Chapter 17: Radial Basis Networks

Brandon Morgan

2/9/2021

E17.10

We are given the following function to approximate:

$$g(p) = 1 + \sin(\frac{\pi}{8}p) \quad -2 \leq p \leq 2$$

We are to approximate this function with an RBF network.

1)

Here we will select ten random training points from the interval given above:

```
actual_function = function(x) 1 + sin(pi*x/8)
set.seed(10)
p = runif(10, -2, 2)
t = actual_function(p)
cbind(p, t)
```

```
##                p         t
##  [1,]  0.02991281 1.0117465
##  [2,] -0.77292598 0.7011119
##  [3,] -0.29236933 0.8854389
##  [4,]  0.77240832 1.2986941
##  [5,] -1.65945612 0.3934874
##  [6,] -1.09825353 0.5819632
##  [7,] -0.90187791 0.6531912
##  [8,] -0.91077974 0.6499145
##  [9,]  0.46331723 1.1809421
## [10,] -0.28131390 0.8897529
```

2)

We create four basis function centers even spaced on the interval given above:

```
W_1=matrix(seq(-2,2, length=4), ncol=1)
W_1
```

```
##            [,1]
## [1,] -2.0000000
## [2,] -0.6666667
## [3,]  0.6666667
## [4,]  2.0000000
```

For the bias of the first layer, we will use Eq. (17.9):

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

Because our function centers are evenly spaced, the maximum distance is $d_{max} = 4/3 = 1.\bar{3}$. From the size of our first weight matrix, 4x1, $S = 4$. Thus, our initial bias for the first layer willbe $\sqrt{4}/(1.\bar{3}) = 1.5$.

```
b_1 = matrix(1.5, ncol=1, nrow=4)
b_1
```

```
##      [,1]
## [1,]  1.5
## [2,]  1.5
## [3,]  1.5
## [4,]  1.5
```

The weights and biases for the second layer were shown from the textbook in Eq. (17.31) to be: $X^* = [U^T U + \rho I]^{-1} U^T I$; however, we are assuming there is no regularation, $\rho = 0$. Therefore, our weights and bias become $X^* = [U^T U]^{-1} U^T I$, where $U_i = z_i = \begin{bmatrix} a_q^1 & 1 \end{bmatrix}$ and where $a_q^1 = radbas(n_q^1)$ where $n_q^1 = ||p_q - w^1||b_i$.

Here's n^1:

```
n = matrix(0, ncol=10, nrow=4)
for(q in 1:10) {
  for(i in 1:4) {
    # you're supposed to take the norm of the values p-w, but since its a single
    # entry, its the same as absolute
    n[i,q] = (abs(p[q]-W_1[i][1])*b_1[i][1])
  }
}
n
```

```
##           [,1]     [,2]     [,3]      [,4]      [,5]      [,6]      [,7]
## [1,] 3.0448692 1.840611 2.561446 4.1586125 0.5108158 1.3526197 1.6471831
## [2,] 1.0448692 0.159389 0.561446 2.1586125 1.4891842 0.6473803 0.3528169
## [3,] 0.9551308 2.159389 1.438554 0.1586125 3.4891842 2.6473803 2.3528169
## [4,] 2.9551308 4.159389 3.438554 1.8413875 5.4891842 4.6473803 4.3528169
##           [,8]      [,9]     [,10]
## [1,] 1.6338304 3.6949758 2.5780292
## [2,] 0.3661696 1.6949758 0.5780292
## [3,] 2.3661696 0.3050242 1.4219708
## [4,] 4.3661696 2.3050242 3.4219708
```

The $radbas()$ function is defind to be e^{-n^2}. Thus, our a^1 vector is:

```
radbas = function(x) {
  result = matrix(0, ncol=ncol(x), nrow=nrow(x))
  for(i in 1:nrow(x)) {
    for(j in 1:ncol(x)) {
      result[i,j] = exp(-(x[i,j])^2)
    }
  }
  result
}
a_1 = radbas(n)
a_1
```

```
##              [,1]         [,2]         [,3]         [,4]         [,5]
## [1,] 9.409284e-05 3.378126e-02 1.414463e-03 3.085220e-08 7.703327e-01
## [2,] 3.356281e-01 9.749151e-01 7.296272e-01 9.470175e-03 1.088625e-01
## [3,] 4.016096e-01 9.438477e-03 1.262568e-01 9.751559e-01 5.160865e-06
## [4,] 1.612108e-04 3.065357e-08 7.329131e-06 3.368482e-02 8.207499e-14
##              [,6]         [,7]         [,8]         [,9]        [,10]
## [1,] 1.604809e-01 6.632341e-02 6.929367e-02 1.176641e-06 1.298904e-03
## [2,] 6.576376e-01 8.829562e-01 8.745199e-01 5.653230e-02 7.159695e-01
## [3,] 9.040531e-04 3.943261e-03 3.702457e-03 9.111573e-01 1.323903e-01
## [4,] 4.169130e-10 5.907668e-09 5.258411e-09 4.926451e-03 8.212243e-06
```

To create our U matrix, we simply add a column of ones to end of the matrix, after transposing, a coefficient for the biases.

```
ones = matrix(1, nrow=10, ncol=1)
U_1 = cbind(t(a_1), ones)
U_1
```

```
##               [,1]        [,2]         [,3]         [,4] [,5]
##  [1,] 9.409284e-05 0.335628064 4.016096e-01 1.612108e-04    1
##  [2,] 3.378126e-02 0.974915146 9.438477e-03 3.065357e-08    1
##  [3,] 1.414463e-03 0.729627164 1.262568e-01 7.329131e-06    1
##  [4,] 3.085220e-08 0.009470175 9.751559e-01 3.368482e-02    1
##  [5,] 7.703327e-01 0.108862513 5.160865e-06 8.207499e-14    1
##  [6,] 1.604809e-01 0.657637603 9.040531e-04 4.169130e-10    1
##  [7,] 6.632341e-02 0.882956152 3.943261e-03 5.907668e-09    1
##  [8,] 6.929367e-02 0.874519945 3.702457e-03 5.258411e-09    1
##  [9,] 1.176641e-06 0.056532300 9.111573e-01 4.926451e-03    1
## [10,] 1.298904e-03 0.715969506 1.323903e-01 8.212243e-06    1
```

Now we can finally estimate the weights of the second layer via multiple linear repression with zero regularization:

```
x = solve(t(U_1)%*%U_1)%*%t(U_1)%*%t
W_2 = x[1:4,]
b_2 = x[5,1]
W_2
```

```
## [1] -0.8721046 -0.3866168  0.1468913  2.2116062
```

```
b_2
```

```
## [1] 1.08126
```

Now, we have the final neural network:

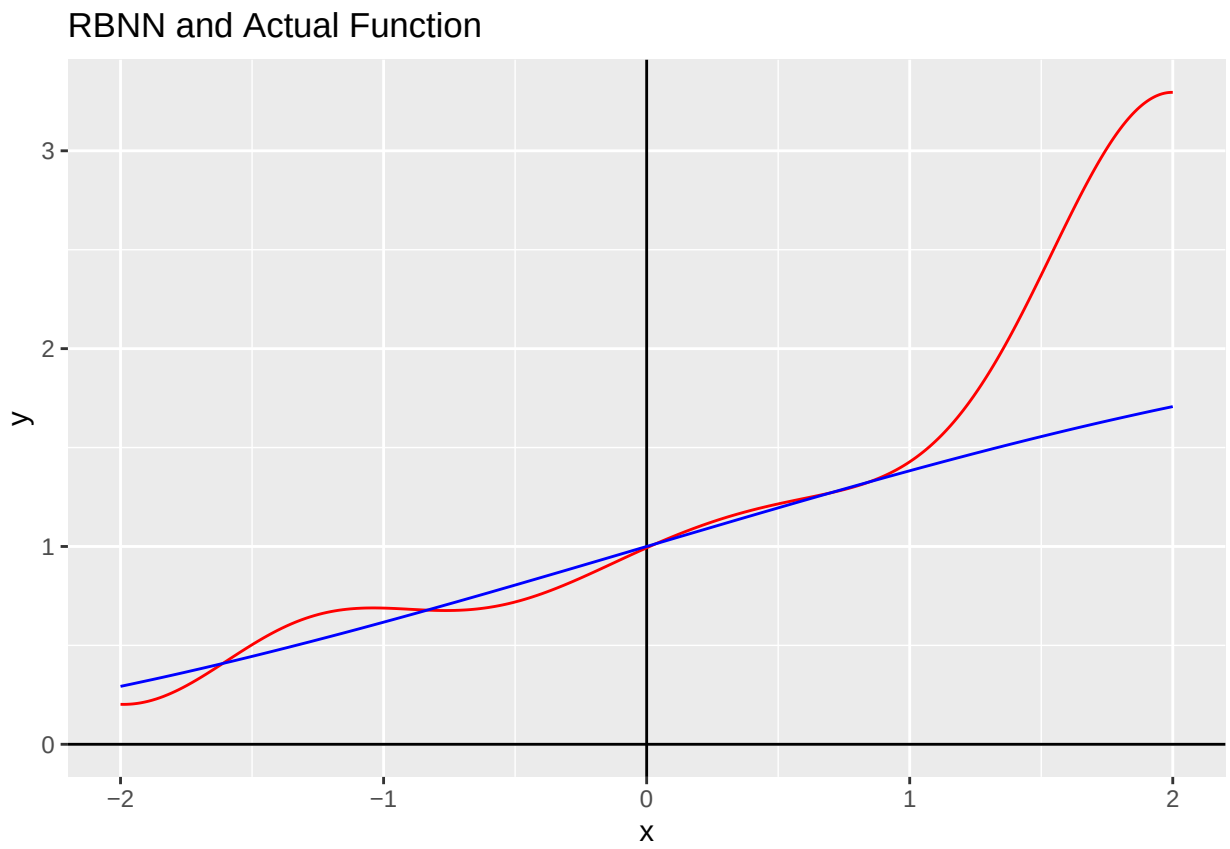
```
# Radial Basis Neural Network
RBNN = function(p) {
  n = matrix(0, ncol=1, nrow=4)
  b_1 = matrix(1.5, ncol=1, nrow=4)
  W_1=matrix(seq(-2,2, length=4), ncol=1)
  for(i in 1:4) {
      # you're supposed to take the norm of the values p-w, but since its a single
      # entry, its the same as absolute
      n[i,1] = (abs(p-W_1[i][1])*b_1[i][1])
  }
  radbas = function(x) {
    result = matrix(0, ncol=ncol(x), nrow=nrow(x))
    for(i in 1:nrow(x)) {
      for(j in 1:ncol(x)) {
        result[i,j] = exp(-(x[i,j])^2)
      }
    }
    result
  }
  a_1 = radbas(n)

  W_2 = matrix(c(-0.8721046 ,-0.3866168 , 0.1468913  ,2.2116062), nrow=1)
  b_2 =  1.08126

  a_2 = W_2%*%a_1+b_2
  a_2[1,1]

}
```

Now if we plot our approximation against the actual function:

```
library(ggplot2)
```

```
## Warning: package 'ggplot2' was built under R version 3.6.3
```

```
df = data.frame(x=seq(-2,2, length=1000))
y = matrix(0, nrow = 1000, ncol=1)
for(i in 1:1000) {
  y[i][1] = RBNN((df$x)[i])
}
df$y = y
ggplot(df, aes(x=x,y=y))+geom_hline(yintercept=0)+geom_vline(xintercept=0)+
  geom_line(colour="red")+stat_function(fun=actual_function, colour="blue")+
  ggtitle("RBNN and Actual Function")
```



The sum of squared errors for this problem is $\Sigma(t - a^2)^2$:

```
a_2 = matrix(0, nrow = 1, ncol=10)
for(i in 1:10) {
  a_2[1,i] = RBNN(p[i])
}
norm(a_2-t, "F")
```

```
## [1] 0.1553618
```

As we can see, we actually have a small sum of squared errors even though our network did not generalize well to the overall function, probably due to the fact that our test input space did not contain any value near 2.

Doubling Bias

Now, if we were to double the initial bias, it can be shown with the same process described above we would get the following final layer weights and bias:

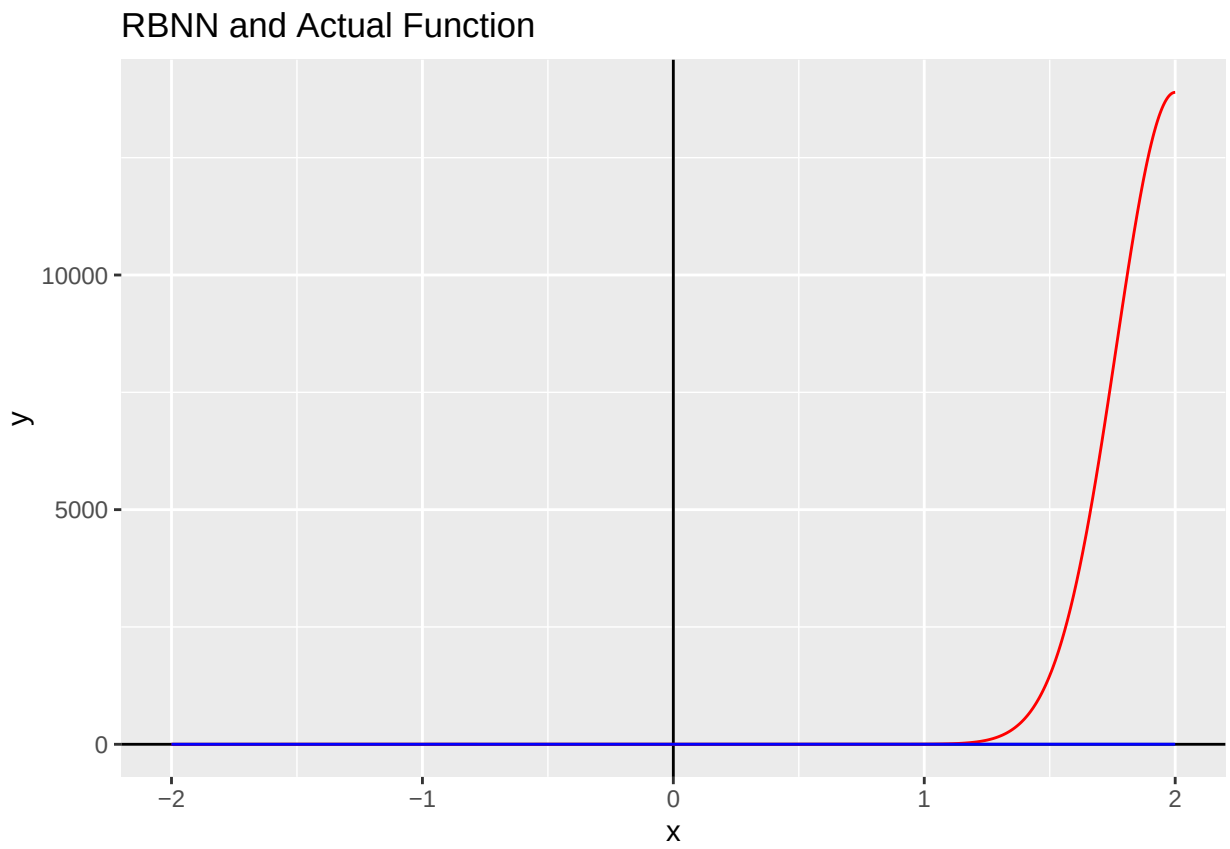
$$W_2 = \begin{bmatrix} -1.3846784 & -0.2837879 & 0.4425128 & 13891.5009956 \end{bmatrix}, \; b_2 = [0.8806578]$$

This will yield the following approximation:

```
# Radial Basis Neural Network
RBNN = function(p) {
  n = matrix(0, ncol=1, nrow=4)
  b_1 = matrix(3, ncol=1, nrow=4)
  W_1=matrix(seq(-2,2, length=4), ncol=1)
  for(i in 1:4) {
      # you're supposed to take the norm of the values p-w, but since its a single
      # entry, its the same as absolute
      n[i,1] = (abs(p-W_1[i][1])*b_1[i][1])
  }
  radbas = function(x) {
    result = matrix(0, ncol=ncol(x), nrow=nrow(x))
    for(i in 1:nrow(x)) {
      for(j in 1:ncol(x)) {
        result[i,j] = exp(-(x[i,j])^2)
      }
    }
    result
  }
  a_1 = radbas(n)

  W_2 = matrix(c(-1.3846784  , -0.2837879   ,  0.4425128 ,13891.5009956 ), nrow=1)
  b_2 =  0.8806578

  a_2 = W_2%*%a_1+b_2
  a_2[1,1]

}
```

Now if we plot our approximation against the actual function:

```
library(ggplot2)
df = data.frame(x=seq(-2,2, length=1000))
y = matrix(0, nrow = 1000, ncol=1)
for(i in 1:1000) {
  y[i][1] = RBNN((df$x)[i])
}
df$y = y
ggplot(df, aes(x=x,y=y))+geom_hline(yintercept=0)+geom_vline(xintercept=0)+
  geom_line(colour="red")+stat_function(fun=actual_function, colour="blue")+
  ggtitle("RBNN and Actual Function")
```



The sum of squared errors for this problem is $\Sigma(t - a^2)^2$:

```
a_2 = matrix(0, nrow = 1, ncol=10)
for(i in 1:10) {
  a_2[1,i] = RBNN(p[i])
}
norm(a_2-t, "F")
```

```
## [1] 0.3200008
```

As we can see, our sum of squared errors is extremely small, almost perfect; however, note that it did not generalize well to the total input space as our test input space did not contain any values near two, where the network diverged.

Halving Initial Bias

Now, if we were to halve the initial bias, it can be shown with the same process described above we would get the following final layer weights and bias:

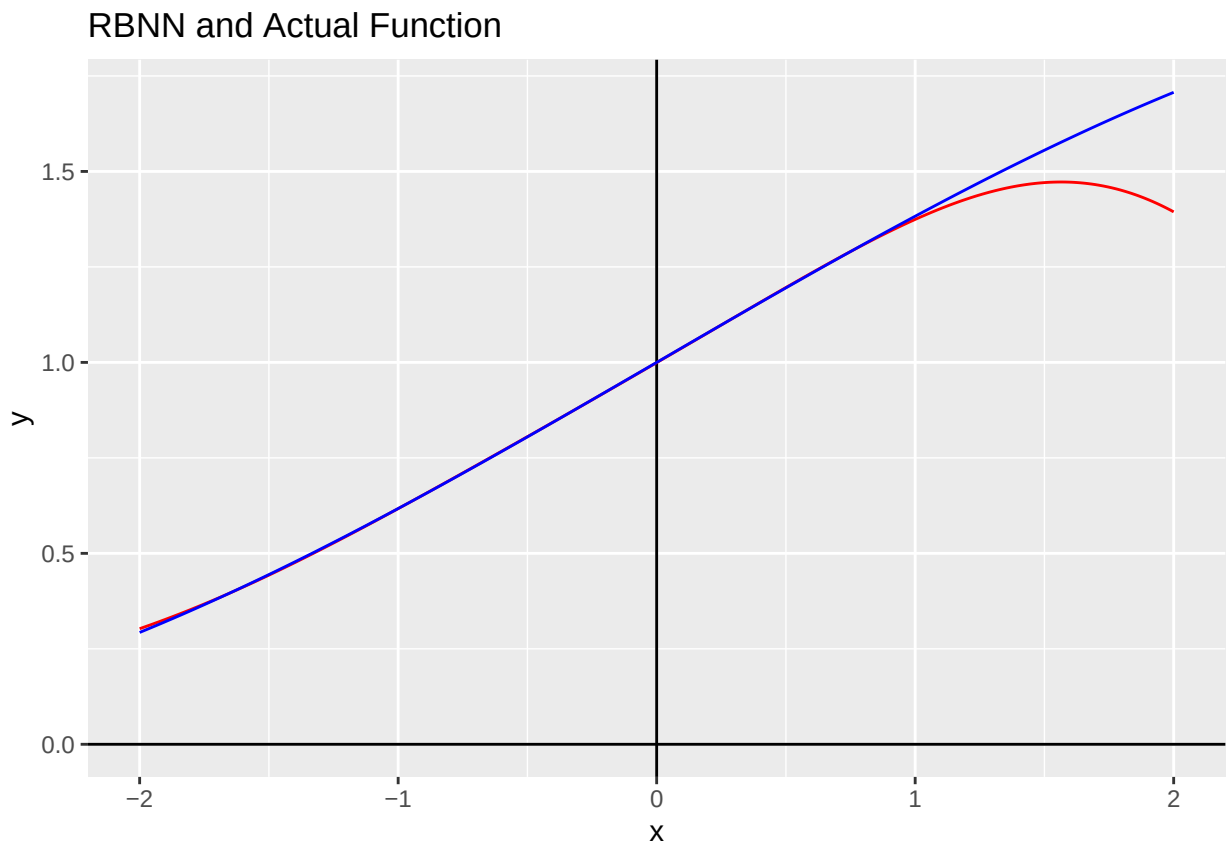
$$W_2 = \begin{bmatrix} 0.04540426 & 0.36222023 & 0.62917920 & 1.04362491 \end{bmatrix}, \; b_2 = [0.112438]$$

This will yield the following approximation:

```
# Radial Basis Neural Network
RBNN = function(p) {
  n = matrix(0, ncol=1, nrow=4)
  b_1 = matrix(0.75, ncol=1, nrow=4)
  W_1=matrix(seq(-2,2, length=4), ncol=1)
  for(i in 1:4) {
      # you're supposed to take the norm of the values p-w, but since its a single
      # entry, its the same as absolute
      n[i,1] = (abs(p-W_1[i][1])*b_1[i][1])
  }
  radbas = function(x) {
    result = matrix(0, ncol=ncol(x), nrow=nrow(x))
    for(i in 1:nrow(x)) {
      for(j in 1:ncol(x)) {
        result[i,j] = exp(-(x[i,j])^2)
      }
    }
    result
  }
  a_1 = radbas(n)

  W_2 = matrix(c(0.04540426, 0.36222023, 0.62917920 ,1.04362491), nrow=1)
  b_2 =  0.112438

  a_2 = W_2%*%a_1+b_2
  a_2[1,1]

}
```

Now if we plot our approximation against the actual function:

```
library(ggplot2)
df = data.frame(x=seq(-2,2, length=1000))
y = matrix(0, nrow = 1000, ncol=1)
for(i in 1:1000) {
  y[i][1] = RBNN((df$x)[i])
}
df$y = y
ggplot(df, aes(x=x,y=y))+geom_hline(yintercept=0)+geom_vline(xintercept=0)+
  geom_line(colour="red")+stat_function(fun=actual_function, colour="blue")+
  ggtitle("RBNN and Actual Function")
```



The sum of squared errors for this problem is $\Sigma(t - a^2)^2$:

```
a_2 = matrix(0, nrow = 1, ncol=10)
for(i in 1:10) {
  a_2[1,i] = RBNN(p[i])
}
norm(a_2-t, "F")
```

```
## [1] 0.001534613
```

After halving the intial bias we also get a good result when compared to the initial bias as its sum of squared errors is a lot less and it generalizes well to the total input space when compared to doulbing the initial bias.