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.

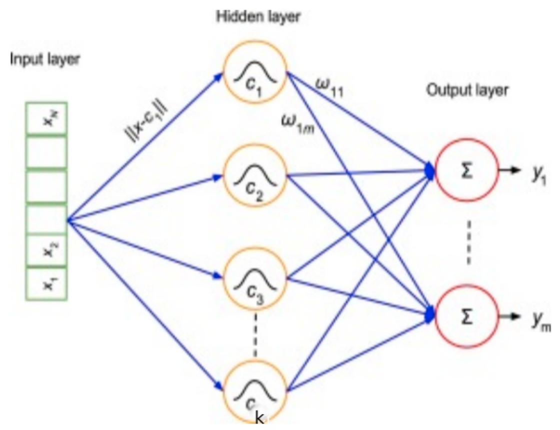
$$\begin{array}{lcll}
\phi(r,\sigma) & = & \exp\left(\dfrac{-r^2}{\sigma^2}\right) & \text{Gaussian} \\
\phi(r) & = & r^3 & \text{Cubic} \\
\phi(r) & = & r^2\log(r) & \text{Thin Plate Spline} \\
\phi(r) & = & \dfrac{1}{r+1} & \text{Cauchy} \\
\phi(r,\beta) & = & \sqrt{r^2+\beta} & \text{Multiquadric}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

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.



Training the RBF can be split into two steps. The first step is to locate the centers and to determine the transfer function (and its parameters). The second step is to build the design matrix Φ and solve the linear system of equations for the weights and biases, W and $\boldsymbol{b}$.

It is possible for us to include center locations and the transfer function parameters in the error function. There are lots of journal articles about how this might be done, but none have turned out to be that popular. This is probably due to the fact that separating centers and transfer function parameters out makes the problem fast and linear- this is the primary reason one would use RBFs instead of other methods.

12.2.2 Determining the Centers and transfer function

Center Selection

We will primarily use one of the following techniques to select the centers “offline” (meaning this can be and is usually done independently from finding the weights and biases).

- $k-$means clustering (or any clustering that returns cluster centers).

  Data clustering is popular, but what we’re really doing is spreading the clusters around the input data. The desired targets are typically not taken into account.

- Random center selection: This seems like it would be a terrible thing to do since it guarantees a suboptimal solution, but for smaller data sets, this does actually work pretty well.

- Orthogonal Least Squares: This is Matlab’s default algorithm, and we’ll discuss this later in this section.

- Center placement as part of the optimization problem. This is possible, but as we noted before, for larger problems this might become an unreasonable thing to do (in practice). We won’t do this, although it is interesting to see where the centers end up.

Notes on the transfer function

Some transfer functions do not require us to provide values for extra parameters. The cubic function can be an attractive option that way. However, the Gaussian is probably the most popular choice for transfer function, and so it is good to have some way to set the spread. It is possible to always set the spread to 1 by re-scaling the data appropriately, but there may be reasons not to do that.

When looking at $k-$means, some recommend using the distance between centers as a guide for setting the spread. If we define the spread as `spr`, where

$$\phi(r) = \mathrm{e}^{-\mathtt{spr}\, r^2}$$

then, if M is the maximum distance between our k centers,

$$\mathtt{spr} = \frac{M}{\sqrt{2k}}.$$

Matlab’s formula for calculating `spr` is, given the desired spread s (default is $s = 1$) then

$$\mathtt{spr} = \frac{\sqrt{-\ln(1/2)}}{s} = \frac{\sqrt{\ln(2)}}{s}$$

At this point, we’ll assume that the **location**, **number** of centers, and the **transfer function** with its parameters has been chosen. We continue on to train the net.

12.2.3 Training the RBF

As usual, we should start by examining the data we’re given, scale it if necessary, and split it into training and testing sets. We don’t necessarily require k-fold cross validation for the remainder, although we could use it to try to get a better estimate of the error (of course, that requires extra data that we may not have). Given all that, let’s assume that we have p training points, each point is in $\mathbb{R}^n$. Further, we have p targets, each in $\mathbb{R}^m$. We’re using k centers and the transfer function ϕ has been defined and its parameters set.

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 $\boldsymbol{t}$ pairing to build the system of equations which we will solve using the least squares error.

1. Build the design matrix Φ, where
$$\Phi_{i,j} = \phi(\|\boldsymbol{x}_j - \boldsymbol{c}_i\|)$$
As defined here, Φ will be $k \times p$ (number of centers by number of points).

2. Let T be the $m \times p$ matrix representing p target vectors in $\mathbb{R}^m$.

3. Now the weight matrix takes us from the middle layer (in $\mathbb{R}^k$) to the outer layer (in $\mathbb{R}^m$), so it should be $m \times k$. Further, the bias vector $\mathbf{b}$ should be $m \times 1$. The equation we need to solve for $W, \boldsymbol{b}$ is given by:
$$W\Phi + \boldsymbol{b} = T \tag{12.2}$$

4. It's best to convert the equation in the previous step to a linear equation (rather than affine). We modify W and Φ to take care of that:
$$\hat{W} = [W \quad \boldsymbol{b}], \qquad \hat{\Phi} = \begin{bmatrix} \Phi \\ 1, 1, 1, \cdots 1 \end{bmatrix}$$
Now, $\hat{W}$ is $m \times k + 1$ and $\hat{\Phi}$ is $k + 1 \times p$, and our previous equation becomes:
$$\hat{W}\hat{\Phi} = T$$

5. Finally, take the pseudoinverse of Φ (manually using the SVD or by using a built-in command) and solve, giving
$$\hat{W} = T\hat{\Phi}^\dagger.$$
We also recall that $\hat{W} = [W\, \boldsymbol{b}]$, so if we want these separated, we can do this now.

12.2.4 Predicting data using the RBF

Once we have constructed the RBF, we can predict how new data will behave. To do this, remember that we need the following:

- The set of centers, usually as a matrix.
- Which transfer function ϕ is being used, and its parameters, if necessary.
- The weights and bias values, W and $\boldsymbol{b}$.

Once we have these, given N new input vectors, we would perform the following steps to output the result:

1. Compute the distance matrix between the N data points and k centers. Let's assume this is a $k \times N$ matrix.

2. Apply the transfer function ϕ, and this results in the $k \times N$ matrix Φ.

3. Apply our affine map, where W is $m \times k$:
$$W\Phi + \boldsymbol{b} = Y$$
(This is not good linear algebra notation, but I hope you understand that this notation means to add the column vector $\boldsymbol{b}$ to each column of the matrix $W\Phi$).

4. Y is $m \times N$ and the model output (or prediction).

Example

As an example of the interplay between the number of centers and the error, let's try an experiment.

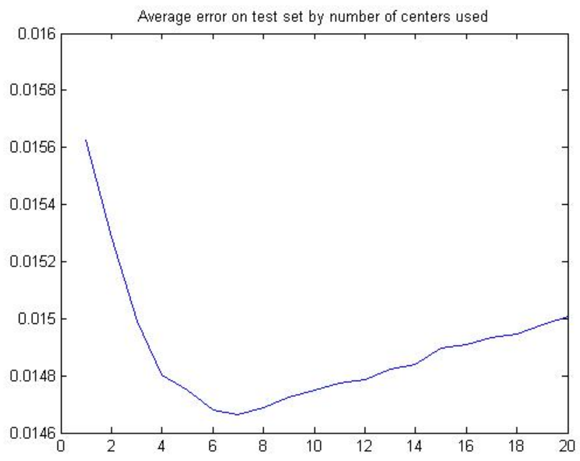
The data will be two dimensional, and the "true" underlying function will be

$$z = \exp\left(-\frac{x^2+y^2}{4}\right)$$

with some noise added (with standard deviation of about 0.5).

We'll have 1500 points in the domain, and we'll randomly select the centers (for each training session) from the data, and then train the network on 300 points. This could give us a lot of variation in the error, so we'll train 30 times (each with a different selection of centers), and take the average for the given number of cluster centers.

We will then consider the error on the **test** set of 1200 points, and plot the error versus the number of centers.



As we expect, the error decreases initially as we increase the number of centers. However, once we go beyond a certain point, the error starts to **increase** again. This is the point at which we should stop adding centers, because it means we're starting to **overfit**.

Exercises

1. 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 for Matlab's choice for approximating σ:

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

(I'm not sure why they stick with $-\ln(1/2)$ rather than just $\ln(2)$!)

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, but does not seem to be widely available in Python yet (Apr 2021).

First, we'll get the big picture, and then we'll look at some of the linear algebra that makes it work. It's most convenient to start with the linear version of the RBF equation:

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

Dimensions are important here, so let's define them again: We'll assume that we have p data points in $\mathbb{R}^n$ as input, and the output dimension is $\mathbb{R}^m$.

Initially, we'll assume that every data point is a center, so we have p centers. That means that W is $m \times (p+1)$, Φ is $(p+1) \times p$, and T is $m \times p$.

The first thing to notice is that j^{th} **row** of Φ (except for the last one) corresponds to the j^{th} center $\boldsymbol{c}_j$ (that is, the row of Φ is constructed by taking distances from $\boldsymbol{c}_j$ and the p points in X).

The second thing to notice is that on the left side of the equation, we're working with the **row space** of $\hat{\Phi}$ and on the left side, we have the **rows** of T. Orthogonal Least Squares (or OLS) assumes that we'll get a "good solution" to the RBF equation if it is possible to select a small number of rows of Φ (corresponding to a small number of centers) from which we can construct the rows of T.

Another way to think of OLS is that we want to try to use a small number of rows of $\hat{\Phi}$ (not really counting the last row) in order to form a basis for the row space of T.

We cannot use the SVD in this case, because we want to use **actual** rows of $\hat{\Phi}$ and not combinations of rows- that's an important restriction to keep in mind.

The OLS Algorithm

Begin with all data in $\hat{\Phi}$, so this is now $(p+1) \times p$.

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

2. Put point $\mathbf{x}_w$ as a center.

3. Remove row w from $\hat{\Phi}$

4. Remove the component of the w^{th} row from the other rows (this makes all the remaining rows orthogonal to row w).

5. Compute the error on the data using the current set of centers as the model.

6. Repeat until the error is small enough, or we have used the maximum number of centers allowed.

This algorithm always stops, either because some maximum number of centers has been reached, or because we have used all the data (and the RBF equation will be interpolating the test data).

Exercises for the OLS

There is some lovely linear algebra here. Let's review it!

1. 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 $-\boldsymbol{y}$) can be found by computing the maximum of:

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

   (Hint: What does this expression represent? Have we seen it before?)

2. To "remove the component of **y** in the direction of **x**" means to do what, exactly? It means that we want to subtract the projection of **y** onto **x** from **y**- which means that the "new version" of **y** will be orthogonal to **x**. Show that this formula will do the trick (that is, show $\mathbf{y}_{\text{new}} \perp \mathbf{x}$).

$$\mathbf{y}_{\text{new}} = \mathbf{y} - \text{Proj}_x(\boldsymbol{y}) = \mathbf{y} - \frac{\mathbf{y}^T\mathbf{x}}{\mathbf{x}^T\mathbf{x}}\mathbf{x}$$

   Side Remark: We did this back in linear algebra when we looked at Gram-Schmidt orthogonalization.

3. Why might OLS not be used much now? (Hint: What if you have a very large number of data points?)

Homework:

Homework will be assigned separately, since it will involve an application of the RBF to several different data sets, and you may choose to use Python or Matlab/Octave.

12.4 Summary of the Radial Basis Functions

1. What is a Radial Basis Function?

   A radial basis function (RBF) is a general model to build a mapping from $\mathbb{R}^n$ to $\mathbb{R}^m$.

2. Choices that need to be made when using an RBF:

   (a) The transfer function ϕ that will be used.
   Some choices: $\phi(r) = r$, $\phi(r) = \mathrm{e}^{-r^2/\sigma^2}$, $\phi(r) = r^3$, etc.

   (b) The model parameters for an RBF are given by:

   - The number of centers, k.
   - The center locations, $\mathbf{c}_i$.
     Common choices:
     - The centers are randomly chosen from the data.
     - The centers are chosen as the centroids of data clusters.
     - The centers are chosen automatically using Orthogonal Least Squares (this is Matlab's default).
     - The centers are chosen through optimization.
     - In all cases, we could also define individual values of each parameter (like the widths of the Gaussians).

3. Once ϕ and the centers have been chosen, we solve for weights and biases using a least squares solution. That is:

- Compute the distance matrix between the k centers and p data points.
- Form the transfer matrix Φ by applying ϕ to the distance matrix, so Φ is $k \times p$.
- Add a row of ones to the bottom of matrix Φ, so that $\hat{\Phi}$ is $(k+1) \times p$.
- Solve $\hat{W}\hat{\Phi} = T$ by using the pseudoinverse, so $\hat{W} = T\hat{\Phi}^{\dagger}$.

4. To test the function on new domain points (we have now fixed the centers and the weight matrix W):
   - Form the distance matrix between the k centers and the $\hat{p}$ new data points, this will be $k \times \hat{p}$.
   - Apply ϕ to the distance matrix to get Φ (dimenesions $k \times \hat{p}$).
   - Add a row to the bottom of matrix Φ to get $\hat{\Phi}$.
   - The new output is $\hat{W}\hat{\Phi}$.

5. Why use an RBF?

   (a) Once the centers have been chosen, this is a linear modeling problem. That is, to find the weights, we are solving a linear equation. Therefore, an RBF is much faster than methods that involve nonlinear optimization.

   (b) There are nice connections to statistics if we use the Gaussian transfer function, although we have not discussed them.

6. Software and the RBF.

   There are two ways of producing an RBF model in Matlab- one is to do it explicitly yourself, the other is to use Matlab's built-in routines using the "Neural Network Toolbox". If you're using the toolbox, then the OLS is built into the `newrb` command.

   Unfortunately, Octave doesn't maintain a version of the neural network toolbox anymore, so the subroutines needed need to be all provided (I have written those separately for you to use).

   Alternatively, there isn't much pre-written for Python for some reason (April 2021), so I will provide you with the functionality needed, along with some examples.