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:

$$f(\mathbf{x}_i) = \mathbf{y}_i$$

In regression, we minimize the error between the desired values, $\mathbf{y}_i$, and the function output, $f(\mathbf{x}_i)$. If we use p data points, then we minimize an error function- for example, the sum of squared error:

$$\min_{\alpha} \sum_{i=1}^{p} \|f(\mathbf{x}_i) - \mathbf{y}_i\|^2$$

where the function f depends on the parameters in α.

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, we can construct a degree p polynomial that interpolates that data.

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

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

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

$$\begin{aligned}
y_1 &= a_p x_1^p + a_{p-2} x_1^{p-2} + \dots + a_1 x_1 + a_0 \\
y_2 &= a_p x_2^p + a_{p-2} x_2^{p-2} + \dots + a_1 x_2 + a_0 \\
y_3 &= a_p x_3^p + a_{p-2} x_3^{p-2} + \dots + a_1 x_3 + a_0 \\
\vdots &\quad \dots \\
y_{p+1} &= a_p x_{p+1}^p + a_{p-2} x_{p+1}^{p-2} + \dots + a_1 x_{p+1} + a_0
\end{aligned}$$

And writing this as a matrix-vector equation:

$$\begin{bmatrix}
x_1^p & x_1^{p-1} & x_1^{p-2} & \dots & x_1 & 1 \\
x_2^p & x_2^{p-1} & x_2^{p-2} & \dots & x_2 & 1 \\
\vdots & & & & \vdots & \vdots \\
x_{p+1}^p & x_{p+1}^{p-1} & x_{p+1}^{p-2} & \dots & x_{p+1} & 1
\end{bmatrix}
\begin{bmatrix} a_p \\ a_{p-1} \\ \vdots \\ a_1 \\ a_0 \end{bmatrix}
=
\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_p \\ y_{p+1} \end{bmatrix} \tag{11.1}$$

or more concisely as $V\mathbf{a} = \mathbf{y}$, where V is called the Vandermonde matrix associated with the data points $x_1, \ldots, x_p$. For future reference, note that in Matlab, there is a built in function, `vander`, but we could build V by taking the domain data as a single column vector $\mathbf{x}$, then form the $p+1$ columns as powers of this vector:

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

The matrix V is square, so it may be invertible. In fact,

Example

Consider the following example in Matlab:

```
>> x=[-1;3;2]
x =
    -1
     3
     2
>> vander(x)
ans =
     1    -1     1
     9     3     1
     4     2     1
```

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.

For example, the determinant of our previous 3×3 matrix is (double products can be visualized best as an array):

$$(x_2 - x_1)(x_3 - x_2) \quad (x_3 - x_1) \quad = (3 - -1)(2 - 3)(2 - -1) = -12$$

In Matlab, look at the following:

```
det(vander(x))
det([x.^0 x.^1 x.^2])
```

(This is an unfortunate consequence of how Matlab codes up the matrix, but it will not matter for us). Of great importance to us is the following Corollary, that follows immediately from the theorem:
Corollary: If the data points $x_1, \ldots, x_p$ are all distinct, then the Vandermonde matrix is invertible.

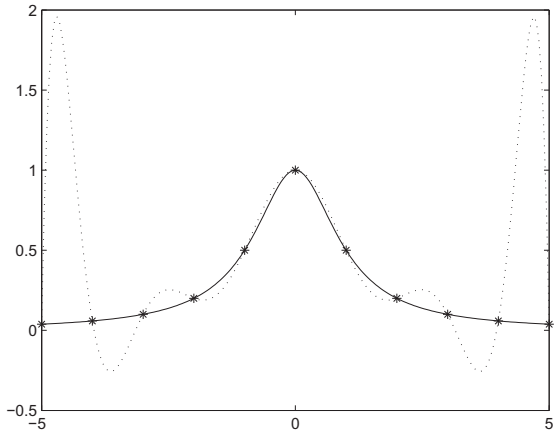

Figure 11.1: An example of the wild oscillations one can get in interpolating data with a high degree polynomial. In this case, we are interpolating 11 data points (asterisks) with a degree 10 polynomial (dotted curve) from the function represented by a solid curve. This is also an example of excellent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

This theorem tells us that we can solve the interpolation problem by simply inverting the Vandermonde matrix:

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

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.

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, so we find the least squares solution to the equation using our pseudo-inverse function (either one we constructed from the SVD or Matlab's built-in function).

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 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:

$$p(x, y) = a_0 + a_1x + a_2y + a_3xy + a_4x^2 + a_5y^2$$

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

$$p(x, y, z) = a_0 + a_1x + a_2y + a_3z + a_4xy + a_5xz + a_6yz + a_7x^2 + a_8y^2 + a_9z^2$$

- 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 "Multidimensional 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:

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

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:

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

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

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

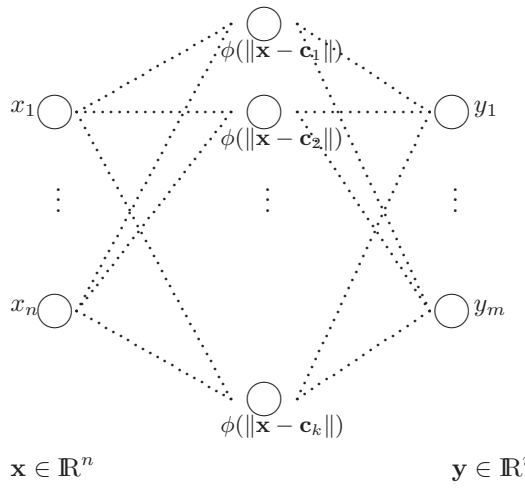

Figure 11.2: The RBF as a neural network. The input layer has as many nodes as input dimensions. The middle layer has k nodes, one for each center $\mathbf{c}_k$. The processing at the middle layer is to first compute the distance from the input vector to the corresponding center, then apply ϕ. The resulting 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.

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 multi-dimensional; 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).

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),:);
```

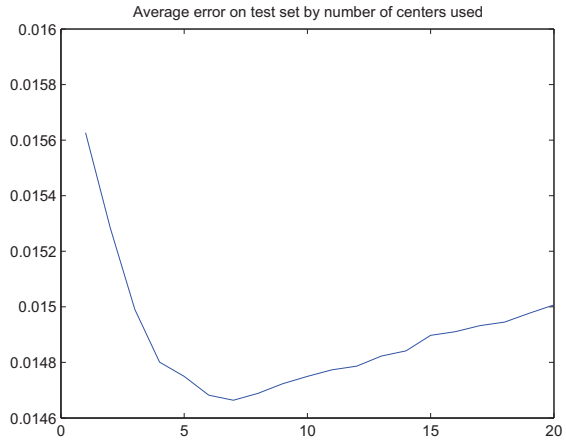

Figure 11.3: The error, averaged over 30 random placements of the centers, of the error on the testing set. The horizontal axis shows the number of centers. As we predicted, the error on the test set initially goes down, but begins to increase as we begin to model the noise in the training set. We would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

```
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:

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

(d) Show that the last exercise implies that, if we are given k and a, then we should take σ to be:

$$\sigma = \frac{a}{\sqrt{-\ln(1-k^2)}} \tag{11.6}$$

The previous exercises give some justification to the following approximation for σ (See, for example, `designrb`, which is in the file `newrb.m`):

$$\sigma = \frac{a}{\sqrt{-\ln(0.5)}}$$

11.5 Orthogonal Least Squares

The following is a summary of the work in the reference [7]. We present the multidimensional extension that is the basis of the method used in Matlab's subfunction: `designrb`, which can be found in the file `newrb`. (Hint: To find the file, in Matlab type `which newrb`). We go through this algorithm in detail so that we can modify it to suit our needs.

Recall that our matrix equation is:

$$W\Phi = Y$$

where, using k centers, our p input data points are in $\mathbb{R}^n$, output in $\mathbb{R}^m$, then W is $m \times (k+1)$, Φ is $(k+1) \times p$, and Y is $m \times p$.

Begin with $k = p$. The basic algorithm does the following:

1. Look for the row of Φ (in $\mathbb{R}^p$) that most closely points in the same direction as a row of Y.

2. Take that row out of Φ (which makes Φ smaller), then deflate the remaining rows. Another way to say this is that we remove the component of the columns of Φ that point in the direction of our "winner" (much like the Gram-Schmidt process).

3. Repeat.

Before continuing, let's review the linear algebra by doing a few exercises that we'll need to perform the operations listed above.

- Show that, if we have a set of vectors $X = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k]$ and a vector $\boldsymbol{y}$, then the vector in X that most closely points in the direction of $\boldsymbol{y}$ (or $-\boldsymbol{y}$) is found by computing the maximum of:

$$\left(\frac{\boldsymbol{x}_i^T \boldsymbol{y}}{\|\boldsymbol{x}_i\| \|\boldsymbol{y}\|} \right)^2 \tag{11.7}$$

- Recall that:

$$\frac{\boldsymbol{v}\boldsymbol{v}^T}{\|\boldsymbol{v}\|^2} \tag{11.8}$$

is an orthogonal projector to the one dimensional subspace spanned by $\boldsymbol{v}$. Using this, show that the following set of Matlab commands will return a matrix whose columns are in the orthogonal complement of the subspace spanned by $\boldsymbol{v}$. That is, $\boldsymbol{v}$ is orthogonal to the columns of the final matrix in the computation:

```
a=v'*P/(v'*v);
P=P-v*a;
```

- Show that, if $X = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k]$ are column vectors, then

$$\texttt{sum(X.*X)} = \left(\|\boldsymbol{x}_1\|^2, \ldots, \|\boldsymbol{x}_k\|^2\right)$$

Write a Matlab function, `mnormal` that will input a matrix, and output the matrix with normalized columns (comes in handy!).

- Let $X = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m]$ and $Y = [\boldsymbol{y}_1, \ldots, \boldsymbol{y}_k]$. Show that the (i, j) component of `X'*Y` is given by:

$$\boldsymbol{x}_i^T \boldsymbol{y}_j$$

- Verify that the following Matlab commands solves the (least squares) equation: $wp + b = t$ for w and b:

```
[pr,pc]=size(p);
x=t/[p;ones(1,pc)];
w=x(:,1:pr);
b=x(:,pr+1);
```

- What does the following set of Matlab commands do? (Assume M is a matrix)

```
replace=find(isnan(M));
M(replace)=zeros(size(replace));
```

We are now ready to examine the Matlab function `newrb` and its main subfunction, `designrb`. These programs use the OLS algorithm [7], which we describe below.

You might compare the notation below to the actual Matlab code.

Algorithm. Orthogonal Least Squares for RBFs

Input the domain and range pairs (p, t) the error goal and gaussian spread (`eg`, `sp`).

Output the centers `w1`, the reciprocal of the widths, 1σ, are stored in `b1`, the weights ω_{ij} are in `w2` and the biases b_j are in the vector `b2`. The other output arguments are optional.

```
function [w1,b1,w2,b2,k,tr] = designrb(p,t,eg,sp)

[r,q] = size(p);
[s2,q] = size(t);
b = sqrt(-log(.5))/sp;

% RADIAL BASIS LAYER OUTPUTS
P = radbas(dist(p',p)*b);
PP = sum(P.*P)';
d = t';
dd = sum(d.*d)';
```

1. **Exercise:** For each matrix above, write down the size (in terms of "number of points" and "dimension").

   So far, we've constructed the interpolation matrix `P`=$\mathbf{\Phi}^T$, and initialized some of our variables. We're now ready to initialize the algorithm:

   ```
   % CALCULATE "ERRORS" ASSOCIATED WITH VECTORS
   e = ((P' * d)' .^ 2) ./ (dd * PP');
   ```

2. **Exercise:** What is the size of `e`? Show that the (j, k) component of `e` is given by (in terms of our previous notation):

   $$\frac{\boldsymbol{\phi}_k^T \boldsymbol{y}_j}{\boldsymbol{y}_j^T \boldsymbol{y}_j \boldsymbol{\phi}_k^T \boldsymbol{\phi}_k}$$

   In view of Equation 11.7, what does `e` represent?

   In the following lines, there is some intriguing code. The integer `pick` is used to store our column choice. The vector `used` will be used to store the columns of $\mathbf{\Phi}$ that have been chosen, and the vector `left` are the columns remaining.

   ```
   % PICK VECTOR WITH MOST "ERROR"
   pick = findLargeColumn(e);
   used = [];
   left = 1:q;
   W = P(:,pick);
   P(:,pick) = []; PP(pick,:) = [];
   e(:,pick) = [];
   ```

   The matrix `W` is initialized as the best column of $\mathbf{\Phi}$. The matrices `P` and `e` has their corresponding column removed, `PP` has one element removed. Now update `used` and `left`.

   ```
   used = [used left(pick)];
   ```

```
left(pick) = [];

% CALCULATE ACTUAL ERROR
w1 = p(:,used)';
a1 = radbas(dist(w1,p)*b);
[w2,b2] = solvelin2(a1,t);
a2 = w2*a1 + b2*ones(1,q);
sse = sumsqr(t-a2);
```

3. **Exercise:** Explain each of the previous lines. What is the difference between `W` and `w1`?

   We now begin the MAIN LOOP:

```
for k = 1:(q-1)

  % CHECK ERROR
  if (sse < eg), break, end

  % CALCULATE "ERRORS" ASSOCIATED WITH VECTORS
  wj = W(:,k);
  a = wj' * P / (wj'*wj);
  P = P - wj * a;
  PP = sum(P.*P)';
  e = ((P' * d)' .^ 2) ./ (dd * PP');
```

4. **Exercise:** Explain the previous lines of code. To assist you, you may want to refer back to Equation 11.8.

```
  % PICK VECTOR WITH MOST "ERROR"
  pick = findLargeColumn(e);
  W = [W, P(:,pick)];
  P(:,pick) = []; PP(pick,:) = [];
  e(:,pick) = [];
  used = [used left(pick)];
  left(pick) = [];

  % CALCULATE ACTUAL ERROR
  w1 = p(:,used)';
  a1 = radbas(dist(w1,p)*b);
  [w2,b2] = solvelin2(a1,t);
  a2 = w2*a1 + b2*ones(1,q);
  sse = sumsqr(t-a2);
end

[S1,R] = size(w1);
b1 = ones(S1,1)*b;
```

5. **Exercise:** If you wanted to be able to output the training errors, how would you modify `designrb`?

6. **Exercise:** The following two subfunctions are also included in `newrb.m`. Go through and explain each line.

```
%=========================================================
function i = findLargeColumn(m)

replace = find(isnan(m));
m(replace) = zeros(size(replace));

m = sum(m .^ 2,1);
i = find(m == max(m));
i = i(1);

%=========================================================

function [w,b] = solvelin2(p,t)

if nargout <= 1
  w= t/p;
else
  [pr,pc] = size(p);
  x = t/[p; ones(1,pc)];
  w = x(:,1:pr);
  b = x(:,pr+1);
end
```

   To see the actual Matlab code, in the command window, type `type newrb` and you will see a printout of the code.

11.6 Homework: Iris Classification

In the following script file, we see the effect of changing the width of the Gaussian on the performance of the classification procedure. We will use Matlab's built-in functions to do the training and testing.

Type in this script file, then use it as a template to investigate how PCA reduction in the domain effects the classification. In particular:

1. Type in the script file.

2. Make appropriate changes so that once the domain set X is loaded, perform a PCA/KL reduction to the best three dimensional space. This will be the new domain for the RBF.

3. Run both the original script and the new script, and compare your answers.

```
%Script file:

numiters=20;  %Number of widths of Gaussians
numtrain=120; %Number of training points to use

X=load('IRIS.NET');
Y=X(:,5:7);          %The targets are the last 3 columns
X(:,5:7)=[];         %The patterns are the first 4 cols

eg=0.1;              %Error goal
sp=linspace(0.05,1.6,numiters);  %Gaussian Widths

[NumPoints,Dim]=size(X);
temp=randperm(NumPoints);
P=X(temp(1:numtrain),:)';
T=Y(temp(1:numtrain),:)';

warning off  %We expect some warnings about invertibility

for i=1:numiters
%print the iteration so you can keep track:
  i
  net=newrb(P,T,eg,sp(i));
  z=sim(net,X');
  e(i)=0;

  [c(i),b]=size(net.IW{1});  %c is the number of weights used

  for k=1:150
    [x,j]=max(z(:,k));
    if Y(k,j)~=1
      e(i)=e(i)+1;           %e is the error in classification
    end
  end
  W(i)=max(max(abs(net.LW{2,1})));  %Max weight size
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Plotting routines
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

subplot(2,2,1)
```

```
plot3(X(1:50,1),X(1:50,2),X(1:50,3),'r*')
hold on
plot3(X(51:100,1),X(51:100,2),X(51:100,3),'b^')
plot3(X(101:150,1),X(101:150,2),X(101:150,3),'go')
hold off
subplot(2,2,2)
plot(sp,e)
title('Error versus Gaussian Width')
subplot(2,2,3)
plot(sp,c)
title('Number of centers from OLS versus Gaussian Width');
subplot(2,2,4)
semilogy(sp,W)
title('Maximum Weight versus Gaussian Width');
```