Chapter 12

Radial Basis Functions

12.1 Introduction

The neural network has been so popular because of it is actually a **universal function approximator**. That is, for any given function (expressed partially as data), there is a neural network that will approximate it. This extraordinary property means that the applications of neural networks are only bounded by our ability to compute and train them.

We're going to start our discussion of the more general neural nets by looking at one class in particular- The Radial Basis Functions (or RBFs). While they had been around for a long time, it was an important work by Dave Broomhead and David Lowe, "Multivariable Functional Interpolation and Adaptive Networks" published in 1988 that connected the RBF to the neural net.

Further, RBFs were initially used (Powell, late 1970s) to perform *interpolation* rather than regression- That means that they were used to find an *exact fit* to a high dimensional function rather than looking for a generalization (in the regression problem). We know from that, that it is possible to drive the training error to zero (which would typically be overfitting), and so we need to keep that in mind when we're training.

We will see that the RBF network has some very attractive features: Training can be split up and computed in layers, perhaps making it more tractable, and they are fast and intuitive as well.

12.2 Radial Basis Functions

The architecture of a radial basis function consists of the following pieces, which we'll go through more carefully in a moment:

- A set of **centers**, $\{\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots, \boldsymbol{c}_k\}$. The centers are in the same space as the input data, $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_p\}$, and can in fact be selected from the input data.
- One or more **transfer functions**, ϕ (a map from $\mathbb{R}$ to $\mathbb{R}$). Common choices will be given shortly.

The action of the RBF network is then defined as follows (given in layers): The first mapping is from the input space, $\mathbb{R}^n$ to $\mathbb{R}^k$ (where k is the number of centers). At this point, the transfer function ϕ is applied to each element, and that is followed by an affine mapping to the output layer, $\mathbb{R}^m$.

$$\mathbf{x} \to \begin{bmatrix} \|\mathbf{x}-\mathbf{c}_1\| \\ \|\mathbf{x}-\mathbf{c}_2\| \\ \vdots \\ \|\mathbf{x}-\mathbf{c}_k\| \end{bmatrix} \to \phi\left(\begin{bmatrix} \|\mathbf{x}-\mathbf{c}_1\| \\ \|\mathbf{x}-\mathbf{c}_2\| \\ \vdots \\ \|\mathbf{x}-\mathbf{c}_k\| \end{bmatrix}\right) \to W \begin{bmatrix} \phi(\|\mathbf{x}-\mathbf{c}_1\|) \\ \phi(\|\mathbf{x}-\mathbf{c}_2\|) \\ \vdots \\ \phi(\|\mathbf{x}-\mathbf{c}_k\|) \end{bmatrix} + \boldsymbol{b} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} \tag{12.1}$$

Before going further, we might just make a note of where the name "radial basis function" comes from.

Definition: A radial function is any function of the form $\phi(\mathbf{x}) = \phi(\|\mathbf{x}\|)$, so that ϕ acts on a vector in $\mathbb{R}^n$, but only through the norm so that $\phi : [0, \infty) \to \mathbb{R}$.

It is possible to then take some set of radial functions and then have a basis for some function space. We're going to use the radial basis functions for function approximation however.

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

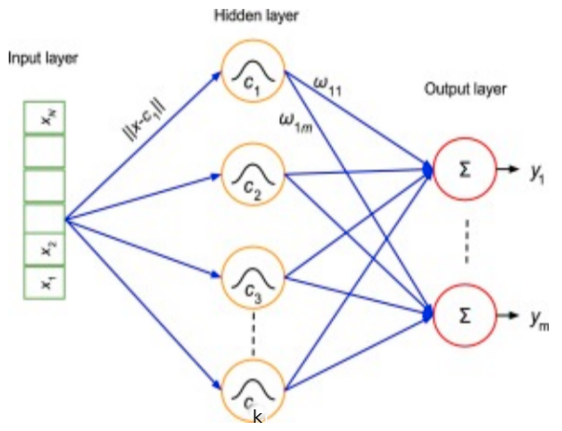
$$\begin{array}{rcll}
\phi(r,\sigma) & = & \exp\left(\dfrac{-r^2}{\sigma^2}\right) & \text{Gaussian} \\
\phi(r) & = & r^3 & \text{Cubic} \\
\phi(r) & = & r^2\log(r) & \text{Thin Plate Spline} \\
\phi(r) & = & \dfrac{1}{r+1} & \text{Cauchy} \\
\phi(r,\beta) & = & \sqrt{r^2+\beta} & \text{Multiquadric}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

The reader may note that some transfer functions have extra parameters- Like the Gaussian, and Multiquadrics. Also, we will assume that ϕ, when applied to a vector or a matrix, will be defined element-wise.

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).

12.2.1 The RBF as a Neural Network

The RBF as a neural network is shown to the right. 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.



12.2.2 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 12.1 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:

$$W\Phi = Y \tag{12.2}$$

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).

12.2.3 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 12.2.2:

```
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
```

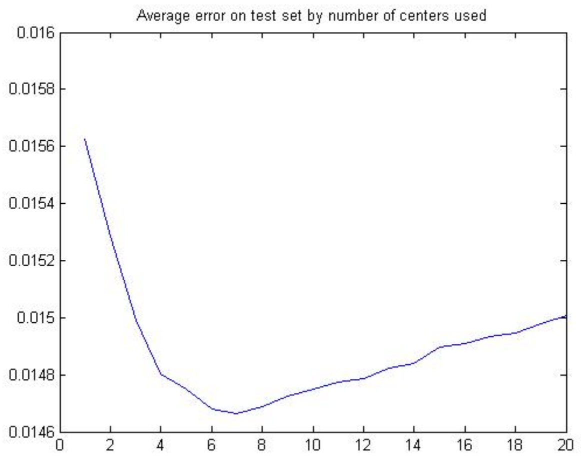


Figure 12.1: The error, averaged over 30 random placements of the centers, of the error on the testing set. The horizontal axis shows the number of centers. As we predicted, the error on the test set initially goes down, but begins to increase as we begin to model the noise in the training set. We would want to stop adding centers at the bottom of the valley shown.

```
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 12.2
   - `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:

$$\sigma = \frac{a}{\sqrt{-\ln(1-k^2)}} \tag{12.3}$$

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

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

12.3 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:

$$\left(\frac{\boldsymbol{x}_i^T \boldsymbol{y}}{\|\boldsymbol{x}_i\|\|\boldsymbol{y}\|}\right)^2 \tag{12.4}$$