Graphically, we can see the classes in the plane in Figure 10.5. There are several ways of performing the desired mapping- for example, the outputs could be $1, 2, 3, 4$. But this may have unintended consequences. In this case, the metric between outputs would imply that Class 4 is much farther away from Class 1 than Class 3. A better method may be to take Class 1 to be the vector $[-1, -1]^T$, Class 2 is the vector $[-1, 1]^T$, Class 3 is $[1, -1]^T$, and Class 4 is $[1, 1]^T$. Now the 4 classes are on the vertices of a square.

Now for the details of the program. First write the inputs as an 2×8 matrix, with a corresponding output matrix that is also 2×8.

Set the learning to 0.04. Set the initial weights to the 2×2 identity, and the initial bias vector b to $[1, 1]^T$.

Let the program run until you think it has converged (this is your choice). We can also set an error bound so that we might stop early. In our code, we would add an "if-statement" that breaks off the computation if our error is small enough. In the code (like after assigning a value to α, you would set a value for the error tolerance, `ErrTol` in this case. Then we would add:

```
if EpochErr(k)<ErrTol
   break;
end
```

See if you can determine where this code fragment should be added.

Finally, plot the points $\mathbf{x} = [x, y]^T$ so that $W\mathbf{x} + \mathbf{b} = \mathbf{0}$, which will be the two lines:

$$W_{11}x + W_{12}y + b_1 = 0, \quad W_{21}x + W_{22}y + b_2 = 0$$

These lines form what is called the *decision boundary*. Here is one way to do that:

```
tt=linspace( min(X(1,:)), max(X(2,:)) );
yy1=(W(1,1)/W(1,2))*tt+(b(1)/W(1,2));
yy2=(W(2,1)/W(2,2))*tt+(b(2)/W(2,2));
plot(tt,yy1,'r',tt,yy2,'b');
hold on
plot(X(1,:),X(2,:),'k*');
hold off
```

10.5 Batch Training

In batch training, all of the data is available to us at the start. In this case, we will assume that that we are working with the linearized version of the neural net, where the matrix of weights is appended with the bias vector at the end, and the input vectors $\boldsymbol{x}$ have an extra "1" appended to them.

To be specific about dimensions, suppose we have p data pairs, where the domain is in $\mathbb{R}^n$ and the range is in $\mathbb{R}^m$. Then for each $\boldsymbol{x}^{(i)} \in \mathbb{R}^n$, the linear network output is $\boldsymbol{y}^{(i)}$ that hopefully approximates our target $\boldsymbol{t}^{(i)}$. This implies that the weight matrix W is $m \times n$ and $\boldsymbol{b} \in \mathbb{R}^m$:

$$W\boldsymbol{x}^{(i)} + \boldsymbol{b} = \boldsymbol{y}^{(i)} \approx \boldsymbol{t}^{(i)}$$

We can put this in matrix form:

$$W\left[\boldsymbol{x}^{(1)},\ \boldsymbol{x}^{(2)},\ \ldots,\ \boldsymbol{x}^{(p)}\right] + [\boldsymbol{b},\ \boldsymbol{b},\ \ldots,\ \boldsymbol{b}] = \left[\boldsymbol{y}^{(1)},\ \boldsymbol{y}^{(2)},\ \ldots,\ \boldsymbol{y}^{(p)}\right]$$

We transform this into a linear problem by appending $\boldsymbol{b}$ to the last column of W and we put a row of 1's as the bottom row in the data:

$$\hat{W} = [W \quad \boldsymbol{b}], \qquad \hat{X} = \begin{bmatrix} \boldsymbol{x}^{(1)} & \boldsymbol{x}^{(2)} & \ldots & \boldsymbol{x}^{(p)} \\ 1 & 1 & & 1 \end{bmatrix}$$

Now the output of the linear network is simply

$$Y = \hat{W}\hat{X}$$

and we solve the following system of equations for $\hat{W}$:

$$\hat{W}\hat{X} = T$$

(Be sure you can give the dimensions of each of those three arrays!)

In this form, we are solving the following for X: $XA = B$. In Matlab, we will use the slash command to solve this. If the matrix A is invertible, the slash will compute the inverse, and if the matrix is not invertible, the slash command will find the least squares solution. In either case, in Matlab, our matrix is easy to find:

```
hatW=T/hatX
```

Example: Associate Memory, Part II

Going back over the associate memory model, we'll try to solve the problem using batch training. The first part of the code is the same until after we set up the vector T. Use Matlab's matrix division to solve it. Then we have:

```
%% Main code start

X=[T1(:) T2(:) G1(:) G2(:) F1(:) F2(:)]; %X is 16 x 6
T=[60 60 0 0 -60 -60];
NumPoints=6;

hatX=[X;ones(1,NumPoints)];  %Puts a row of 1's in the bottom

hatW=T/hatX;

W=hatW(1:16);
b=hatW(17);

%% Output Results
Y=W*X+b*ones(1,NumPoints)
```

Exercise:

Find the linear neural network using batch training for the mapping from X to T (data is ordered) given below.

$$X = \left\{ \begin{pmatrix} 2 \\ 2 \end{pmatrix}, \begin{pmatrix} 1 \\ -2 \end{pmatrix}, \begin{pmatrix} -2 \\ 2 \end{pmatrix}, \begin{pmatrix} -1 \\ 1 \end{pmatrix} \right\} \quad T = \{-1, 1, -1, 1\}$$

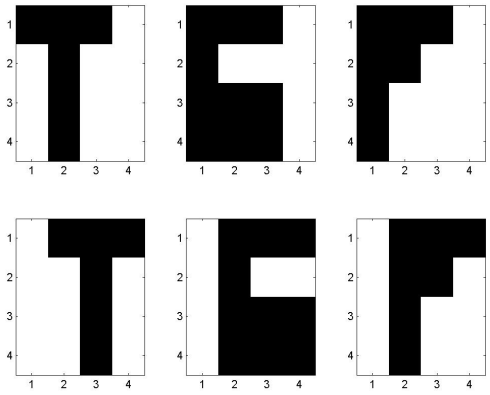

Figure 10.3: The inputs to our linear associative memory model: Three letters, T, G, H, where we have two samples of each letter, and each letter is defined by a 4×4 grid of numbers. We'll be associating T with -60, G with 0, and H with 60.

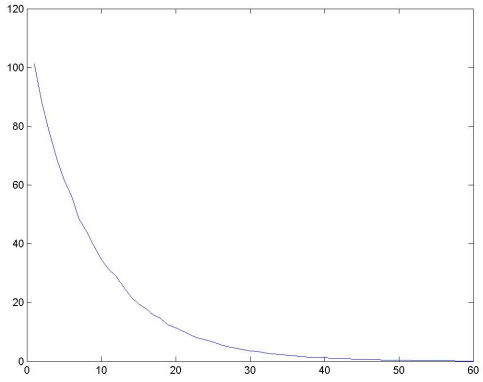

Figure 10.4: The errors for the Widrow-Hoff rule applied to letter recognition (or associative memory. After 60 passes through the data, the associations are very good.

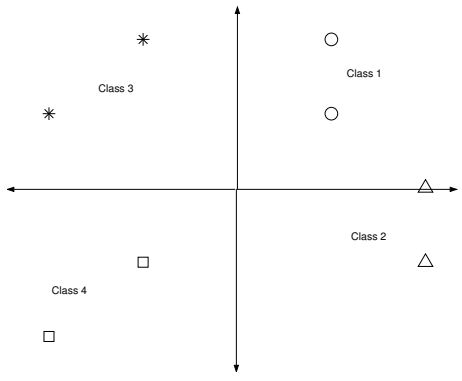

Figure 10.5: Pattern Classification Problem. Each point is a sample of one of the four classes.

Chapter 11

Radial Basis Functions

In this chapter, we begin the modeling of nonlinear relationships in data- in particular, we build up the machinery that will model arbitrary continuous maps from our domain set in $\mathbb{R}^n$ to the desired target set in $\mathbb{R}^m$. The fact that our theory can handle *any* relationship is both satisfying (we are approximating something that theoretically exists) and unsatisfying (we might end up modeling the data **and** the noise.

We've brought up these conflicting goals before- We can build more and more accurate models by increasing the complexity and number of parameters used, but we do so at the expense of *generalization*. We will see this tension in the sections below, but before going further into the nonlinear functions, we first want to look at the general process of modeling data with functions.

11.1 The Process of Function Approximation

There are an infinite number of functions that can model a finite set of domain, range pairings. With that in mind, it will come as no surprise that we will need to make some decisions about what kind of data we're working with, and what kinds of functions make sense to us- Although many of the algorithms in this chapter can be thought of as "black boxes", we still need to always be sure that we assume only as much as necessary, and that the outcome makes sense.

To begin, consider the data. There are two distinct problems in modeling functions from data: Interpolation and Regression. *Interpolation* is the requirement that the function we're building, f, models each data point exactly, so that the model output is equal to the target value at each pair $(\boldsymbol{x}^{(i)}, \boldsymbol{t}^{(i)})$:

$$\boldsymbol{y}^{(i)} = f(\mathbf{x}^{(i)}) = \mathbf{t}^{(i)}$$

In regression, we minimize the error between the target values, $\mathbf{t}_i$, and the function output, $\boldsymbol{y}_i = f(\mathbf{x}_i)$. If we use p data points, then we minimize an error function- for example, the sum of squared errors using parameters contained in a vector α will be:

$$E(\alpha) = \min_{\alpha} \sum_{i=1}^{p} \|\boldsymbol{t}^{(i)} - \boldsymbol{y}^{(i)}\|^2$$

You may be asking yourself why more accuracy is not always better? The answer is: it depends. For some problems, the data that you are given is extremely accurate- In those cases, accuracy may be of primary importance. But, most often, the data has noise and in that case, too much accuracy means that you're modeling the noise.

In regression problems, we will always break our data into two groups: a *training* set and a *testing* set. It is up to personal preference, but people tend to use about 20% of the data to build the model (get the model parameters), and use the remaining 80% of the data to test how well the model performs (or generalizes). Occasionally, there is a third set of data required- a *validation* set that is also used to choose the model

parameters (for example, to prune or add nodes to the network), but we won't use this set here. By using distinct data sets, we hope to get as much accuracy as we can, while still maintaining a good error on unseen data- We will see how this works later.

11.2 Using Polynomials to Build Functions

We may used a fixed set of basis vectors to build up a function- For example, when we construct a Taylor series for a function, we are using a polynomial approximations to get better and better approximations, and if the error between the series and the function goes to zero as the degree increases, we say that the function is analytic.

If we have two data points, we can construct a line that interpolates those points. Given three points, we can construct an interpolating parabola. In fact, given $p+1$ points, (x_i, t_i), we can construct a degree p polynomial that interpolates that data.

We begin with a model function, a polynomial of degree p:

$$f(x) = a_0 + a_1 x + a_2 x^2 + \ldots a_p x^p$$

and using the data, we construct a set of $p+1$ equations (there are $p+1$ coefficients in a degree p polynomial):

$$\begin{array}{ll} t_1 & = a_p x_1^p + a_{p-2} x_1^{p-2} + \ldots + a_1 x_1 + a_0 \\ t_2 & = a_p x_2^p + a_{p-2} x_2^{p-2} + \ldots + a_1 x_2 + a_0 \\ t_3 & = a_p x_3^p + a_{p-2} x_3^{p-2} + \ldots + a_1 x_3 + a_0 \\ \vdots & \cdots \\ t_{p+1} & = a_p x_{p+1}^p + a_{p-2} x_{p+1}^{p-2} + \ldots + a_1 x_{p+1} + a_0 \end{array}$$

And writing this as a matrix-vector equation:

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{p-1} & x_1^p \\ 1 & x_2 & x_2^2 & \cdots & x_2^{p-1} & x_2^p \\ \vdots & & & & & \vdots \\ 1 & x_{p+1} & x_{p+1}^2 & \cdots & x_{p+1}^{p-1} & x_{p+1}^p \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{p-1} \\ a_p \end{bmatrix} = \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_p \\ t_{p+1} \end{bmatrix} \quad \Rightarrow \quad V\mathbf{a} = \mathbf{t} \tag{11.1}$$

This form comes up so often, we give the matrix V a name- It is called the Vandermonde matrix associated with the data points $x_1, \ldots, x_p$.

The main thing to note here is that the column of 1's starts it off, followed by the powers of `x`. This is not a coincidence- The formula for the determinant of this matrix depends on this form.

In Matlab, there is a built in function, `vander`, that will build V, but we can build it manually as well. For some reason, Matlab's first column is the p^{th} power of `x`, so the columns are reversed from our definition. To build V manually is straightforward, however:

```
V=[x.^0  x  x.^2  ...  x.^(p-1)  x.^p]
```

The matrix V is square, so it may be invertible. In fact, if the data in `x` are all distinct, then V is invertible by the following theorem that we state without proof.

Theorem: Given p real numbers $x_1, x_2, \cdots, x_p$, then the determinant of the $p \times p$ Vandermonde matrix computed by taking columns as increasing powers of the vector (different than Matlab) is:

$$\det(V) = \prod_{1 \le i < j \le p} (x_j - x_i)$$

We'll show this for a couple of simple matrices in the exercises. There is a subtle note to be aware of- We had to make a slight change in the definition in order to get the formula.

Note:

If we're not taking the determinant, we could form the Vandermonde matrix any way we want, as long as we're consistent. In these notes, the Vandermonde matrix will start with the column of ones.

Example: Given the vector $\boldsymbol{x} = [1, 2, 3]^T$, then form the Vandermonde matrix and the determinant in two ways- One the "normal" way, and the other using the theorem.

$$V = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \end{bmatrix} \qquad \Rightarrow \quad \det(V) = 2$$

Using the theorem, set $i = 1$ first, then take $j = 2, 3$: $(x_2 - x_1)(x_3 - x_1)$ then multiply that by $i = 2, j = 3$: $(x_3 - x_2)$ to get:

$$(2 - 1)(3 - 1)(3 - 2) = 2$$

If we reversed the columns of V, we would have a determinant of -2, so the only change by reversing the columns is that you may be off by a negative sign.

Whichever way you code the matrix, however, the following corollary is what we were after:

Corollary: If the data points $x_1, \ldots, x_p$ are all distinct, then the Vandermonde matrix is invertible.

Therefore, the corollary tells us that we can solve the interpolation problem by simply inverting the Vandermonde matrix:

$$\mathbf{a} = V^{-1}\mathbf{y}$$

Polynomial Interpolation is Generally Bad

Using polynomials can be bad for interpolation, as we explore more deeply in a course in Numerical Analysis. For now, consider interpolating a fairly nice and smooth function,

$$f(x) = \frac{1}{1 + x^2}$$

over eleven evenly spaced points between (and including) -5 and 5. Construct the degree 10 polynomial, and the result is shown in Figure 11.1. The graph shows the actual function (black curve), the interpolating points (black asterisks), and the degree 10 polynomial *between* interpolating points (dashed curve). As we can see, there is some really bad generalization- And these wild oscillations we see are typical of polynomials of large degree.

11.2.1 Towards a Regression Problem

To partially solve this bad generalization, we will turn to the regression problem. In this case, we will try to fit a polynomial of much smaller degree k to the p data points, with $k << p$.

We can do this quite simply by removing the columns of the Vandermonde matrix that we do not need- or equivalently, in the exercises we will write a function, `vander1.m` that will build the appropriate $p \times k + 1$ matrix, and then we have the matrix-vector equation:

$$V\mathbf{a} = \mathbf{y}$$

Of course now the matrix V is not invertible (it is not square), so we find the least squares solution to the equation using the pseudo-inverse (either one we constructed from the SVD or Matlab's built-in function using the slash).

Using regression rather than interpolation hopefully solves one problem (the wild oscillations), but introduces a new problem: What should the degree of the polynomial be? Low degree polynomials do a great

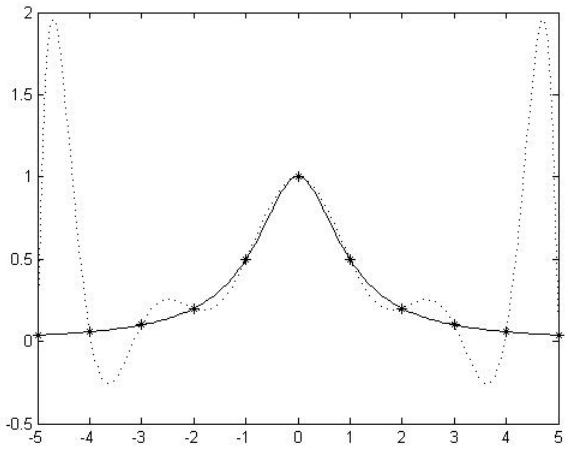

Figure 11.1: An example of the wild oscillations one can get in interpolating data with a high degree polynomial. In this case, we are interpolating 11 data points (asterisks) with a degree 10 polynomial (dotted curve) from the function represented by a solid curve. This is also an example of excellent accuracy on the interpolation data points, but terrible generalization off those points.

job at giving us the general trend of the data, but may be highly inaccurate. High degree polynomials may be exact at the data points used for training, but wildly inaccurate in other places.

There is another reason not to use polynomials. Here is a simple example that will lead us into it:

If you use a polynomial in two variables to approximate a function how many model parameters (unknowns) are there to find?

- If the domain is two dimensional, $z = f(x, y)$, we'll need to construct polynomials in two variables, x and y. The model equation has 6 parameters:

$$p(x, y) = a_0 + a_1 x + a_2 y + a_3 xy + a_4 x^2 + a_5 y^2$$

- If there are 3 domain variables, $w = f(x, y, z)$, a polynomial of degree 2 has 10 parameters:

$$p(x, y, z) = a_0 + a_1 x + a_2 y + a_3 z + a_4 xy + a_5 xz + a_6 yz + a_7 x^2 + a_8 y^2 + a_9 z^2$$

- If there are n domain variables, a polynomial of degree 2 has $(n^2 + 3n + 2)/2$ parameters (see the exercises in this section). For example, using a quite modest degree 2 polynomial with 10 domain variables would take 66 parameters!

The point we want to make in this introduction is: Polynomials do not work well for function building. The explosion in the number of parameters needed for regression is known as **the curse of dimensionality**. Recent results [3, 4] have shown that *any* fixed basis (one that does not depend on the data) will suffer from the curse of dimensionality, and it is this that leads us on to consider a different type of model for nonlinear functions.

Exercises

1. Prove the formula for the determinant of the Vandermonde matrix for the special cases of the 3×3 matrix. Start by row reducing (then find the values for the ??):

$$\begin{vmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \end{vmatrix} \rightarrow (???) \begin{vmatrix} 1 & x_1 & x_1^2 \\ 0 & 1 & ?? \\ 0 & 0 & 1 \end{vmatrix}$$

2. Rewrite Matlab's `vander` command so that `Y=vander1(X,k)` where X is $m \times 1$ (m is the number of points), $k-1$ is the degree of the polynomial and Y is the $m \times k$ matrix whose rows look like those in Equation 11.1.

3. Download and run the sample Matlab file `SampleInterp.m`:

```
f=inline('1./(1+12*x.^2)');
%% Example 1: Seven points
%First define the points for the polynomial:
xp=linspace(-1,1,7);
yp=f(xp);

%Find the coefficients using the Vandermonde matrix:
V=vander(xp);
C=V\yp';

xx=linspace(-1,1,200); %Pts for f, P
yy1=f(xx);
yy2=polyval(C,xx);
plot(xp,yp,'r*',xx,yy1,xx,yy2);
legend('Data','f(x)','Polynomial')
title('Interpolation using 7 points, Degree 6 poly')

%% Example 2: 15 Points
%First define the points for the polynomial:
xp=linspace(-1,1,15);
yp=f(xp);

%Find the coefficients using the Vandermonde matrix:
V=vander(xp);
C=V\yp';

xx=linspace(-1,1,200); %Pts for f, P
yy1=f(xx);
yy2=polyval(C,xx);
figure(2)
plot(xp,yp,'r*',xx,yy1,xx,yy2);
title('Interpolation using 15 points, Degree 14 Poly')
legend('Data','f(x)','Polynomial')

%% Example 3: 30 Points, degree 9
%First define the points for the polynomial:
xp=linspace(-1,1,30);
yp=f(xp);

%Find the coefficients using vander1.m (written by us)
V=vander1(xp',10);
C=V\yp';

xx=linspace(-1,1,200); %Pts for f, P
yy1=f(xx);
yy2=polyval(C,xx);
```

```
figure(3)
plot(xp,yp,'r*',xx,yy1,xx,yy2);
title('Interpolation using 30 points, Degree 10 Poly')
legend('Data','f(x)','Polynomial')
```

Note: In the code, we use one set of domain, range pairs to build up the model (get the values of the coefficients), but then we use a **different** set of domain, range pairs to see how well the model performs. We refer to these sets of data as the training and testing sets.

Which polynomials work "best"?

4. Show that for a second order polynomial in dimension n, there are:

$$\frac{n^2+3n+2}{2} = \frac{(n+2)!}{2!n!}$$

unknown constants, or parameters.

5. In general, if we use a polynomial of degree d in dimension n, there are

$$\frac{(n+d)!}{d!\,n!} \approx n^d$$

parameters. How many parameters do we have if we wish to use a polynomial of degree 10 in dimension 5?

6. Suppose that we have a unit box in n dimensions (corners lie at the points $(\pm 1, \pm 1, \ldots, \pm 1)$), and we wish to subdivide each edge by cutting it into k equal pieces. Verify that there are k^n sub-boxes. Consider what this number will be if k is any positive integer, and n is conservatively large, say $n = 1000$. This rules out any simple, general, "look-up table" type of solution to the regression problem.

11.3 Distance Matrices

A certain matrix called the Euclidean Distance matrix (EDM) will play a natural role in our function buildling problem. Before getting too far, let's define it.

Definition: The Euclidean Distance Matrix (EDM) Let $\{\boldsymbol{x}^{(i)}\}_{i=1}^{P}$ be a set of points in $\mathbb{R}^n$. Let the matrix $\boldsymbol{A}$ be the P by P matrix so that the $(i,j)^{\text{th}}$ entry of $\boldsymbol{A}$ is given by:

$$A_{ij} = \|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\|_2$$

Then $\boldsymbol{A}$ is the Euclidean Distance Matrix (EDM) for the data set $\{\boldsymbol{x}^{(i)}\}_{i=1}^{P}$.

Example: Find the EDM for the one dimensional data: $-1, 2, 3$:

SOLUTION:

$$\begin{bmatrix} |x_1-x_1| & |x_1-x_2| & |x_1-x_3| \\ |x_1-x_2| & |x_2-x_2| & |x_2-x_3| \\ |x_3-x_1| & |x_3-x_2| & |x_3-x_3| \end{bmatrix} = \begin{bmatrix} 0 & 3 & 4 \\ 3 & 0 & 1 \\ 4 & 1 & 0 \end{bmatrix}$$

You might note for future reference that the EDM does not depend on the dimension of the data, but simply on the number of data points. We would get a 3×3 matrix, for example, even if we had three data points in $\mathbb{R}^{1000}$. There is another nice feature about the EDM:

Theorem (Schoenberg, 1937). Let $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_p\}$ be p distinct points in $\mathbb{R}^n$. Then the EDM is invertible.

This theorem implies that, if $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_p\}$ are p distinct points in $\mathbb{R}^n$, and $\{y_1, \ldots, y_p\}$ are points in $\mathbb{R}$, then there exists a function $f : \mathbb{R}^n \to \mathbb{R}$ such that:

$$f(\boldsymbol{x}) = \alpha_1\|\boldsymbol{x} - \boldsymbol{x}_1\| + \ldots + \alpha_p\|\boldsymbol{x} - \boldsymbol{x}_p\| \tag{11.2}$$

and $f(\boldsymbol{x}_i) = y_i$. Therefore, this function f solves the interpolation problem. In matrix form, we are solving:

$$A\boldsymbol{\alpha} = Y$$

where $A_{ij} = \|\boldsymbol{x}_i - \boldsymbol{x}_j\|$ and Y is a column vector.
Example: Use the EDM to solve the interpolation using the data points:

$$(-1, 5) \quad (2, -7) \quad (3, -5)$$

SOLUTION: The model function is:

$$y = \alpha_1|x + 1| + \alpha_2|x - 2| + \alpha_3|x - 3|$$

Substituting in the data, we get the matrix equation:

$$\begin{bmatrix} 0 & 3 & 4 \\ 3 & 0 & 1 \\ 4 & 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} = \begin{bmatrix} 5 \\ -7 \\ -5 \end{bmatrix} \quad \Rightarrow \quad \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} = \begin{bmatrix} -2 \\ 3 \\ -1 \end{bmatrix}$$

So the function is:

$$f(x) = -2|x + 1| + 3|x - 2| - |x - 3|$$

In Matlab, we could solve and plot the function:

```
x=[-1;2;3]; y=[5;-7;-5];
A=edm(x,x);
c=A\y;
xx=linspace(-2,4); %xx is 1 x 100
v=edm(xx',x);  %v is 100 x 3
yy=v*c;
plot(x,y,'r*',xx,yy);
```

Coding the EDM

Computing the Euclidean Distance Matrix is straightforward in theory, but we should be careful in computing it, especially if we begin to have a lot of data.

We will build the code to assume that the matrices are organized as *Number of Points* $\times$ *Dimension*, and give an error message if that is not true. Here is our code. Be sure to write it up and save it in Matlab:

```
function z=edm(w,p)
% A=edm(w,p)
% Input: w, number of points by dimension
% Input: p is number of points by dimension
% Ouput: Matrix z, number points in w by number pts in p
% which is the distance from one point to another
[S,R] = size(w);
[Q,R2] = size(p);
p=p';  %Easier to compute this way

%Error Check:
if (R ~= R2), error('Inner matrix dimensions do not match.\n'), end
```

```
z = zeros(S,Q);
if (Q<S)
   p = p';
   copies = zeros(1,S);
   for q=1:Q
      z(:,q) = sum((w-p(q+copies,:)).^2,2);
   end
else
   w = w';
   copies = zeros(1,Q);
   for i=1:S
      z(i,:) = sum((w(:,i+copies)-p).^2,1);
   end
end
z = z.^0.5;
```

Exercises

1. Revise the Matlab code given for the interpolation problem to work with the following data, where X represents three points in $\mathbb{R}^2$ and Y represents three points in $\mathbb{R}^3$. Notice that in this case, we will not be able to plot the resulting function, but show that you do get Y with your model. (Hint: Be sure to enter X and Y as their transposes to get EDM to work correctly).

$$X = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 1 \end{bmatrix} \qquad Y = \begin{bmatrix} 2 & 1 & -1 \\ 1 & 1 & 1 \\ -1 & 1 & 1 \end{bmatrix}$$

2. As we saw in Chapter 2, the invertibility of a matrix depends on its smallest eigenvalue. A recent theorem states "how invertible" the EDM is:

   Theorem [2]: Let $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_p\}$ be p distinct points in $\mathbb{R}^n$. Let ϵ be the smallest element in the EDM, so that

   $$\|\boldsymbol{x}_i - \boldsymbol{x}_j\| \geq \epsilon, \; i \neq j$$

   Then we have that all eigenvalues λ_i are all bounded away from the origin- there is a constant c so that:

   $$\lambda_i \geq \frac{c\epsilon}{\sqrt{n}} \tag{11.3}$$

   Let's examine what this is saying by verifying it with data in $\mathbb{R}^4$. In Matlab, randomly select 50 points in $\mathbb{R}^4$, and compute the eigenvalues of the EDM and the minimum off-diagonal value of the EDM. Repeat 100 times, and plot the pairs (x_i, y_i), where x_i is the minimum off-diagonal on the i^{th} trial, and y_i is the smallest eigenvalue. Include the line $(x_i, x_i/2)$ for reference.

3. **An interesting problem to think about:** A problem that is related to using the EDM is "Multi-dimensional Scaling": That is, given a $p \times p$ distance or similarity matrix that represents how similar p objects are to one another, construct a set of data points in $\mathbb{R}^n$ (n to be determined, but typically 2 or 3) so that the EDM is as close as possible to the given similarity matrix. A classic test of new algorithms is to give the algorithm a matrix of distances between cities on the map, and see if it produces a close approximation of where those cities are. Of course, one configuration could be given a rigid rotation and give an equivalent EDM, so the solutions are not unique.

11.4 Radial Basis Functions

Definition: A radial basis function (RBF) is a function of the distance of the point to the origin. That is, ϕ is an RBF if $\phi(\mathbf{x}) = \phi(\|\mathbf{x}\|)$, so that ϕ acts on a vector in $\mathbb{R}^n$, but only through the norm. This means that ϕ can be thought of as a scalar function.

This ties us in directly to the EDM, and we modify Equation 11.2 by applying ϕ. This gives us the model function we will use in RBFs:

$$f(\boldsymbol{x}) = \alpha_1\phi\left(\|\boldsymbol{x} - \boldsymbol{x}_1\|\right) + \ldots + \alpha_p\phi\left(\|\boldsymbol{x} - \boldsymbol{x}_p\|\right)$$

where $\phi : \mathbb{R}^+ \to \mathbb{R}$, is typically nonlinear and is referred to as the *transfer function*. The following table gives several common used formulas for ϕ.

Definition: The Transfer Function and Matrix Let $\phi : \mathbb{R}^+ \to \mathbb{R}$ be chosen from the list below.

$$\begin{array}{lcll}
\phi(r,\sigma) & = & \exp\left(\dfrac{-r^2}{\sigma^2}\right) & \text{Gaussian} \\
\phi(r) & = & r^3 & \text{Cubic} \\
\phi(r) & = & r^2\log(r) & \text{Thin Plate Spline} \\
\phi(r) & = & \dfrac{1}{r+1} & \text{Cauchy} \\
\phi(r,\beta) & = & \sqrt{r^2+\beta} & \text{Multiquadric} \\
\phi(r,\beta) & = & \dfrac{1}{\sqrt{r^2+\beta}} & \text{Inverse Multiquadric} \\
\phi(r) & = & r & \text{Identity}
\end{array}$$

We will use the Matlab notation for applying the scalar function ϕ to a matrix- ϕ will be applied element-wise so that:

$$\phi(A)_{ij} = \phi(A_{ij})$$

When ϕ is applied to the EDM, the result is referred to as the *transfer matrix*.

There are other transfer functions one can choose. For a broader definition of transfer functions, see Micchelli [30]. We will examine the effects of the transfer function on the radial approximation shortly, but we focus on a few of them. Matlab, for example, uses only a Gaussian transfer function, which may have some undesired consequences (we'll see in the exercises).

Once we decide on which transfer function we can use, we use the data to find the coefficients $\alpha_1, \ldots, \alpha_p$ by setting up the p equations:

$$\begin{array}{rl}
\alpha_1\phi\left(\|\boldsymbol{x}_1 - \boldsymbol{x}_1\|\right) + \ldots + \alpha_p\phi\left(\|\boldsymbol{x}_1 - \boldsymbol{x}_p\|\right) & = y_1 \\
\alpha_1\phi\left(\|\boldsymbol{x}_2 - \boldsymbol{x}_1\|\right) + \ldots + \alpha_p\phi\left(\|\boldsymbol{x}_2 - \boldsymbol{x}_p\|\right) & = y_2 \\
\vdots \quad \vdots & \\
\alpha_1\phi\left(\|\boldsymbol{x}_p - \boldsymbol{x}_1\|\right) + \ldots + \alpha_p\phi\left(\|\boldsymbol{x}_p - \boldsymbol{x}_p\|\right) & = y_p
\end{array}$$

As long as the vectors $\mathbf{x}_i \neq \mathbf{x}_j$, for $i \neq j$, then the $p \times p$ matrix in this system of equations will be invertible. The resulting function will *interpolate* the data points.

In order to balance the accuracy versus the complexity of our function, rather than using all p data points in the model:

$$f(\boldsymbol{x}) = \alpha_1\phi\left(\|\boldsymbol{x} - \boldsymbol{x}_1\|\right) + \ldots + \alpha_p\phi\left(\|\boldsymbol{x} - \boldsymbol{x}_p\|\right)$$

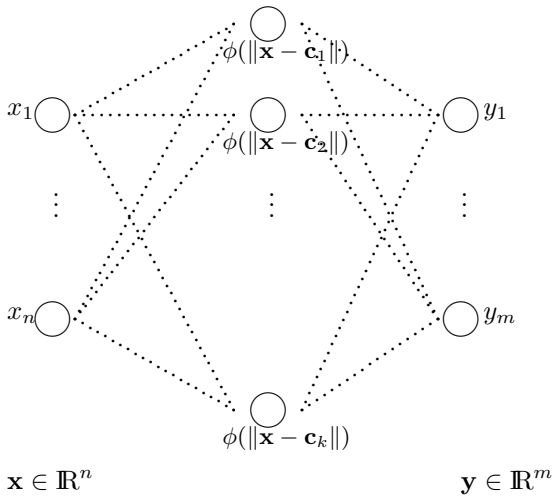

Figure 11.2: The RBF as a neural network. The input layer has as many nodes as input dimensions. The middle layer has k nodes, one for each center $\mathbf{c}_k$. The processing at the middle layer is to first compute the distance from the input vector to the corresponding center, then apply ϕ. The resulting scalar value is passed along to the output layer, $\mathbf{y}$. The last layer is linear in that we will be taking linear combinations of the values of the middle layer.

Following [5], we will use k points, $\mathbf{c}_1, \ldots, \mathbf{c}_k$ for the function:

$$f(\boldsymbol{x}) = \alpha_1 \phi\left(\|\boldsymbol{x} - \boldsymbol{c}_1\|\right) + \ldots + \alpha_p \phi\left(\|\boldsymbol{x} - \boldsymbol{c}_k\|\right)$$

We will refer to these vectors $\mathbf{c}$'s as the centers of the RBF, and typically k will be much smaller than p, the number of data points.

Of course, the simpler function comes at some cost: Interpolation becomes regression, and we have to decide on k and how to place the centers.

Example: If we use one center at the origin, take $\mathbf{x}$ to be in the plane, y be scalar, and $\phi(r) = r^3$, then:

$$f(x_1, x_2) = \alpha(x_1^2 + x_2^2)^{3/2}$$

whose graph is a cone in the plane (vertex at the origin, opening upwards).

Example: Let x, y be scalars, and let us use two centers- $c_1 = -1$ and $c_2 = 2$. The transfer function will be the Gaussian with $\sigma = 1$. Then the model function is:

$$f(x) = \alpha_1 \mathrm{e}^{-(x+1)^2} + \alpha_2 \mathrm{e}^{-(x-2)^2}$$

so the graph will be the linear combination of two Gaussians.

We will generalize the function slightly to assume that the output is multidimensional; that the vector of coefficients, α, becomes a matrix of coefficients, W. We can visualize the RBF as a neural network, as seen in Figure 11.2.

In a slightly different diagrammatic form, we might think of the RBF network as a layer of two mappings,

the first is a mapping of $\mathbb{R}^n$ to $\mathbb{R}^k$, then to the output layer, $\mathbb{R}^m$:

$$\mathbf{x} \rightarrow \begin{bmatrix} \|\mathbf{x}-\mathbf{c}_1\| \\ \|\mathbf{x}-\mathbf{c}_2\| \\ \vdots \\ \|\mathbf{x}-\mathbf{c}_k\| \end{bmatrix} \rightarrow \phi \left(\begin{bmatrix} \|\mathbf{x}-\mathbf{c}_1\| \\ \|\mathbf{x}-\mathbf{c}_2\| \\ \vdots \\ \|\mathbf{x}-\mathbf{c}_k\| \end{bmatrix} \right) \rightarrow \tag{11.4}$$

$$W \begin{bmatrix} \phi(\|\mathbf{x}-\mathbf{c}_1\|) \\ \phi(\|\mathbf{x}-\mathbf{c}_2\|) \\ \vdots \\ \phi(\|\mathbf{x}-\mathbf{c}_k\|) \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

where W is an $m \times k$ matrix of weights. As is our usual practice, we could add a bias vector (shown) as well and keep the problem linear by increasing the size of W to $m \times k+1$

In fact, we are now ready to train the RBF. We see that, once the centers (or points) $\mathbf{c}_1, \ldots, \mathbf{c}_k$ have been fixed, we train the network by using the data to find the matrix W (and possibly the bias vector $\mathbf{b}$). As we've seen before, this can be done at once with all the data (as a batch), or we can update the weights and biases using Widrow-Hoff.

11.4.1 Training the RBF

Training should begin by separating the data into a training and testing set. Having done that, we decide on the number of centers and their placement (these decisions will be investigated more in the next section). We also decide on the transfer function ϕ.

Training proceeds by setting up the linear algebra problem for the weights and biases- We use the diagram in Equation 11.4 for each input $\boldsymbol{x}$ output $\boldsymbol{y}$ pairing to build the system of equations which we will solve using the least squares error.

Let Y be the $m \times p$ matrix constructed so that we have p column vectors in $\mathbb{R}^m$. In matrix form, the system of equations we need to solve is summarized as:

$$W\Phi = Y \tag{11.5}$$

where Φ is $k \times p$- transposed from before; think of the j^{th} column in terms of subtracting the j^{th} data point from each of the k centers:

$$\Phi_{i,j} = \phi(\|\boldsymbol{x}_j - \boldsymbol{c}_i\|)$$

We should increase Φ to $k+1 \times p$ by appending a final row of ones (for the bias terms). This makes the matrix of weights, W have size $m \times k+1$ as mentioned previously. The last column of W corresponds to a vector of biases in $\mathbb{R}^m$.

Finally, we solve for W by using the pseudo-inverse of Φ (either with Matlab's built in `pinv` command or by using the SVD):

$$W = Y\Phi^{\dagger}$$

Now that we have the weights and biases, we can evaluate our RBF network at a new point by using the RBF diagram as a guide (also see the implementation below).

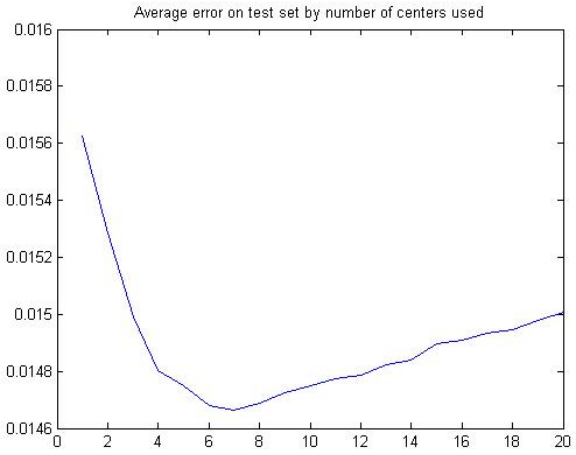

Figure 11.3: The error, averaged over 30 random placements of the centers, of the error on the testing set. The horizontal axis shows the number of centers. As we predicted, the error on the test set initially goes down, but begins to increase as we begin to model the noise in the training set. We would want to stop adding centers at the bottom of the valley shown.

11.4.2 Matlab Notes

Although Matlab has some RBF commands built-in, it is good practice to program these routines in ourselves- Especially since the batch training (least squares solution) is straightforward.

We already have the EDM command to build a matrix of distances. I like to have an additional function, `rbf1.m` that will compute any of the transfer functions ϕ that I would like- and then apply that to any kind of input- scalar, vector or matrix.

We should write a routine that will input a training set consisting of a matrix X and Y, a way of choosing ϕ, and a set of centers C. It should output the weight matrix W (and I would go ahead and separate the bias vector $\boldsymbol{b}$.

When you're done writing, it is convenient to be able to write, as Matlab:

```
Xtrain=...
Xtest=...
Ytrain=...
Ytest=...
Centers=...
phi=2;  %Choose a number from the list

[W,b]=rbfTrain1(Xtrain,Ytrain,Centers,phi);
Z=rbfTest(Xtest,Centers,phi,W,b);
```

To illustrate what happens with training sessions, let's take a look at some. In the following case, we show how the error on the training set tends to go down as we increase the number of centers, but the error on the test set goes up after a while (that is the point at which we would want to stop adding centers). Here is the code that produced the graph in Figure 11.4.2:

```
X=randn(1500,2);
Y=exp(-(X(:,1).^2+X(:,2).^2)./4)+0.5*randn(1500,1);  %Actual data

temp=randperm(1500);
```

```
Xtrain=X(temp(1:300),:); Xtest=X(temp(301:end),:);
Ytrain=Y(temp(1:300),:); Ytest=Y(temp(301:end),:);

for k=1:30
for j=1:20
    NumClusters=j;
    temp=randperm(300);
    C=Xtrain(temp(1:NumClusters),:);

    A=edm(Xtrain,C);
    Phi=rbf1(A,1,3);

    alpha=pinv(Phi)*Ytrain;
    TrainErr(k,j)=(1/length(Ytrain))*norm(Phi*alpha-Ytrain);
    %Compute the error using all the data:
    A=edm(Xtest,C);
    Phi=rbf1(A,1,3);
    Z=Phi*alpha;
    Err(k,j)=(1/length(Ytest))*norm(Ytest-Z);
end
end
figure(1)
plot(mean(TrainErr));
title('Training error tends to always decrease...');
figure(2)
plot(mean(Err));
title('Average error on test set by number of centers used');
```

Using Matlab's Neural Networks Toolbox

For some reason Matlab's Neural Network Toolbox only has Gaussian RBFs. It uses the an approximation to the width as described in the exercises below, and gives you the option of running interpolation or regression, and the regression uses Orthogonal Least Squares, which is described in the next section. Using it is fairly simple, and here are a couple of sample training sessions (from the help documentation):

```
P = -1:.1:1;
T = [-.9602 -.5770 -.0729  .3771  .6405  .6600  .4609 ...
      .1336 -.2013 -.4344 -.5000 -.3930 -.1647  .0988 ...
      .3072  .3960  .3449  .1816 -.0312 -.2189 -.3201];

eg = 0.02; % sum-squared error goal
sc = 1;    % width of Gaussian
net = newrb(P,T,eg,sc);

%Test the network on a new data set X:
X = -1:.01:1;
Y = sim(net,X);
plot(P,T,'+',X,Y,'k-');
```

Issues coming up

Using an RBF so far, we need to make 2 decisions:

1. What should the transfer function be? If it has an extra parameter (like the Gaussian), what should it be?

   There is no generally accepted answer to this question. We might have some external reason for choosing one function over another, and some people stay mainly with their personal favorite.

   Having said that, there are some reasons for choosing the Gaussian in that the exponential function has some attracting statistical properties. There is a rule of thumb for choosing the width of the Gaussian, which is explored further in the exercises:

   The width should be wider than the distance between data points, but smaller than the diameter of the set.

2. How many centers should I use, and where should I put them?

   Generally speaking, use as few centers as possible, while still maintaining a desired level of accuracy. Remember that we can easily zero out the error on the training set, so this is where the testing set can help balance the tradeoff between accuracy and simplicity of the model. In the next section, we will look at an algorithm for choosing centers.

Exercises

1. Write the Matlab code discussed to duplicate the sessions we looked at previously:

   - `function Phi=rbf1(X,C,phi,opt)` where we input data in the matrix X, a matrix of centers C, and a number corresponding to the nonlinear function ϕ. The last input is optional, depending on whether $\phi(r)$ depends on any other parameters.
   - `function [W,b]=rbfTrain(X,Y,C,phi,opt)` Constructs and solves the RBF Equation 11.5
   - `function Z=rbfTest(X,C,W,b,phi,opt)` Construct the RBF Equation and output the result as Z (an $m \times p$ matrix of outputs).

2. The following exercises will consider how we might set the width of the Gaussian transfer function.

   (a) We will approximate:

   $$\left(\int_{-b}^{b} \mathrm{e}^{-x^2}\, dx\right)^2 = \int_{-b}^{b}\int_{-b}^{b} \mathrm{e}^{-(x^2+y^2)}\, dx\, dy \approx \int\int_{B} \mathrm{e}^{-(x^2+y^2)}\, dB$$

   where B is the disk of radius b. Show that this last integral is:

   $$\pi\left(1-\mathrm{e}^{-b^2}\right)$$

   (b) Using the previous exercise, conclude that:

   $$\int_{-\infty}^{\infty} \mathrm{e}^{\frac{-x^2}{\sigma^2}}\, dx = \sigma\sqrt{\pi}$$

   (c) We'll make a working definition of the *width* of the Gaussian: It is the value a so that k percentage of the area is between $-a$ and a (so k is between 0 and 1). The actual value of k will be problem-dependent.

   Use the previous two exercises to show that our working definition of the "width" a, means that, given a we would like to find σ so that:

   $$\int_{-a}^{a} \mathrm{e}^{\frac{-x^2}{\sigma^2}}\, dx \approx k \int_{-\infty}^{\infty} \mathrm{e}^{\frac{-x^2}{\sigma^2}}\, dx$$