Chapter 17: Radial Basis Networks

Brandon Morgan

2/9/2021

E17.11

Here we will repeat E17.10 except with regularization parameter $\rho = 0.2$ and ten neurons in the hidden layer

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 have the same 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=10), ncol=1)
W_1
```

```
##              [,1]
##  [1,] -2.0000000
##  [2,] -1.5555556
##  [3,] -1.1111111
##  [4,] -0.6666667
##  [5,] -0.2222222
##  [6,]  0.2222222
##  [7,]  0.6666667
##  [8,]  1.1111111
##  [9,]  1.5555556
## [10,]  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 $d_{max} = 0.\bar{4}$. From the size of our first weight matrix, 10x1, $S = 10$. Thus, our initial bias for the first layer willbe $\sqrt{10}/(0.\bar{4}) = 7.115$.

```
b_1 = matrix(7.115, ncol=1, nrow=10)
b_1
```

```
##         [,1]
##  [1,] 7.115
##  [2,] 7.115
##  [3,] 7.115
##  [4,] 7.115
##  [5,] 7.115
##  [6,] 7.115
##  [7,] 7.115
##  [8,] 7.115
##  [9,] 7.115
## [10,] 7.115
```

The weights and biases for the second layer were shown from the textbook in Eq. (17.31) to be: $X^* = [U^T U + \rho I]^{-1} U^T I$, 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=10)
for(q in 1:10) {
  for(i in 1:10) {
    # 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]
##  [1,] 14.442830  8.730632 12.1497922 19.7256852  2.4229697  6.41592610
##  [2,] 11.280607  5.568409  8.9875700 16.5634630  0.7392525  3.25370388
```

```
##  [3,]  8.118385  2.406187  5.8253477 13.4012408  3.9014748  0.09148166
##  [4,]  4.956163  0.756035  2.6631255 10.2390186  7.0636970  3.07074056
##  [5,]  1.793941  3.918257  0.4990967  7.0767963 10.2259192  6.23296278
##  [6,]  1.368281  7.080479  3.6613189  3.9145741 13.3881414  9.39518501
##  [7,]  4.530504 10.242702  6.8235411  0.7523519 16.5503637 12.55740723
##  [8,]  7.692726 13.404924  9.9857634  2.4098703 19.7125859 15.71962945
##  [9,] 10.854948 16.567146 13.1479856  5.5720926 22.8748081 18.88185167
## [10,] 14.017170 19.729368 16.3102078  8.7343148 26.0370303 22.04407390
##            [,7]      [,8]      [,9]      [,10]
##  [1,]  7.813139  7.749802 17.526502 12.2284516
##  [2,]  4.650916  4.587580 14.364280  9.0662294
##  [3,]  1.488694  1.425358 11.202058  5.9040072
##  [4,]  1.673528  1.736864  8.039835  2.7417849
##  [5,]  4.835750  4.899087  4.877613  0.4204373
##  [6,]  7.997972  8.061309  1.715391  3.5826595
##  [7,] 11.160195 11.223531  1.446831  6.7448817
##  [8,] 14.322417 14.385753  4.609053  9.9071039
##  [9,] 17.484639 17.547976  7.771276 13.0693262
## [10,] 20.646861 20.710198 10.933498 16.2315484
```

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,] 2.559763e-91  7.877130e-34  7.772414e-65 1.034816e-169  2.820667e-03
##  [2,] 5.433921e-56  3.417835e-14  8.303270e-36 7.112862e-120  5.789760e-01
##  [3,] 2.379259e-29  3.058786e-03  1.829605e-15  1.008418e-78  2.451231e-07
##  [4,] 2.148746e-11  5.646276e-01  8.315347e-04  2.948844e-46  2.140537e-22
##  [5,] 4.002610e-02  2.149760e-07  7.795040e-01  1.778599e-22  3.855461e-46
##  [6,] 1.537859e-01  1.688234e-22  1.507201e-06  2.212682e-07  1.432335e-78
##  [7,] 1.218721e-09  2.734578e-46  6.010892e-21  5.677731e-01 1.097559e-119
##  [8,] 1.992083e-26  9.136143e-79  4.944491e-44  3.005008e-03 1.734707e-169
##  [9,] 6.716225e-52 6.295792e-120  8.389174e-76  3.280434e-14 5.655087e-228
## [10,] 4.670442e-86 8.948551e-170 2.935835e-116  7.386385e-34 3.802483e-295
##               [,6]          [,7]          [,8]          [,9]         [,10]
##  [1,]  1.326341e-18  3.079174e-27  8.251241e-27 3.927985e-134  1.142252e-65
##  [2,]  2.525241e-05  4.034277e-10  7.242461e-10  2.459903e-90  2.006813e-36
##  [3,]  9.916660e-01  1.090215e-01  1.311197e-01  3.177467e-55  7.272214e-16
##  [4,]  8.032356e-05  6.076776e-02  4.896262e-02  8.465625e-29  5.435523e-04
##  [5,]  1.341947e-17  6.986334e-11  3.771171e-11  4.652131e-11  8.379746e-01
##  [6,]  4.624264e-39  1.656687e-28  5.991046e-29  5.273024e-02  2.664620e-06
##  [7,]  3.286736e-69  8.103009e-55  1.963107e-55  1.232771e-01  1.747651e-20
```

```
##  [8,] 4.818391e-108  8.174613e-90  1.326784e-90  5.944564e-10  2.364226e-43
##  [9,] 1.456981e-155 1.700995e-133 1.849573e-134  5.912518e-27  6.596872e-75
## [10,] 9.087009e-212 7.300517e-186 5.318115e-187  1.212942e-52 3.796664e-115
```

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,]  2.559763e-91  5.433921e-56 2.379259e-29 2.148746e-11 4.002610e-02
##  [2,]  7.877130e-34  3.417835e-14 3.058786e-03 5.646276e-01 2.149760e-07
##  [3,]  7.772414e-65  8.303270e-36 1.829605e-15 8.315347e-04 7.795040e-01
##  [4,] 1.034816e-169 7.112862e-120 1.008418e-78 2.948844e-46 1.778599e-22
##  [5,]  2.820667e-03  5.789760e-01 2.451231e-07 2.140537e-22 3.855461e-46
##  [6,]  1.326341e-18  2.525241e-05 9.916660e-01 8.032356e-05 1.341947e-17
##  [7,]  3.079174e-27  4.034277e-10 1.090215e-01 6.076776e-02 6.986334e-11
##  [8,]  8.251241e-27  7.242461e-10 1.311197e-01 4.896262e-02 3.771171e-11
##  [9,] 3.927985e-134  2.459903e-90 3.177467e-55 8.465625e-29 4.652131e-11
## [10,]  1.142252e-65  2.006813e-36 7.272214e-16 5.435523e-04 8.379746e-01
##                [,6]          [,7]          [,8]          [,9]         [,10]
##  [1,] 1.537859e-01  1.218721e-09  1.992083e-26  6.716225e-52  4.670442e-86
##  [2,] 1.688234e-22  2.734578e-46  9.136143e-79 6.295792e-120 8.948551e-170
##  [3,] 1.507201e-06  6.010892e-21  4.944491e-44  8.389174e-76 2.935835e-116
##  [4,] 2.212682e-07  5.677731e-01  3.005008e-03  3.280434e-14  7.386385e-34
##  [5,] 1.432335e-78 1.097559e-119 1.734707e-169 5.655087e-228 3.802483e-295
##  [6,] 4.624264e-39  3.286736e-69 4.818391e-108 1.456981e-155 9.087009e-212
##  [7,] 1.656687e-28  8.103009e-55  8.174613e-90 1.700995e-133 7.300517e-186
##  [8,] 5.991046e-29  1.963107e-55  1.326784e-90 1.849573e-134 5.318115e-187
##  [9,] 5.273024e-02  1.232771e-01  5.944564e-10  5.912518e-27  1.212942e-52
## [10,] 2.664620e-06  1.747651e-20  2.364226e-43  6.596872e-75 3.796664e-115
##       [,11]
##  [1,]     1
##  [2,]     1
##  [3,]     1
##  [4,]     1
##  [5,]     1
##  [6,]     1
##  [7,]     1
##  [8,]     1
##  [9,]     1
## [10,]     1
```

Now we can finally estimate the weights of the second layer via multiple linear repression with 0.2 regularization:

```
rho = 0.2
regular = rho*diag(11)
x = solve(t(U_1)%*%U_1+regular)%*%t(U_1)%*%t
W_2 = x[1:10,]
b_2 = x[11,1]
W_2
```

```
##  [1] -2.204712e-03 -4.525455e-01 -2.182553e-01 -1.464140e-01  8.573983e-02
##  [6]  2.022808e-01  5.964153e-01  2.227111e-03  2.431238e-14  5.474293e-34
```

```
b_2
```

```
## [1] 0.8118322
```

Now, we have the final neural network:

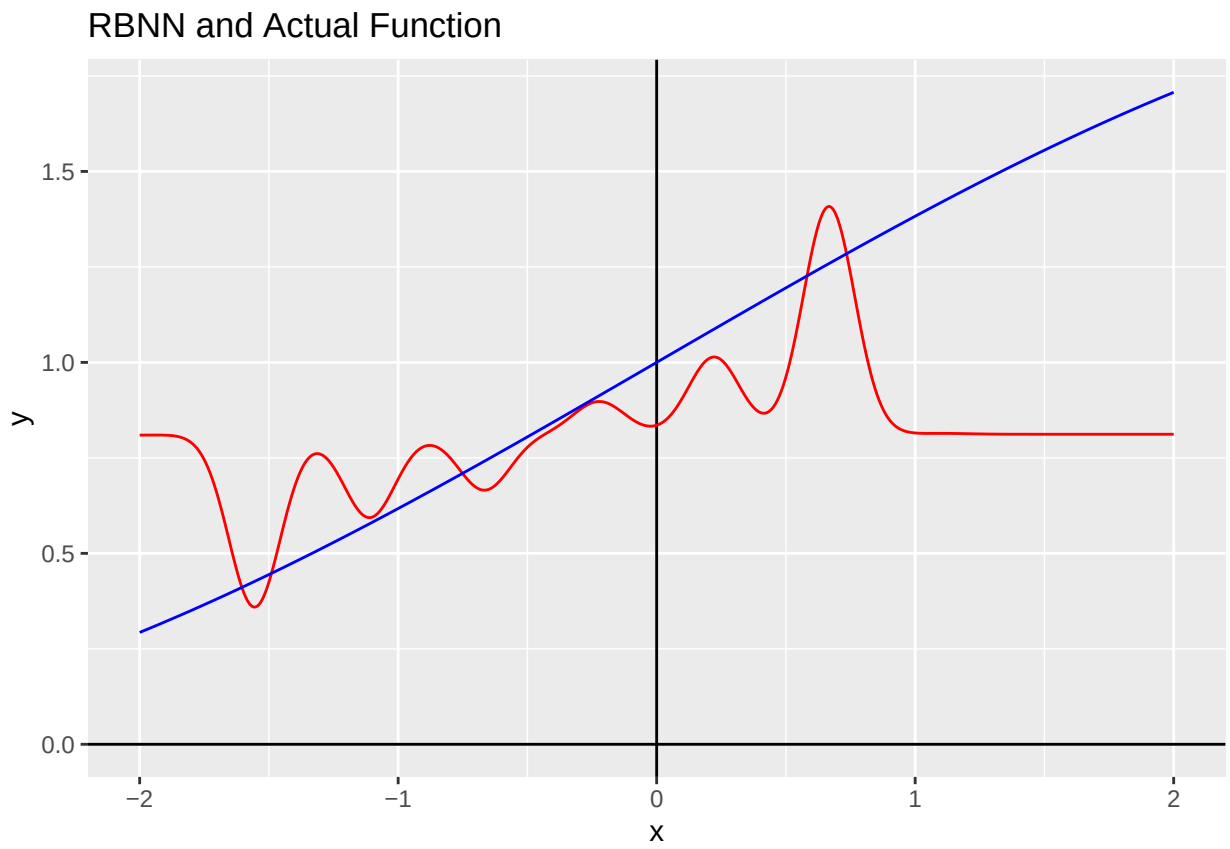
```
# Radial Basis Neural Network
RBNN = function(p) {
  n = matrix(0, ncol=1, nrow=10)
  b_1 = matrix(7.115, ncol=1, nrow=10)
  W_1=matrix(seq(-2,2, length=10), ncol=1)
  for(i in 1:10) {
      # you're supposed to take the norm of the values p-w, but since its a single
      # entry, its the same as 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(-2.204712e-03, -4.525455e-01, -2.182553e-01, -1.464140e-01,
                 8.573983e-02,  2.022808e-01,5.964153e-01,  2.227111e-03,
                 2.431238e-14,  5.474293e-34), nrow=1)
  b_2 =    0.8118322
  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] 0.4332711
```

As we can see, we actually have a sort of small sum of squared errors even though our network did not generalize well to the overall function. It seems that adding more neurons produced a model that overfitted the testing data and did not generalize well to the complete input space.

Doubling Initial Bias

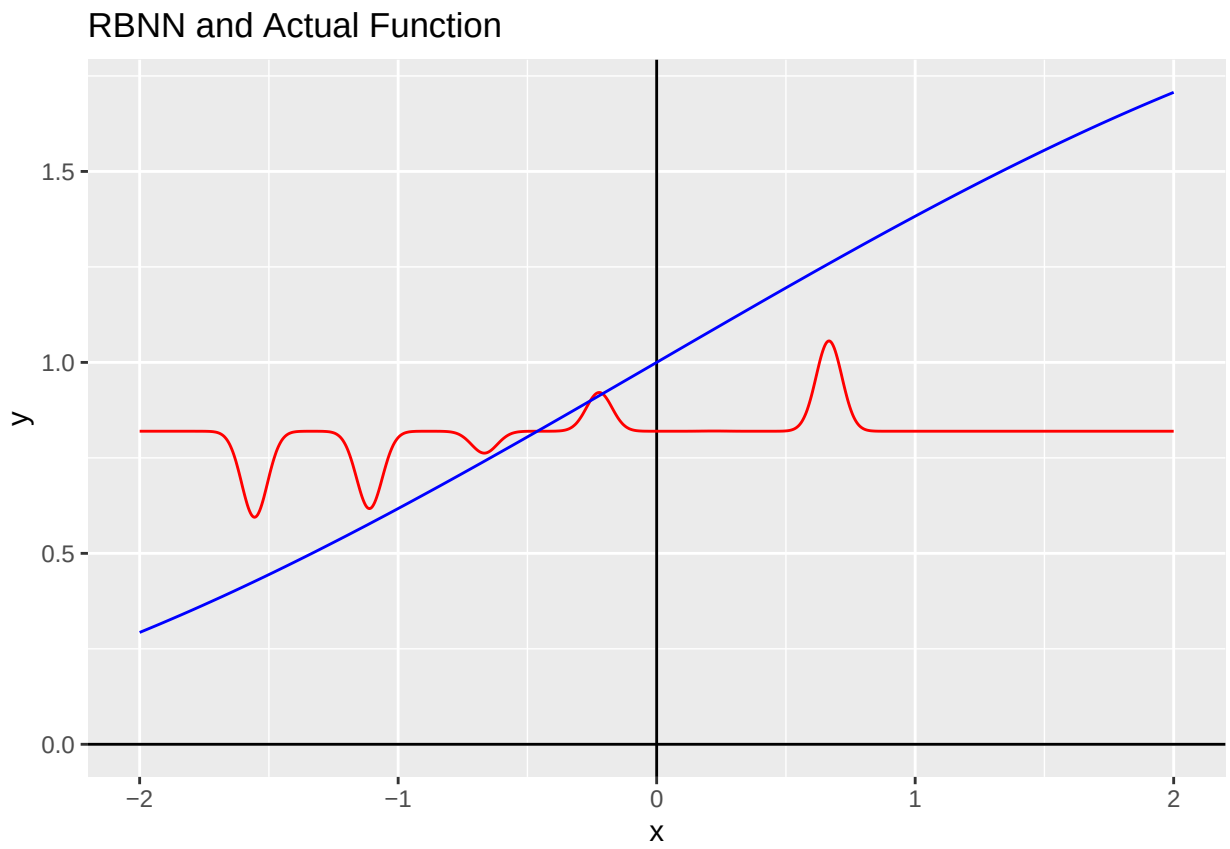
The updated weights and biases for the second layer were calcualted using the same procedure above but are not listed here.

```
# Radial Basis Neural Network
RBNN = function(p) {
  n = matrix(0, ncol=1, nrow=10)
  b_1 = matrix(2*7.115, ncol=1, nrow=10)
  W_1=matrix(seq(-2,2, length=10), ncol=1)
  for(i in 1:10) {
```

```
      # 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(-1.268974e-10,  -2.252619e-01 , -2.026167e-01 , -5.733735e-02,
                 1.014461e-01,  5.509417e-04  , 2.365128e-01 ,  1.852556e-10 ,
                 2.630960e-54,  6.762647e-133), nrow=1)
  b_2 =    0.8197361
  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] 0.7789259
```

Halving Initial Bias

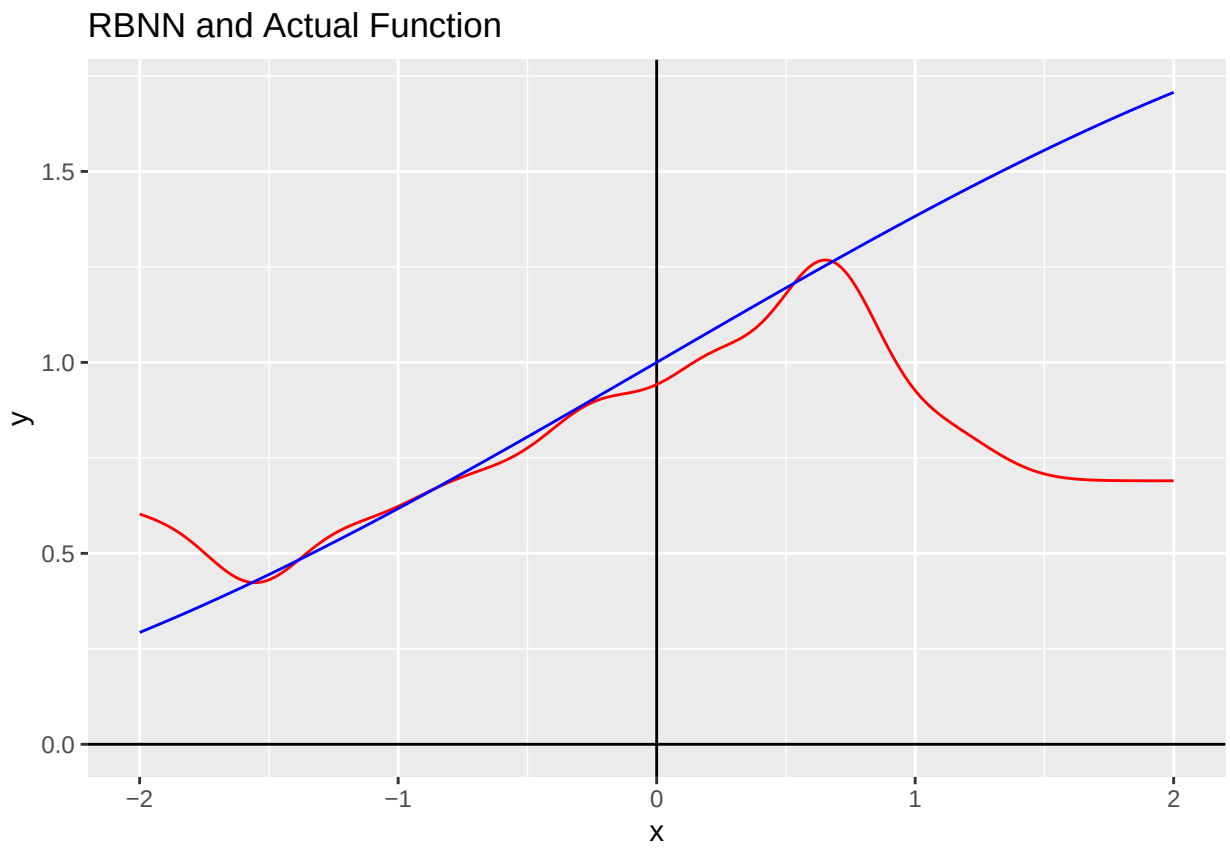
The updated weights and biases for the second layer were calcualted using the same procedure above but are not listed here.

```
# Radial Basis Neural Network
RBNN = function(p) {
  n = matrix(0, ncol=1, nrow=10)
  b_1 = matrix(0.5*7.115, ncol=1, nrow=10)
  W_1=matrix(seq(-2,2, length=10), ncol=1)
  for(i in 1:10) {
      # 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(-6.600601e-02 ,-2.547319e-01, -7.858525e-02 , 2.203105e-02 ,
                 1.876504e-01 , 2.796572e-01,   5.441831e-01 , 1.203897e-01  ,
                 2.173765e-04 , 2.662244e-09), nrow=1)
  b_2 =    0.6899307
  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] 0.1366464
```

The Network does well at halve initial bias with a small sum of squared error, but does not generalize well around the input 2 as it is not contained in the test input set.