that we had used for the fuzzy systems) since it can then provide a function that naturally comes on in a circular region, and a function that comes on everywhere but in a circular region. (Develop and sketch one to convince yourself of this.)

Similar issues in structure choice arise in the multi-output classifiers that we discuss next.

Multiple-Output Classifiers

Another perhaps more common way to formulate the classification problem as a function approximation problem is to construct a multi-output approximator with n_c outputs. View this multi-output system as n_c multi-input single-output systems. Consider how to train the j^{th} output to classify whether the input vector x is a member of class j where $j \in \{1, 2, \ldots, n_c\}$. Suppose that we have M examples that pair objects with their classes but in a different way than in the last subsection. Here, suppose that we have M_j data pairs $(x(i), y(i))$ where $x(i)$ is a specific data vector and $y(i) \in \{0, 1\}$ where if $x(i)$ is in class j, then $y(i) = 1$ and if it is not in class j, then $y(i) = 0$. The entire data set for training the classifier is simply the union of the data sets used to train each classifier (then $M = \sum_{j=1}^{n_c} M_j$ for the data set G). Now, suppose that we have trained the n_c approximators with these data sets.

How does the classification process work? Suppose that we consider only the j^{th} classifier that tries to decide if the input vector is in the j^{th} class. Suppose that we call the approximator that was trained for this task $F_j(x, \theta)$ (of course, θ is the parameter vector that resulted from the training process). For a given x, we could test if $F_j(x, \theta) \geq 0.5$ and if it is, then we could indicate that x has class j (and if it is not, then it is not of class j). There are several possible problems with such an approach. First, for a given x there may be more than one output that is greater than 0.5 so that a single vector could be classified as being in two different classes (and often you would not want this). Second, it is possible that there is no j such that the value of $F_j(x, \theta) \geq 0.5$ and in this case, it does not know how to classify.

Hence, the common approach is to pick the output, say j^*, that has a maximum value and then indicate that x has class j^*. Mathematically, we say that we decide that the given input x is of class j^* where

$$j^* = \arg \max_{j=1,2,\ldots,n_c} \{F_j(x, \theta)\}$$

(if there is more than one value that has the maximum value, then you simply arbitrarily pick one). Note that with this approach, we will always have a unique classification. But, of course, if all the values of $F_j(x, \theta)$ are close to zero, we may not be very confident in the classification. In fact, in some applications it may make sense to use the output of the classifier to indicate the confidence in the classification.

11.6.2 Clustering Methods: Gradient Approaches

In this section we take a different approach to the classification problem from in the last subsection. Here, we specify functions that are designed to partition data in certain ways and try to adjust the parameters of these functions so that they group the data into clusters. We do not explicitly focus on a function approximation problem; however, we note that these methods can be used with other methods to construct approximators (e.g., see the next section, where we couple a clustering method with a least squares approach to form a function approximator). The clustering methods of this section could be used in a similar role.

Cluster Functions

First, we give some examples of how to specify what we will call "cluster functions" that are nonlinear functions designed to partition data. There are a wide variety of possibilities for such functions and we only consider two here (the first one will be studied in more detail in the next section).

Polynomial-Based Function: Let

$$v^j = [v_1^j, v_2^j, \ldots, v_n^j]^\top$$

denote the j^{th} "cluster center" where $j = 1, 2, \ldots, R$. Let

$$p_j(x) = \left[\sum_{k=1}^{R} \left(\frac{|x - v^j|^2}{|x - v^k|^2} \right)^{\frac{1}{m-1}} \right]^{-1} \tag{11.27}$$

$j = 1, 2, \ldots, R$ be the "polynomial-based" cluster functions. Here, we must have $m > 1$. Note that m controls the "width" of all the clusters.

As an example, consider the case where $n = 1$ and $R = 3$. Let $m = 2$. A plot of p_j, $j = 1, 2, 3$ for the case where $v^1 = -5$, $v^2 = 0$, and $v^3 = 5$ is shown in Figure 11.29 (see top plot). We use a solid line for p_1, a dashed line for p_2, and a dotted line for p_3. Notice that the clusters provide "soft" partitions for the x domain. When one function is near one, the others are near zero. At the outer edges of the domain of x (i.e., for large $|x|$ values), the cluster function values all approach the same value.

A plot of p_j, $j = 1, 2, 3$ for the case where $v^1 = -3$, $v^2 = 6$, and $v^3 = 1$ is shown in Figure 11.30 (see top plot). In this case, notice that it also achieves a good partitioning of the x axis.

Gaussian-Based Function: For $j = 1, 2, \ldots, R$, let

$$\mu_j(x) = \prod_{i=1}^{n} \exp\left(-\frac{1}{2} \left(\frac{x_i - c_i^j}{\sigma_i^j} \right)^2 \right)$$

where c_i^j is the point in the i^{th} input x_i where the function achieves a maximum, and $\sigma_i^j > 0$ is the "width" of the function for the i^{th} input. (This is simply the Gaussian premise membership function used earlier for fuzzy systems.) We will

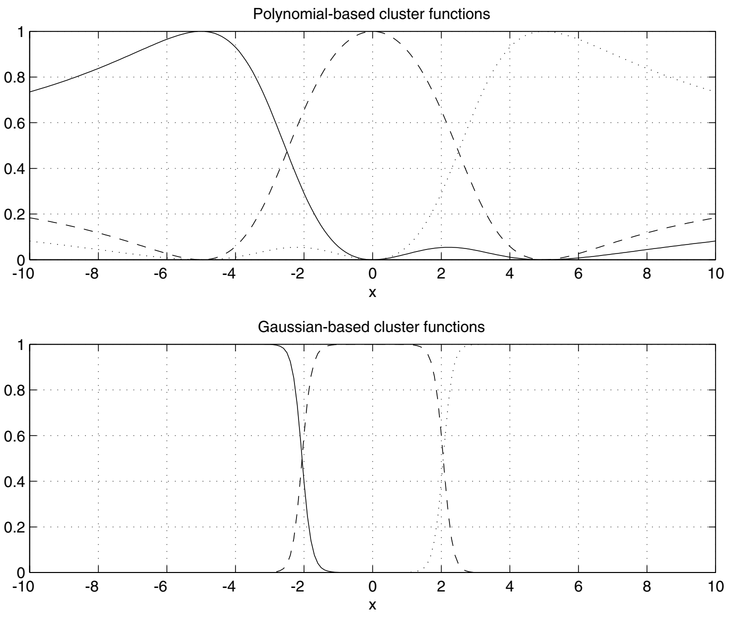

Figure 11.29: Polynomial and Gaussian-based cluster functions.

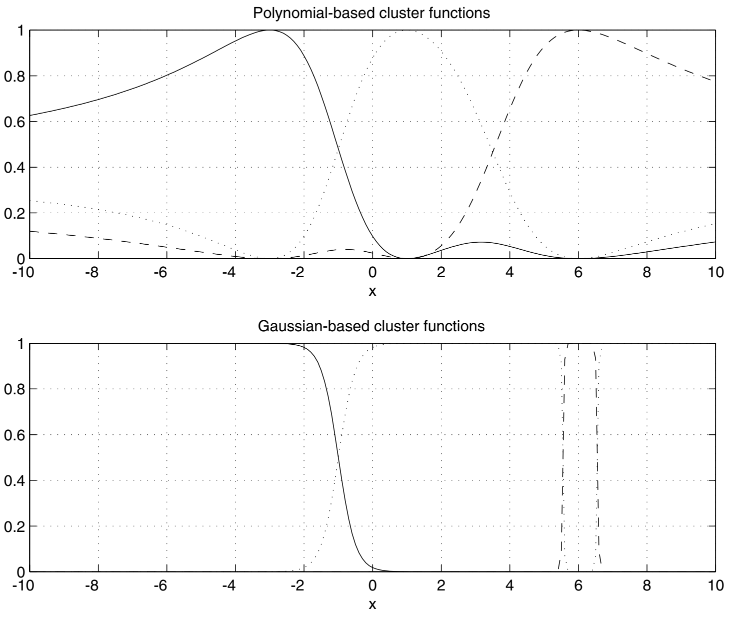

Figure 11.30: Polynomial and Gaussian-based cluster functions.

use this function to construct another type of cluster function. In particular, we "normalize the Gaussian functions" by letting

$$\xi_j = \frac{\mu_j(x)}{\sum_{i=1}^{R} \mu_i(x)}$$

$j = 1, 2, \ldots, R$. We can use these functions as cluster functions. Note that in Takagi-Sugeno fuzzy systems, we used them to turn on and off different consequent functions.

As an example, consider the case where $n = 1$ and $R = 3$. A plot of ξ_j, $j = 1, 2, 3$, for the case where $c_1^1 = -5$, $c_1^2 = 0$, and $c_1^3 = 5$, with $\sigma_1^1 = 1$, $\sigma_1^2 = 0.7$, and $\sigma_1^3 = 1$, is shown in Figure 11.29 (see bottom plot). We use a solid line for ξ_1, a dashed line for ξ_2, and a dotted line for ξ_3. Notice that these functions provide a different type of partitioning of the domain from the polynomial-based function. Besides the shapes being different, at the outer edges of the domain, all the points would be grouped together into the outermost cluster. Notice also that the widths for all the functions can be different, while in the polynomial-based case, the widths are all controlled by one parameter. This added flexibility may or may not be useful (and of course, the polynomial-based function can be modified to provide this characteristic).

It is also possible that a cluster of this type exhibits other shapes that may be useful. For instance, it can be the case that one cluster can come on (i.e., achieve a value near one) in more than one region of the space. However, note that due to the normalization (i.e., the division by the sum of the μ_i), the sum of the cluster function values at any one x point must be one. This ensures that as one cluster function increases, the others must decrease so that any input is in a cluster in varying amounts and never completely in more than one cluster.

As an example, a plot of ξ_j, $j = 1, 2, 3$, for the case where $c_1^1 = -3$, $c_1^2 = 6$, and $c_1^3 = 1$, with $\sigma_1^1 = 1$, $\sigma_1^2 = 0.1$, and $\sigma_1^3 = 1$, is shown in Figure 11.30 (see bottom plot). Compare this to the result from the polynomial-based function and notice that the result is quite different. Notice that here ξ_3 is very near one (i.e., it is on) for $x \in [0, 5]$ *and* $x \geq 7$. This characteristic of this cluster function could be useful in some applications (but may be bad for others) and provides for some interesting cluster shapes (not just the standard ones). For instance, in the case where $n = 2$, it is possible to have what you may call circular concentric clusters with a circle in the middle and doughnut-shaped clusters centered around it.

Clustering Cost Functions

Clustering via a gradient method involves adjusting functions representing clusters to minimize a measure of how the data are grouped and separated.

In this section we provide cost functions that we will seek to minimize to make the cluster functions partition the data.

Cost for Polynomial-Based Function: Consider the function

$$J(\theta) = \sum_{i=1}^{M} \sum_{j=1}^{R} (\mu_{ij})^m |x(i) - v^j|^2 \tag{11.28}$$

where $m > 1$, v^j are the cluster centers, typically $M >> R$, and the μ_{ij} are scalars. Here, the parameter vector θ holds both the cluster centers and the μ_{ij} scalars. Intuitively, the μ_{ij} for $i = 1, ..., M$ and $j = 1, ..., R$ are the grades of membership of $x(i)$ in the j^{th} cluster. Typically, you would require that for each $i = 1, 2, \ldots, M$, $\sum_{j=1}^{R} \mu_{ij} = 1$ so that the centers are placed so that no more than one will have a value of 1 at any point on the input domain (this forces us to solve a nonlinear optimization problem with constraints, and this will be discussed below). The terms $|x(i) - v^j|^2$ are included to try to get the clusters to be in the middle of the data. The $(\mu_{ij})^2$ values weight the terms $|x(i) - v^j|^2$ and they are adjusted so that the cluster centers will separate to find different groups of data.

Cost for Gaussian-Based Function: Recall that c_i^j is the point in the i^{th} input where the j^{th} cluster center reaches a maximum. Let

$$c^j = [c_1^j, c_2^j, \ldots, c_n^j]^\top$$

and think of this as a cluster center. Consider the function

$$J(\theta) = \sum_{i=1}^{M} \sum_{j=1}^{R} \xi_j(x(i)) |x(i) - c^j|^2 \tag{11.29}$$

Here, θ can hold both the c^j and σ_i^j values. Sometimes, however, it may be convenient to use the same value for all the σ_i^j and you may only want to adjust that single value. Alternatively, you may simply want to fix the values of the spreads, for instance, to be all the same value (this can simplify the optimization problem). Conceptually, the cost function is closely related to the one used for the polynomial-based function. Notice, however, that the clustering problem for the Gaussian-based function is a nonlinear optimization problem *without* constraints.

Cluster Adjustment Methods

For the cost function for the Gaussian-based function defined in the last section, it is possible to define a gradient update formula and use it to iteratively update the parameters of the cluster functions. The gradient $\nabla J(\theta)$ can be found and used with the methods of the last section. For instance, you may want to use a steepest descent or Levenberg-Marquardt method to solve the minimization problem. Clearly, standard initialization and termination issues for gradient algorithms are relevant. Also, we must emphasize that there are no convergence guarantees, so we will not know if we have found a global minimum of the cost function.

Specification of gradient update formulas for cluster functions requires the same general approach as for approximators.

The cost function for the polynomial-based function can also be minimized but in doing so, we must guarantee that the method ensures that for each $i = 1, 2, \ldots, M$, $\sum_{j=1}^{R} \mu_{ij} = 1$ (this is a constrained minimization problem). In the next section we will show one method to do this.

11.6.3 Fuzzy C-Means Clustering and Function Approximation

We can use clustering methods to tune the parameters that enter nonlinearly, and linear least squares to tune the parameters that enter linearly. This is just one of many possible "hybrid" training methods.

As indicated above, "clustering" is the partitioning of data into subsets or groups based on similarities between the data. Here, we will introduce a method to perform fuzzy clustering, where we seek to use fuzzy sets to define soft boundaries to separate data into groups. The methods here are related to conventional ones that have been developed in the field of pattern recognition. In the c-means approach, we continue in the spirit of the previous methods in that we use optimization to pick the clusters and, hence, the premise membership function parameters. The consequent parameters are chosen using the weighted least squares approach developed earlier. In this way, we show one way to use a clustering method in the construction of function approximators. The combined least squares-clustering method has been called "clustering with optimal output predefuzzification."

Clustering for Specifying Rule Premises

Fuzzy clustering is the partitioning of a collection of data into fuzzy subsets or "clusters" based on similarities between the data, and can be implemented using an algorithm called fuzzy c-means.

C-Means Cost Function: Fuzzy c-means is an iterative algorithm used to find grades of membership μ_{ij} (scalars) and cluster centers v^j (vectors of dimension $n \times 1$) to minimize the cost function

$$J(\theta) = \sum_{i=1}^{M} \sum_{j=1}^{R} (\mu_{ij})^m |x(i) - v^j|^2 \tag{11.30}$$

where $m > 1$ is a design parameter. Here, M is the number of input-output data pairs in the training data set G, R is the number of clusters (number of rules) we wish to calculate, $x(i)$ for $i = 1, ..., M$ is the input portion of the input-output training data pairs, $v^j = [v_1^j, v_2^j, \ldots, v_n^j]^\top$ for $j = 1, ..., R$ are the cluster centers, μ_{ij} for $i = 1, ..., M$, and $j = 1, ..., R$ is the grade of membership of $x(i)$ in the j^{th} cluster. Also, $|x| = \sqrt{x^\top x}$ where x is a vector. Intuitively, minimization of J results in cluster centers being placed to represent groups (clusters) of data.

The Premises and Fuzzy System to be Constructed: Fuzzy clustering will be used to form the premise portion of the If-Then rules in the fuzzy system we wish to construct. The process of "optimal output predefuzzification" (least squares training for consequent parameters) is used to form the consequent portion of the rules. We will combine fuzzy clustering and optimal output predefuzzification to construct multi-input single-output fuzzy systems. Extension of our discussion to multi-input multi-output systems can be done by repeating the process for each of the outputs.

In this section we utilize a Takagi-Sugeno fuzzy system in which the consequent portion of the rule-base is a function of the crisp inputs such that

$$\textbf{If } H^j \textbf{ Then } g_j(x) = a_{j,0} + a_{j,1}x_1 + \cdots + a_{j,n}x_n \tag{11.31}$$

where n is the number of inputs and H^j is an input fuzzy set given by

$$H^j = \{(x, \mu_{H^j}(x)) : x \in \mathcal{X}_1 \times \cdots \times \mathcal{X}_n\} \tag{11.32}$$

where $\mathcal{X}_i$ is the i^{th} universe of discourse, and $\mu_{H^j}(x)$ is the membership function associated with H^j that represents the premise certainty for rule j; and $g_j(x) = \underline{a}_j^\top \hat{x}$ where $\underline{a}_j = [a_{j,0}, a_{j,1} \ldots, a_{j,n}]^\top$ and $\hat{x} = [1, x^\top]^\top$ where $j = 1, \ldots, R$. The resulting fuzzy system is a weighted average of the output $g_j(x)$ for $j = 1, ..., R$ and is given by

$$F_{ts}(x, \theta) = \frac{\sum_{j=1}^R g_j(x)\mu_{H^j}(x)}{\sum_{j=1}^R \mu_{H^j}(x)} \tag{11.33}$$

where R is the number of rules in the rule-base. Next, we will use the Takagi-Sugeno fuzzy model, fuzzy clustering, and optimal output defuzzification to determine the parameters $\underline{a}_j$ and $\mu_{H^j}(x)$, which define the fuzzy system. We will do this via a simple example.

Clustering Algorithm

We first discuss the choice of some of the parameters and initialization. Then we will provide a method to iteratively update the cluster centers and μ_{ij}. To do this, we will use a simple example with the training data set

$$G = \left\{ \left(\begin{bmatrix} 0 \\ 2 \end{bmatrix}, 1 \right), \left(\begin{bmatrix} 2 \\ 4 \end{bmatrix}, 5 \right), \left(\begin{bmatrix} 3 \\ 6 \end{bmatrix}, 6 \right) \right\} \tag{11.34}$$

as shown in Figure 11.31. For the clustering method, we will only use the input portion of the training data; however, when we seek to form our approximator, we will also use the output data.

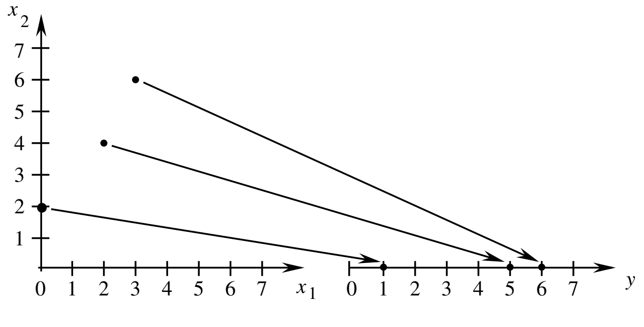

Figure 11.31: A simple training data set G.

Initialization: To specify the clustering algorithm, we first specify a "fuzziness factor" $m > 1$, which is a parameter that determines the amount of overlap of the clusters. If $m > 1$ is large, then points with less membership in the j^{th} cluster have less influence on the determination of the new cluster centers. Next, we specify the number of clusters R we wish to calculate. The number of clusters R equals the number of rules in the rule-base and must be less than or equal to the number of data pairs in the training data set G (i.e., $R \leq M$). We also specify the error tolerance $\epsilon_c > 0$, which is the amount of error allowed in calculating the cluster centers. We initialize the cluster centers v_0^j via a random number generator so that each component of v_0^j is no larger (smaller) than the largest (smallest) corresponding component of the input portion of the training data. The selection of v_0^j, although somewhat arbitrary, may affect the final solution.

For our simple example, we choose $m = 2$ (a typical choice) and $R = 2$, and let $\epsilon_c = 0.001$. Our initial cluster centers were randomly chosen to be

$$v_0^1 = \begin{bmatrix} 1.89 \\ 3.76 \end{bmatrix}$$

and

$$v_0^2 = \begin{bmatrix} 2.47 \\ 4.76 \end{bmatrix}$$

so that each component lies in between $x_1(i)$ and $x_2(i)$ for $i = 1, 2, 3$ (see the definition of G in Equation (11.34)).

Cluster Center Calculations: Next, we compute the new cluster centers v^j based on the previous cluster centers to try to minimize the cost function in Equation (11.30). The necessary conditions for minimizing J are given by using Lagrange multiplier theory as

$$v_{new}^j = \frac{\sum_{i=1}^{M} x(i)(\mu_{ij}^{new})^m}{\sum_{i=1}^{M}(\mu_{ij}^{new})^m} \tag{11.35}$$

where

$$\mu_{ij}^{new} = \left[\sum_{k=1}^{R}\left(\frac{|x(i) - v_{old}^j|^2}{|x(i) - v_{old}^k|^2}\right)^{\frac{1}{m-1}}\right]^{-1} \tag{11.36}$$

for each $i = 1, \ldots, M$ and for each $j = 1, 2, \ldots, R$ such that $\sum_{j=1}^{R} \mu_{ij}^{new} = 1$ (and $|x|^2 = x^\top x$). In Equation (11.36), we see that it is possible that there exists an $i = 1, 2, \ldots, M$ such that $|x(i) - v_{old}^j|^2 = 0$ for some $j = 1, 2, \ldots, R$. In this case, the μ_{ij}^{new} is undefined. To fix this problem, let μ_{ij} for all i be any nonnegative numbers such that $\sum_{j=1}^{R} \mu_{ij} = 1$ and $\mu_{ij} = 0$, if $|x(i) - v_{old}^j|^2 \neq 0$.

Using Equation (11.36) for our example with $v_{old}^j = v_0^j$, $j = 1, 2$, we find that $\mu_{11}^{new} = 0.6729, \mu_{12}^{new} = 0.3271, \mu_{21}^{new} = 0.9197, \mu_{22}^{new} = 0.0803, \mu_{31}^{new} = 0.2254,$

and $\mu_{32}^{new} = 0.7746$. We use these μ_{ij}^{new} from Equation (11.36) to calculate the new cluster centers

$$v_{new}^1 = \begin{bmatrix} 1.366 \\ 3.4043 \end{bmatrix}$$

and

$$v_{new}^2 = \begin{bmatrix} 2.5410 \\ 5.3820 \end{bmatrix}$$

using Equation (11.35).

Testing for Termination: Next, we compare the distances between the current cluster centers v_{new}^j and the previous cluster centers v_{old}^j (which for the first step is v_0^j). If $|v_{new}^j - v_{old}^j| < \epsilon_c$ for all $j = 1, 2, \ldots, R$, then the cluster centers v_{new}^j accurately represent the input data, the fuzzy clustering algorithm is terminated, and we proceed to the optimal output defuzzification algorithm (see below) where we use a least squares method. Otherwise, we continue to iteratively use Equations (11.35) and (11.36) until we find cluster centers v_{new}^j that satisfy $|v_{new}^j - v_{old}^j| < \epsilon_c$ for all $j = 1, 2, \ldots, R$. For our example, $v_{old}^j = v_0^j$, and we see that $|v_{new}^j - v_{old}^j| = 0.6328$ for $j = 1$ and 0.6260 for $j = 2$. Both of these values are greater than ϵ_c, so we continue to update the cluster centers.

Proceeding to the next iteration, let $v_{old}^j = v_{new}^j$, $j = 1, 2, \ldots, R$ from the last iteration, and apply Equations (11.35) and (11.36) to find $\mu_{11}^{new} = 0.8233, \mu_{12}^{new} = 0.1767, \mu_{21}^{new} = 0.7445, \mu_{22}^{new} = 0.2555, \mu_{31}^{new} = 0.0593$, and $\mu_{32}^{new} = 0.9407$ using the cluster centers calculated above, yielding the new cluster centers

$$v_{new}^1 = \begin{bmatrix} 0.9056 \\ 2.9084 \end{bmatrix}$$

and

$$v_{new}^2 = \begin{bmatrix} 2.8381 \\ 5.7397 \end{bmatrix}$$

Computing the distances between these cluster centers and the previous ones, we find that $|v_{new}^j - v_{old}^j| > \epsilon_c$, so the algorithm continues. It takes 14 iterations before the algorithm terminates (i.e., before we have $|v_{new}^j - v_{old}^j| \leq \epsilon_c = 0.001$ for all $j = 1, 2, \ldots, R$). When it does terminate, name the final membership grade values μ_{ij} and cluster centers v^j, $i = 1, 2, \ldots, M$, $j = 1, 2, \ldots, R$.

Finding the Final Cluster Center Values: For our example, after 14 iterations the algorithm finds $\mu_{11} = 0.9994$, $\mu_{12} = 0.0006$, $\mu_{21} = 0.1875$, $\mu_{22} = 0.8125$, $\mu_{31} = 0.0345$, $\mu_{32} = 0.9655$,

$$v^1 = \begin{bmatrix} 0.0714 \\ 2.0725 \end{bmatrix}$$

and

$$v^2 = \begin{bmatrix} 2.5854 \\ 5.1707 \end{bmatrix}$$

Notice that the clusters have converged so that v^1 is near $x(1) = [0,2]^\top$ and v^2 lies in between $x(2) = [2,4]^\top$ and $x(3) = [3,6]^\top$.

Specifying the Premise Membership Function: The final values of v^j, $j = 1, 2, \ldots, R$, are used to specify the premise membership functions for the i^{th} rule. In particular, we specify the premise membership functions as

$$\mu_{H^j}(x) = \left[\sum_{k=1}^{R} \left(\frac{|x - v^j|^2}{|x - v^k|^2}\right)^{\frac{1}{m-1}}\right]^{-1} \tag{11.37}$$

$j = 1, 2, \ldots, R$ where v^j, $j = 1, 2, \ldots, R$ are the cluster centers from the last iteration that uses Equations (11.35) and (11.36). It is interesting to note that for large values of m, we get smoother (less distinctive) membership functions. This is the primary guideline to use in selecting the value of m; however, often a good first choice is $m = 2$. Next, note that $\mu_{H^j}(x)$ is a premise membership function that is different from any that we have considered. With the premises of the rules defined, we next specify the consequent portion.

Least Squares for Specifying Rule Consequents

We apply "optimal output predefuzzification" to the training data to calculate the function $g_j(x) = \underline{a}_j^\top \hat{x}$, $j = 1, 2, \ldots, R$ for each rule (i.e., each cluster center), by determining the parameters $\underline{a}_j$. There are two methods you can use to find the $\underline{a}_j$.

Approach 1: For each cluster center v^j, in this approach we wish to minimize the squared error between the function $g_j(x)$ and the output portion of the training data pairs. Let $\hat{x}(i) = [1, (x(i))^\top]^\top$ where $(x(i), y(i)) \in G$. We wish to minimize the cost function J_j given by

We may use the μ_{ij} from the clusters to weight the batch least squares calculation so that the linear approximations pertain to each cluster.

$$J_j = \sum_{i=1}^{M} (\mu_{ij})^2 \left(y(i) - (\hat{x}(i))^\top \underline{a}_j\right)^2 \tag{11.38}$$

for each $j = 1, 2, \ldots, R$ where μ_{ij} is the grade of membership of the input portion of the i^{th} data pair for the j^{th} cluster that resulted from the clustering algorithm after it converged, $y(i)$ is the output portion of the i^{th} data pair from G, $(x(i), y(i))$, and the multiplication of $(\hat{x}(i))^\top$ and $\underline{a}_j$ defines the output $g_j(x)$ associated with the j^{th} rule for the i^{th} training data point.

Looking at Equation (11.38), we see that the minimization of J_j via the choice of the $\underline{a}_j$ is a weighted least squares problem. From Equation (10.2) on page 424, the solution $\underline{a}_j$ for $j = 1, 2, \ldots, R$ to the weighted least squares problem is given by

$$\underline{a}_j = (\hat{X}^\top D_j^2 \hat{X})^{-1} \hat{X}^\top D_j^2 Y \tag{11.39}$$

where

$$\begin{aligned} \hat{X} &= \begin{bmatrix} 1 & \dots & 1 \\ x(1) & \dots & x(M) \end{bmatrix}^\top \\ Y &= [y(1), \dots, y(M)]^\top, \\ D_j^2 &= (\mathrm{diag}([\mu_{1j}, \dots, \mu_{Mj}]))^2 \end{aligned}$$

For our example, the parameters that satisfy the linear function $g_j(x) = \underline{a}_j^\top \hat{\underline{x}}(i)$ for $j = 1, 2$ such that J_j in Equation (11.38) is minimized, were found to be $\underline{a}_1 = [3, 2.999, -1]^\top$ and $\underline{a}_2 = [3, 3, -1]^\top$, which are very close to each other.

Approach 2: As an alternative approach, rather than solving R least squares problems, one for each rule, we can use the least squares methods to specify the consequent parameters of the Takagi-Sugeno fuzzy system. To do this, we simply parameterize the Takagi-Sugeno fuzzy system in Equation (11.33) in a form so that it is linear in the consequent parameters; then we can use batch or recursive least squares methods to find the parameters. Unless we indicate otherwise, we will always use approach 1 in this book.

Testing the Approximator

Suppose that we use approach 1 to specify the rule consequents. To test how accurately the constructed fuzzy system represents the training data set G in Figure 11.31 on page 537, suppose that we choose the test point x' such that $(x', y') \notin G$. Specifically, we choose

$$x' = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

We would expect from Figure 11.31 that the output of the fuzzy system would lie somewhere between 1 and 5. The output is 3.9999, so we see that the trained Takagi-Sugeno fuzzy system seems to interpolate adequately. Notice also that if we let $x = x(i)$, $i = 1, 2, 3$ where $(x(i), y(i)) \in G$, we get values very close to the $y(i)$, $i = 1, 2, 3$, respectively. That is, for this example, the fuzzy system nearly perfectly maps the training data pairs. We also note that if the input to the fuzzy system is $x = [2.5, 5]^\top$, the output is 5.5, so the fuzzy system seems to perform good interpolation near the training data points.

Finally, we note that the $\underline{a}_j$ will clearly not always be as close to each other as for this example. For instance, if we add the data pair $([4, 5]^\top, 5.5)$ to G (i.e., make $M = 4$), then the cluster centers converge after 13 iterations (using the same parameters m and ϵ_c as we did earlier). Using approach 1 to find the consequent parameters, we get

$$\underline{a}_1 = [-1.458, 0.7307, 1.2307]^\top$$

and

$$\underline{a}_2 = [2.999, 0.00004, 0.5]^\top$$

For the resulting fuzzy system, if we let $x = [1, 2]^\top$ in Equation (11.33), we get an output value of 1.8378, so we see that it performs differently from the case for $M = 3$ but still provides a reasonable interpolated value.

11.7 Neural or Fuzzy: Which is Better? Bad Question!

If you are asking this question, it shows that you do not understand the fundamental concepts!

- You should be concerned about whether your training data carries the proper information to perform good approximation. Is the training data set large enough? Is the test set large enough? Does your measure of approximation accuracy properly reflect your approximation goals? Did you start with a simple linear (or affine) approximator and a linear least squares method? For many applications this can be sufficient; you only need all the capabilities of neural or fuzzy system approximators if you have a nonlinear approximation problem.

- You should realize that for practical applications, the choice of which is the best approximator structure is very difficult and you cannot quickly conclude that one is better than another.

- You should be concerned with approximator complexity and approximator tunability for your application. For example, via experience have you found that a certain type of structure works well? Or, based on physical insights, can you use nonlinear functions of input data as inputs to your approximator?

- You should ask whether to use a local or globally supported basis function (i.e., one that only has a local influence on the approximator mapping, or one that, if it is changed, can change the shape over the whole region of the mapping).

- You should ask whether you have too many inputs (i.e., too large a value for n) so that it is not possible to use a grid if you are using a locally supported basis function (i.e., you should be concerned with the impact of how many inputs you have on the computational complexity in approximator structure choice).

- You should ask whether you should use a linear or nonlinear in the parameter approximator, since this affects tuning flexibility and training algorithm performance.

The names "neural" or "fuzzy" are largely attached simply for historical purposes due to the fields that they came from. Really you need to think of the basics, not this terminology. Focus on generalization, overfitting, complexity, and composition of the data set. Generally, structure choice is quite difficult so it needs attention; however, methods to *automate* the construction of the structure are discussed in the "For Further Study" section at the end of this part.

11.8 Exercises and Design Problems

Exercise 11.1 (Matlab for Neural Network Training):

(a) Suppose you use a multilayer perceptron with two layers, the first layer has $n_1 = 11$ logistic function neurons, and the output layer has a single linear neuron. Use a software package (e.g., the Matlab Neural Networks toolbox) to match the training data shown in Figure 9.10 (this defines G and here use $M = 121$). Train with the Levenberg-Marquardt method. While you train with 121 data pairs, test with about 10 times that many. Plot the approximator mapping and data on the same plot to evaluate the accuracy of the interpolation.

(b) Train with a conjugate gradient method and compare to the result in (a).

(c) Train with steepest descent and compare to the results in (a) and (b).

Exercise 11.2 (Levenberg-Marquardt Update Formulas for Neural Networks):

(a) Derive the Levenberg-Marquardt parameter update formulas for a two-layer multilayer perceptron with a linear output layer and hyperbolic tangent activation functions in the hidden layer. Assume that you update all weights and biases in the network (i.e., both the parameters that enter linearly and those that enter in a nonlinear fashion).

(b) Repeat (a), but for a radial basis function neural network where the output is computed as a sum of Gaussian receptive field units. Assume that you update all parameters in the network (i.e., both the parameters that enter linearly and those that enter in a nonlinear fashion).

(c) For both (a) and (b), solve the function approximation theme problem given in this chapter. Clearly explain all your choices for the approximator structure and training method. Illustrate generalization properties of the approximators after they are trained.

Design Problem 11.1 (Fuzzy C-Means and Least Squares for Approximator Tuning):

(a) Use fuzzy c-means and least squares for tuning the special type of Takagi-Sugeno fuzzy system given in the chapter to solve the function approximation theme problem studied throughout this chapter.

(b) Illustrate its generalization capabilities and that it makes reasonable choices for cluster placement (plot the final clusters on the same plot as the function you are trying to approximate).

(c) Compare the results to what you obtained in Exercise 11.2 where neural networks and Levenberg-Marquardt training were used.

Design Problem 11.2 (Structural Plasticity and Approximators)*: The human brain and the brains of many other animals learn not only via parameter adjustment (the adjustment of strengths between connections in the biological neural network), but also by growing new neuronal connections and destroying others. In this problem you will learn methods for constructing the structure of approximators. For example, in the case of multilayer perceptrons, some methods automatically pick the number of neurons used in each layer, and some of the methods use biomimicry concepts based on biological neural networks. Alternatively, with our unified view of approximators, we can consider how to construct the structure of a fuzzy system. For instance, we may study how to automatically pick the number of rules or membership functions.

(a) First, you must conduct some background research. Read the papers [295, 431] and see the book [412] for ideas on how to construct the number of rules in a fuzzy system.

(b) Explain in detail how the neural network methods can be used for fuzzy systems. To do this, pick a standard fuzzy system and define the algorithms for its construction. Are there methods developed in the area of neural networks that do not seem to apply to any fuzzy system?

(c) Choose a method from one of the above references, specify a structure construction/destruction algorithm, develop code to implement it, and test it for the theme problem that was studied in this chapter. For many methods this will involve specifying how structure is adjusted, and the use of a standard training method (e.g, gradient or least squares) as found in the chapter. Be sure to use appropriate training and test sets, and clearly illustrate the performance of the method. If possible, compare it to the results in the chapter where only the parameters were tuned, not the structure.

(d) Invent a method for tuning structure of an approximator. You choose the type of approximator you want to study. Hint: Suppose that you have a low-dimensional function to approximate (e.g., one output and two inputs). Suppose that your training data set is G and test set is Γ. Suppose that you grid the input space, calling each subregion a "cell," and label these c_i, $i = 1, 2, \ldots, N_g$ where N_g is the number of cells created by the number of partitions on the j^{th} input space x_j (we assume that the number of divisions on each input dimension is the same, but clearly this is just for convenience). Suppose that you use a test set Γ_a and a cost function J_a that is defined to be the approximation error in each cell c_i between the approximator and the actual function for Γ_a. In particular, if c_i is a cell (e.g., a square if the dimension of the input space is 2), then $J_a(c_i)$ could be defined to be the average mean squared error over points in the test set Γ_a that lie in c_i (clearly, then, to make this a reasonable definition you would want Γ_a to have points in each cell created by the input space gridding). Suppose that there are no common points in Γ_a, G, and Γ. Suppose that $|G| < |\Gamma|$ (with the difference in size large enough so that you can achieve good function approximation for a fixed size approximator, and for any value that you adjust p to be in your structure adjustment method). Also, suppose that $|\Gamma_a|$ is large enough to be representative of the approximation error, no matter how you adjust the structure (e.g., it could be that $|\Gamma_a| = |\Gamma|$).

Now, view the approximator construction problem as a two-level optimization problem. In particular, we will view it as a type of multilevel reinforcement learning approach, and hence, it is a gradient-type method. For any fixed structure (i.e., fixed p), we will tune with a standard gradient method (e.g., Levenberg-Marquardt). This tuning will occur interleaved with structure adjustments; there will be a structure adjustment, then multiple steps of the standard gradient method will be executed (e.g., until some termination criterion is satisfied) before the next structure adjustment. How do we then make structure adjustments? There are many ways. One way is to adjust the structure of the approximator to try to achieve the minimization of J_a. Choose some threshold $\varepsilon > 0$ that represents what you consider to be an acceptable level of approximation error in any cell c_i. Suppose that we try to adjust an approximator structure that is based on gridding the input space with basis functions (e.g., the radial basis function neural network or several types of fuzzy systems). To be more concrete, suppose that we adjust radial basis function neural networks with their radial basis functions defined to be Gaussian functions with centers that our structure adjustment method will place (for simplicity, let the parameters that enter linearly be adjusted only after structure adjustments are made in the step where we use gradient training). Adjust structure as follows:

1. Compute $J_a(c_i)$, $i = 1, 2, \ldots, N_g$, over the test set Γ_a.
2. If for some i, $J_a(c_i) > \varepsilon$, then randomly place $\lambda_{add} int(|J_a(c_i) - \varepsilon|)$ ($int(\cdot)$ is the integer part of its argument) new radial basis functions in the region c_i, where $\lambda_{add} > 0$ can be thought of as a step size for the structure adjustment algorithm in the case where structure is added.
3. If for some i, $J_a(c_i) \leq \varepsilon$, then randomly remove $\lambda_{sub} int(|J_a(c_i) - \varepsilon|)$ radial basis functions from the region c_i, where $\lambda_{sub} > 0$ can be thought of as a step size for the structure adjustment

algorithm in the case where structure is deleted.

4. Go to standard gradient method for parameter adjustments.

The goal of the method is to try to achieve an error of ε in each cell. Why not just try for zero approximation error in each cell? This will in general require an infinite number of radial basis functions; we pay for accuracy with complexity. The addition of more radial basis functions allows for more accurate function approximation in regions where they are added. Removal of radial basis functions can result in lower approximation accuracy where they are removed. The algorithm will tend to redistribute the centers so as to allocate them where more accuracy is needed.

Fully test this algorithm for both $n = 1$ and $n = 2$, showing how it reshapes the approximator mapping (show plots) and reallocates the radial basis function centers. Explain why you can view the above approach as a reinforcement-based learning method for structure, and in particular, write down the update equation that clearly shows it is a gradient-type method. What is the reinforcement function? Next, can you achieve a simpler approximator structure with this approach than with the one you would construct manually? Does the gridding approach that this method is based on make it impossible to apply to high-dimensional function approximation problems? If not, explain. If so, which method from the literature would do better? Next, explain how you could redesign the algorithm so that it can be used for online function construction.