17 Radial Basis Networks

Objectives

The multilayer networks discussed in Chapters 11 and 12 represent one type of neural network structure for function approximation and pattern recognition. As we saw in Chapter 11, multilayer networks with sigmoid transfer functions in the hidden layers and linear transfer functions in the output layer are universal function approximators. In this chapter we will discuss another type of universal approximation network, the radial basis function network. This network can be used for many of the same applications as multilayer networks.

This chapter will follow the structure of Chapter 11. We will begin by demonstrating, in an intuitive way, the universal approximation capabilities of the radial basis function network. Then we will describe three different techniques for training these networks. They can be trained by the same gradient-based algorithms discussed in Chapters 11 and 12, with derivatives computed using a form of backpropagation. However, they can also be trained using a two-stage process, in which the first layer weights are computed independently from the weights in the second layer. Finally, these networks can be built in an incremental way - one neuron at a time.

Theory and Examples

The radial basis function network is related to the multilayer perceptron network of Chapter 11. It is also a universal approximator and can be used for function approximation or pattern recognition. We will begin this chapter with a description of the network and a demonstration of its abilities for function approximation and pattern recognition.

The original work in radial basis functions was performed by Powell and others during the 1980's [Powe87]. In this original work, radial basis functions were used for exact interpolation in a multidimensional space. In other words, the function created by the radial basis interpolation was required to pass exactly through all targets in the training set. The use of radial basis functions for exact interpolation continues to be an important application area, and it is also an active area of research.

For our purposes, however, we will not be considering exact interpolation. Neural networks are often used on noisy data, and exact interpolation often results in overfitting when the training data is noisy, as we discussed in Chapter 13. Our interest is in the use of radial basis functions to provide robust approximations to unknown functions based on generally limited and noisy measurements. Broomhead and Lowe [BrLo88] were the first to develop the radial basis function neural network model, which produces a smooth interpolating function. No attempt is made to force the network response to exactly match target outputs. The emphasis is on producing networks that will generalize well to new situations.

In the next section we will demonstrate the capabilities of the radial basis function neural network. In the following sections we will describe procedures for training these networks.

Radial Basis Network

RBF

The radial basis network is a two-layer network. There are two major distinctions between the radial basis function (RBF) network and a two layer perceptron network. First, in layer 1 of the RBF network, instead of performing an inner product operation between the weights and the input (matrix multiplication), we calculate the distance between the input vector and the rows of the weight matrix. (This is similar to the LVQ network shown in Figure 14.13.) Second, instead of adding the bias, we multiply by the bias. Therefore, the net input for neuron i in the first layer is calculated as follows:

$$n_i^1 = \| \mathbf{p} - {}_i\mathbf{w}^1 \| b_i^1 . \quad (17.1)$$

Each row of the weight matrix acts as a center point - a point where the net input value will be zero. The bias performs a scaling operation on the transfer (basis) function, causing it to stretch or compress.

We should note that most papers and texts on RBF networks use the terms standard deviation, variance or spread constant, rather than bias. We have used the bias in order to maintain a consistency with other networks in this text. This is simply a matter of notation and pedagogy. The operation of the network is not affected. When a Gaussian transfer function is used, the bias is related to the standard deviation as follows: $b = 1/(\sigma\sqrt{2})$.

MLP

The transfer functions used in the first layer of the RBF network are different than the sigmoid functions generally used in the hidden layers of multilayer perceptrons (MLP). There are several different types of transfer function that can be used (see [BrLo88]), but for clarity of presentation we will consider only the Gaussian function, which is the one most commonly used in the neural network community. It is defined as follows

$$a = e^{-n^2}, \tag{17.2}$$

and it is plotted in Figure 17.1.

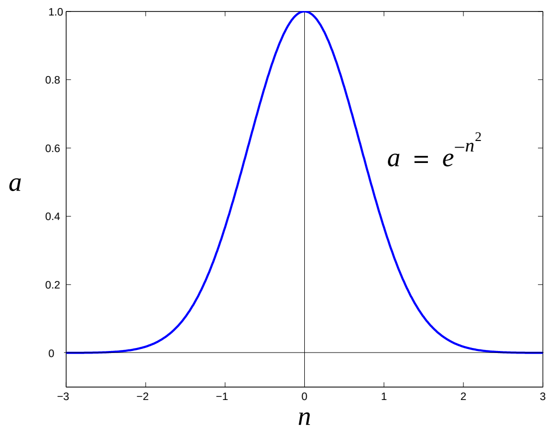


Figure 17.1 Gaussian Basis Function

local function

global function

A key property of this function is that it is *local*. This means that the output is close to zero if you move very far in either direction from the center point. This is in contrast to the *global* sigmoid functions, whose output remains close to 1 as the net input goes to infinity.

The second layer of the RBF network is a standard linear layer:

$$\mathbf{a}^2 = \mathbf{W}^2\mathbf{a}^1 + \mathbf{b}^2 \tag{17.3}$$

Figure 17.2 shows the complete RBF network.

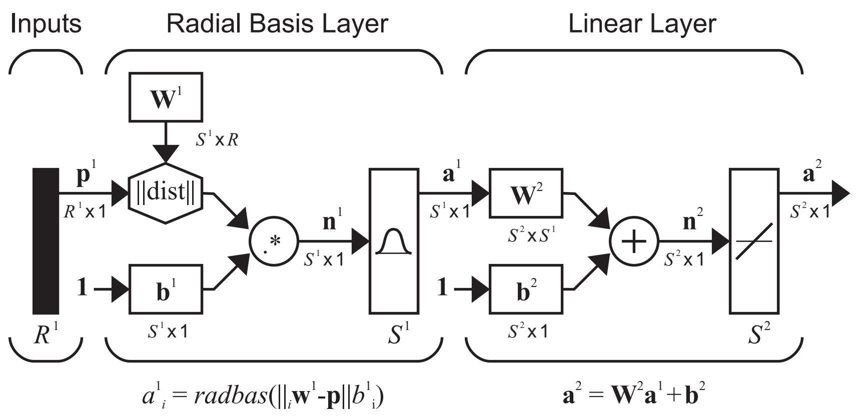


Figure 17.2 Radial Basis Network

Function Approximation

This RBF network has been shown to be a universal approximator [PaSa93], just like the MLP network. To illustrate the capability of this network, consider a network with two neurons in the hidden layer, one output neuron, and with the following default parameters:

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

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

The response of the network with the default parameters is shown in Figure 17.3, which plots the network output a^2 as the input p is varied over the range $[-2, 2]$.

Notice that the response consists of two hills, one for each of the Gaussian neurons (basis functions) in the first layer. By adjusting the network parameters, we can change the shape and location of each hill, as we will see in the following discussion. (As you proceed through this example, it may be helpful to compare the response of this sample RBF network with the response of the sample MLP network in Figure 11.5.)

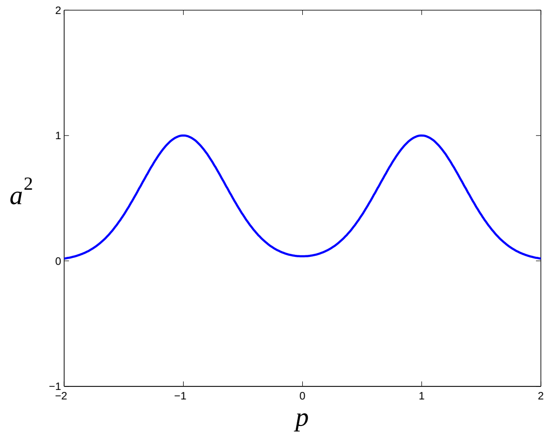


Figure 17.3 Default Network Response

Figure 17.4 illustrates the effects of parameter changes on the network response. The blue curve is the nominal response. The other curves correspond to the network response when one parameter at a time is varied over the following ranges:

$$0 \le w^1_{2,1} \le 2\,,\ -1 \le w^2_{1,1} \le 1\,,\ 0.5 \le b^1_2 \le 8\,,\ -1 \le b^2 \le 1\,. \tag{17.4}$$

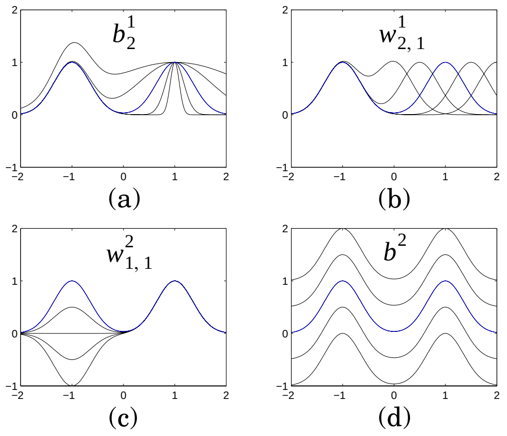


Figure 17.4 Effect of Parameter Changes on Network Response

Figure 17.4 (a) shows how the network biases in the first layer can be used to change the width of the hills - the larger the bias, the narrower the hill. Figure 17.4 (b) illustrates how the weights in the first layer determine the location of the hills; there will be a hill centered at each first layer weight. For multidimensional inputs there will be a hill centered at each row of the weight matrix. For this reason, each row of the first layer weight matrix is often called the *center* for the corresponding neuron (basis function).

center

Notice that the effects of the weight and the bias in first layer of the RBF network are much different than for the MLP network, which was shown in Figure 11.6. In the MLP network, the sigmoid functions create steps. The weights change the slopes of the steps, and the biases change the locations of the steps.

Figure 17.4 (c) illustrates how the weights in the second layer scale the height of the hills. The bias in the second layer shifts the entire network response up or down, as can be seen in Figure 17.4 (d). The second layer of the RBF network is the same type of linear layer used in the MLP network of Figure 11.6, and it performs a similar function, which is to create a weighted sum of the outputs of the layer 1 neurons.

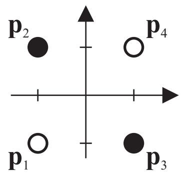
This example demonstrates the flexibility of the RBF network for function approximation. As with the MLP, it seems clear that if we have enough neurons in the first layer of the RBF network, we can approximate virtually any function of interest, and [PaSa93] provides a mathematical proof that this is the case. However, although both MLP and RBF networks are universal approximators, they perform their approximation in different ways. For the RBF network, each transfer function is only active over a small region of the input space - the response is *local*. If the input moves far from a given center, the output of the corresponding neuron will be close to zero. This has consequences for the design of RBF networks. We must have centers adequately distributed throughout the range of the network inputs, and we must select biases in such a way that all of the basis functions overlap in a significant way. (Recall that the biases change the width of each basis function.) We will discuss these design considerations in more detail in later sections.

To experiment with the response of this RBF network, use the MATLAB® Neural Network Design Demonstration RBF Network Function (**nnd17nf**).

Pattern Classification

To illustrate the capabilities of the RBF network for pattern classification, consider again the classic exclusive-or (XOR) problem. The categories for the XOR gate are

$$\text{Category 1: } \left\{ \mathbf{p}_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, \mathbf{p}_3 = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \right\}, \text{ Category 2: } \left\{ \mathbf{p}_1 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}, \mathbf{p}_4 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}.$$



The problem is illustrated graphically in the figure to the left. Because the two categories are not linearly separable, a single-layer network cannot perform the classification.

RBF networks can classify these patterns. In fact, there are many different RBF solutions. We will consider one solution that demonstrates in a simple way how to use RBF networks for pattern classification. The idea will be to have the network produce outputs greater than zero when the input is near patterns $\mathbf{p}_2$ or $\mathbf{p}_3$, and outputs less than zero for all other inputs. (Note that the procedures we will use to design this example network are not suitable for complex problems, but they will help us illustrate the capabilities of the RBF network.)

From the problem statement, we know that the network will need to have two inputs and one output. For simplicity, we will use only two neurons in the first layer (two basis functions), since this will be sufficient to solve the XOR problem. As we discussed earlier, the rows of the first-layer weight matrix will create centers for the two basis functions. We will choose the centers to be equal to the patterns $\mathbf{p}_2$ and $\mathbf{p}_3$. By centering a basis function at each pattern, we can produce maximum network outputs there. The first layer weight matrix is then

$$\mathbf{W}^1 = \begin{bmatrix} \mathbf{p}_2^T \\ \mathbf{p}_3^T \end{bmatrix} = \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix}. \qquad (17.5)$$

The choice of the bias in the first layer depends on the width that we want for each basis function. For this problem, we would like the network function to have two distinct peaks at $\mathbf{p}_2$ and $\mathbf{p}_3$. Therefore, we don't want the basis functions to overlap too much. The centers of the basis functions are each a distance of $\sqrt{2}$ from the origin. We want the basis function to drop significantly from its peak in this distance. If we use a bias of 1, we would get the following reduction in that distance:

$$a = e^{-n^2} = e^{-(1 \cdot \sqrt{2})^2} = e^{-2} = 0.1353 . \qquad (17.6)$$

Therefore, each basis function will have a peak of 1 at the centers, and will drop to 0.1353 at the origin. This will work for our problem, so we select the first layer bias vector to be

$$\mathbf{b}^1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}. \qquad (17.7)$$

The original basis function response ranges from 0 to 1 (see Figure 17.1). We want the output to be negative for inputs much different than $\mathbf{p}_2$ and $\mathbf{p}_3$, so we will use a bias of -1 for the second layer, and we will use a value of 2 for the second layer weights, in order to bring the peaks back up to 1. The second layer weights and biases then become

$$\mathbf{W}^2 = \begin{bmatrix} 2 & 2 \end{bmatrix} ,\ b^2 = \begin{bmatrix} -1 \end{bmatrix} . \tag{17.8}$$

For the network parameter values given in (17.5), (17.7) and (17.8), the network response is shown in Figure 17.5. This figure also shows where the surface intersects the plane at $a^2 = 0$, which is where the decision takes place. This is also indicated by the contours shown underneath the surface. These are the function contours where $a^2 = 0$. They are almost circles that surround the $\mathbf{p}_2$ and $\mathbf{p}_3$ vectors. This means that the network output will be greater than 0 only when the input vector is near the $\mathbf{p}_2$ and $\mathbf{p}_3$ vectors.

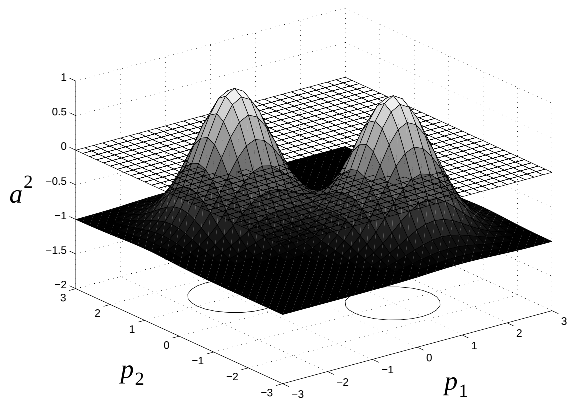


Figure 17.5 Example 2-Input RBF Function Surface

Figure 17.6 illustrates more clearly the decision boundaries. Whenever the input falls in the blue regions, the output of the network will be greater than zero. Whenever the network input falls outside the blue regions, the network output will be less than zero.

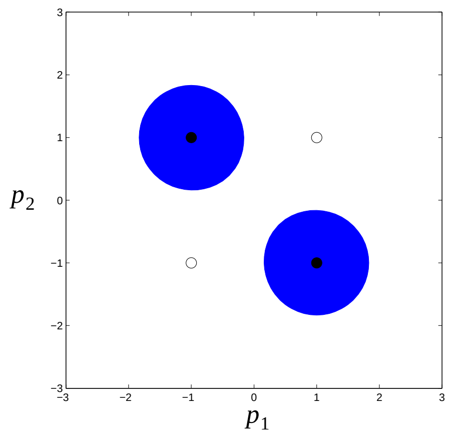


Figure 17.6 RBF Example Decision Regions

This network, therefore, classifies the patterns correctly. It is not the best solution, in the sense that it does not always assign input patterns to the closest prototype vector, unlike the MLP solution shown in Figure 11.2. You will notice that the decision regions for this RBF network are circles, unlike the linear boundaries that we see in single layer perceptrons. The MLP can put linear boundaries together to create arbitrary decision regions. The RBF network can put circular boundaries together to create arbitrary decision regions. In this problem, the linear boundaries are more efficient. Of course, when many neurons are used, and the centers are close together, the elementary RBF boundaries are no longer purely circular, and the elementary MLP boundaries are no longer purely linear. However, associating circular boundaries with RBF networks and linear boundaries with MLP networks can be helpful in understanding their operation as pattern classifiers.

To experiment with the RBF network for pattern classification, use the MATLAB® Neural Network Design Demonstration RBF Pattern Classification (**`nnd17pc`**).

Now that we see the power of RBF networks for function approximation and pattern recognition, the next step is to develop general training algorithms for these networks.

Global vs. Local

Before we discuss the training algorithms, we should say a final word about the advantages and disadvantages of the global transfer functions used by the MLP networks and the local transfer functions used by the RBF networks. The MLP creates a distributed representation, because all of the transfer functions overlap in their activity. At any given input value, many sigmoid functions in the first layer will have significant outputs. They must sum or cancel in the second layer in order to produce the appropriate response at each point. In the RBF network, each basis function is only active over a small range of the input. For any given input, only a few basis functions will be active.

There are advantages and disadvantages to each approach. The global approach tends to require fewer neurons in the hidden layer, since each neuron contributes to the response over a large part of the input space. For the RBF network, however, basis centers must be spread throughout the range of the input space in order to provide an accurate approximation. This leads to the problem of the "curse of dimensionality," which we will discuss in the next section. Also, if more neurons, and therefore more parameters, are used, then there is a greater likelihood that the network will overfit the training data and fail to generalize well to new situations.

On the other hand, the local approach generally leads to faster training, especially when the two-stage algorithms, which will be discussed in the next section, are used. Also, the local approach can be very useful for adaptive training, in which the network continues to be incrementally trained while it is being used, as in adaptive filters (nonlinear versions of the filters in Chapter 10) or controllers. If, for a period of time, training data only appears in a certain region of the input space, then a global representation will tend to improve its accuracy in those regions at the expense of its representation in other regions. Local representations will not have this problem to the same extent. Because each neuron is only active in a small region of the input space, its weights will not be adjusted if the input falls outside that region.

Training RBF Networks

Unlike the MLP network, which is almost always trained by some gradient-based algorithm (steepest descent, conjugate gradient, Levenberg-Marquardt, etc.), the RBF network can be trained by a variety of approaches.

RBF networks can be trained using gradient-based algorithms. However, because of the local nature of the transfer function and the way in which the first layer weights and biases operate, there tend to be many more unsatisfactory local minima in the error surfaces of RBF networks than in those of MLP networks. For this reason, gradient-based algorithms are often unsatisfactory for the complete training of RBF networks. They are used on occasion, however, for fine-tuning of the network after it has been initially trained using some other method. Later in this chapter we will discuss the modifications to the backpropagation equations in Chapter 11 that are needed to compute the gradients for RBF networks.

The most commonly used RBF training algorithms have two stages, which treat the two layers of the RBF network separately. The algorithms differ mainly in how the first layer weights and biases are selected. Once the first layer weights and biases have been selected, the second layer weights can be computed in one step, using a linear least-squares algorithm. We will discuss linear least squares in the next section.

The simplest of the two-stage algorithms arranges the centers (first layer weights) in a grid pattern throughout the input range and then chooses a constant bias so that the basis functions have some degree of overlap. This procedure is not optimal, because the most efficient approximation would place more basis functions in regions of the input space where the function to be approximated is most complex. Also, for many practical cases the full range of the input space is not used, and therefore many basis functions could be wasted. One of the drawbacks of the RBF network, especially when the centers are selected on a grid, is that they suffer from the *curse of dimensionality*. This means that as the dimension of the input space increases, the number of basis functions required increases geometrically. For example, suppose that we have one input variable, and we specify a grid of 10 basis functions evenly spaced across the range of the input variable. Now increase the number of input variables to 2. To maintain the same grid coverage for both input variables, we would need 10^2, or 100 basis functions.

Curse of Dimensionali

Another method for selecting the centers is to select some random subset of the input vectors in the training set. This ensures that basis centers will be placed in areas where they will be useful to the network. However, due to the randomness of the selection, this procedure is not optimal. A more efficient approach is to use a method such as the Kohonen competitive layer or the feature map, described in Chapter 16, to cluster the input space. The cluster centers then become basis function centers. This ensures that the basis functions are placed in regions with significant activity. We will discuss this method in a later section.

A final procedure that we will discuss for RBF training is called orthogonal least squares. It is based on a general method for building linear models called subset selection. This method starts with a large number of possible centers - typically all of the input vectors from the training data. At each stage of the procedure, it selects one center to add to the first layer weight. The selection is based on how much the new neuron will reduce the sum squared error. Neurons are added until some criteria is met. The criteria is typically chosen to maximize the generalization capability of the network.

Linear Least Squares

In this section we will assume that the first layer weights and biases of the RBF network are fixed. This can be done by fixing the centers on a grid, or by randomly selecting the centers from the input vectors in the training data set (or by using the clustering method which is described in a later section). When the centers are randomly selected, all of the biases can be selected using the following formula [Lowe89]:

$$b_i^1 = \frac{\sqrt{S^1}}{d_{max}}, \tag{17.9}$$

where d_{max} is the maximum distance between neighboring centers. This is designed to ensure an appropriate degree of overlap between the basis functions. Using this method, all of the biases have the same value. There are other methods which use different values for each bias. We will discuss one such method later, in the section on clustering.

Once the first layer parameters have been set, the training of the second layer weights and biases is equivalent to training a linear network, as in Chapter 10. For example, consider that we have the following training points

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}, \tag{17.10}$$

where $\mathbf{p}_q$ is an input to the network, and $\mathbf{t}_q$ is the corresponding target output. The output of the first layer for each input $\mathbf{p}_q$ in the training set can be computed as

$$n_{i,q}^1 = \left\| \mathbf{p}_q - {}_i\mathbf{w}^1 \right\| b_i^1, \tag{17.11}$$

$$\mathbf{a}_q^1 = \mathbf{radbas}(\mathbf{n}_q^1). \tag{17.12}$$

Since the first layer weights and biases will not be adjusted, the training data set for the second layer then becomes

$$\{\mathbf{a}_1^1, \mathbf{t}_1\}, \{\mathbf{a}_2^1, \mathbf{t}_2\}, \ldots, \{\mathbf{a}_Q^1, \mathbf{t}_Q\}. \tag{17.13}$$

The second layer response is linear:

$$\mathbf{a}^2 = \mathbf{W}^2\mathbf{a}^1 + \mathbf{b}^2. \tag{17.14}$$

We want to select the weights and biases in this layer to minimize the sum square error performance index over the training set:

$$F(\mathbf{x}) = \sum_{q=1}^{Q} (\mathbf{t}_q - \mathbf{a}_q^2)^T (\mathbf{t}_q - \mathbf{a}_q^2) \tag{17.15}$$

Our derivation of the solution to this linear least squares problem will follow the linear network derivation starting with Eq. (10.6). To simplify the discussion, we will assume a scalar target, and we will lump all of the parameters we are adjusting, including the bias, into one vector:

$$\mathbf{x} = \begin{bmatrix} {}_1\mathbf{w}^2 \\ b^2 \end{bmatrix}. \tag{17.16}$$

Similarly, we include the bias input "1" as a component of the input vector

$$\mathbf{z}_q = \begin{bmatrix} \mathbf{a}_q^1 \\ 1 \end{bmatrix}. \tag{17.17}$$

Now the network output, which we usually write in the form

$$a_q^2 = ({}_1\mathbf{w}^2)^T \mathbf{a}_q^1 + b^2 , \tag{17.18}$$

can be written as

$$a_q = \mathbf{x}^T \mathbf{z}_q . \tag{17.19}$$

This allows us to conveniently write out an expression for the sum square error:

$$F(\mathbf{x}) = \sum_{q=1}^{Q} (e_q)^2 = \sum_{q=1}^{Q} (t_q - a_q)^2 = \sum_{q=1}^{Q} (t_q - \mathbf{x}^T\mathbf{z}_q)^2 . \tag{17.20}$$

To express this in matrix form, we define the following matrices:

$$\mathbf{t} = \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_Q \end{bmatrix}, \quad \mathbf{U} = \begin{bmatrix} {}_1\mathbf{u}^T \\ {}_2\mathbf{u}^T \\ \vdots \\ {}_Q\mathbf{u}^T \end{bmatrix} = \begin{bmatrix} \mathbf{z}_1^T \\ \mathbf{z}_2^T \\ \vdots \\ \mathbf{z}_Q^T \end{bmatrix}, \quad \mathbf{e} = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_Q \end{bmatrix}. \tag{17.21}$$

The error can now be written

$$\mathbf{e} = \mathbf{t} - \mathbf{U}\mathbf{x} , \tag{17.22}$$

and the performance index become

$$F(\mathbf{x}) = (\mathbf{t} - \mathbf{U}\mathbf{x})^T(\mathbf{t} - \mathbf{U}\mathbf{x}) . \tag{17.23}$$

If we use regularization, as we discussed in Chapter 13, to help in preventing overfitting, we obtain the following form for the performance index:

$$F(\mathbf{x}) = (\mathbf{t}-\mathbf{U}\mathbf{x})^T(\mathbf{t}-\mathbf{U}\mathbf{x}) + \rho\sum_{i=1}^{n} x_i^2 = (\mathbf{t}-\mathbf{U}\mathbf{x})^T(\mathbf{t}-\mathbf{U}\mathbf{x}) + \rho\mathbf{x}^T\mathbf{x}\,, \quad (17.24)$$

where $\rho = \alpha/\beta$ from Eq. (13.4). Let's expand this expression to obtain

$$\begin{aligned} F(\mathbf{x}) &= (\mathbf{t}-\mathbf{U}\mathbf{x})^T(\mathbf{t}-\mathbf{U}\mathbf{x}) + \rho\mathbf{x}^T\mathbf{x} = \mathbf{t}^T\mathbf{t} - 2\mathbf{t}^T\mathbf{U}\mathbf{x} + \mathbf{x}^T\mathbf{U}^T\mathbf{U}\mathbf{x} + \rho\mathbf{x}^T\mathbf{x} \\ &= \mathbf{t}^T\mathbf{t} - 2\mathbf{t}^T\mathbf{U}\mathbf{x} + \mathbf{x}^T[\mathbf{U}^T\mathbf{U} + \rho\mathbf{I}]\mathbf{x} \end{aligned} \quad (17.25)$$

Take a close look at Eq. (17.25), and compare it with the general form of the quadratic function, given in Eq. (8.35) and repeated here:

$$F(\mathbf{x}) = c + \mathbf{d}^T\mathbf{x} + \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x}\ . \quad (17.26)$$

Our performance function is a quadratic function, where

$$c = \mathbf{t}^T\mathbf{t},\ \mathbf{d} = -2\mathbf{U}^T\mathbf{t} \text{ and } \mathbf{A} = 2[\mathbf{U}^T\mathbf{U} + \rho\mathbf{I}]\,. \quad (17.27)$$

From Chapter 8 we know that the characteristics of the quadratic function depend primarily on the Hessian matrix $\mathbf{A}$. For example, if the eigenvalues of the Hessian are all positive, then the function will have one unique global minimum.

In this case the Hessian matrix is $2[\mathbf{U}^T\mathbf{U} + \rho\mathbf{I}]$, and it can be shown that this matrix is either positive definite or positive semidefinite (see Exercise E17.4), which means that it can never have negative eigenvalues. We are left with two possibilities. If the Hessian matrix has only positive eigenvalues, the performance index will have one unique global minimum (see Figure 8.7). If the Hessian matrix has some zero eigenvalues, the performance index will either have a weak minimum (see Figure 8.9) or no minimum (see Problem P8.7), depending on the vector $\mathbf{d}$. In this case, it must have a minimum, since $F(\mathbf{x})$ is a sum square function, which cannot be negative.

Now let's locate the stationary point of the performance index. From our previous discussion of quadratic functions in Chapter 8, we know that the gradient is

$$\nabla F(\mathbf{x}) = \nabla\left(c + \mathbf{d}^T\mathbf{x} + \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x}\right) = \mathbf{d} + \mathbf{A}\mathbf{x} = -2\mathbf{U}^T\mathbf{t} + 2[\mathbf{U}^T\mathbf{U} + \rho\mathbf{I}]\mathbf{x}\,. \quad (17.28)$$

The stationary point of $F(\mathbf{x})$ can be found by setting the gradient equal to zero:

$$-2\mathbf{Z}^T\mathbf{t} + 2[\mathbf{U}^T\mathbf{U} + \rho\mathbf{I}]\mathbf{x} = 0 \quad \Rightarrow \quad [\mathbf{U}^T\mathbf{U} + \rho\mathbf{I}]\mathbf{x} = \mathbf{U}^T\mathbf{t}\,. \quad (17.29)$$

Therefore, the optimum weights $\mathbf{x}^*$ can be computed from

$$[\mathbf{U}^T\mathbf{U} + \rho\mathbf{I}]\mathbf{x}^* = \mathbf{U}^T\mathbf{t}. \tag{17.30}$$

If the Hessian matrix is positive definite, there will be a unique stationary point, which will be a strong minimum:

$$\mathbf{x}^* = [\mathbf{U}^T\mathbf{U} + \rho\mathbf{I}]^{-1}\mathbf{U}^T\mathbf{t} \tag{17.31}$$

Let's demonstrate this procedure with a simple problem.

Example

To illustrate the least squares algorithm, let's choose a network and apply it to a particular problem. We will use an RBF network with three neurons in the first layer to approximate the following function

$$g(p) = 1 + \sin\left(\frac{\pi}{4}p\right) \text{ for } -2 \le p \le 2. \tag{17.32}$$

To obtain our training set we will evaluate this function at six values of p:

$$p = \{-2, -1.2, -0.4, 0.4, 1.2, 2\}. \tag{17.33}$$

This produces the targets

$$t = \{0, 0.19, 0.69, 1.3, 1.8, 2\}. \tag{17.34}$$

We will choose the basis function centers to be spaced equally throughout the input range: -2, 0 and 2. For simplicity, we will choose the bias to be the reciprocal of the spacing between points. This produces the following first layer weight and bias.

$$\mathbf{W}^1 = \begin{bmatrix} -2 \\ 0 \\ 2 \end{bmatrix}, \mathbf{b}^1 = \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \end{bmatrix}. \tag{17.35}$$

The next step is to compute the output of the first layer, using the following equations.

$$n^1_{i,q} = \left\|\mathbf{p}_q - {}_i\mathbf{w}^1\right\| b^1_i, \tag{17.36}$$

$$\mathbf{a}^1_q = \mathbf{radbas}(\mathbf{n}^1_q). \tag{17.37}$$

This produces the following $\mathbf{a}^1$ vectors

$$\mathbf{a}^1 = \left\{ \begin{bmatrix} 1 \\ 0.368 \\ 0.018 \end{bmatrix}, \begin{bmatrix} 0.852 \\ 0.698 \\ 0.077 \end{bmatrix}, \begin{bmatrix} 0.527 \\ 0.961 \\ 0.237 \end{bmatrix}, \begin{bmatrix} 0.237 \\ 0.961 \\ 0.527 \end{bmatrix}, \begin{bmatrix} 0.077 \\ 0.698 \\ 0.852 \end{bmatrix}, \begin{bmatrix} 0.018 \\ 0.368 \\ 1 \end{bmatrix} \right\} \tag{17.38}$$

We can use Eq. (17.17) and Eq. (17.21) to create the **U** and **t** matrices

$$\mathbf{U}^T = \begin{bmatrix} 1 & 0.852 & 0.527 & 0.237 & 0.077 & 0.018 \\ 0.368 & 0.698 & 0.961 & 0.961 & 0.698 & 0.368 \\ 0.018 & 0.077 & 0.237 & 0.527 & 0.852 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}, \tag{17.39}$$

$$\mathbf{t}^T = \begin{bmatrix} 0 & 0.19 & 0.69 & 1.3 & 1.8 & 2 \end{bmatrix}. \tag{17.40}$$

The next step is to solve for the weights and biases in the second layer using Eq. (17.30). We will begin with the regularization parameter set to zero.

$$\begin{aligned} \mathbf{x}^* &= [\mathbf{U}^T\mathbf{U} + \rho\mathbf{I}]^{-1}\mathbf{U}^T\mathbf{t} \\ &= \begin{bmatrix} 2.07 & 1.76 & 0.42 & 2.71 \\ 1.76 & 3.09 & 1.76 & 4.05 \\ 0.42 & 1.76 & 2.07 & 2.71 \\ 2.71 & 4.05 & 2.71 & 6 \end{bmatrix}^{-1} \begin{bmatrix} 1.01 \\ 4.05 \\ 4.41 \\ 6 \end{bmatrix} = \begin{bmatrix} -1.03 \\ 0 \\ 1.03 \\ 1 \end{bmatrix} \end{aligned} \tag{17.41}$$

The second layer weight and bias are therefore

$$\mathbf{W}^2 = \begin{bmatrix} -1.03 & 0 & 1.03 \end{bmatrix}, \mathbf{b}^2 = \begin{bmatrix} 1 \end{bmatrix}. \tag{17.42}$$

Figure 17.7 illustrates the operation of this RBF network. The blue line represents the RBF approximation, and the circles represent the six data points. The dotted lines in the upper axis represent the individual basis functions scaled by the corresponding weights in the second layer (including the constant bias term). The sum of the dotted lines will produce the blue line. In the small axis at the bottom, you can see the unscaled basis functions, which are the outputs of the first layer.

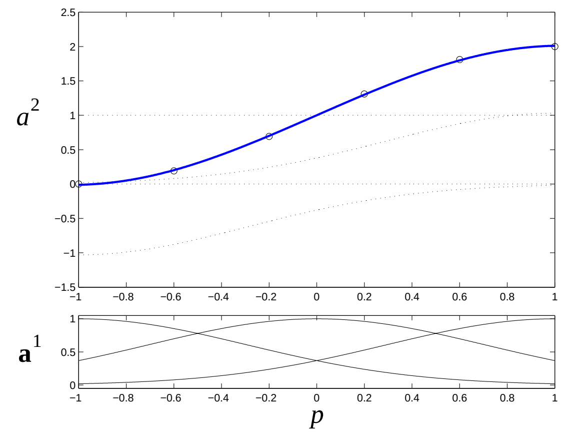


Figure 17.7 RBF Sine Approximation

The RBF network design process can be sensitive to the choice of the center locations and the bias. For example, if we select six basis functions and six data points, and if we choose the first layer biases to be 8, instead of 0.5, then the network response will be as shown in Figure 17.8. The spread of the basis function decreases as the inverse of the bias. When the bias is this large, there is not sufficient overlap in the basis functions to provide a smooth approximation. We match each data point exactly. However, because of the local nature of the basis function, the approximation to the true function is not accurate between the training data points.

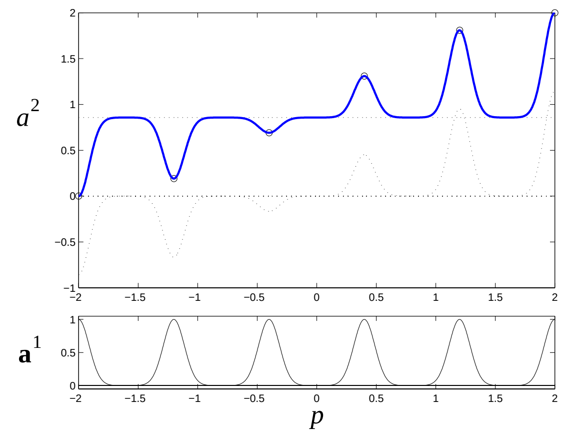


Figure 17.8 RBF Response with Bias Too Large

To experiment with the linear least squares fitting, use the MATLAB® Neural Network Design Demonstration RBF Linear Least Squares (`nnd17lls`).

Orthogonal Least Squares

In the previous section we assumed that the weights and biases in the first layer were fixed. The centers were fixed on a grid, or were randomly selected from the input vectors in the training data set. In this section we consider a different approach for selecting the centers. We will assume that there exists a number of potential centers. These centers could include the entire set of input vectors in the training set, vectors chosen in a grid pattern, or vectors chosen by any other procedure one might think of. We will then select vectors one at a time from this set of potential centers, until the network performance is satisfactory. We will build up the network one neuron at a time.

Subset Selection

The basic idea behind this method comes from statistics, and it is called *subset selection* [Mill90]. The general objective of subset selection is to choose an appropriate subset of independent variables to provide the most efficient prediction of a target dependent variable. For example, suppose that we have 10 independent variables, and we want to use them to predict our target dependent variable. We want to create the simplest predictor possible, so we want to use the minimum number of independent variables for the prediction. Which subset of the 10 independent variables should we use? The optimal approach, called an exhaustive search, tries all combinations of subsets and finds the smallest one that provides satisfactory performance. (We will define later what we mean by satisfactory performance.)

Unfortunately, this strategy is not practical. If we have Q variables in our original set, the following expression gives the number of distinct subsets:

$$\sum_{q=1}^{Q} \frac{Q!}{q!(Q-q)!} . \qquad (17.43)$$

If $Q = 10$, this number is 1023. If $Q = 20$, the number is more than 1 million. We need to have a less expensive strategy than the exhaustive search. There are several suboptimal procedures. They are not guaranteed to find the optimal subset, but they require significantly less computation. One procedure is called *forward selection*. This method begins with an empty model and then adds variables one at a time. At each stage, we add the independent variable that provides the largest reduction in squared error. We stop adding variables when the performance is adequate. Another approach, called *backward elimination*, starts with all independent variables selected for the model. At each stage we eliminate the variable that would cause the least increase in the squared error. The process continues until the performance is inadequate. There are other approaches which combine

Forward Selection

Backward Elimination

forward selection and backward elimination, so that variables can be added and deleted at each iteration.

Any of the standard subset selection techniques can be used for selecting RBF centers. For purposes of illustration, we will consider one specific form of forward selection, called orthogonal least squares [ChCo91]. Its main feature is that it efficiently calculates the error reduction provided by the addition of each potential center to the RBF network.

To develop the orthogonal least squares algorithm, we begin with Eq. (17.22), repeated here in slightly different form:

$$\mathbf{t} = \mathbf{U}\mathbf{x} + \mathbf{e}\,. \tag{17.44}$$

We will use our standard notation for matrix rows and columns to individually identify both the rows and the columns of the matrix **U**:

$$\mathbf{U} = \begin{bmatrix} {}_1\mathbf{u}^T \\ {}_2\mathbf{u}^T \\ \vdots \\ {}_Q\mathbf{u}^T \end{bmatrix} = \begin{bmatrix} \mathbf{z}_1^T \\ \mathbf{z}_2^T \\ \vdots \\ \mathbf{z}_Q^T \end{bmatrix} = \begin{bmatrix} \mathbf{u}_1 & \mathbf{u}_2 & \dots & \mathbf{u}_n \end{bmatrix} \tag{17.45}$$

Here each row of the matrix $\mathbf{U}$ represents the output of layer 1 of the RBF network for one input vector from the training set. There will be a column of the matrix **U** for each neuron (basis function) in layer 1 plus the bias term ($n = S^1 + 1$). Note that for the OLS algorithm, the potential centers for the basis functions are often chosen to be all of the input vectors in the training set. In this case, n will equal $Q + 1$, since the constant “1” for the bias term is included in $\mathbf{z}$, as shown in Eq. (17.17).

Eq. (17.44) is in the form of a standard linear regression model. The matrix $\mathbf{U}$ is called the regression matrix, and the columns of $\mathbf{U}$ are called the regressor vectors.

The objective of OLS is to determine how many columns of $\mathbf{U}$ (numbers of neurons or basis functions) should be used. The first step is to calculate how much each potential column would reduce the squared error. The problem is that the columns are generally correlated with each other, and so it is difficult to determine how much each individual column would reduce the error. For this reason, we need to first orthogonalize the columns. Orthogonalizing the columns means that we can decompose $\mathbf{U}$ as follows:

$$\mathbf{U} = \mathbf{M}\mathbf{R}\,, \tag{17.46}$$

where $\mathbf{R}$ is an upper triangular matrix, with ones on the diagonal:

$$\mathbf{R} = \begin{bmatrix} 1 & r_{1,2} & r_{1,3} & \dots & r_{1,n} \\ 0 & 1 & r_{2,3} & \dots & r_{2,n} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ 0 & 0 & 0 & \dots & r_{n-1,n} \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}, \tag{17.47}$$

and $\mathbf{M}$ is a matrix with orthogonal columns $\mathbf{m}_i$. This means that $\mathbf{M}$ has the following properties

$$\mathbf{M}^T\mathbf{M} = \mathbf{V} = \begin{bmatrix} v_{1,1} & 0 & \dots & 0 \\ 0 & v_{2,2} & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & v_{n,n} \end{bmatrix} = \begin{bmatrix} \mathbf{m}_1^T\mathbf{m}_1 & 0 & \dots & 0 \\ 0 & \mathbf{m}_2^T\mathbf{m}_2 & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & \mathbf{m}_n^T\mathbf{m}_n \end{bmatrix} \tag{17.48}$$

Now Eq. (17.44) can be written

$$\mathbf{t} = \mathbf{MRx} + \mathbf{e} = \mathbf{Mh} + \mathbf{e}, \tag{17.49}$$

where

$$\mathbf{h} = \mathbf{Rx}. \tag{17.50}$$

The least squares solution for Eq. (17.49) is

$$\mathbf{h}^* = [\mathbf{M}^T\mathbf{M}]^{-1}\mathbf{M}^T\mathbf{t} = [\mathbf{V}]^{-1}\mathbf{M}^T\mathbf{t}, \tag{17.51}$$

and because $\mathbf{V}$ is diagonal, the elements of $\mathbf{h}^*$ can be computed

$$h_i^* = \frac{\mathbf{m}_i^T\mathbf{t}}{v_{i,i}} = \frac{\mathbf{m}_i^T\mathbf{t}}{\mathbf{m}_i^T\mathbf{m}_i}. \tag{17.52}$$

From $\mathbf{h}^*$ we can compute $\mathbf{x}^*$ using Eq. (17.50). Since $\mathbf{R}$ is upper-triangular, Eq. (17.50) can be solved by back-substitution and does not require a matrix inversion.

There are a number of ways to obtain the orthogonal vectors $\mathbf{m}_i$, but we will use the Gram-Schmidt orthogonalization process of Eq. (5.20), starting with the original columns of $\mathbf{U}$.

$$\mathbf{m}_1 = \mathbf{u}_1, \tag{17.53}$$

$$\mathbf{m}_k = \mathbf{u}_k - \sum_{i=1}^{k-1} r_{i,k}\mathbf{m}_i , \qquad (17.54)$$

where

$$r_{i,k} = \frac{\mathbf{m}_i^T\mathbf{u}_k}{\mathbf{m}_i^T\mathbf{m}_i}, \; i = 1, \ldots, k-1 . \qquad (17.55)$$

Now let's see how orthogonalizing the columns of $\mathbf{U}$ enables us to efficiently calculate the squared error contribution of each basis vector. Using Eq. (17.49), the total sum square value of the targets is given by

$$\mathbf{t}^T\mathbf{t} = [\mathbf{Mh} + \mathbf{e}]^T[\mathbf{Mh} + \mathbf{e}] = \mathbf{h}^T\mathbf{M}^T\mathbf{Mh} + \mathbf{e}^T\mathbf{Mh} + \mathbf{h}^T\mathbf{M}^T\mathbf{e} + \mathbf{e}^T\mathbf{e} . \qquad (17.56)$$

Consider the second term in the sum:

$$\mathbf{e}^T\mathbf{Mh} = [\mathbf{t} - \mathbf{Mh}]^T\mathbf{Mh} = \mathbf{t}^T\mathbf{Mh} - \mathbf{h}^T\mathbf{M}^T\mathbf{Mh} . \qquad (17.57)$$

If we use the optimal $\mathbf{h}^*$ from Eq. (17.51), we find

$$\mathbf{e}^T\mathbf{Mh}^* = \mathbf{t}^T\mathbf{Mh}^* - \mathbf{t}^T\mathbf{MV}^{-1}\mathbf{M}^T\mathbf{Mh}^* = \mathbf{t}^T\mathbf{Mh}^* - \mathbf{t}^T\mathbf{Mh}^* = 0 . \qquad (17.58)$$

Therefore the total sum square value from Eq. (17.56) becomes

$$\mathbf{t}^T\mathbf{t} = \mathbf{h}^T\mathbf{M}^T\mathbf{Mh} + \mathbf{e}^T\mathbf{e} = \mathbf{h}^T\mathbf{Vh} + \mathbf{e}^T\mathbf{e} = \sum_{i=1}^{n} h_i^2\mathbf{m}_i^T\mathbf{m}_i + \mathbf{e}^T\mathbf{e} . \qquad (17.59)$$

The first term on the right of Eq. (17.59) is the contribution to the sum squared value explained by the regressors, and the second term is the remaining sum squared value that is not explained by the regressors. Therefore, regressor (basis function) i contributes

$$h_i^2\mathbf{m}_i^T\mathbf{m}_i \qquad (17.60)$$

to the squared value. This also represents how much the squared error can be reduced by including the corresponding basis function in the network. We will use this number, after normalizing by the total squared value, to determine the next basis function to include at each iteration:

$$o_i = \frac{h_i^2\mathbf{m}_i^T\mathbf{m}_i}{\mathbf{t}^T\mathbf{t}} . \qquad (17.61)$$

This number always falls between zero and one.

Now let's put all these ideas together into an algorithm for selecting centers.

The OLS Algorithm

To begin the algorithm, we start with all potential basis functions included in the regression matrix $\mathbf{U}$. (As we explained below Eq. (17.45), if all input vectors in the training set are to be considered potential basis function centers, then the $\mathbf{U}$ matrix will be Q by $Q+1$.) This matrix represents only potential basis functions, since we start with no basis functions included in the network.

The first stage of the OLS algorithm consists of the following three steps, for $i = 1, \ldots, Q$:

$$\mathbf{m}_1^{(i)} = \mathbf{u}_i , \tag{17.62}$$

$$h_1^{(i)} = \frac{\mathbf{m}_1^{(i)T}\mathbf{t}}{\mathbf{m}_1^{(i)T}\mathbf{m}_1^{(i)}} , \tag{17.63}$$

$$o_1^{(i)} = \frac{(h_1^{(i)})^2 \mathbf{m}_1^{(i)T}\mathbf{m}_1^{(i)}}{\mathbf{t}^T\mathbf{t}} . \tag{17.64}$$

We then select the basis function that creates the largest reduction in error:

$$o_1 = o_1^{(i_1)} = max\{o_1^{(i)}\} , \tag{17.65}$$

$$\mathbf{m}_1 = \mathbf{m}_1^{(i_1)} = \mathbf{u}_{i_1} . \tag{17.66}$$

The remaining iterations of the algorithm continue as follows (for iteration *k*):

For $i = 1, \ldots, Q$, $i \neq i_1$, ..., $i \neq i_{k-1}$

$$r_{j,k}^{(i)} = \frac{\mathbf{m}_j^T\mathbf{u}_i}{\mathbf{m}_j^T\mathbf{m}_j} , \; j = 1, \ldots, k-1 , \tag{17.67}$$

$$\mathbf{m}_k^{(i)} = \mathbf{u}_i - \sum_{j=1}^{k-1} r_{j,k}^{(i)}\mathbf{m}_j , \tag{17.68}$$

$$h_k^{(i)} = \frac{\mathbf{m}_k^{(i)T}\mathbf{t}}{\mathbf{m}_k^{(i)T}\mathbf{m}_k^{(i)}}, \tag{17.69}$$

$$o_k^{(i)} = \frac{(h_k^{(i)})^2\mathbf{m}_k^{(i)T}\mathbf{m}_k^{(i)}}{\mathbf{t}^T\mathbf{t}}, \tag{17.70}$$

$$o_k = o_k^{(i_k)} = max\{o_k^{(i)}\}, \tag{17.71}$$

$$r_{j,k} = r_{j,k}^{(i_k)},\ j = 1, \dots, k-1. \tag{17.72}$$

$$\mathbf{m}_k = \mathbf{m}_k^{(i_k)}. \tag{17.73}$$

The iterations continue until some stopping criterion is met. One choice of stopping criterion is

$$1 - \sum_{j=1}^{k} o_j < \delta, \tag{17.74}$$

where δ is some small number. However, if δ is chosen too small, we can have overfitting, since the network will become too complex. An alternative is to use a validation set, as we discussed in the chapter on generalization. We would stop when the error on the validation set increased.

After the algorithm has converged, the original weights $\mathbf{x}$ can be computed from the transformed weights $\mathbf{h}$ by using Eq. (17.50). This produces, by back substitution,

$$x_n = h_n,\ x_k = h_k - \sum_{j=k+1}^{n} r_{j,k}x_j, \tag{17.75}$$

where n is the final number of weights and biases in the second layer (adjustable parameters).

To experiment with orthogonal least squares learning, use the MATLAB® Neural Network Design Demonstration RBF Orthogonal Least Squares (**`nnd17ols`**).

Clustering

There is another approach [MoDa89] for selecting the weights and biases in the first layer of the RBF network. This method uses the competitive networks described in Chapter 16. Recall that the competitive layer of Kohonen (see Figure 14.2) and the Self Organizing Feature Map (see Figure

14.9) perform a clustering operation on the input vectors of the training set. After training, the rows of the competitive networks contain prototypes, or cluster centers. This provides an approach for locating centers and selecting biases for the first layer of the RBF network. If we take the input vectors from the training set and perform a clustering operation on them, the resulting prototypes (cluster centers) could be used as centers for the RBF network. In addition, we could compute the variance of each individual cluster and use that number to calculate an appropriate bias to be used for the corresponding neuron.

Consider again the following training set:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \dots, \{\mathbf{p}_Q, \mathbf{t}_Q\} . \quad (17.76)$$

We want to perform a clustering of the input vectors from this training set:

$$\{\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_Q\} . \quad (17.77)$$

We will train the first layer weights of the RBF network to perform a clustering of these vectors, using the Kohonen learning rule of Eq. (14.13), and repeated here:

$${}_{i*}\mathbf{w}^1(q) = {}_{i*}\mathbf{w}^1(q-1) + \alpha(\mathbf{p}(q) - {}_{i*}\mathbf{w}^1(q-1)) , \quad (17.78)$$

where $\mathbf{p}(q)$ is one of the input vectors in the training set, and ${}_{i*}\mathbf{w}^1(q-1)$ is the weight vector that was closest to $\mathbf{p}(q)$. (We could also use other clustering algorithms, such as the Self Organizing Feature Map, or the k-means clustering algorithm, which was suggested in [MoDa89].) As described in Chapter 16, Eq. (17.78) is repeated until the weights have converged. The resulting converged weights will represent cluster centers of the training set input vectors. This will insure that we will have basis functions located in areas where input vectors are most likely to occur.

In addition to selecting the first layer weights, the clustering process can provide us with a method for determining the first layer biases. For each neuron (basis function), locate the n_c input vectors from the training set that are closest to the corresponding weight vector (center). Then compute the average distance between the center and its neighbors.

$$dist_i = \frac{1}{n_c}\left(\sum_{j=1}^{n_c} \left\|\mathbf{p}_j^i - {}_i\mathbf{w}^1\right\|^2\right)^{\frac{1}{2}} \quad (17.79)$$

where $\mathbf{p}_1^i$ is the input vector that closest to ${}_i\mathbf{w}^1$, and is $\mathbf{p}_2^i$ the next closest input vector. From these distances, [MoDa89] recommends setting the first layer biases as follows:

$$b_i^1 = \frac{1}{\sqrt{2}dist_i} . \tag{17.80}$$

Therefore, when a cluster is wide, the corresponding basis function will be wide as well. Notice that in this case each bias in the first layer will be different. This should provide a network that is more efficient in its use of basis functions than a network with equal biases.

After the weights and biases of the first layer are determined, linear least squares is used to find the second layer weights and biases.

There is a potential drawback to the clustering method for designing the first layer of the RBF network. The method only takes into account the distribution of the input vectors; it does not consider the targets. It is possible that the function we are trying to approximate is more complex in regions for which there are fewer inputs. For this case, the clustering method will not distribute the centers appropriately. On the other hand, one would hope that the training data is located in regions where the network will be most used, and therefore the function approximation will be most accurate in those areas.

Nonlinear Optimization

It is also possible to train RBF networks in the same manner as MLP networks - using nonlinear optimization techniques, in which all weights and biases in the network are adjusted at the same time. These methods are not generally used for the full training of RBF networks, because these networks tend to have many more unsatisfactory local minima in their error surfaces. However, nonlinear optimization can be used for the fine-tuning of the network parameters, after initial training by one of the two-stage methods we presented in earlier sections.

We will not present the nonlinear optimization methods in their entirety here, since they were treated extensively in Chapters 11 and 12. Instead, we will simply indicate how the basic backpropagation algorithm for computing the gradient in MLP networks can be modified for RBF networks.

The derivation of the gradient for RBF networks follows the same pattern as the gradient development for MLP networks, starting with Eq. (11.9), which you may wish to review at this time. Here we will only discuss the one step where the two derivations differ. The difference occurs with Eq. (11.20). The net input for the second layer of the RBF network has the same form as its counterpart in the MLP network, but the first layer net input has a different form (as given in Eq. (17.1) and repeated here):

$$n_i^1 = \| \mathbf{p} - {}_i\mathbf{w}^1 \| b_i^1 = b_i^1 \sqrt{\sum_{j=1}^{S^1} (p_j - w_{i,j}^1)^2} . \tag{17.81}$$

If we take the derivative of this function with respect to the weights and biases, we obtain

$$\frac{\partial n_i^1}{\partial w_{i,j}^1} = b_i^1 \frac{1/2}{\sqrt{\sum_{j=1}^{S^1} (p_j - w_{i,j}^1)^2}} 2(p_j - w_{i,j}^1)(-1) = \frac{b_i^1 (w_{i,j}^1 - p_j)}{\left\| \mathbf{p} - {}_i\mathbf{w}^1 \right\|}, \tag{17.82}$$

$$\frac{\partial n_i^1}{\partial b_i^1} = \left\| \mathbf{p} - {}_i\mathbf{w}^1 \right\|. \tag{17.83}$$

This produces the modified gradient equations (compare with Eq. (11.23) and Eq. (11.24)) for Layer 1 of the RBF network

$$\frac{\partial \hat{F}}{\partial w_{i,j}^1} = s_i^1 \frac{b_i^1 (w_{i,j}^1 - p_j)}{\left\| \mathbf{p} - {}_i\mathbf{w}^1 \right\|}, \tag{17.84}$$

$$\frac{\partial \hat{F}}{\partial b_i^1} = s_i^1 \left\| \mathbf{p} - {}_i\mathbf{w}^1 \right\| . \tag{17.85}$$

Therefore, if we look at the summary of the gradient descent backpropagation algorithm for MLP networks, from Eq. (11.44) to Eq. (11.47), we find that the only difference for RBF networks is that we substitute Eq. (17.84) and Eq. (17.85) for Eq. (11.46) and Eq. (11.47) when $m = 1$. When $m = 2$ the original equations remain the same.

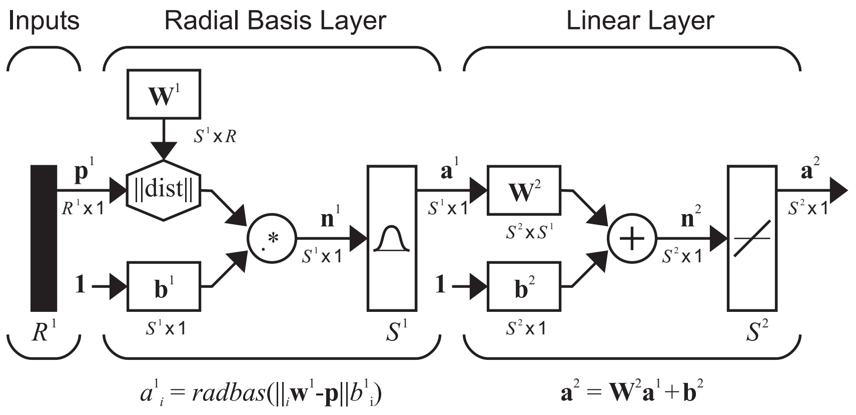
To experiment with nonlinear optimization learning, use the MATLAB® Neural Network Design Demonstration RBF Nonlinear Optimization (**nnd17no**).

Other Training Techniques

In this chapter we have only touched the surface of the variety of training techniques that have been proposed for RBF networks. We have attempted to present the principal concepts, but there are many variations. For example, the OLS algorithm has been extended to handle multiple outputs [ChCo92] and regularized performance indices [ChCh96]. It has also been used in combination with a genetic algorithm [ChCo99], which was used to select the first layer biases and the regularization parameter. The expectation maximization algorithm has also been suggested by several authors for optimizing the center locations, starting with [Bish91]. [OrHa00] used a regression tree approach for center selection. There have also been many variations on the use of clustering and on the combination of clustering for initialization and nonlinear optimization for fine-tuning. The architecture of the RBF network lends itself to many training approaches.

Summary of Results

Radial Basis Network



Training RBF Networks

Linear Least Squares

$$\mathbf{x} = \begin{bmatrix} {}_1\mathbf{w}^2 \\ b^2 \end{bmatrix}, \quad \mathbf{z}_q = \begin{bmatrix} \mathbf{a}^1_q \\ 1 \end{bmatrix}$$

$$\mathbf{t} = \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_Q \end{bmatrix}, \quad \mathbf{U} = \begin{bmatrix} {}_1\mathbf{u}^T \\ {}_2\mathbf{u}^T \\ \vdots \\ {}_Q\mathbf{u}^T \end{bmatrix} = \begin{bmatrix} \mathbf{z}_1^T \\ \mathbf{z}_2^T \\ \vdots \\ \mathbf{z}_Q^T \end{bmatrix}, \quad \mathbf{e} = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_Q \end{bmatrix}$$

$$F(\mathbf{x}) = (\mathbf{t} - \mathbf{U}\mathbf{x})^T(\mathbf{t} - \mathbf{U}\mathbf{x}) + \rho\mathbf{x}^T\mathbf{x}$$

$$[\mathbf{U}^T\mathbf{U} + \rho\mathbf{I}]\mathbf{x}^* = \mathbf{U}^T\mathbf{t}$$

Orthogonal Least Squares

Step 1

$$\mathbf{m}_1^{(i)} = \mathbf{u}_i ,$$

$$h_1^{(i)} = \frac{\mathbf{m}_1^{(i)T}\mathbf{t}}{\mathbf{m}_1^{(i)T}\mathbf{m}_1^{(i)}} ,$$

$$o_1^{(i)} = \frac{(h_1^{(i)})^2\mathbf{m}_1^{(i)T}\mathbf{m}_1^{(i)}}{\mathbf{t}^T\mathbf{t}} .$$

$$o_1 = o_1^{(i_1)} = max\{o_1^{(i)}\}$$

$$\mathbf{m}_1 = \mathbf{m}_1^{(i_1)} = \mathbf{u}_{i_1} .$$

Step *k*

For $i = 1, \ldots, Q$, $i \neq i_1$, ..., $i \neq i_{k-1}$

$$r_{j,k}^{(i)} = \frac{\mathbf{m}_j^T\mathbf{u}_k}{\mathbf{m}_j^T\mathbf{m}_j} , \; j = 1, \ldots, k ,$$

$$\mathbf{m}_k^{(i)} = \mathbf{u}_i - \sum_{j=1}^{k-1} r_{j,k}^{(i)}\mathbf{m}_j ,$$

$$h_k^{(i)} = \frac{\mathbf{m}_k^{(i)T}\mathbf{t}}{\mathbf{m}_k^{(i)T}\mathbf{m}_k^{(i)}} ,$$

$$o_k^{(i)} = \frac{(h_k^{(i)})^2\mathbf{m}_k^{(i)T}\mathbf{m}_k^{(i)}}{\mathbf{t}^T\mathbf{t}} ,$$

$$o_k = o_k^{(i_k)} = max\{o_k^{(i)}\} ,$$

$$\mathbf{m}_k = \mathbf{m}_k^{(i_k)} .$$

Clustering

Training the weights

$$ {}_{i^*}\mathbf{w}^1(q) = {}_{i^*}\mathbf{w}^1(q-1) + \alpha(\mathbf{p}(q) - {}_{i^*}\mathbf{w}^1(q-1)) $$

Selecting the bias

$$ dist_i = \frac{1}{n_c}\left(\sum_{j=1}^{n_c} \left\| \mathbf{p}_j^i - {}_i\mathbf{w}^1 \right\|^2\right)^{\frac{1}{2}} $$

$$ b_i^1 = \frac{1}{\sqrt{2}dist_i} $$

Nonlinear Optimization

Replace Eq. (11.46) and Eq. (11.47) in standard backpropagation with

$$ \frac{\partial \hat{F}}{\partial w_{i,j}^1} = s_i^1 \frac{b_i^1(w_{i,j}^1 - p_j)}{\left\| \mathbf{p} - {}_i\mathbf{w}^1 \right\|}, $$

$$ \frac{\partial \hat{F}}{\partial b_i^1} = s_i^1 \left\| \mathbf{p} - {}_i\mathbf{w}^1 \right\| . $$

Solved Problems

P17.1 Use the OLS algorithm, to approximate the following function:

$$g(p) = \cos(\pi p) \text{ for } -1 \le p \le 1 .$$

To obtain our training set we will evaluate this function at five values of p:

$$p = \{-1, -0.5, 0, 0.5, 1\} .$$

This produces the targets

$$t = \{-1, 0, 1, 0, -1\} .$$

Perform one iteration of the OLS algorithm. Assume that the inputs in the training set are the potential centers and that the biases are all equal to 1.

First, we compute the outputs of the first layer:

$$n^1_{i,q} = \| \mathbf{p}_q - {}_i\mathbf{w}^1 \| b^1_i ,$$

$$\mathbf{a}^1_q = \mathbf{radbas}(\mathbf{n}^1_q) ,$$

$$\mathbf{a}^1 = \left\{ \begin{bmatrix} 1.000 \\ 0.779 \\ 0.368 \\ 0.105 \\ 0.018 \end{bmatrix}, \begin{bmatrix} 0.779 \\ 1.000 \\ 0.779 \\ 0.368 \\ 0.105 \end{bmatrix}, \begin{bmatrix} 0.368 \\ 0.779 \\ 1.000 \\ 0.779 \\ 0.368 \end{bmatrix}, \begin{bmatrix} 0.105 \\ 0.368 \\ 0.779 \\ 1.000 \\ 0.779 \end{bmatrix}, \begin{bmatrix} 0.018 \\ 0.105 \\ 0.368 \\ 0.779 \\ 1.000 \end{bmatrix} \right\} .$$

We can use Eq. (17.17) and Eq. (17.21) to create the **U** and **t** matrices:

$$\mathbf{U}^T = \begin{bmatrix} 1.000 & 0.779 & 0.368 & 0.105 & 0.018 \\ 0.779 & 1.000 & 0.779 & 0.368 & 0.105 \\ 0.368 & 0.779 & 1.000 & 0.779 & 0.368 \\ 0.105 & 0.368 & 0.779 & 1.000 & 0.779 \\ 0.018 & 0.105 & 0.368 & 0.779 & 1.000 \\ 1.000 & 1.000 & 1.000 & 1.000 & 1.000 \end{bmatrix} ,$$

$$\mathbf{t}^T = \begin{bmatrix} -1 & 0 & 1 & 0 & -1 \end{bmatrix} .$$

Now we perform step one of the algorithm:

$$\mathbf{m}_1^{(i)} = \mathbf{u}_i ,$$

$$\mathbf{m}_1^{(1)} = \begin{bmatrix} 1.000 \\ 0.779 \\ 0.368 \\ 0.105 \\ 0.018 \end{bmatrix}, \mathbf{m}_1^{(2)} = \begin{bmatrix} 0.779 \\ 1.000 \\ 0.779 \\ 0.368 \\ 0.105 \end{bmatrix}, \mathbf{m}_1^{(3)} = \begin{bmatrix} 0.368 \\ 0.779 \\ 1.000 \\ 0.779 \\ 0.368 \end{bmatrix}, \mathbf{m}_1^{(4)} = \begin{bmatrix} 0.105 \\ 0.368 \\ 0.779 \\ 1.000 \\ 0.779 \end{bmatrix},$$

$$\mathbf{m}_1^{(5)} = \begin{bmatrix} 0.018 \\ 0.105 \\ 0.368 \\ 0.779 \\ 1.000 \end{bmatrix}, \mathbf{m}_1^{(6)} = \begin{bmatrix} 1.000 \\ 1.000 \\ 1.000 \\ 1.000 \\ 1.000 \end{bmatrix},$$

$$h_1^{(i)} = \frac{\mathbf{m}_1^{(i)T}\mathbf{t}}{\mathbf{m}_1^{(i)T}\mathbf{m}_1^{(i)}},$$

$$h_1^{(1)} = -0.317 , h_1^{(2)} = -0.045 , h_1^{(3)} = 0.106 , h_1^{(4)} = -0.045 , h_1^{(5)} = -0.317 ,$$
$$h_1^{(6)} = -0.200 ,$$

$$o_1^{(i)} = \frac{(h_1^{(i)})^2 \mathbf{m}_1^{(i)T}\mathbf{m}_1^{(i)}}{\mathbf{t}^T\mathbf{t}},$$

$$o_1^{(1)} = 0.0804 , o_1^{(2)} = 0.0016 , o_1^{(3)} = 0.0094 , o_1^{(4)} = 0.0016 , o_1^{(5)} = 0.0804 ,$$
$$o_1^{(6)} = 0.0667 .$$

We see that the first and fifth centers would produce a 0.0804 reduction in the error. This means that the error would be reduced by 8.04%, if the first or fifth center were used in a single-neuron first layer. We would typically select the first center, since it has the smallest index.

If we were to stop at this point, we would add the first center to the hidden layer. Using Eq. (17.75), we would find that $w^2_{1,1} = x_1 = h_1 = h_1^{(1)} = -0.317$. Also, $b_2 = 0$, since the bias center, $\mathbf{m}_1^{(6)}$, was not selected on the first iteration. Note that if we continue to add neurons in the hidden layer, the first weight will change. This can be seen from Eq. (17.75). This equation to find x_k is only used after all of the h_k are found. Only x_n will exactly equal h_n.

If we continued the algorithm, the first column would be removed from $\mathbf{U}$. We would then orthogonalize all remaining columns of $\mathbf{U}$ with respect to $\mathbf{m}_1$, which was chosen on the first iteration, using Eq. (17.54). It is interesting to note that the error reduction on the second iteration would be much higher than the reduction on the first iteration. The sequence of reductions would be 0.0804, 0.3526, 0.5074, 0.0448, 0.0147, 0, and the centers would be chosen in the following order: 1, 2, 5, 3, 4, 6. The reason that reductions in later iterations are higher is that it takes a combination of basis functions to produce the best approximation. This is why forward selection is not guaranteed to produce the optimal combination, which can be found with an exhaustive search. Also, notice that the bias is selected last, and it produces no reduction in the error.

P17.2 **Figure P17.1 illustrates a classification problem, where Class I vectors are represented by dark circles, and Class II vectors are represented by light circles. These categories are not linearly separable. Design a radial basis function network to correctly classify these categories.**

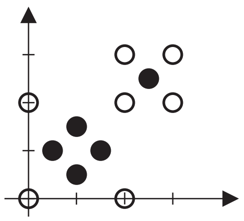


Figure P17.1 Classification Problem for Problem P17.2

From the problem statement, we know that the network will need to have two inputs, and we can use one output to distinguish the two classes. We will choose a positive output for Class I vectors, and a negative output for Class II vectors. The Class I region is made up of two simple subregions, and it appears that two neurons should be sufficient to perform the classification. The rows of the first-layer weight matrix will create centers for the two basis functions, and we will choose each center to be located in the middle of one subregion. By centering a basis function in each subregion, we can produce maximum network outputs there. The first layer weight matrix is then

$$\mathbf{W}^1 = \begin{bmatrix} 1 & 1 \\ 2.5 & 2.5 \end{bmatrix}.$$

The choice of the biases in the first layer depends on the width that we want for each basis function. For this problem, the first basis function should be wider than the second. Therefore, the first bias will be smaller than the second bias. The boundary formed by the first basis function should have a radius of approximately 1, while the second basis function boundary should have a radius of approximately $1/2$. We want the basis functions to drop significantly from their peaks in these distances. If we use a bias of 1 for the first neuron and a bias of 2 for the second neuron, we get the following reductions within one radius of the centers:

$$a = e^{-n^2} = e^{-(1\cdot 1)^2} = e^{-1} = 0.3679, \quad a = e^{-n^2} = e^{-(2\cdot 0.5)^2} = e^{-1} = 0.3679$$

This will work for our problem, so we select the first layer bias vector to be

$$\mathbf{b}^1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}.$$

The original basis function response ranges from 0 to 1 (see Figure 17.1). We want the output to be negative for inputs outside the decision regions, so we will use a bias of -1 for the second layer, and we will use a value of 2 for the second layer weights, in order to bring the peaks back up to 1. The second layer weights and biases then become

$$\mathbf{W}^2 = \begin{bmatrix} 2 & 2 \end{bmatrix}, \quad b^2 = \begin{bmatrix} -1 \end{bmatrix}.$$

For these network parameter values, the network response is shown on the right side of Figure P17.2. This figure also shows where the surface intersects the plane at $a^2 = 0$, which is where the decision takes place. This is also indicated by the contours shown underneath the surface. These are the function contours where $a^2 = 0$. These decision regions are shown more clearly on the left side of Figure P17.2.

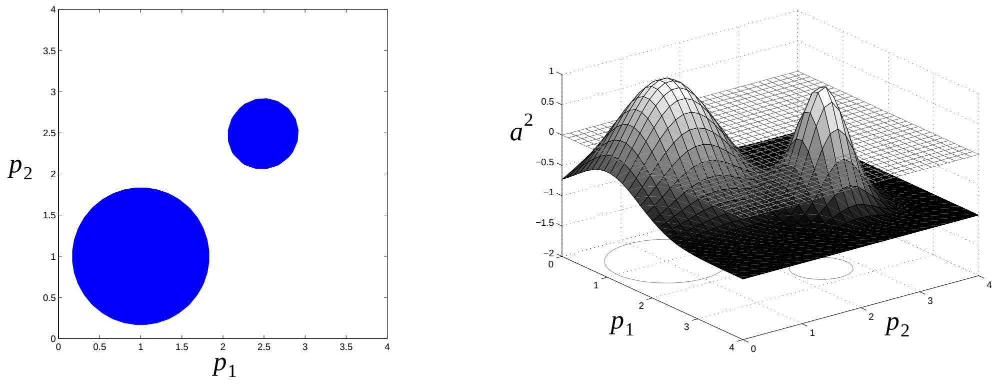


Figure P17.2 Decision Regions for Problem P17.2

P17.3 For an RBF network with one input and one neuron in the hidden layer, the initial weights and biases are chosen to be

$$w^1(0) = 0\ ,\ b^1(0) = 1\ ,\ w^2(0) = -2\ ,\ b^2(0) = 1\ .$$

An input/target pair is given to be

$$((p = -1),(t = 1))\ .$$

Perform one iteration of steepest descent backpropagation with $\alpha = 1$**.**

The first step is to propagate the input through the network.

$$n^1 = \|p - w^1\| b^1 = 1\sqrt{(-1-0)^2} = 1$$

$$a^1 = radbas(n^1) = e^{-n^2} = e^{-1} = 0.3679$$

$$n^2 = w^2 a^1 + b^2 = (-2)(0.3679) + 1 = 0.2642$$

$$a^2 = purelin(n^2) = n^2 = 0.2642$$

$$e = (t - a^2) = (1 - (0.2642)) = 0.7358$$

Now we backpropagate the sensitivities using Eq. (11.44) and Eq. (11.45).

$$\mathbf{s}^2 = -2\dot{\mathbf{F}}^2(\mathbf{n}^2)(\mathbf{t} - \mathbf{a}) = -2[1](e) = -2[1]0.7358 = -1.4716$$

$$\mathbf{s}^1 = \dot{\mathbf{F}}^1(\mathbf{n}^1)(\mathbf{W}^2)^T\mathbf{s}^2 = [2n^1]w^2\mathbf{s}^2 = [2 \times 1](-2)(-1.4716) = 5.8864$$

Finally, the weights and biases are updated using Eq. (11.46) and Eq. (11.47) for Layer 2, and Eq. (17.84) and Eq. (17.85) for Layer 1:

$$w^2(1) = w^2(0) - \alpha s^2 (a^1)^T = (-2) - 1(-1.4716)(0.3679) = -1.4586\quad ,$$

$$w^1(1) = w^1(0) - \alpha s^1\left(\frac{b^1(w^1 - p)}{\|p - w^1\|}\right) = (0) - 1(5.8864)\left(\frac{1(0-(-1))}{\|-1-0\|}\right) = -5.8864\,,$$

$$b^2(1) = b^2(0) - \alpha s^2 = 1 - 1(-1.4716) = 2.4716\,,$$

$$b^1(1) = b^1(0) - \alpha s^1\|p - w^1\| = 1 - 1(5.8864)\|-1-0\| = -4.8864\,.$$

Epilogue

The radial basis function network is an alternative to the multilayer perceptron network for problems of function approximation and pattern recognition. In this chapter we have demonstrated the operation of the RBF network, and we have described several techniques for training the network. Unlike the MLP network, RBF training usually consists of two stages. In the first stage, the weights and biases in the first layer are found. In the second stage, which typically involves linear least squares, the second layer weights and biases are calculated.

Further Reading

[Bish91] C. M. Bishop, "Improving the generalization properties of radial basis function neural networks," *Neural Computation*, Vol. 3, No. 4, pp. 579-588, 1991.

First published use of the expectation-maximization algorithm for optimizing cluster centers for the radial basis network.

[BrLo88] D.S. Broomhead and D. Lowe, "Multivariable function interpolation and adaptive networks," *Complex Systems*, vol.2, pp. 321-355, 1988.

This seminal paper describes the first use of radial basis functions in the context of neural networks.

[ChCo91] S. Chen, C.F.N. Cowan, and P.M. Grant, "Orthogonal least squares learning algorithm for radial basis function networks," *IEEE Transactions on Neural Networks*, Vol.2, No.2, pp.302-309, 1991.

The first description of the use of the subset selection technique for selecting centers for radial basis function networks.

[ChCo92] S. Chen, P. M. Grant, and C. F. N. Cowan, "Orthogonal least squares algorithm for training multioutput radial basis function networks," *Proceedings of the Institute of Electrical Engineers*, Vol. 139, Pt. F, No. 6, pp. 378–384, 1992.

This paper extends the orthogonal least squares algorithm to the case of multiple outputs.

[ChCh96] S. Chen, E. S. Chng, and K. Alkadhimi, "Regularised orthogonal least squares algorithm for constructing radial basis function networks," *International Journal of Control*, Vol. 64, No. 5, pp. 829–837, 1996.

Modifies the orthogonal least squares algorithm to handle regularized performance indices.

[ChCo99] S. Chen, C.F.N. Cowan, and P.M. Grant, "Combined Genetic Algorithm Optimization and Regularized Orthogonal Least Squares Learning for Radial Basis Function Networks," *IEEE Transactions on Neural Networks*, Vol.10, No.5, pp.302-309, 1999.

Combines a genetic algorithm with orthogonal least squares to compute the regularization parameter and basis function spread, while also selecting centers and solving for the optimal second layer weights of radial basis function networks.

[Lowe89] D. Lowe, "Adaptive radial basis function nonlinearities, and the problem of generalization," *Proceedings of the First IEE International Conference on Artificial Neural Networks*, pp. 171 - 175, 1989.

This paper describes the use of gradient-based algorithms for training all of the parameters of an RBF network, including basis function centers and widths. It also provides a formula for setting the basis function widths, if the centers are randomly chosen from the training data set.

[Mill90] A.J. Miller, *Subset Selection in Regression*. Chapman and Hall, N.Y., 1990.

This book provides a very complete and clear discussion of the general problem of subset selection. This involves choosing an appropriate subset from a large set of independent input variables, in order to provide the most efficient prediction of some dependent variable.

[MoDa89] J. Moody and C.J. Darken, "Fast Learning in Networks of Locally-Tuned Processing Units," *Neural Computation*, Vol. 1, pp. 281–294, 1989.

The first published use of clustering methods to find radial basis function centers and variances.

[OrHa00] M. J. Orr, J. Hallam, A. Murray, and T. Leonard, "Assessing rbf networks using delve," IJNS, 2000.

This paper compares a variety of methods for training radial basis function networks. The methods include forward selection with regularization and also regression trees.

[Powe87] M.J.D. Powell, "Radial basis functions for multivariable interpolation: a review," *Algorithms for Approximation*, pp. 143-167, Oxford, 1987.

This paper provides the definitive survey of the original work on radial basis functions. The original use of radial basis functions was in exact multivariable interpolation.

[PaSa93] J. Park and I.W. Sandberg, "Universal approximation using radial-basis-function networks," *Neural Computation*, vol. 5, pp. 305-316, 1993.

This paper proves the universal approximation capability of radial basis function networks.

Exercises

E17.1 Design an RBF network to perform the classification illustrated in Figure E17.1. The network should produce a positive output whenever the input vector is in the shaded region and a negative output otherwise.

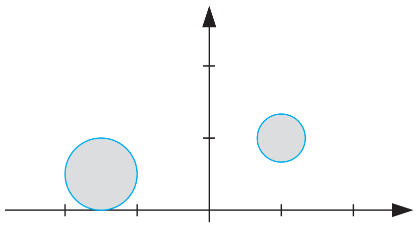


Figure E17.1 Pattern Classification Regions

E17.2 Choose the weights and biases for an RBF network with two neurons in the hidden layer and one output neuron, so that the network response passes through the points indicated by the blue circles in Figure E17.2.

Use the MATLAB® Neural Network Design Demonstration RBF Network Function (**nnd17rbnf**) *to check your result.*

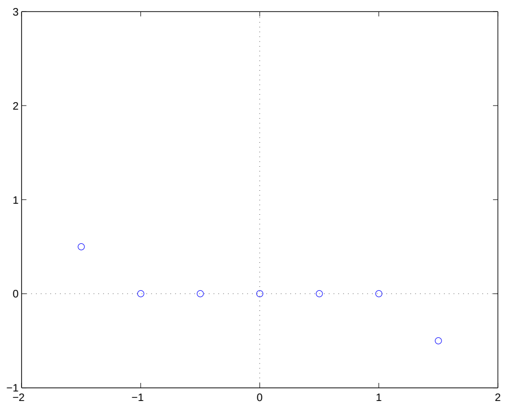


Figure E17.2 Function Approximation Exercise

E17.3 Consider a 1-2-1 RBF network (two neurons in the hidden layer and one output neuron). The first layer weights and biases are fixed as follows:

$$\mathbf{W}^1 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, \mathbf{b}^1 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}.$$

Assume that the bias in the second layer is fixed at 0 ($b^2 = 0$). The training set has the following input/target pairs:

$$\{p_1 = 1, t_1 = -1\}, \{p_2 = 0, t_2 = 0\}, \{p_3 = -1, t_3 = 1\}.$$

- **i.** Use linear least squares to solve for the second layer weights, assuming that the regularization parameter $\rho = 0$.
- **ii.** Plot the contour plot for the sum squared error. Recall that it will be a quadratic function. (See Chapter 8.)
- **iii.** Write a MATLAB® M-file to check your answers to parts i. and ii.
- **iv.** Repeat parts i. to iii., with $\rho = 4$. Plot regularized squared error.

E17.4 The Hessian matrix for the performance index of the RBF network, given in Eq. (17.25), is

$$2[\mathbf{U}^T\mathbf{U} + \rho\mathbf{I}].$$

Show that this matrix is at least positive semidefinite for $\rho \geq 0$, and show that it is positive definite if $\rho > 0$.

E17.5 Consider an RBF network with the weights and biases in the first layer fixed. Show how the LMS algorithm of Chapter 10 could be modified for learning the second layer weights and biases.

E17.6 Suppose that a Gaussian transfer function in the first layer of the RBF network is replaced with a linear transfer function.

- **i.** In Solved Problem P11.8, we showed that a multilayer perceptron with linear transfer functions in each layer is equivalent to a single-layer perceptron. If we use a linear transfer function in each layer of an RBF network, is that equivalent to a single-layer network? Explain.
- **ii.** Work out an example, equivalent to Figure 17.4, to demonstrate the operation of the RBF network with linear transfer function in the first layer. Use MATLAB® to plot your figures. Do you think that the RBF network will be a universal approximator, if the first layer transfer function is linear? Explain your answer.

E17.7 Write a MATLAB® program to implement the linear least squares algorithm for the $1 - S^1 - 1$ RBF network with first layer weights and biases fixed. Train the network to approximate the function

$$g(p) = 1 + \sin\left(\frac{\pi}{8}p\right) \quad \text{for } -2 \le p \le 2 .$$

i. Select 10 training points at random from the interval $-2 \le p \le 2$.

ii. Select four basis function centers evenly spaced on the interval $-2 \le p \le 2$. Then, use Eq. (17.9) to set the bias. Finally, use linear least squares to find the second layer weights and biases, assuming that there is no regularization. Plot the network response for $-2 \le p \le 2$, and show the training points on the same plot. Compute the sum squared error over the training set.

iii. Double the bias from part ii and repeat.

iv. Decrease the bias by half from part ii, and repeat.

v. Compare the final sum squared errors for all cases and explain your results.

E17.8 Use the function described in Exercise E17.7, and use an RBF network with 10 neurons in the hidden layer.

i. Repeat Exercise E17.7 ii. with regularization parameter $\rho = 0.2$. Describe the changes in the RBF network response.

ii. Add random noise to the training targets. Repeat Exercise E17.7 ii. with no regularization and with regularization parameter $\rho = 0.2, 2, 20$. Which case produces the best results. Explain.

E17.9 Write a MATLAB® program to implement the orthogonal least squares algorithm. Repeat Exercise E17.7 using the orthogonal least squares algorithm. Use the 10 random training point inputs as the potential centers, and use Eq. (17.9) to set the bias. Use only the first four selected centers. Compare your final sum squared errors with the result from E17.7 part ii.

E17.10 Write a MATLAB® program to implement the steepest descent algorithm for the $1 - S^1 - 1$ RBF network. Train the network to approximate the function

$$g(p) = 1 + \sin\left(\frac{\pi}{8}p\right) \quad \text{for } -2 \le p \le 2 .$$

You should be able to use a slightly modified version of the program you wrote for Exercise E11.11.

i. Select 10 data points at random from the interval $-2 \le p \le 2$.

ii. Initialize all parameters (weights and biases in both layers) as small random numbers, and then train the network to convergence. (Experiment with the learning rate α, to determine a stable value.) Plot the network response for $-2 \le p \le 2$, and show the training points on the same plot. Compute the sum squared error over the training set. Use 2, 4 and 8 centers.

iii. Repeat part ii., but use a different method for initializing the parameters. Start by setting the parameters as follows. First, select basis function centers evenly spaced on the interval $-2 \le p \le 2$. Then, use Eq. (17.9) to set the bias. Finally, use linear least squares to find the second layer weights and biases. Compute the squared error for these initial weights and biases. Starting from these initial conditions, train all parameters with steepest descent.

iv. Compare the final sum squared errors for all cases and explain your results.

E17.11 Suppose that a radial basis function layer (Layer 1 of the RBF network) were used in the second or third layer of a multilayer network. How could you modify the backpropagation equation, Eq. (11.35), to accommodate this change. (Recall that the weight update equations would be modified from Eq. (11.23) and Eq. (11.24) to Eq. (17.84) and Eq. (17.85).)

E17.12 Consider again Exercise E14.7, in which you trained a feature map to cluster the input space

$$0 \le p_1 \le 1, \; 2 \le p_2 \le 3 .$$

Assume that over this input space, we wish to use an RBF network to approximate the following function:

$$t = \sin(2\pi p_1)\cos(2\pi p_2) .$$

```
» 2 + 2
ans =
     4
```

i. Use MATLAB to randomly generate 200 input vectors in the region shown above.

ii. Write a MATLAB M-file to implement a four-neuron by four-neuron (two-dimensional) feature map. Calculate the net input by finding the distance between the input and weight vectors directly, as is done by the LVQ network, so the vectors do not need to be normalized. Use the feature map to cluster the input vectors.

iii. Use the trained feature map weight matrix from part ii as the weight matrix of the first layer of an RBF network. Use Eq. (17.79) to determine the average distance between each cluster and its center, and then use Eq. (17.80) to set the bias for each neuron in the first layer of the RBF network.

iv. For each of the 200 input vectors in part i, compute the target response for the function above. Then use the resulting input/target pairs to determine the second-layer weights and bias for the RBF network.

v. Repeat parts ii to iv, using a two by two feature map. Compare your results.