

# CS545 - Assignment 5 : neural network with Scaled Conjugate Gradient

Geoffrey Sewell

October 27, 2009

---

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Neural Network</b>	<b>1</b>
2.1	Set Up . . . . .	1
2.1.1	makeStandardizeF . . . . .	1
2.1.2	makeNN . . . . .	2
2.1.3	useNN . . . . .	5
2.2	Observation and Discussion . . . . .	6
<b>3</b>	<b>Noisy Sine Curve</b>	<b>6</b>
3.1	10 Hidden Units . . . . .	8
3.1.1	Set up . . . . .	8
3.1.2	Results and Discussion . . . . .	8
3.2	Various Number of Hidden Units . . . . .	10
3.2.1	Set up . . . . .	10
3.2.2	Discussion . . . . .	11
<b>4</b>	<b>Automobile Miles Per Gallon Data</b>	<b>11</b>
4.1	Set Up . . . . .	12
4.2	Discussion . . . . .	14
<b>5</b>	<b>Conclusion</b>	<b>14</b>

---

## 1 Introduction

A neural network is yet another form of linear modeling. Given a training and testing set, the neural network can search for a set of weights that, when applied to the training data set, will result in an output that hopefully matches the expected output completely. In this paper, the process of creating and training a neural network will be covered. Also, how the different parameters play a role in determining the success of a neural network will be further explored in various test conditions such as for a noisy sine curve and for a real data set.

## 2 Neural Network

In order to create a neural network, two functions have been created. One function is responsible for training the neural network, with help from the scaled conjugate gradient, and the other function applies a set of

data to the neural network and returns the predicted results. One thing to keep in mind, is that this neural network is used for prediction of continuous values.

## 2.1 Set Up

Creating the neural network requires a bunch of steps in order to set up an accurate neural network. The functions listed below are vital in to setting up and training a neural network.

### 2.1.1 makeStandardizeF

The makeStandardizeF function, which was provided by Professor Anderson, creates a standardize function based off of the mean and standard of deviation of the data set that is passed to it. While this functionality is still there, the ability to create an unstandardize function was added. The code that was added to modify the makeStandardizeF function is below

```
makeStandardizeF <- function(X, StanAndUnstan=FALSE) {
  if (missing(X)) {
    cat("Usage:
      standardize <- makeStandardizeF(X)
          ## X is nSamples x nDimensions
      Xs <- standardize(X)
      X2s <- standardize(X2)\n")
    return(invisible())
  }
  ## X is nSamples x nDimensions
  mu <- colMeans(X)
  sigma <- sd(X) ##sd should be named colSDs

  standardize <- function(newX) {
    nr <- nrow(newX)
    nc <- ncol(newX)
    (newX - matrix(mu, nr, nc, byrow=TRUE))
      / matrix(sigma, nr, nc, byrow=TRUE)
  }

  unstandardize <- function(newX) {
    nr <- nrow(newX)
    nc <- ncol(newX)
    #(newX - matrix(mu, nr, nc, byrow=TRUE))
      / matrix(sigma, nr, nc, byrow=TRUE)
    newX*matrix(sigma, nr, nc, byrow=TRUE)
      + matrix(mu, nr, nc, byrow=TRUE)
  }

  if (StanAndUnstan)
    list(standardize=standardize,
         unstandardize=unstandardize)
  else
    standardize
}
```

This modified makeStandardizeF function can now take a second argument that determines whether it should return a list of both the standardize and unstandardize function or just the standardize function.

The unstandardize function uses the mean and standard of deviation value previously calculated to turn a standardized set of data, that used the same mean and standard of deviation, to its original values.

The ability to unstandardize are necessary for after the target attributes are predicted from the neural network. The outputted results will come back as standardize according to the function used to standardize the target attributes. In order to get back the actual predicted value of the target attributes, the output from the neural network must be unstandardized first.

### 2.1.2 makeNN

The function makeNN is responsible for the training of the neural network. Given a set of input data and the expected target data, the neural network is trained and is hopefully able to successfully predict the values of the target attributes when given a data set to predict.

The API for the makeNN function is

```
makeNN <- function(Xtrain, Ttrain, numHiddenUnits,
                  lambda=0, nIterations=10000, xPrecision=0, fPrecision=0)
```

Passed arguments to the function are the input attributes and target attributes of the training data set, the number of hidden units to use in the neural network, a value for lambda to avoid over fitting, and the number of max iterations, xPrecision, and fPrecision which are all used by the scaled conjugate gradient function provided by Professor Anderson. While the input attributes, target attributes, and number of hidden units must be passed to makeNN; lambda, the number of iterations, xPrecision, and fPrecision are optional arguments to this function because they have default values.

The makeNN function makes use of a number of other functions which are used to help train the neural network. All of the following functions are local to the makeNN function. One of these functions is the pack function which is shown below

```
pack <- function(V, W)
{
  matrix(c(V,W))
}
```

The pack function is responsible for combining the Hidden and Output weights in to a single column matrix which is helpful when passing the weights around to various functions.

Another function that is found inside of the makeNN function is the unpack function. This does the opposite of pack and splits the column matrix up into the respective V(hidden) and W(output) weights and returns them in a list. The code for this is shown below

```
unpack <- function(weights)
{
  list(V = matrix(weights[1:((ni+1)*nh)], ni+1, nh),
       W = matrix(weights[-(1:((ni+1)*nh))], nh+1, no))
}
```

The size of the V and W weight vectors is dependent upon the weights and attributes of the data set. The size of the V weights column is dependent upon the number of input attributes and number of hidden units while the W weights column depends on the number of hidden units and the number of output weights.

The sqErrorF function determines how close the weights are to the actual values of the training set using the root mean square error and helps guide the scaled conjugate gradient in determining when it should stop modifying the weights.

```
sqErrorF <- function(weights)
{
  weight <- unpack(weights)

  X1 <- stanXTrain(Xtrain)

  X1 <- cbind(1, X1)
  Z <- tanh(X1 %*% weight$V)
  Y <- cbind(1, Z) %*% weight$W
```

```

T <- stanTTrain(Ttrain)
error <- mean((Y - T)^2) + lambda
          * (t(weight$V[-1,]) %*% weight$V[-1,])
}

```

The sqErrorF function takes weights passed in and splits them into their individual V and W weights. Before calculating the root mean square error, the training input data set must be standardized and have a bias column appended to the beginning. Once that has been done, the new standardized training set must be sent forward through the neural network with the new weights. Once the output from the neural network has been calculated the must be compared with the standardized target attributes. The root mean square error plus the lambda value times the hidden weights to reduce over fitting.

The last function that is placed in the makeNN function is the gradF function. This function is responsible for calculating the gradient of the squared error of the weights. The code for the gradF function is located below

```

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

  #Forward Pass
  X1 <- stanXTrain(Xtrain)
  X1 <- cbind(1,X1)
  Z <- tanh(X1 %*% V)
  Y <- cbind(1,Z) %*% W

  T <- stanTTrain(Ttrain)
  error <- Y-T

  #Backward Pass

  if (is.matrix(W[-1,]))
    V <- ( t(X1) %*% ( error %*% t(W[-1,])
                    * (1-Z^2) ) ) / (nrow(Xtrain) * ncol(Ttrain))
    + lambda*rbind(0,V[-1,])
  else
    V <- ( t(X1) %*% ( error %*% as.matrix(W[-1,])
                    * (1-Z^2) ) ) / (nrow(Xtrain) * ncol(Ttrain))
    + lambda*rbind(0,as.matrix(V[-1,]))

  W <- ( t(cbind(1,Z)) %*% error ) / (nrow(Xtrain)
    * ncol(Ttrain))

  pack(V,W)
}

```

The weights are unpacked in to V and W, the standardized input attributes are sent forward through the neural network. Once the output is calculated, the error is calculated and used in the backward pass. Since the gradient of the error is needed for the backward pass, the actual values are subtracted from the output values of the forward pass. Also, when calculating the backward pass for the hidden weights, the lambda value multiplied by the hidden weights to account for possible over fitting. The gradient values are then packed in to a single column matrix and returned.

The if statement used is to determine whether the W weight, without the first row, is still a matrix. From earlier experiences, I have found that R will automatically convert a matrix with 1 row to a vector. As a result of the matrix being converted to a vector, the transpose of the vector does not work and a

non-conformable arguments error is thrown. In the event that the matrix, without the first row, is a vector, the resulting vector is turned into a column matrix. The column matrix is actually the transpose of a vector, so the matrix multiplication works out when that is done. If W turns out to be a matrix, then the transpose of it can be taken even without the first row.

To get the training of the neural network going, some variable must first be initialized.

```
ni <- ncol(Xtrain)
nh <- numHiddenUnits
no <- ncol(Ttrain)

V <- matrix(runif((ni+1)*nh, -.01,.01), ni+1, nh)
W <- matrix(runif((nh+1)*no, -.01,.01), nh+1, no)
```

The number of input attributes, target attributes, and the number of hidden units must be kept track of. These values determine how the weight attributes are going to be split apart in the unpack method mentioned above. Also, V and W must be initialized to a small value. This determines the starting point on the gradient for using scaled conjugate gradient. If the weights are too big, then the gradient may find itself on a flat curve and will be unable to modify the weights enough to improve the neural network's ability to predict.

Next the standardize function for the training input attributes as well as the standardize and unstandardize function for the training target attributes must be calculated. These standardize functions will be used in the gradF function as well as the sqErrorF function in order to calculate the error as well as accurately calculate the forward pass and backward pass of the neural network. The code for standardization is given below

```
stanXTrain <- makeStandardizeF(Xtrain)
makeResults <- makeStandardizeF(Ttrain, TRUE)

stanTTrain <- makeResults$standardize
unstanTTrain <- makeResults$unstandardize
```

Now that the values and standardize functions are created, as shown above, the scaled conjugate gradient function can be run. From the arguments passed to the makeNN, such as nIterations, fPrecision, and xPrecision; the scaled conjugate gradient is able to run using those arguments. The gradF and sqErrorF functions are passed as arguments to scaled conjugate gradient function to determine the direction of the gradient and see how close the output with the current weights is in regards to the actual values of the target attributes.

```
scgResult <- scg(pack(V,W), sqErrorF, gradF,
                 nIterations = nIterations,
                 xPrecision = xPrecision,
                 fPrecision = fPrecision,
                 ftracep = TRUE, xtracep = TRUE)

VW <- unpack(scgResult$x)
V <- VW[[1]]
W <- VW[[2]]
```

```
list(V=V, W=W, standardizeTtrain=stanTTrain,
     standardizeXtrain=stanXTrain,
     unstandardizeTtrain=unstanTTrain,
     scgResults=scgResult, ni=ni, nh=nh, no=no)
```

Once the scaled conjugate gradient function has finished running, the weights can be retrieved. They first must be split into their V and W components and sent in a list. Other values returned that are necessary for the neural network to make predictions are the standardize functions for both the input attributes and target attributes, the unstandardize function for the target attributes. Also returned are the number of input

attributes, number of hidden units, number of output attributes, and the progress information from when the scaled conjugate gradient was ran. This information is helpful in plotting the evolution of the weights and determining how well the overall output is in regards to the actual value.

Once the makeNN is finished, the resulting list is used to predict the target attributes for a set of testing data.

### 2.1.3 useNN

The useNN function predicts the target attributes based off of the neural network that was just trained given a particular data set and returns the predicted values.

```
useNN <- function(nnet , Xtest)
{
  standardize <- nnet$standardizeXtrain
  unstandardize <- nnet$unstandardizeTtrain
  V <- nnet$V
  W <- nnet$W

  Xs <- standardize(Xtest)
  X1 <- cbind(1, Xs)

  Z <- tanh(X1 %*% V)
  Y <- cbind(1, Z) %*% W

  output <- unstandardize(Y)
}
```

The useNN function takes the output from the makeNN function and the input attributes of a data set whose target attributes are to be predicted as an argument. The standardize and unstandardize functions as well as the weights for V and W are stored in order to access them quicker. Next the data set is standardized and a column of 1s is added to the very beginning of the standardized matrix. The newly formed matrix is then sent forward through the neural network using the weights pulled out from the list returned from makeNN. Once the output from the neural network is calculated it must be unstandardized in order to see how the data actually relates with the expected data. While the comparison between the predicted results from the neural network and the actual results can be done by standardizing the actual results, I feel that it is better to unstandardize the neural network results. That way the predicted results will be more useful when comparing that with the rest of the data set.

## 2.2 Observation and Discussion

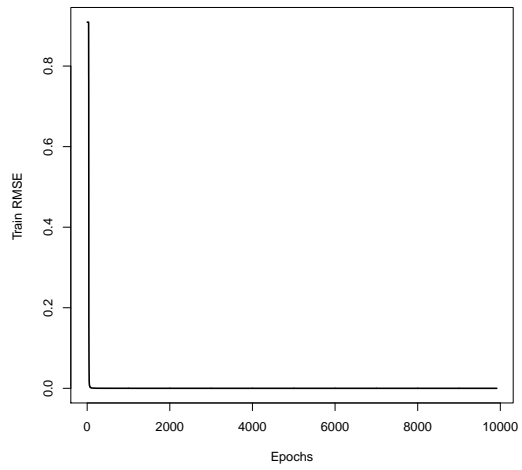
To make sure that the neural network was coded correctly, a very small sample was used to test it. A very small function such as

$$f(x) = x^2 \tag{1}$$

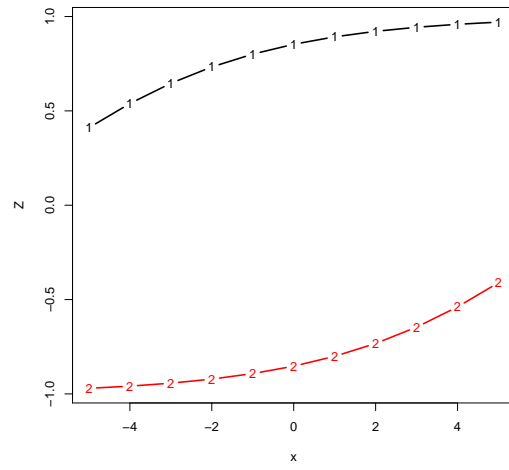
was used on a small set of samples as a way of testing how well the neural network was trained.

Figure 1 displays the information for the neural network set up for the Equation 1. Figure 1(a) displays the root mean square error of the predicted results from the neural network compared with the actual results from the function. Figure 1(b) displays the hidden units used in training the neural network. Figure 1(c) shows a diagram of the neural network and the weight placed upon the different weights to form the neural network. And finally, Figure 1(d) plots the expected results of the neural network against the actual results from the function.

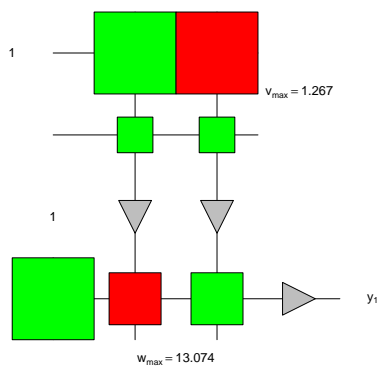
By looking at Figure 1(d), The results from the Equation 1 is fairly close to being exactly right. While the graph depicts a perfect prediction using the neural network, the actual numeric results prove to be a bit different. Compared to the actual results the predicted results were a couple thousandths off from the actual value which is still really close. Figure 1(a) shows how the error decreases over a certain number of iterations of training the neural network. Although it is very difficult to see, the error between the neural network's



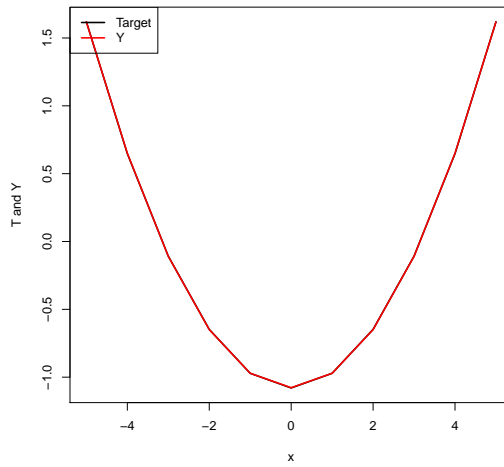
(a) RMSE Error vs. Number of Iterations



(b) Hidden Units



(c) Picture of the neural network



(d) Comparison to Actual Data

Figure 1: Training of neural network for  $f(x) = x * x$

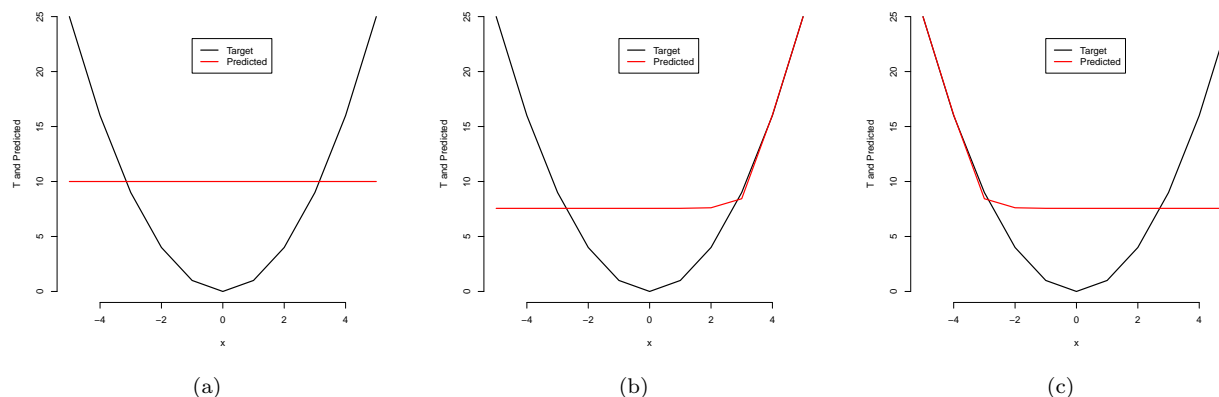


Figure 2: Examples of a bad neural network for a Quadratic function with 1 hidden unit

predictions and the actual results took a few iterations before there was barely any error. However, the error dropped fairly quickly.

Something that I realized from this little test function is that a good number of hidden units is vital for a neural network to do well in prediction. Originally, I trained my neural network on the function

$$f(x) = x \tag{2}$$

with only 1 hidden unit. For a linear function, 1 hidden unit was all that was necessary to come up with a good model. Problems arose when I tried to fit one hidden unit, to a quadratic function. Figure 2 shows a comparison of the actual results of a quadratic function compared to the output of a neural network for the same values of X. Notice that the results are different for Figure 2(a), Figure 2(b), and Figure 2(c) which may be a result of different initializations of the weights. Depending on the initialization of the weights before calling upon the Scaled Conjugate Gradient function, the predictions would match a small piece of it but never the entire figure. Sometimes the predictions for the training data would produce a constant value as in Figure 2(a). Other times, it would start off constant and then begin to follow the curve when  $x = 3$  as in Figure 2(b) or it may follow the curve originally and then become constant as in Figure 2(c). This brings up the question, what is a suitable number of hidden units for a function? This will be explored in the next section.

### 3 Noisy Sine Curve

Before trying to train our neural network to predict real-life data, a neural network will be trained on a noisy sine curve function to see how well it does. As mentioned in previous section, the number of hidden units plays a role in how well a neural network is able to predict a value. This function will allow us to see how the ability for a neural network to predict the target attributes is affected by the number of hidden units.

#### 3.1 10 Hidden Units

Since the noisy sine curve is a much more complicated function than something like Equation 1

##### 3.1.1 Set up

The data set provided for the noisy sine curve consist of 40 values between 0 to 20 and a noisy sine function that produces noise based off of a Gaussian distribution. The function and the code to generate the data was provided by Professor Anderson and is shown below

```
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)

```

A function this complex will more than likely require more hidden units than 1 as explained above. Therefore, for this case we will start off with 10 hidden units. The same steps apply for training a neural network as the steps mentioned above. The `makeNN` function is called to train the neural network with the 10 hidden units being passed to it.

```
nnet <- makeNN(Xtrain, Ttrain, 10)
```

Once finished training, the neural network can be used to predict the target attributes of a new data set.

```
results <- useNN(nnet, Xtest)
```

Results will contain the predicted values of the test data from the neural network. These results can then be compared against the expected results for the test set to see how accurate it was in prediction.

### 3.1.2 Results and Discussion

The results of running the neural network on the noisy sine curve is shown below

Figure 3 is a collection of images that help to show how well the neural network did in representing the noisy sine curve.

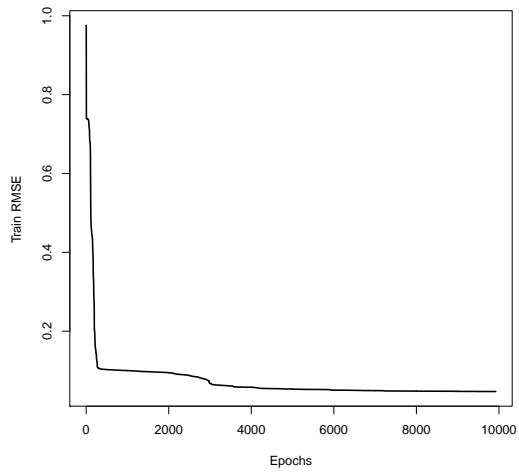
Figure 3(a) shows the progression of the root mean square error over iterations when calculating the best weights for the noisy sine curve prediction. Similar to the Figure 1(a), it did not seem like it took many iterations for it to find a minimum point in RMSE. However, after about 300 iterations, the RMSE dropped even further to close to 0. While gradually decreasing for the rest of the iterations. Keep in mind that the RMSE is calculated only for the training data, and provides no hint as to how well the neural network will do on the test data. Although, a low RMSE is generally a good thing, nothing can be promised when checking the test data.

Figure 3(b) displays how the hidden units responded in regards to the training data. Each hidden unit in this graph depicts how each one individually affects the neural network. If the hidden unit has a negative slope then the weight on that hidden unit is negative, if there is a positive slope, then the weight for that particular hidden unit is positive. If the weight is shifted to either the right or the left of the center of the graph, then the bias weight for that hidden unit is either positive or negative respectively.

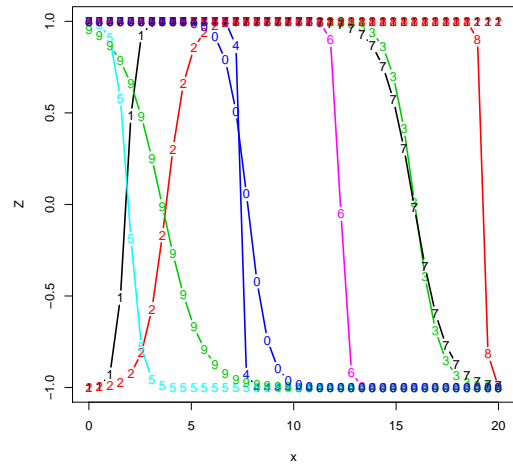
Figure 3(e) shows the weights involved with the hidden and output units in regards to the neural network. The larger the square is, than the larger the magnitude of that particular weight. The color of the block shows whether the weight is negative or positive. This diagram should correlate with the graph in Figure 3(b) in regards to how the weights are shown. For example, hidden unit 8 of Figure 3(e), has a high positive bias weight with a negative weight applied to the hidden unit. In Figure 3(b), hidden unit 8 is shifted far right, indicating a high positive bias weight, and has a negative slope, indicating a negative weight on the actual hidden unit.

Figure 3(c) plots the actual output of the noisy sine curve, the black line, compared with the output from the neural network, the red line, for the training data. The neural network does a pretty decent job in comparison with the actual values for the training data. The first curve is fairly accurate, but things begin to go down hill as X increases. Although close to the expected output, the predicted outputs are not as sharp when it's changing its slope from positive to negative and vice versa. The more sharp points concentrated in an area of the graph for the expected output, the predicted output "cuts corners" and does not get as pointy but is much smoother.

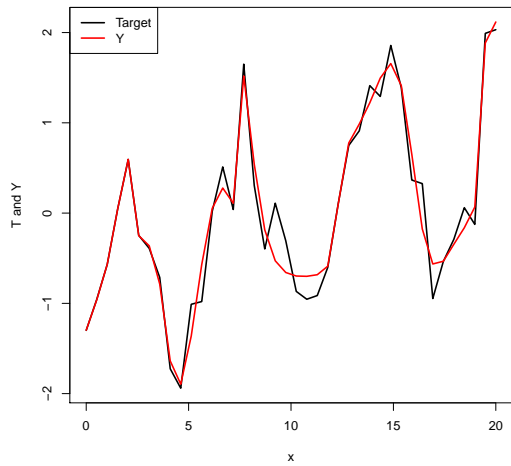
Figure 3(d) plots the actual output of the noisy sine curve verse the predicted output from the neural network for the test data. As can be seen, the prediction does not appear to do a good job at correctly predicting the value the noisy sine curve. The predicted results is much smoother at the local minimums and maximums than the actual output. If there was not as much noise in the actual output the predicted value would not be so far off. The general shape of the curve is fairly accurate. It mainly gets messed up



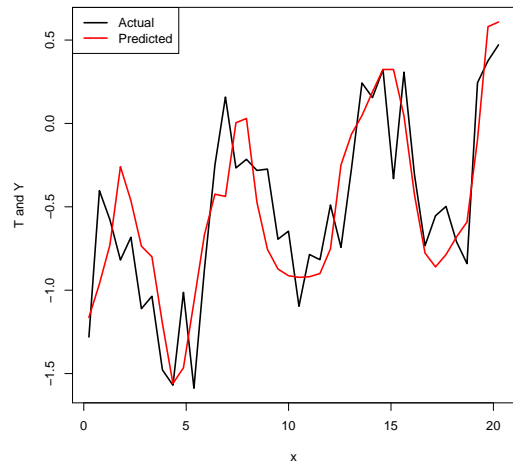
(a) RMSE Error vs. Number of Iterations



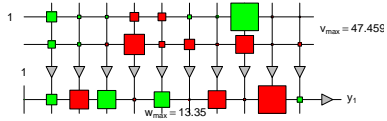
(b) Hidden Units



(c) Comparison to training target data



(d) Comparison to testing target data



(e) Picture of neural network

Figure 3: neural network for  $f(x) = -1 + 0.05 * x + 0.4 * \sin(x) + 0.3 * \text{rnorm}(\text{length}(x))$  with 10 Hidden Units

when it gets real jagged. Although, the neural network was able to match the actual data from the output fairly well, it was far from being accurate for the test data. This makes sense because the neural network should be more accurate in predicting the data used to train it. However, if prediction of the test data is not accurate, then the neural network is not a good predictor. Next, the number of hidden units will be varied to see if that has a better accuracy.

## 3.2 Various Number of Hidden Units

The neural network created using 10 hidden units was pretty accurate for the training data but did not do so well for the actual testing data. This next experiment will look at the accuracy of a neural network created using different numbers of hidden units.

### 3.2.1 Set up

To check how the number of hidden units effects the neural networks ability to predict the output, the same data set will be used. The same steps are used to create and run the neural network. The `makeNN` function creates and trains the neural network according to the training data and the number of hidden units passed to it, and `useNN` uses the neural network output to predict the results of the data set passed to it. Some additional changes have been added to it to ensure that the RMSE that are observed are legitimate and not a result of an outlier or anything of that nature. The code for running these test is shown below

```
testRMSE <- NULL
trainRMSE <- NULL
for (numh in 1:20)
{
  trainForHiddenUnit <- NULL
  testForHiddenUnit <- NULL
  for (trial in 1:5)
  {
    nnet <- makeNN(Xtrain, Ttrain, numh)
    V <- nnet$V
    W <- nnet$W
    resultsTrain <- useNN(nnet, Xtrain)
    trainForHiddenUnit <- rbind(trainForHiddenUnit,
                                sqrt(mean((resultsTrain-Ttrain)^2)))

    resultsTest <- useNN(nnet, Xtest)
    testForHiddenUnit <- rbind(testForHiddenUnit,
                               sqrt(mean((resultsTest-Ttest)^2)))
  }

  trainRMSE <- rbind(trainRMSE, cbind(numh, mean(trainForHiddenUnit)))
  testRMSE <- rbind(testRMSE, cbind(numh, mean(testForHiddenUnit)))
}
```

Rather than creating a neural network once for each Hidden Unit, a neural network is trained 5 times for each hidden unit. Each time the neural network for a particular number of hidden units is run, the training data set RMSE and test data set RMSE are collected and stored. Once all 5 trials are ran the RMSE stored for that specific number of hidden units is the average of all the previously gathered RMSE values.

Once all the different number of hidden units were tested and the test data RMSE and training data RMSE were calculated, Figure 4 was produced using the results.

### 3.2.2 Discussion

Figure 4 is a plot of the RMSE values for both the training and testing data sets in regards to the number of hidden units.

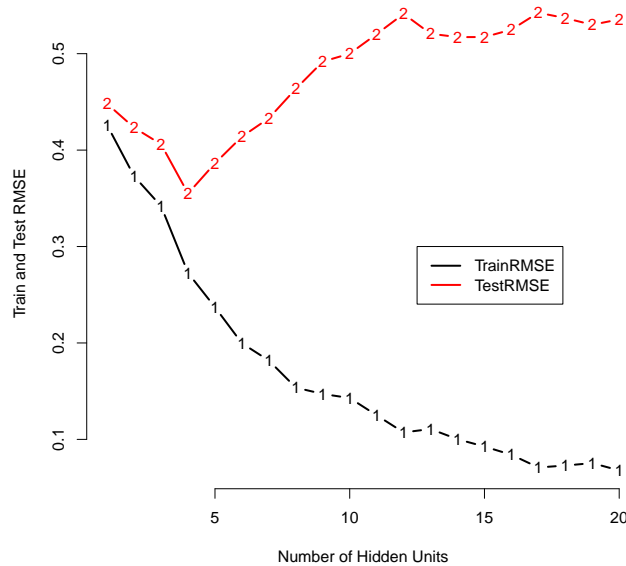


Figure 4: Graph of Training and Test RMSE vs. Number of Hidden Units

Looking only at the training data set RMSE, increasing the number of hidden units improves the accuracy of the neural network by quite a bit. When the number of hidden units hits 14, the RMSE of the training set is less than .1, which is great for a complicated function like the noisy sine curve. Unfortunately the testing data set RMSE does not follow the same trend.

The testing data set RMSE for a small number of hidden units follows a similar trend to that of the training data set RMSE. As the number of hidden units increases the RMSE value decreases. However this changes when the number of hidden units goes beyond 5. Once it is above the 5 the RMSE increases fairly quickly in relation to an increase to the number of hidden units. Eventually it levels off at about 0.5 RMSE. From looking at the testing RMSE it appears that 4 and 5 appear to be the best number of hidden units. This could be a result of the neural networking over fitting the data.

To combat over fitting, many different approaches can be taken such as limiting the number of epochs for the scaled conjugate gradient. This paper, however, will focus on limiting the magnitude of the weight like what was done in Ridge Regression by using a lambda value.

## 4 Automobile Miles Per Gallon Data

Applying the neural network to the Automobile Data, an attempt will be made at accurately predicting real-world data. Using the neural network accurate prediction will attempted to be made using on the miles-per-gallon and horsepower attribute of the data set. Also with this data, the neural networks prediction ability will be tested with various numbers of hidden units and different values from lambda.

### 4.1 Set Up

Before training a neural network with this data set, the data must be modified in order for the training to work correctly. The Automobile Miles Per Gallon data set, provided by the University of California Irvine Machine Learning Repository, must first be read in to R. After being read in, the last column of the data set can be removed. This is because the last column contains the car associated with each sample in the data

set. Since a weight is not going to be placed on the type of car or the manufacturer of the car, that column can be deleted. The code to read in the data set and remove the last column is below

```
dataOrig <- read.table(
  "http://archive.ics.uci.edu/ml/
  machine-learning-databases/auto-mpg/auto-mpg.data")
dataOrig <- dataOrig[,-ncol(dataOrig)] ## remove name and make numeric

testRMSE <- NULL
trainRMSE <- NULL
```

Also, to gather the RMSE based off of the number of hidden units and value of lambda, 2 variables are created which will hold either the training data set RMSE and the testing data set RMSE in regards to the Neural network.

A problem arises after reading in this data set and trying to use mathematical operators on the data. Since the horse power attribute for some samples is unknown, a question mark is placed in that slot. For the neural network to successfully train on the data set, the samples with a question mark in some of the attributes must be removed.

```
mask2 <- apply(as.matrix(dataOrig[,4]), 1, function(ps) all(ps != "?"))
dataOrig <- dataOrig[mask2,]

dataOrig <- apply(dataOrig, 2, as.numeric)
```

In the code above, the samples that do not have a question mark in the horse power column are kept while the samples with a '?' are thrown away. While this removes the ?s from data, the problem of having to apply mathematical operators to the data set occurs because the columns are seen as strings. To solve this problem, each column of the matrix must be coerced into being numeric. Once completed, the neural network can now be trained using the data set.

Since the only way to test our data is based off of a subset of the miles per gallon data, the data must be split into training and testing partitions. The training percent was set to 80 to get a decent training size for the neural network. To ensure that the same training set is not being used every time, the data set must be randomized so that different samples are used for training and testing. Once randomized the columns miles per gallon, column 1, and horse power, column 4, must be removed from the input attributes and into a target attributes matrix. From that point the data can finally be split into training and testing sets. The code for this is shown below

```
trainingFrac <- .80

data <- dataOrig[sample(nrow(dataOrig)),] ## randomly rearrange data
target <- data[,c(1,4)]
data <- data[, -c(1,4)] ## remove target columns from data

Xtrain <- data[1:trainingNum,]
Ttrain <- target[1:trainingNum,]
Xtest <- data[(trainingNum+1):nrow(data),]
Ttest <- target[(trainingNum+1):nrow(target),]
```

The neural network will be created using the same steps mentioned earlier in the paper. makeNN will be run with the training input attributes, training target attributes, a specific number of hidden units, and, also, a value for lambda which is responsible for limiting the magnitude of the weights. Once the predicted outputs are calculated from the neural network, both the target attributes of the test data and the output from the neural network are standardized and then compared with each other to determine how well the neural network did in prediction. Below is an example of code that creates a neural network and run it on the training and testing data and then calculates the root mean square error.

```
nnet <- makeNN(Xtrain, Ttrain, numHidden, lambda=lambdaVal)
```

```

resultsTrain <- useNN(nnet, Xtrain)
sResults <- nnet$standardizeTtrain(resultsTrain)
sResultsAct <- nnet$standardizeTtrain(Ttrain)
temp <- sqrt(mean((sResults - sResultsAct)^2))
trainForHiddenUnit <- rbind(trainForHiddenUnit, temp)

resultsTest <- useNN(nnet, Xtest)
sResultsTest <- nnet$standardizeTtrain(resultsTest)
sResultsActTest <- nnet$standardizeTtrain(Ttest)
tempTest <- sqrt(mean((sResultsTest - sResultsActTest)^2))

```

The creation of the neural network, running the data through the neural network for training and testing, and comparing the outputs from the neural network will be run for different numbers of hidden units and different lambda values. Also, for each pair of number of hidden units and lambda values, the data will be randomized a few times and for each randomization the neural network will be created and tested a few times. Only running the neural network once for each combination of hidden units and lambda values may provide different values than running it another time since the initialization of the weights and the samples that make up the training and testing sets are different. By running it multiple times, it will provide a better idea of how each combination of hidden unit and lambda effects the overall accuracy of the neural network. Once the all the trials have been tested a few times for each of the different randomizations, the average RMSE value for the training and testing data is calculated and is considered the RMSE value for that combo of hidden unit and lambda values.

```

for (numHidden in 5:10)
{
  for (lambdaVal in c(.01, .001, 1, 0, .10))
  {
    cat("HiddenUnits :", numHidden, ", lambdaVal :", lambdaVal, "\n")
    for (randomization in 1:3)
    {
      #Randomize the Data and split into training and testing set
      #Create the neural network
      #Run the neural network on Training and Testing Data
      #Calculate the RMSE for both the training and testing
      #Store the RMSE in matrix for averaging later
    }

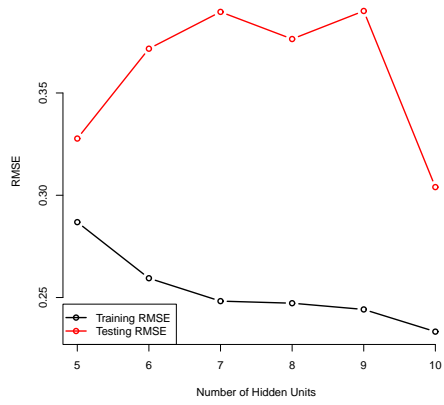
    #Add the average of the RMSE values for the training and testing data to the matrix
    #that hold the RMSE information for different combos of hidden units and lambda values
  }
}

```

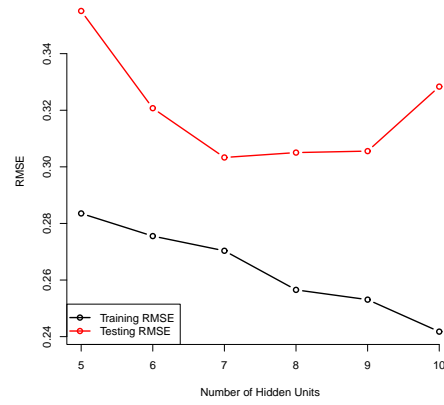
I would have liked to have included trials for each randomization because the neural network will have different outputs depending on the initialization of the weights. However, due to the time constraint, I am not able to run more trials for a new randomized data set. The time it takes for running through all the iterations is fairly long.

## 4.2 Discussion

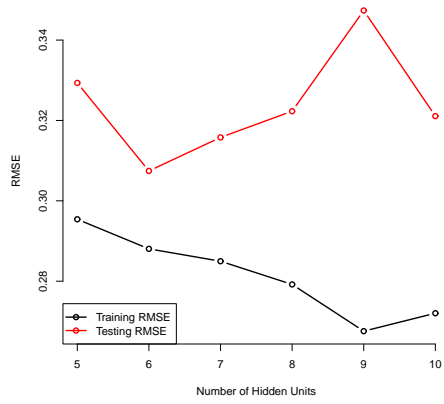
To determine how the combination of hidden units and lambda values effects the neural networks, 5 values for both lambda and hidden units were selected. The number of hidden units test were between 5 and 10 because those number of hidden units appeared to provide a decent balance in minimizing the RMSE for the training data and the RMSE for the testing data. The lambda values were chosen completely at random but were not made too big because the weights should not be limited too much when building the neural network.



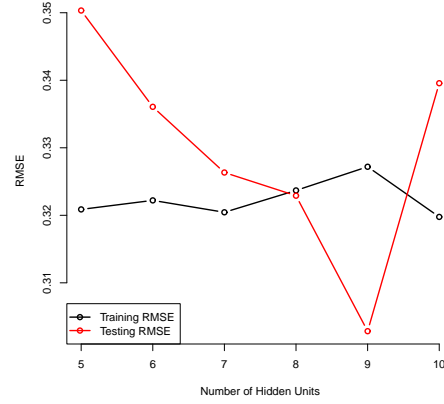
(a) Lambda Value = 0



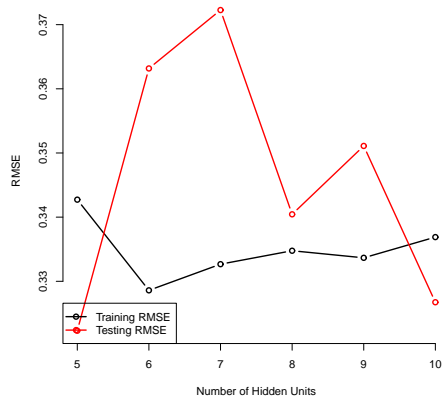
(b) Lambda Value = .001



(c) Lambda Value = .01



(d) Lambda Value = .1



(e) Lambda Value = 1

Figure 5: RMSE for different values of lambda for various hidden units

Figure 5 displays the RMSE values vs. the Number of hidden units for each of the lambda values tested. From looking at the graphs, it does not seem that one combination of lambda and number of hidden units provides the best RMSE for both training and testing. Also, it is hard to pick out a pattern that would correctly describe all the different lambda values tested.

What seemed particularly odd to me is how the testing RMSE was found to be lower than the training RMSE for Figure 5(e) and Figure 5(d). While the testing RMSE is not much lower than the training RMSE (smaller by about .02), it still strikes me as weird because the neural network was trained using the training data. Each lambda value showed some sort of an increase when the number of hidden units increased, however, the increase occurred at different numbers of hidden units depending on the lambda value. Also, the best RMSE for training and testing data occur for 2 different lambdas. The best training RMSE seems to occur for a lambda value of 0 while the best testing RMSE seems to occur for a lambda value of 0.1.

All in all, it did not seem like there is any clear indication of what combination of lambda and number of hidden values will provide the best results. Maybe if more hidden units and more varied lambda values were included in this test, would a better idea on what the best parameters are for this data set. Keep in mind, for another data set, the best parameters could be totally different.

## 5 Conclusion

A lot of factors are involved in creating and training a successful neural network. As with most algorithms for prediction, there is no clear cut answer as to what parameters give the best results for the neural network as the results change depending on what is to be predicted. As mentioned in class, splitting up the data set in to training and testing set may not be enough to fully test how successful a neural network will be. The steps taken above for the Miles Per Gallon data is just a validation RMSE. Therefore the neural network's ability to predict values may not be as successful as was mentioned due to the fact that there is no guarantee that new data will be like the test data.

The hardest aspects for me to get straight for this assignment was adding the lambda value to the gradient error and also determining whether the results from the neural network needed to be standardized or not. When calculating the gradient error on the hidden units, it took me a while to grasp how exactly the lambda value was being included in the equation and how the first row of V had to be removed so that a bias was not affected. I forgot to take out the bias column and as a result the output wasn't what I expected it to be, or I would get an error in the middle of my calculations. When to standardize caused some problems for me as well. Throughout the assignment, I would either forget to standardize or standardize when I was not supposed to in calculating my results. This especially caused a problem when comparing the output from my neural network with the actual target attributes for the Miles Per Gallon data. The first time through, I tested various numbers of hidden units and lambda values but did not standardize any of my results. However, when I attempted to compute the RMSE value, the output was way out of range. I was getting values like 9 and 8. Originally I thought that the values should be back to its original units when calculating the RMSE, however that was not the case. After standardizing both my results and the target attributes of the actual data did my RMSE drop and begin to look the way it was supposed to.

How neural networks work was something I did not know about before taking this class. I heard of them but knew nothing about how the math behind it or the calculations used. This assignment gave me a better understanding of what really goes on behind the scenes of neural networks and how the weights for the input attributes are calculated.

## References

- [1] Anderson, C., *CS545: Artificial Neural Networks*, <http://www.cs.colostate.edu/~anderson/cs545/Lectures/week7day2week7day2.pdf>, 2009.
- [2] Asuncion A. and DJ Newman, *UCI Machine Learning Repository*, [http://www.ics.uci.edu/~sim\\$mllearn/{MLR}epository.html](http://www.ics.uci.edu/~sim$mllearn/{MLR}epository.html), 2007.