

# CS545: Assignment 5

Zach Cashero

October 27, 2009

---

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Method</b>	<b>1</b>
<b>3</b>	<b>Implementation</b>	<b>2</b>
<b>4</b>	<b>Experiments</b>	<b>4</b>
4.1	Verification . . . . .	4
4.2	Fitting a Sine Curve . . . . .	5
4.3	Auto-MPG Data . . . . .	6
<b>5</b>	<b>Conclusions</b>	<b>10</b>

---

## 1 Introduction

Neural network methods can be used to generate predictions to a non-linear regression problem. A neural network consists of two or more layers of non-linear and linear functions. The inputs are first passed through the non-linear functions, and then these non-linear outputs are passed through a linear layer to produce the outputs. The parameters to all of the layers can be adjusted over time and trained on data in order to reduce the error. Therefore, a neural network can generate a fairly compact model that is still general purpose. In this paper, I will discuss an implementation of a neural network and apply it some contrived and real data.

## 2 Method

Neural networks can contain many layers, but for now we will just be considering a neural network with two layers, a hidden layer and an output layer. The hidden layer is at the front end, and the inputs are fed directly into this layer. This layer consists of hidden units where each contains a set of weights for the inputs, which we will refer to as  $V$ . The inputs are transformed by the weights on each hidden unit, and a non-linear function is then applied to it. The output of the hidden layer, which we will refer to as  $Z$ , can then be defined as  $Z = h(XV)$  where  $h$  is some non-linear function.

The  $h$  function can be chosen as anything if the nature of the data suggests that a certain non-linear function will work better than others. However, two common choices are the sigmoid function and the  $\tanh$  function. These both have nice properties since the derivatives approach zero as the magnitude of  $X$  increases. In this approach, we will be using the  $\tanh$  function for  $h$ . This hidden layer can then be thought of as a set of non-linear basis functions that are all fixed to be this  $h$  function. However, they are not truly fixed since the weights,  $V$ , can be adapted to transform the output of each hidden unit.

The number of hidden units is picked empirically based on the data. Essentially, more hidden units can create a more complex model that can represent more complex data. However, using too many hidden units can overfit to the training data since the hidden units will start to model the noise in the data.

The outputs of this hidden layer are then fed into the output layer. The number of units in the output layer must be fixed so that each output has one unit associated with it. Since the output of the hidden layer is used as inputs for the output layer, each output unit contains weights,  $W$ , for each of the hidden layer outputs. Essentially, each output unit generates an output based on a linear combination of the non-linear outputs from the hidden layer. Therefore the output,  $Y$ , is represented as  $Y = ZW$  or  $Y = h(XV)W$ .

The parameters of a neural net that can be adjusted to increase performance are the set of weights  $V$  and  $W$ . Using a training data set, the parameters are chosen to minimize the error function:

$$E = \frac{1}{N} \frac{1}{K} \sum_{n=1}^N \sum_{k=1}^K (y_{n,k} - t_{n,k})^2$$

An optimization function can then be used to find an optimal set of weights. In this paper, the scaled conjugate gradient method was used [1]. This will only find a local optimal though, so the final model that is learned is dependent on the initialization of these weights. The gradient of the error function can only be determined by finding two separate gradients, one with respect to  $W$  and the other with respect to  $V$ . The gradient with respect to  $W$  is:

$$\frac{1}{N} \frac{1}{K} Z^T (Y - T)$$

The gradient with respect to  $V$  can be found by backpropagating the errors to the hidden layer and comes out to be:

$$\frac{1}{N} \frac{1}{K} X^T ((Y - T)W^T * (1 - Z^2))$$

Since neural networks are susceptible to overfitting like other methods, some steps can be taken to limit this and improve the testing RMSE. We will explore one of these approaches by limiting the magnitudes of the weights. This is accomplished by introducing a new term that penalizes a certain model based on the magnitude of its weights. It is introduced into the error function so that the optimization algorithm will continue to reduce the amount of error until it gets to some balancing point where the error cannot be reduced much more without increasing the magnitudes of the weights even further. In my implementation, I only penalized the  $V$  weights, so the resulting error function would become:

$$E = \frac{1}{N} \frac{1}{K} \sum_{n=1}^N \sum_{k=1}^K (y_{n,k} - t_{n,k})^2 + \lambda * V_{n,k}^2$$

and the gradient with respect to  $V$  would change to:

$$\frac{1}{N} \frac{1}{K} X^T ((Y - T)W^T * (1 - Z^2)) + \lambda * V$$

Therefore, the  $\lambda$  parameter can be tuned to determine the impact of this weight penalty on the error function and gradient.

### 3 Implementation

The neural network described above has been implemented in R, and the code is shown below. The *makeNN* function takes as input the training data, number of hidden units, and a  $\lambda$  value if a weight magnitude penalty is used. The *nIterations*, *xPrecision*, and *fPrecision* parameters are all passed into the scaled conjugate gradient algorithm to determine the stopping criteria.

This function defines four internal functions which I will talk about, but notice that besides that, it is really very simple. The first step is to standardize the data and initialize the weights to random values. The weights start off with very small values, between -0.01 and 0.01, that are picked from a uniform distribution. The internal functions are then defined after that so that they use the standardized training data. After

that, the weights are then trained on the data using the scaled conjugate gradient method and the results are returned. Therefore, most of the work is being done in the scaled conjugate gradient method.

Two of the internal functions *pack* and *unpack* are merely used to transform the  $V$  and  $W$  weight matrices between their matrix forms and a single vector which contains all weights. The *sqErrorF* function takes a set of weights and calculates the squared error as shown above with the weight magnitude penalty. Since the *sqErrorF* is passed into the *scg* method, it is called at each iteration after the weights have been modified to determine the new error. The *gradF* function is used in much the same way by *scg*. It takes a set of weights and returns the gradient based on these. Then the *scg* method uses this to recompute the gradient each time the weights change.

```
makeNN <- function(Xtrain, Ttrain, nh, lambda=0, nIterations=10000, xPrecision=0,
                  fPrecision=0, hFunc=tanh) {
  N <- nrow(Xtrain)
  if(is.null(N))
    N <- length(Xtrain)
  ni <- ncol(Xtrain)
  if(is.null(ni))
    ni <- 1
  no <- ncol(Ttrain)
  if(is.null(no))
    no <- 1

  ### standardize the data
  standardizeX <- makeStandardizeF(Xtrain, addBias=TRUE)
  standardizeT <- makeStandardizeF(Ttrain)
  unstandardizeT <- makeUnstandardizeF(Ttrain)

  Xtrain <- standardizeX(Xtrain)
  Ttrain <- standardizeT(Ttrain)

  ### initialize the weights to random values
  V <- matrix(runif(ni+1, -0.01, 0.01), ni+1, nh)
  W <- matrix(runif(nh+1, -0.01, 0.01), nh+1, no)

  ### packs two weight matrices into a single vector
  pack <- function(V,W)
    matrix(c(V,W))

  ### unpacks weights from a single vector and builds two matrices from it
  unpack <- function(weights)
    list(V = matrix(weights[1:((ni+1)*nh)], ni+1, nh),
         W = matrix(weights[-(1:((ni+1)*nh))], nh+1, no))

  ### calculates the mean squared error given this set of weights
  sqErrorF <- function(weights) {
    unpacked <- unpack(weights)
    V <- unpacked$V
    W <- unpacked$W
    Z <- cbind(1, hFunc(Xtrain %*% V))
    Y <- Z %*% W

    mean((Y - Ttrain)^2) + lambda * sum(V[-1, ]^2)
  }

  ### calculates and returns the gradient with these weights
```

```

gradF <- function(weights) {
  unpacked <- unpack(weights)
  V <- unpacked$V
  W <- unpacked$W

  N <- nrow(Xtrain)
  K <- ncol(Ttrain)
  Z <- cbind(1, hFunc(Xtrain %*% V))
  Y <- Z %*% W
  gradV <- t(Xtrain) %*% ((Y - Ttrain) %*% t(W[-1,]) * (1 - Z[, -1]^2))
  + rbind(0, V[-1,]) * lambda
  gradW <- t(Z) %*% (Y - Ttrain)

  return(pack(gradV, gradW) / (N * K))
}

scgResult <- scg(pack(V,W), sqErrorF, gradF, nIterations=nIterations,
  xPrecision=xPrecision, fPrecision=fPrecision, ftracep=TRUE, xtracep=TRUE)
unpacked <- unpack(scgResult$x)
V <- unpacked$V
W <- unpacked$W

list(V=V, W=W, standardizeX=standardizeX, standardizeT=standardizeT,
  unstandardizeT=unstandardizeT, lambda=lambda, hFunc=hFunc, scg=scgResult)
}

```

The following code shows the *useNN* function which takes a neural net model that was created with the *makeNN* function and a set of testing data. First, it standardizes the data and computes the outputs of the hidden layer,  $Z$ . It then uses  $Z$  to compute the final outputs,  $Y$ . Notice that since the target variables were standardized in *makeNN*, the computed outputs must be standardized before they are returned from *useNN*.

```

useNN <- function(nnet, Xtest) {
  Xtest <- nnet$standardizeX(Xtest)

  Z <- cbind(1, nnet$hFunc(Xtest %*% nnet$V))
  Y <- Z %*% nnet$W
  list(Y=nnet$unstandardizeT(Y), Z=Z)
}

```

## 4 Experiments

### 4.1 Verification

This verification test uses a very simple data set with predictable outcomes in order to demonstrate correct functionality. The code below shows how the data was generated and tested. The data is simply a sequence of integers from 1 to 100, and the targets are the same as the  $x$  values. This would just produce a linear relation. This code uses the *makeNN* function with only a single hidden unit in order to easily verify that the weights are adapted correctly.

```

Xtrain <- seq(1,100)
Ttrain <- Xtrain

nnet <- makeNN(Xtrain, Ttrain, 1, nIterations=10000, xPrecision=1e-8,
  fPrecision=1e-8, hFunc=tanh)

```

Since we know the data is linear, we expect the non-linear hidden unit to approximate a linear function. Using the *tanh* function, this can be accomplished by multiplying  $x$  by a weight with a very small magnitude. When looking at the *tanh* function, multiplying the input by a small weight basically reduces the  $x$  range and can be thought of as almost “zooming in” on the curve. If we “zoom in” far enough, the *tanh* curve becomes almost linear. Besides a small weight on the input of the hidden layer, we should also expect the bias weights to be almost zero, since there is no shifting of the data. Also, since the input to the hidden layer is multiplied by a small weight, and the output  $Z$  will reflect this. Therefore the output layer can reverse this effect if its weight is the reciprocal of the hidden layer weight. The R output below shows the weights that were trained on the data above. It shows that all the observations we made do hold true.

V:

```

                [,1]
bias -0.003309599
X1    0.264623970

```

W:

```

                [,1]
bias 0.01220707
Z1   3.92968636

```

## 4.2 Fitting a Sine Curve

This next experiment tests the neural network when trying to model a noisy sine curve. The code below shows the function used to generate the output as well as how the training and testing data is generated. The training data is a set of 40 samples evenly spaced between 0 and 20. The testing data is in the same range as the training data, but it is offset so that each sample falls directly between two of the training samples. Therefore, all the data can be in the same range while still having two distinct sets of data. The random noise also ensures that the two data sets will be different. The data is then applied to a neural network with 10 hidden units.

```

f <- function (x) -1 + 0.05 * x + 0.4 * sin(x) + 0.3 * rnorm(length(x))
nSamples <- 40
xmax <- 20
Xtrain <- matrix(seq(0,xmax,length=nSamples),nSamples,1)
Ttrain <- f(Xtrain)
Xtest <- Xtrain + xmax/nSamples/2
Ttest <- f(Xtest)

nh <- 10
nnet <- makeNN(Xtrain, Ttrain, nh, nIterations=10000, xPrecision=1e-8,
              fPrecision=1e-8, hFunc=tanh)
resultTrain <- useNN(nnet, Xtrain)
resultTest <- useNN(nnet, Xtest)

```

The plots in Figure 1 shows the results of this test on the noisy sine curve. Figure 1(a) shows how the RMSE changes over time as the weights are being trained. In this case, an epoch is an iteration of the scaled conjugate gradient method. This shows that the RMSE drops off quickly and then slowly settles over time. This suggests that the number of iterations could have been reduced dramatically with little effect on the performance. Figure 1(b) shows an interesting look at the outputs from the hidden layer. There are 10 different curves, one for each hidden unit, that are all variations of the *tanh* function. It is immediately obvious that there are probably too many hidden units in this network because many of the outputs are very similar and almost duplicated. Figures 1(c) and 1(d) show the output of the neural network in relation to the target for both the training and testing data, respectively. As expected, the output closely fits many of the spikes and random noise of the training data. Although the testing data cannot be matched to this extent, it does still seem to capture the overall trend of a badly formed sine curve. Figure 1(e) contains a

lot of information about the weights in the network. It shows a box for each of the weights in the hidden and output layers, where the size of the box is relative to the magnitude of the weight. The color of the box indicates the sign of the weight, where green is positive and red is negative. Again, it is obvious that many of the hidden units are redundant, even when considering the weights in the output layer associated with those hidden outputs. The first three and the last two hidden units are probably not adding much more information than just one of each. Some of the hidden outputs can be associated directly with features in the data. For instance, the 7th hidden unit with the largest weight magnitudes can be seen as an almost vertical drop in Figure 1(b). Since the output weight is positive for this hidden output, it should be correlated with a drop in the input data, which can be seen right around  $x = 15$ .

The next experiment with this data tests the effect that the number of hidden units has on performance. The following code shows how this data was collected. The number of hidden units was varied from 1 to 20, and the train and test RMSE was collected for each. For a given number of hidden units, the trial was repeated 20 times, and the data was averaged in order to reduce some of the variance due to the stochasticity of the scaled conjugate gradient algorithm.

```
### vary the number of hidden units
rmse <- NULL
for(i in 1:20) {
  nh <- i
  for(j in 1:20) {
    nnet <- makeNN(Xtrain, Ttrain, nh, nIterations=10000, xPrecision=1e-8,
                  fPrecision=1e-8, hFunc=tanh)
    resultTrain <- useNN(nnet, Xtrain)
    resultTest <- useNN(nnet, Xtest)
    rmse <- rbind(rmse, c(nh, RMSE(resultTrain$Y, Ttrain),
                           RMSE(resultTest$Y, Ttest)))
  }
}

### gather the means
meanRMSEs <- NULL
for(nh in unique(rmse[,1]))
  meanRMSEs <- rbind(meanRMSEs, colMeans(rmse[rmse[,1]==nh, ]))
```

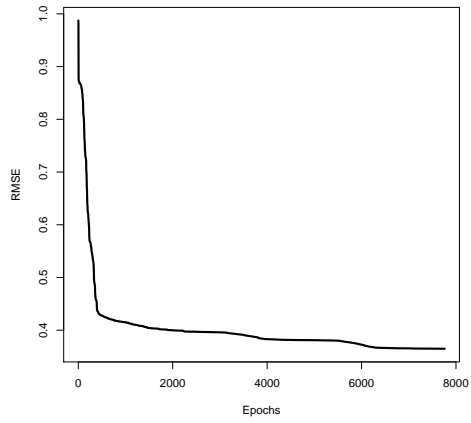
The plot in Figure 2 shows the train and test RMSE as the number of hidden units is varied. As expected, the train RMSE continually decreases since more hidden units are able to model more complex data and therefore are able to fit the data and its noise better. It was a little surprising that the test RMSE did not change very drastically. However, there does seem to be a definite minimum somewhere around five hidden units.

### 4.3 Auto-MPG Data

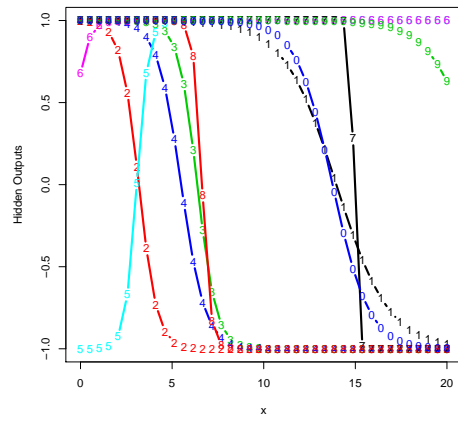
This next experiment uses a set of real-world data to test the neural network. This data contains six input attributes, number of cylinders, displacement, weight, acceleration, model year, and origin. The outputs from the network will be miles per gallon and horsepower. The following code shows how this data was read in and organized. The origin attribute was converted to indicator variables since it consisted of three discrete values which seemed to be arbitrary.

```
### Read mpg data from UCI
mpg <- read.table("auto-mpg.data")

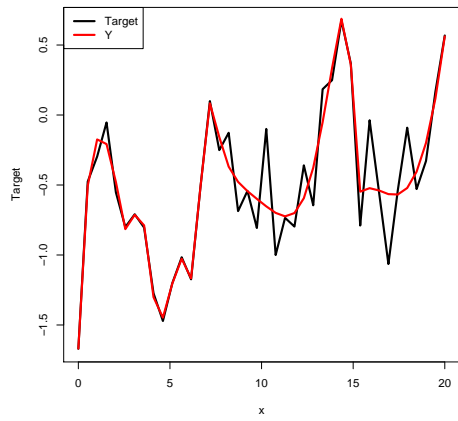
### Remove all samples that have at least one "?"
keepRows <- apply(mpg != "?", 1, all)
mpg <- apply(mpg[keepRows,1:8], 2, as.numeric)
mpg <- cbind(mpg[, -8], makeIndicatorVars(mpg[, 8])[, -3])
mpgnames <- c("mpg", "cylinders", "displacement", "horsepower",
```



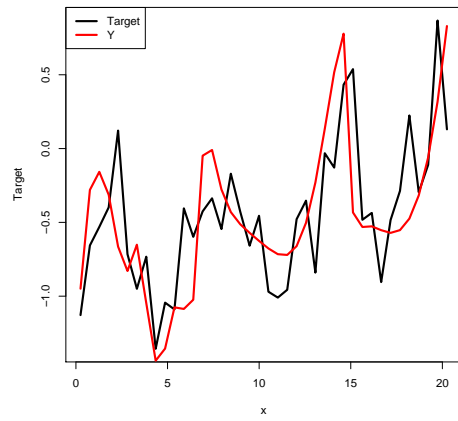
(a) Training RMSE decreases with more epochs



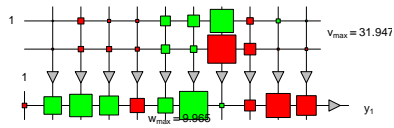
(b) Output of hidden units



(c) Output of neural net against target for training data



(d) Output of neural net against target for testing data



(e) Diagram of all weights in the neural net

Figure 1: Performance of neural network when trying to fit a noisy sine curve

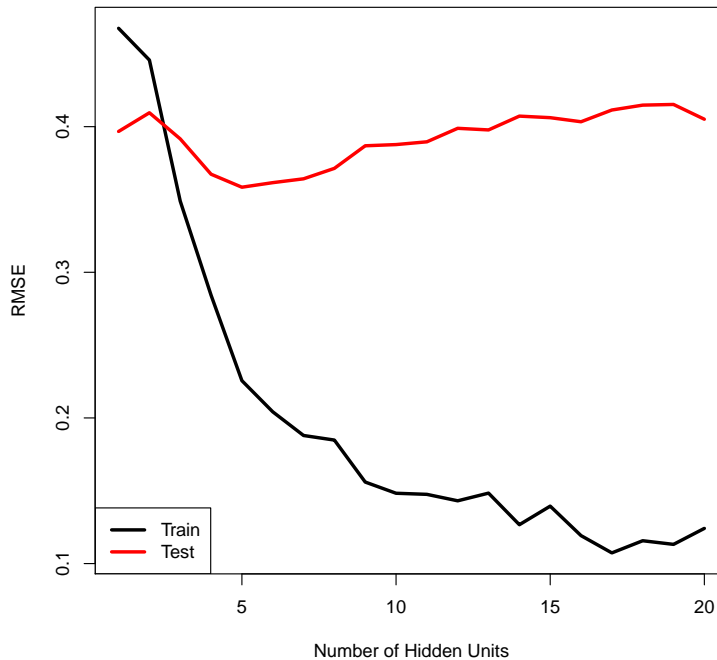


Figure 2: This shows the changes in the train and test RMSE as the number of hidden units is changed.

```
      "weight", "acceleration", "year", "origin1", "origin2")
colnames(mpg) <- mpgnames
```

```
X <- mpg[, c(-1, -4)]
T <- mpg[, c(1, 4)]
```

```
partition = makePartitionF(X, 0.8)
Xtrain <- partition(X, isTrainSet=TRUE)
Xtest <- partition(X, isTrainSet=FALSE)
Ttrain <- partition(T, isTrainSet=TRUE)
Ttest <- partition(T, isTrainSet=FALSE)
```

This first experiment tested the effect of the number of hidden units on this data. In the interest of time, only 5 different values were tested, again repeating each of these 20 times and averaging over the data. The number of hidden units tested were 4, 8, 12, 16, and 20. The results are shown in Figure 3. Again, the train RMSE continues to decrease as the number of hidden units increases. In this data set, it seems that the optimal number of hidden units that minimizes the test RMSE is around four. Even though there is a higher number of dimensions, this seems to suggest that the data is rather simple to be able to be modeled well with only four hidden units.

```
hiddenResults <- NULL
for(nh in seq(4, 20, by=4)) {
  for(i in 1:20) {
    nnet <- makeNN(Xtrain, Ttrain, nh, xPrecision=1e-8, fPrecision=1e-8)
    resultTrain <- useNN(nnet, Xtrain)
    resultTest <- useNN(nnet, Xtest)
    hiddenResults <- rbind(hiddenResults, c(nh, RMSE(resultTrain$Y, Ttrain),
```

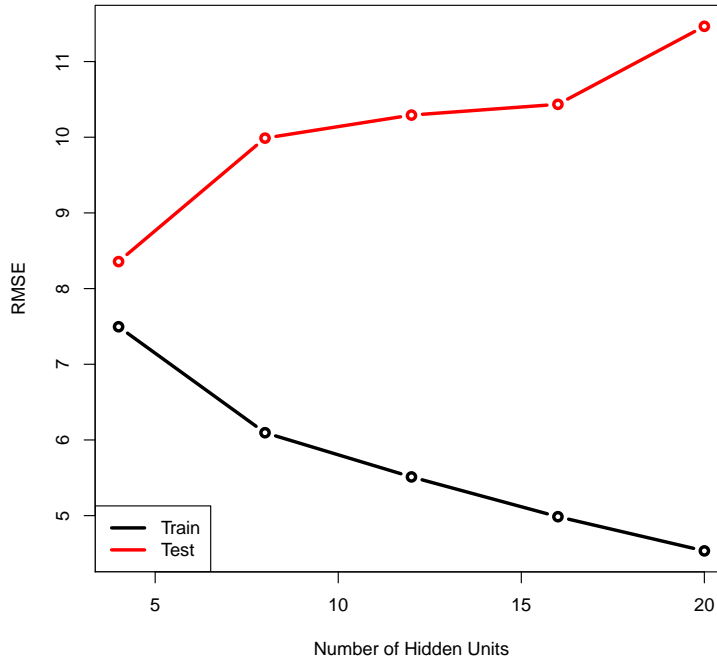


Figure 3: This shows the changes in train and test RMSE as the number of hidden units is varied.

```

    RMSE(resultTest$Y, Ttest)))
  }
}

meanHiddenResults <- NULL
for(nh in unique(hiddenResults[,1])) {
  meanHiddenResults <- rbind(meanHiddenResults,
    colMeans(hiddenResults[hiddenResults[,1]==nh, ]))
}

```

The next experiment with this data tests the effects of the  $\lambda$  parameter.  $\lambda$  was tested at five different values, 0, 0.0001, 0.001, 0.01, and 0.1. As  $\lambda$  is increased, the weight magnitudes are reduced to potentially limit overfitting, but it also can reduce the power of the model. I used powers of 10 as values for  $\lambda$  in order to capture the interesting results that occur when  $\lambda$  is very small. The results shown in Figure 4 show similar trends that can be seen in previous weight penalty methods such as ridge regression. The test RMSE starts off high when  $\lambda = 0$ , and then is reduced by using a small  $\lambda$  value, but eventually rises as  $\lambda$  grows. The minimum  $\lambda$  value is the third point which corresponds to  $\lambda = 0.001$ . It appears that with this data,  $\lambda$  would be most useful in the range of 0 to 0.02.

```

lambdaResults <- NULL
for(lambda in c(0, 0.0001, 0.001, 0.01, 0.1)) {
  for(i in 1:20) {
    nnet <- makeNN(Xtrain, Ttrain, 10, lambda=lambda, xPrecision=1e-8,
      fPrecision=1e-8)
    resultTrain <- useNN(nnet, Xtrain)
    resultTest <- useNN(nnet, Xtest)
    lambdaResults <- rbind(lambdaResults, c(lambda,

```

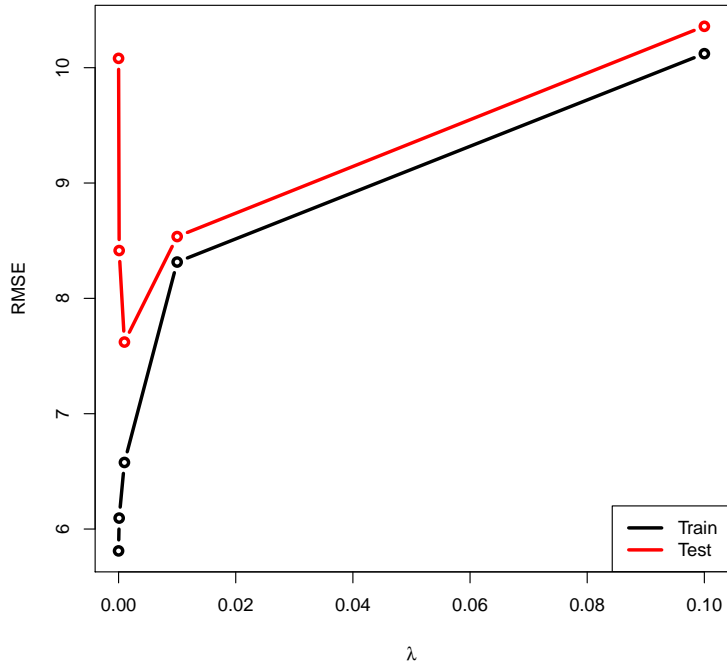


Figure 4: This shows the changes in train and test RMSE as  $\lambda$  is varied.

```

    RMSE(resultTrain$Y, Ttrain), RMSE(resultTest$Y, Ttest)))
  }
}

meanLambdaResults <- NULL
for(lambda in unique(lambdaResults[,1])) {
  meanLambdaResults <- rbind(meanLambdaResults,
    colMeans(lambdaResults[lambdaResults[,1]==lambda, ]))
}

```

## 5 Conclusions

In conclusion, a neural network is a powerful tool that is able to model complex and non-linear data. One advantage of neural networks seems to be the ability to model a complex relationship in a relatively compact model since they have been shown to be useful with a small number of hidden units. Two methods to reduce overfitting, limiting the number of hidden units and using a weight magnitude penalty, have both been shown to be effective. In practice, it would be beneficial to find the optimal values for the number of hidden units and  $\lambda$ .

## References

- [1] Martin F. Moller. A scaled conjugate gradient algorithm for fast supervised learning. *Neural Networks*, 6:525–533, 1993.