

# CS545: Assignment 5

Steve Barriault

October 28, 2009

---

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Simple Neural Networks</b>	<b>1</b>
2.1	Theory . . . . .	1
2.2	Implementation in R . . . . .	2
2.3	A few small examples . . . . .	5
<b>3</b>	<b>A neural network for a noisy sine curve</b>	<b>7</b>
<b>4</b>	<b>The MPG Dataset</b>	<b>13</b>
<b>5</b>	<b>The sigmoid function</b>	<b>19</b>
<b>6</b>	<b>Conclusion</b>	<b>24</b>
<b>7</b>	<b>References</b>	<b>24</b>

---

## 1 Introduction

This is a report on my experimentation using neural networks to predict data. These neural networks were created using a Scale-Conjugate Gradient (SCG) algorithm based on the work of Moller and Nabney on MATLAB and adapted by Anderson for the R platform. [1]

## 2 Simple Neural Networks

### 2.1 Theory

Neural networks are divided into two parts - a forward pass and a feedback pass. The forward pass attempts to make the best predictions possible given what is already known about the data, while the feedback pass enables the weights to be changed so the predictions made by the forward pass can hopefully be more accurate in the future. Both passes' two equations are listed as follows:

*Forward pass*

$$Z = h(\tilde{X}V)$$

$$Y = \tilde{Z}W$$

*Feedback pass*

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

$$W \leftarrow W - \rho_o \frac{1}{N} \frac{1}{K} \tilde{Z}^T (Y - T)$$

Since I am using SCG, there is no need to use any rho constants, and only the gradient part of the feedback pass is required. However, I also needed to implement a weight magnitude penalty to reduce possible overfitting. The updated feedback pass is as follows:

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

$$W \leftarrow \frac{1}{N} \frac{1}{K} \tilde{Z}^T (Y - T) + \lambda W$$

## 2.2 Implementation in R

My code is an adaptation of code that was already provided to us. However, this code needed to be significantly changed to use SCG and to respect the function-like behavior that was required.

First, I built a *makeNN* function that trains a neural network. It accepts as inputs some training data and the actual results of that data as well as the number of hiddenUnits the network will have. Users also have the choice to input a lambda value to prevent overfitting (default is 0), the number of times the network will iterate (default is 10,000) as well as the degree of precision on the data that comes out of the function to calculate the error (*f()*) and the function to produce results (*gradF()*).

```
makeNN <- function( Xtrain , Ttrain , numHiddenUnits ,
                    lambda=0, nIterations=10000, xPrecision=0, fPrecision=0 )
{
  ni = ncol(Xtrain);           #number of input components
  nh = numHiddenUnits;        #number of hidden units
  no = ncol(Ttrain);          #number of output components

  # Helper functions
  pack <- function(V,W)
  {
    matrix(c(V,W))
  }

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

  sqErrorF <- function(weights)
  {
    items <- unpack(weights)
    V <- items$V
    W <- items$W
    Z <- tanh(Xtrain \%*\% V)
    Y <- cbind(1,Z) \%*\% W
    weight <- matrix(V[-1,])

    error <- mean((Y - Ttrain)^2) + (lambda * (t(weight) \%*\% weight))
  }

  gradF <- function(weights)
```

```

{
  items <- unpack(weights)
  V <- items$V
  W <- items$W
  Z <- tanh(Xtrain \%*\% V)
  Y <- cbind(1,Z) \%*\% W
  error <- (Y - Ttrain)
  # 1/N 1/K divider
  divider <- nrow(Xtrain) * ncol(Ttrain)

  gradV <- ((t(Xtrain)/divider) \%*\% (error \%*\% t(W[-1,]) * (1-Z^2)) +
            (lambda * rbind(0,V[-1,])))
  gradW <- ((t(cbind(1,Z))/divider) \%*\% error)
  result <- pack(gradV,gradW)
}

# PART 4
unstandardizeTtrain <- function(X)
{
  nr <- nrow(X)
  nc <- ncol(X-1)
  reversedResults <- ((X * matrix(unSigma, nr, nc, byrow=TRUE)) +
                      matrix(unMu, nr, nc, byrow=TRUE))
}

## STEP 2
# Standardize function for Xtrain
standardizeXtrain <- makeStandardizeF(Xtrain)
Xtrains <- standardizeXtrain(Xtrain)
Xtrain <- Xtrains

#make mu and sigma available to unstandardize
unMu <- colMeans(Ttrain)
unSigma <- sd(Ttrain)

## STEP 3
# Standardize function for Ttrain
standardizeTtrain <- makeStandardizeF(Ttrain)
Ttrains <- standardizeTtrain(Ttrain)
Ttrain <- Ttrains

#make mu and sigma available to unstandardize
unMu <- colMeans(Ttrain)
unSigma <- sd(Ttrain)

## STEP 5
# initial values between -0.01 and 0.01
V <- matrix(0.01*(runif((ni+1)*nh)-0.05), ni+1,nh)
W <- matrix(0.01*(runif((nh+1)*no)-0.05), nh+1,no)

## STEP 6
Xtrain <- cbind(1,Xtrain)

```

```
## STEP 7
```

```
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]]
```

```
## Graphs
```

```
plot(scgResult$ftrace, type="l", lwd=2, xlab="Epochs", ylab="Train RMSE", main="Train RMSE")
drawNNNet(list(W=list(V,W)))
Z <- tanh(Xtrain \%*\% V)
Y <- cbind(1,Z) \%*\% W
matplot(Xtrain[,2], Z, type="b", lwd=2, lty=1, xlab="x", ylab="Z", main="Train - X vs Z")
matplot(Xtrain[,2], cbind(Ttrain, Y), lty=1, type="l", lwd=2, xlab="x",
        ylab="T and Y", main="Train - actual/predicted")
legend("topleft", c("Target", "Y"), lty=1, lwd=2, col=1:2)
```

```
## STEP 8
```

```
return(list(V=V, W=W, standardizeInputs = standardizeXtrain,
           standardizeOutputs = standardizeTtrain,
           unstandardizeOutputs = unstandardizeTtrain,
           lambda = lambda))
}
```

The name of the steps is associated with the steps that were provided to us in the assignment explanations. This code also produces 4 graphs based on the results of the neural network - an evolution through time of the RMSE (based on the number of iteration, or epochs), a neural network graphc describing the relationships and weights associated with different inputs and outputs, a graph explaining the relationships between X and Z, and finally a graph showing the values predicted versus the actual values.

The SCG function attempts to minimize the output of the function *sqErrorF*, which simply returns the square error between target values' actual and predicted values after each forward pass in the neural network. To do so, it injects the existing V and W and then applies these values inside the *gradF* function. That function attempts to refine the prediction using a *tanh* function. Weights V and W are fed to the SCG as a bundle because SCG requires a vector, not a matrix, so I also provided a *pack* and *unpack* functions to provide an easy transition from a matrix structure to a vector structure and then back to a matrix structure.

Finally, the function returns a number of parameters, including the functions used to standardize the data (since SCG works best with standardized data), a function to unstandardize the data for target output, and both the V and W weights that were registered during the last iteration. That data is then used by the function *useNN*, which attempts to make accurate predictions on test data based on the latest V and W values produced by the *makeNN* function.

```
useNN <- function( nnet, Xtest )
{
  ## STEP 1
  Xtests <- nnet$standardizeInputs(Xtest)

  ## STEP 2
  X1 <- cbind(1, Xtests)

  ## STEP 3
  V <- nnet$V
  W <- nnet$W
```

```

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

## STEP 4
results <- nnet$unstandardizeOutputs(Y)
}

```

This latest function is much simpler than the previous one. It simply takes the test values, standardize them using the very same standardization function used in *makeNN* to standardize the training data, and then runs that data through a single forward pass in the neural network, using the latest V and W values available. The data is then unstandardized and returned.

### 2.3 A few small examples

I then proceeded to perform a few tests based on simple values function. The first one was an easy one - I simply had the target value being exactly the same as the input value. Although it was not required, I also ran that neural network against a few test values

The following is the code I used:

```

# Test data generation
f <- function(x) 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)

hiddenUnits <- 10

par(mfcol=c(3,2), bty="n")

myNeuralNetwork <- makeNN(Xtrain, Ttrain, hiddenUnits, lambda =0, nIterations=10000)

error <- myNeuralNetwork$errorTrace

results <- useNN(myNeuralNetwork, Xtest)

matplot(Xtest, cbind(Ttest, results), lty=1, type="l", lwd=2, xlab="x", ylab="T and Y",
         main="Test - actual/predicted")
legend("topleft", c("Target", "Y"), lty=1, lwd=2, col=1:2)
matplot(Xtest, ((Ttest-results)^2), type="l", ylab="RMSE", xlab="values of X",
         main="RMSE test - actual/predicted")

```

My results are displayed in Figure 1. As expected, the neural network had no problem to immediately make excellent predictions, as demonstrated by a RMSE that declined to the vicinity of 0 almost immediately. The predictions (graph at top right) also shows that the neural network was right on the mark. However, when trying to run the test data, it did not do a good job - I believe this is because this relationship is linear, and the slope of the data changed when we generated the test data. Thus, the behavior of the test and training data are not the same.

I also trained a small neural network based on the quadratic equation

$$f(x) = x^2 + 2x + 1$$

by using the following code:

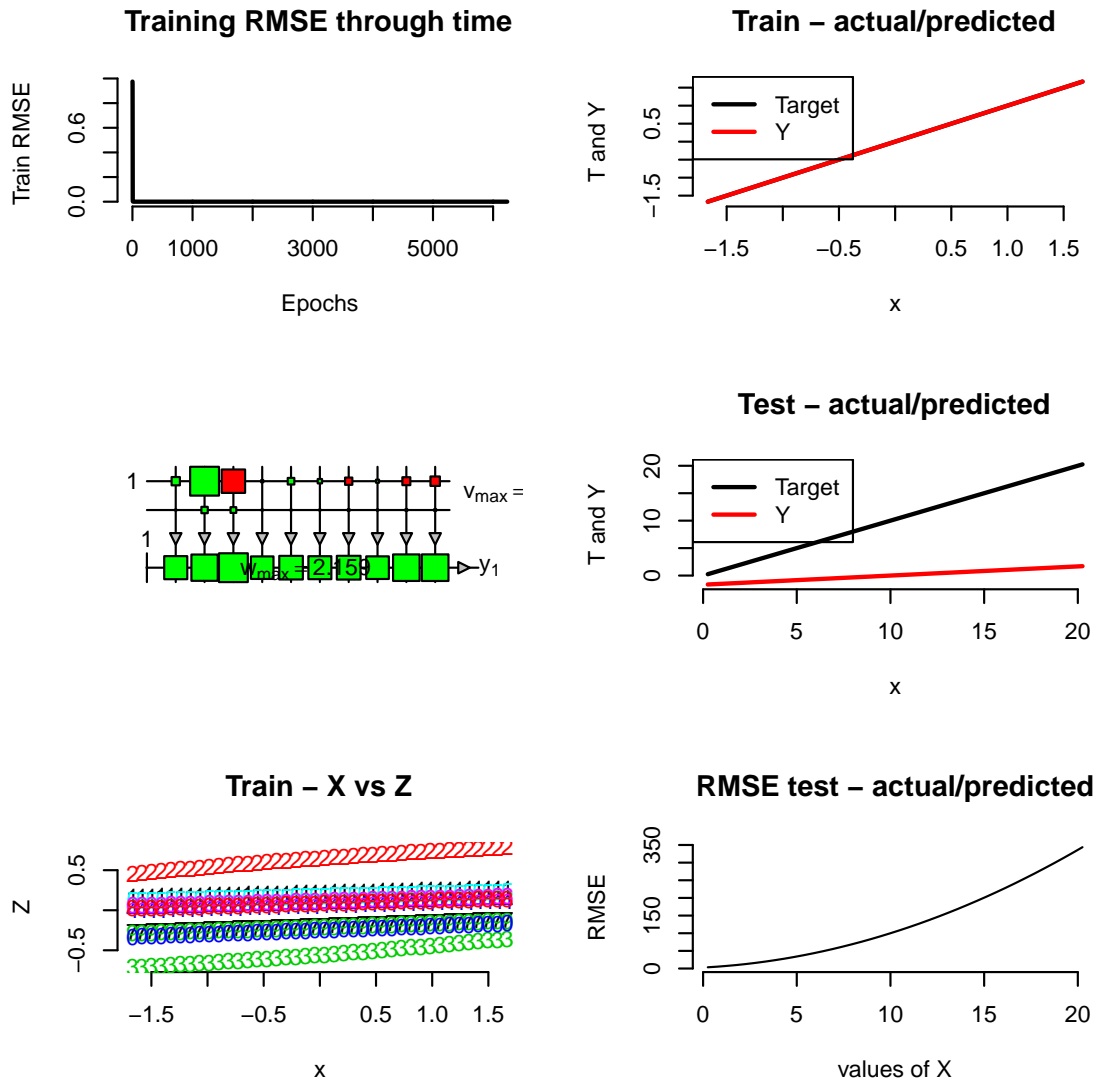


Figure 1: Neural network results based on  $f(x) = x$

```
f <- function(x) (x^2)+(2*x)+1
nSamples <- 40
xmax <- 20
Xtrain <- matrix(seq(0,xmax,length=nSamples),nSamples,1)
Ttrain <- f(Xtrain)
Xtest <- matrix(seq(0,xmax,length=nSamples),nSamples,1)
Ttest <- f(Xtest)
```

Once again, the data converged very quickly, and the predictions done on the training data was on the money. However, test data did not converge. Perhaps this is an example of overfitting - since the data is so simple, our algorithm may simply refine the weights so much that they cannot be used for any other purposes but analyze training data (Figure 2).

### 3 A neural network for a noisy sine curve

I then ran my code against a noisy sine curve. The way to create this sine curve was provided to us in the assignment:

```
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)
```

I also made a few modifications to one of my graphs. First, I saved the untrained data for both Xtrain and Ttrain in XtrainU and TtrainU, respectively, then I unstandardized Y, so as to have a graph showing the fit to the training data on x between 0 to 20, which was required by the assignment:

```
YU <- unstandardizeTtrain(Y)
matplot(XtrainU,cbind(TtrainU,YU),lty=1,type="l",lwd=2,xlab="x",
        ylab="T and Y",main="Train - actual/predicted")
```

The rest of the code remained untouched. I then ran two runs of that noisy sine curve neural networks. I still used 10,000 iterations and no lambda weight penalty. The results are displayed in Figures 3 and 4.

Both runs show very different results, based on the data that was produced. The neural network graph (second center on the left) show different weight and proportionality relationships. But what is really surprising is that the data convergence on both the training and test data is remarkable in the light of the sheer complexity of the original data. There is a clear tendency for the red line (predicted values) to follow the general pattern of the actual target data (in black). This is especially the case for training data. However, the fit to the test data is really not bad, with RMSEs that in general are very low.

What is also remarkable is that while the result on test data is fairly close to the mark, there are spikes in the RMSE spectrum. In other words, it seems that neural networks can do a fair prediction on noisy data most of the time - but when they fail to do an accurate prediction, they fail by a wide margin. That would explain the spiking behavior of the RMSE graph (bottom-right). It is either fairly accurate or not accurate at all, with little to no middle ground.

I also wanted to know if this behavior was fairly constant, or if more refinements to the network could eliminate these spikes. I first tried to run the algorithm for 60,000 iterations. This did not seem to help significantly, as spikes were still present. For that long a wait, it would be better to have some more significant signs of improvement (Figure 6).

I also experimented with smoothing the curve, by using a lambda value of 0.05 in my *makeNN* function. This didn't get rid of the spiking behavior in RMSE values, nor did I thought it would (Figure ??).

As requested, I also experienced with the number of hidden units. To do this, I slightly modified my code following data generation as follows:

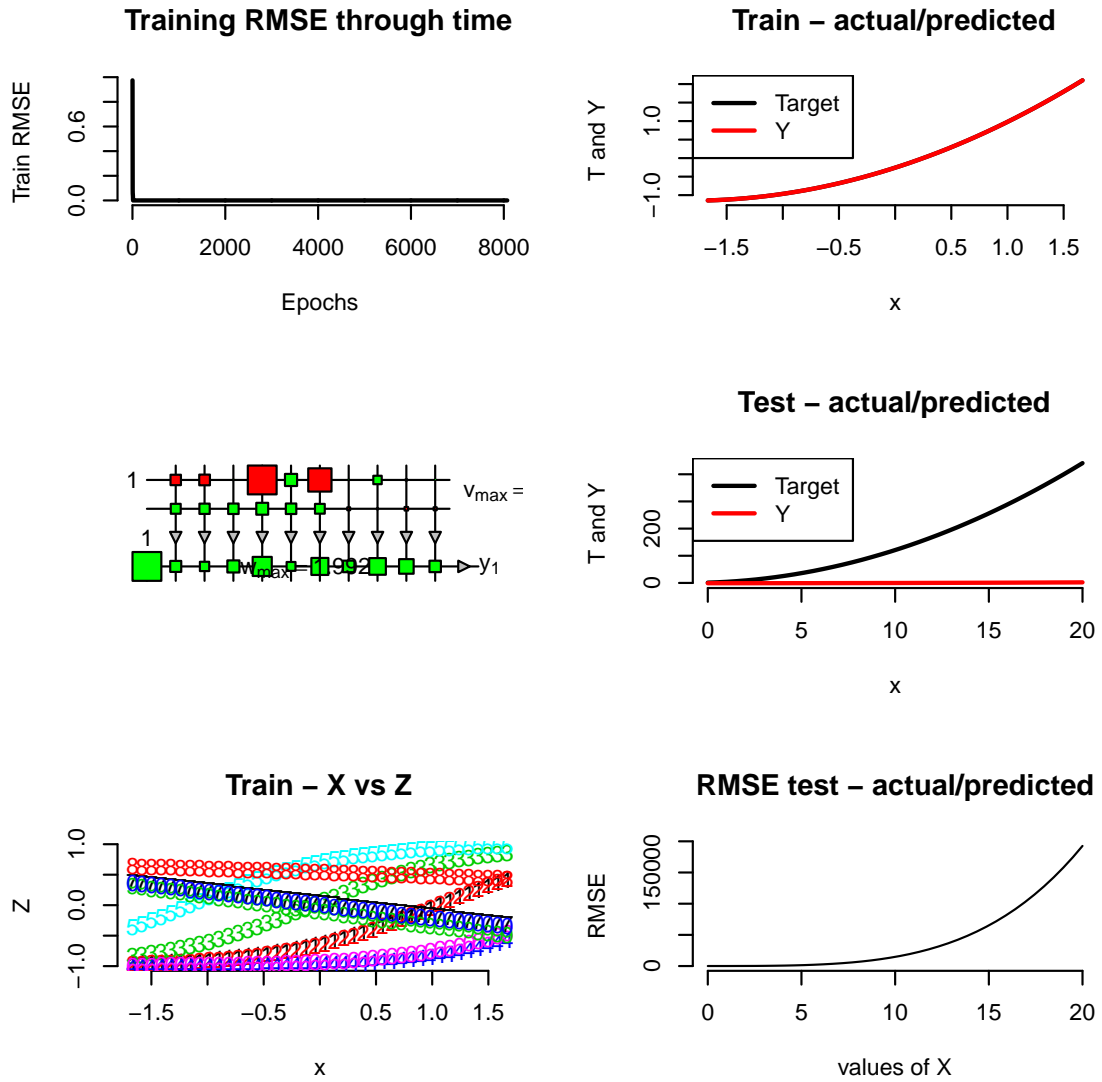


Figure 2: Neural network results based on a simple quadratic equation

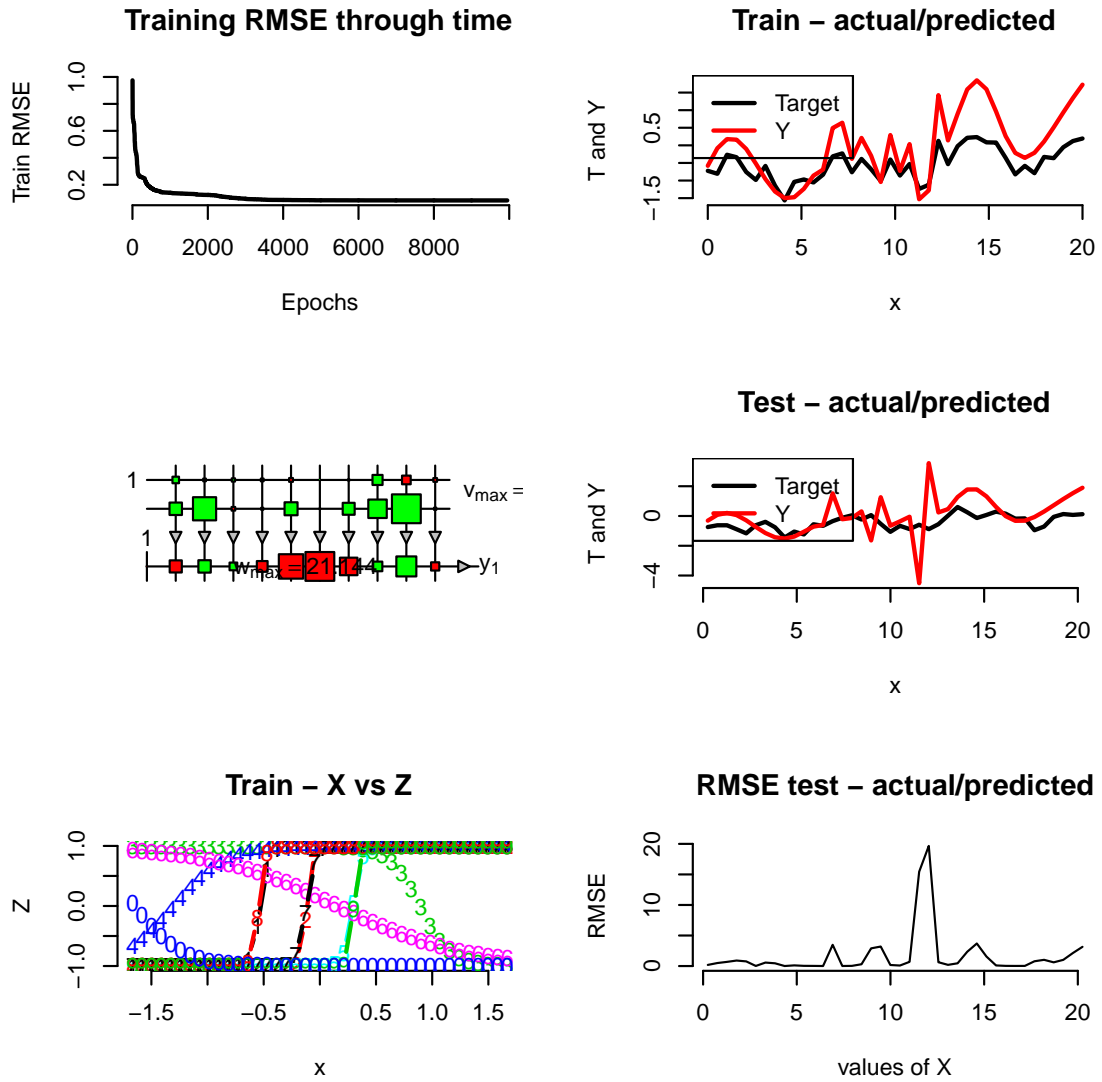


Figure 3: Neural network results based on a noisy sine curve - first run

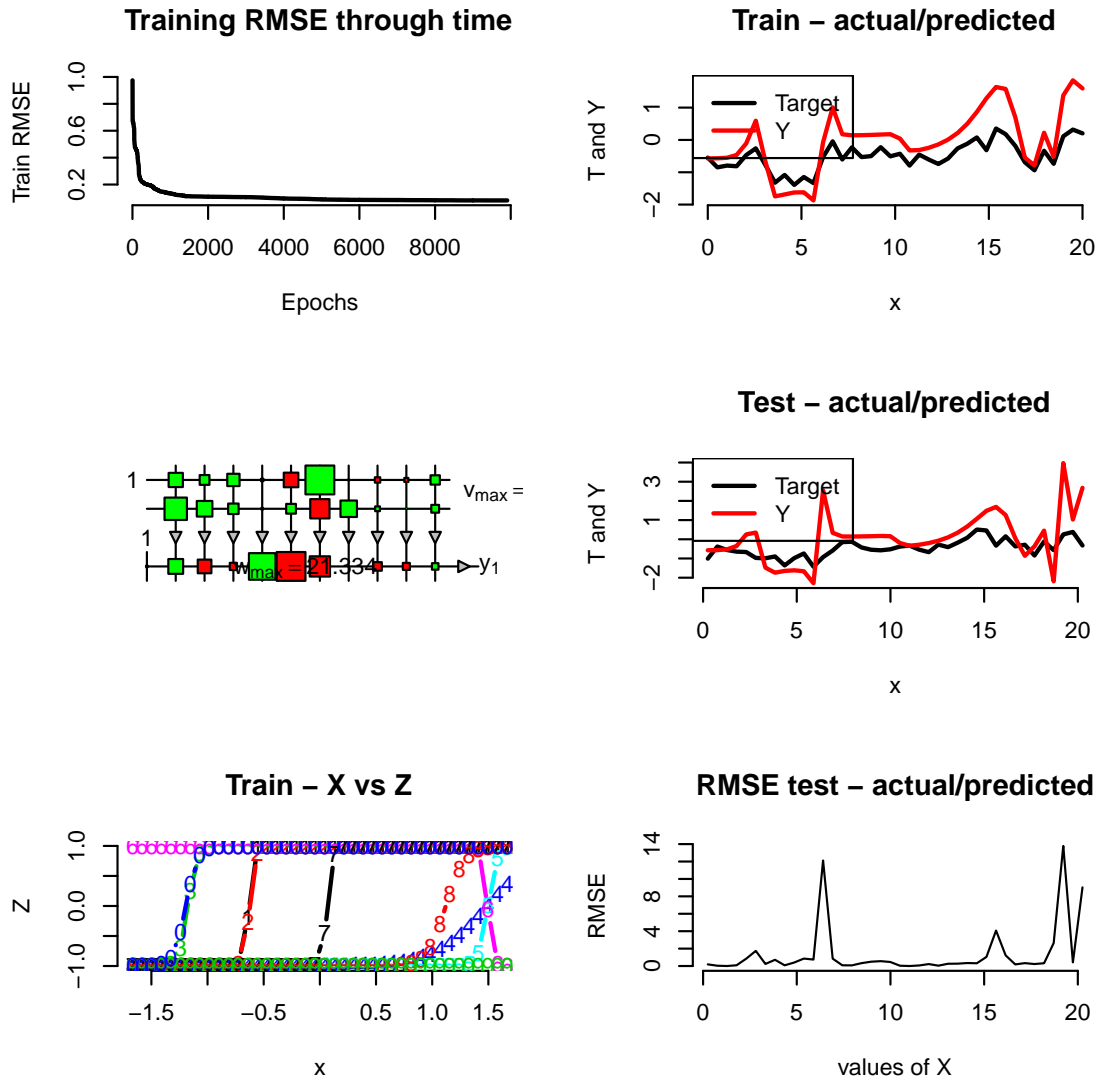


Figure 4: Neural network results based on a noisy sine curve - second run

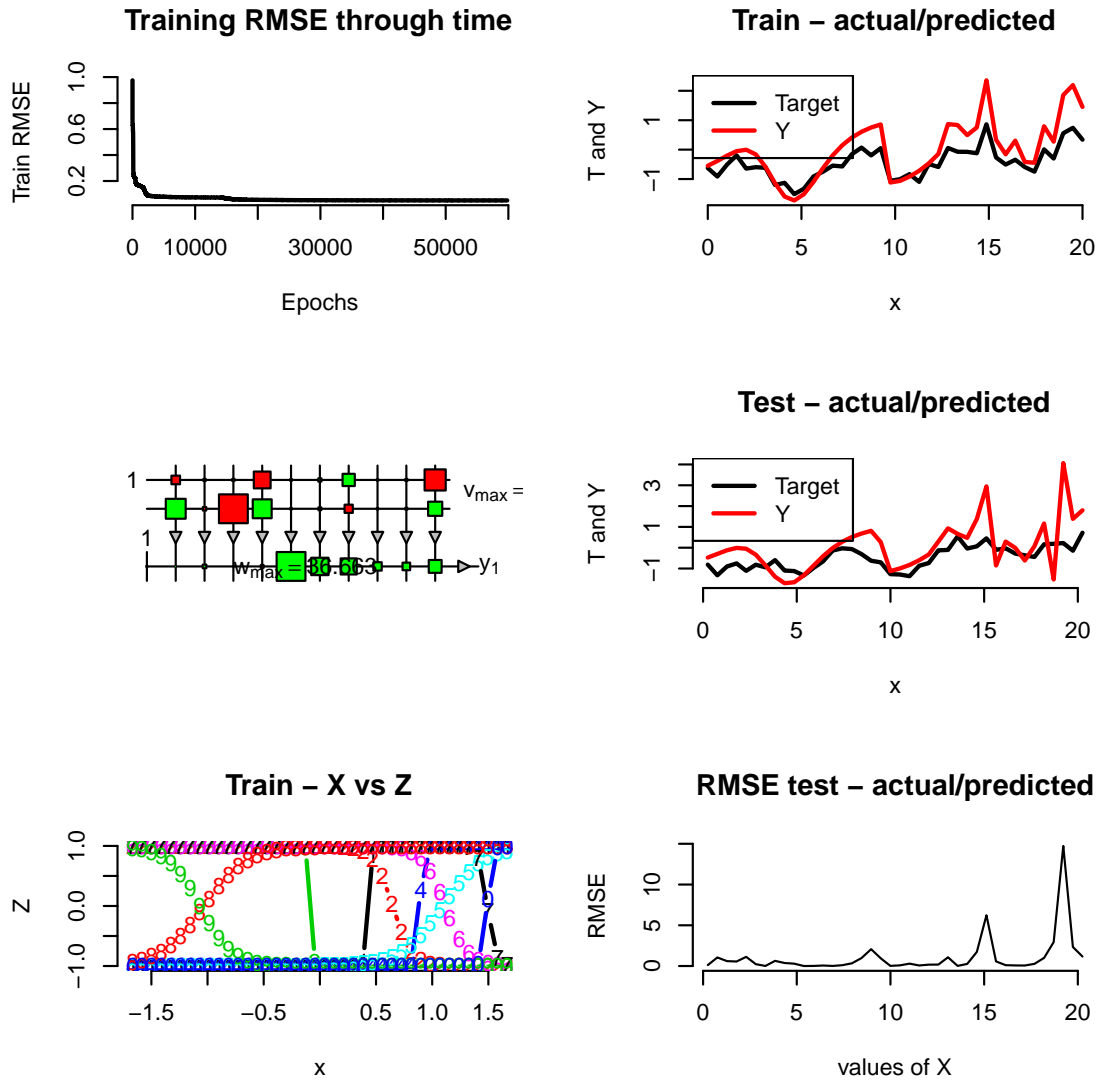


Figure 5: Neural network results based on a noisy sine curve - 60,00 iterations

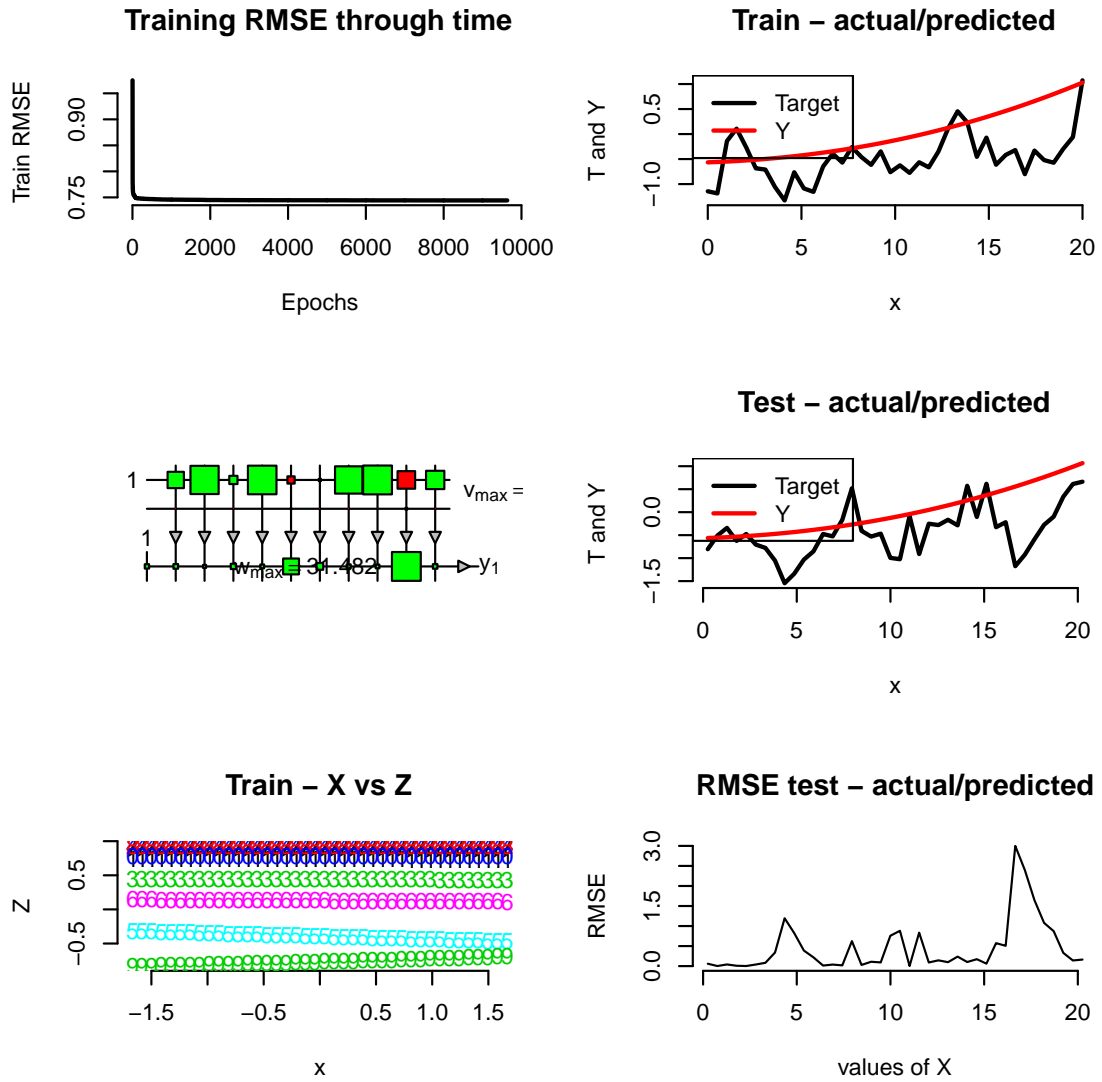


Figure 6: Neural network results based on a noisy sine curve - lambda = 0.05

```

# Test data generation
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)

# Relationship between hidden units and RMSE

RMSE <- c()
test_RMSE <- NULL

for(reps in 1:20)
{
  hiddenUnits <- reps

  par(mfcol=c(3,2), bty="n")

  myNeuralNetwork <- makeNN(Xtrain, Ttrain, hiddenUnits, lambda =0, nIterations=1000)
  results <- useNN(myNeuralNetwork, Xtest)
  matplot(Xtest, cbind(Ttest, results), lty=1, type="l", lwd=2, xlab="x", ylab="T and Y",
    main="Test - actual/predicted")
  legend("topleft", c("Target", "Y"), lty=1, lwd=2, col=1:2)
  matplot(Xtest, ((Ttest-results)^2), type="l", ylab="RMSE", xlab="values of X",
    main="RMSE test - actual/predicted")
  test_RMSE <- mean((Ttest - results)^2)
  RMSE_run <- c(reps, myNeuralNetwork$error, test_RMSE)
  RMSE <- rbind(RMSE, RMSE_run)
}
par(mfcol=c(2,1), bty="n")

matplot(RMSE[,1], RMSE[,2], type="l", ylab="RMSE", xlab="Number of hidden units",
  main="RMSE Train")
matplot(RMSE[,1], RMSE[,3], type="l", ylab="RMSE", xlab="Number of hidden units",
  main="RMSE Test")

```

My main observation is that the more hidden units that is put in the code, the more "bumps" are going to be allowed in the results, and the more fit the prediction can become (at the expense of computation time, of course). See Figures 7 and 8 for more details.

Finally, I plotted the RMSE for different numbers of hidden units, from 1 to 20, for both the training and test data. On the training side, as expected, the more hidden units, the tighter the fit tends to be - although after 12-13 units the RMSE remains relatively stable. On the test side, the picture is a lot more murky, with RMSE being minimized at 6 hidden units, augmenting steadily until 10 hidden units, and then stabilizing after 12 hidden units (Figure 9).

## 4 The MPG Dataset

To put the neural network concept under greater scrutiny, I used the MPG dataset from the URI repository to train a neural network and test it. Based on the R code provided to us in the class, I first removed the names of the vehicles from the data, kept 80% for training and the rest for testing. I treated MPG and horsepower as target values, and used the rest (all six of them) as predictors of MPG and horsepower.

Since this neural network would be generating two outputs, I also modified my code to graph two separate

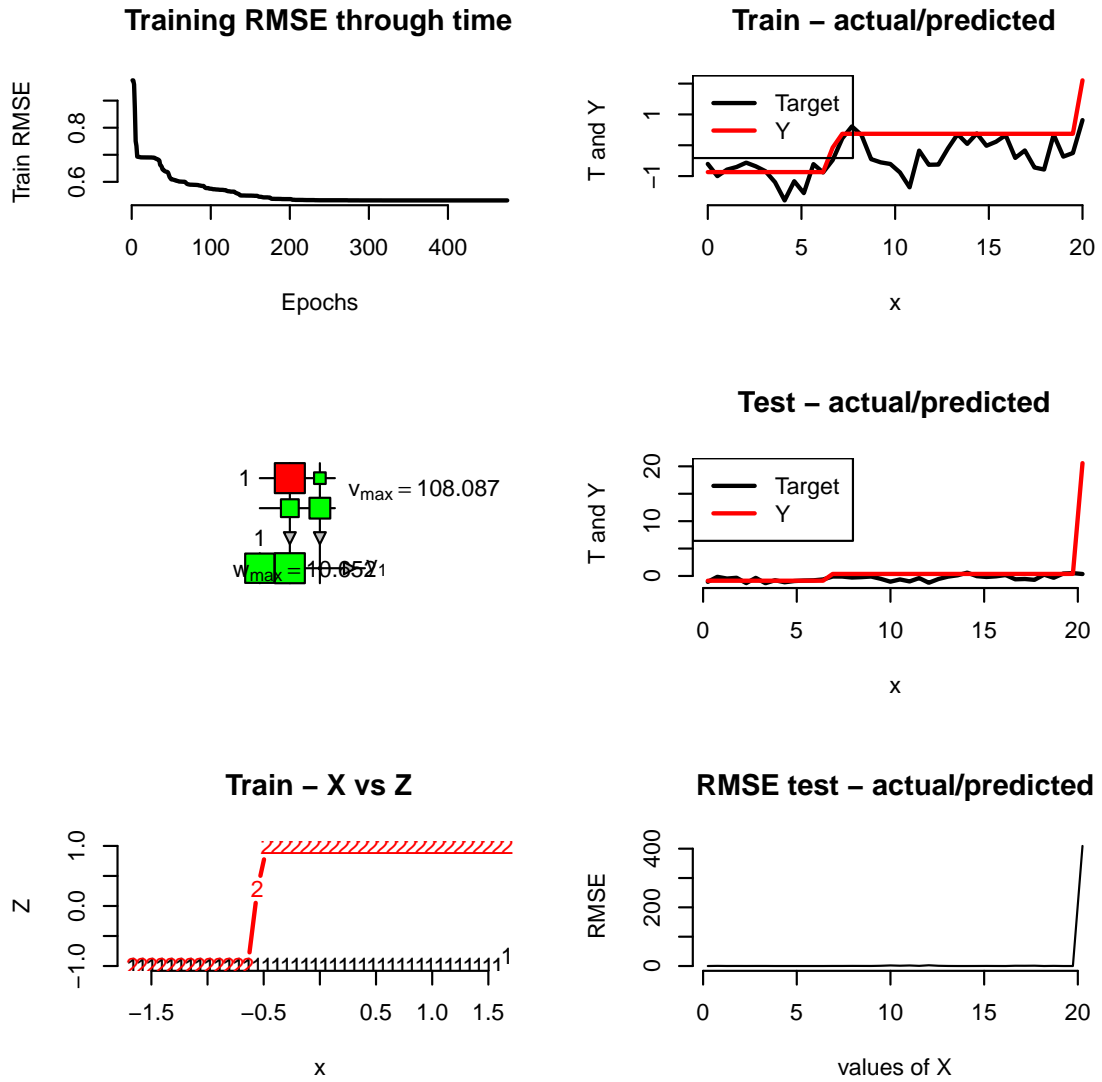


Figure 7: Neural network results based on a noisy sine curve - two hidden units

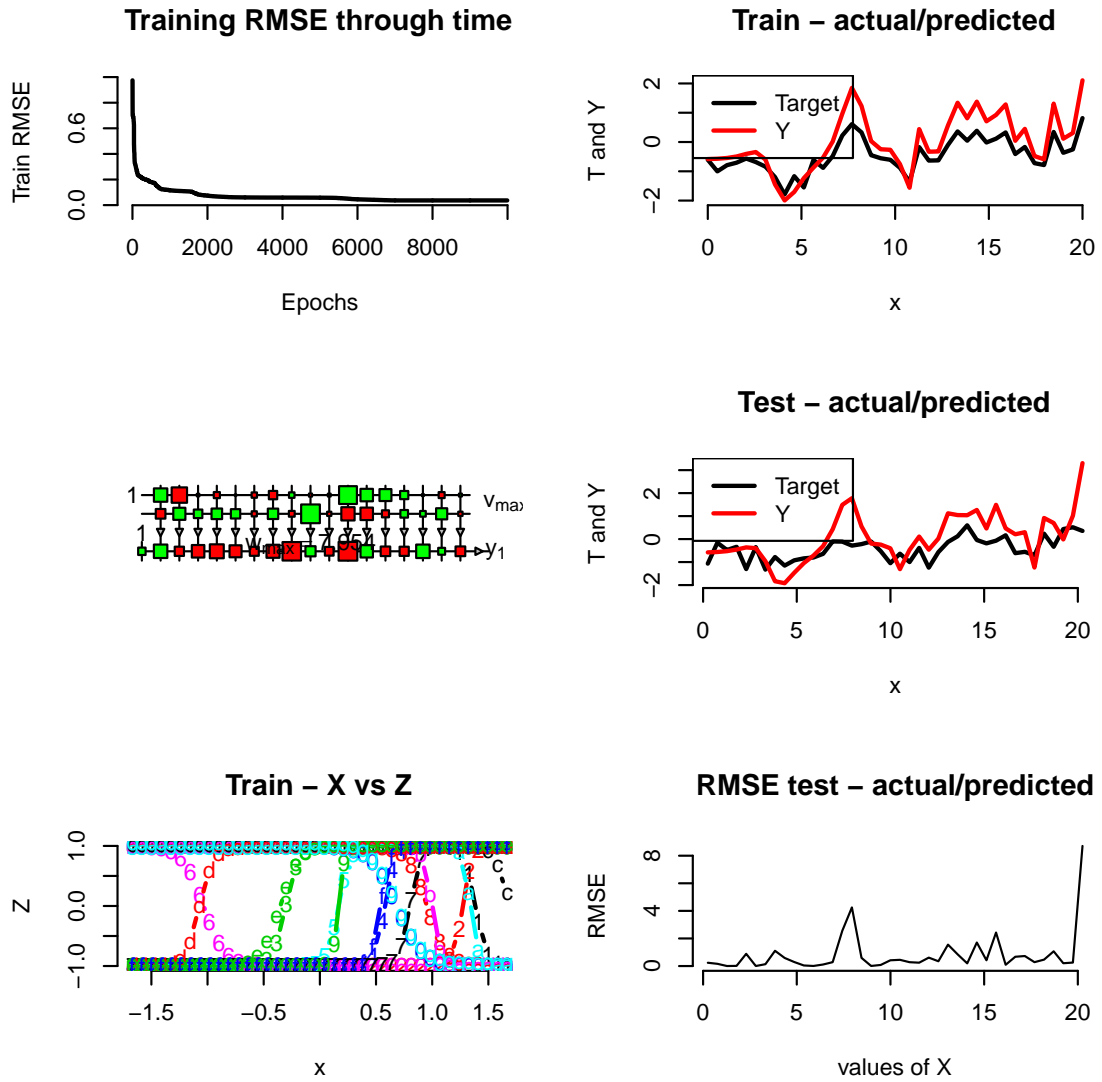


Figure 8: Neural network results based on a noisy sine curve -17 hidden units

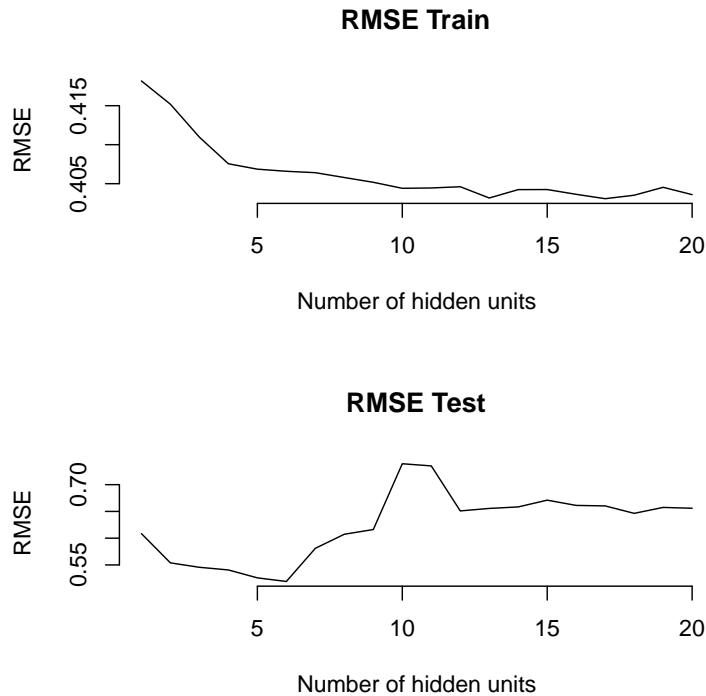


Figure 9: Train and test RMSE for sineCurve for different number of hidden units

sets of graphs, one for each target output, so as to improve readability and relevance. And since the number in each category was of a very different scale, I decided to display standardized inputs and outputs.

The code is as follows:

```
# Test data import
mpg <- read.table("auto-mpg.data")
MPG_names <- c("MPG", "cylinders", "displacement", "horsepower", "weight", "acceleration",
              "model year", "origin", "car name")
colnames(mpg) <- MPG_names

#### Remove all samples that have at least one "?"
keepRows <- apply(mpg != "?", 1, all)
mpg <- mpg[keepRows,]

# Discard vehicle names
mpg <- mpg[, -9]

#### Randomly pick 80\% of samples for training partition
randorder <- sample(nrow(mpg))
nTrain <- round(nrow(mpg)*0.8)
trainRows <- randorder[1:nTrain]

#### Convert all values to numeric
mpg <- apply(mpg, 2, as.numeric)
```

```

### Assemble X and T matrices
Xtrain <- mpg[trainRows, 2:8]
Ttrain <- mpg[trainRows, 1:4, drop=FALSE]
Xtrain <- Xtrain[, -3]
Ttrain <- Ttrain[, -3]
Ttrain <- Ttrain[, -2]

Xtest <- mpg[-trainRows, 2:8]
Ttest <- mpg[-trainRows, 1:4, drop=FALSE]
Xtest <- Xtest[, -3]
Ttest <- Ttest[, -3]
Ttest <- Ttest[, -2]

hiddenUnits <- 10

par(mfcol=c(2,2), bty="n")

print("Entering neural network")
myNeuralNetwork <- makeNN(Xtrain, Ttrain, hiddenUnits, lambda =0, nIterations=10000)
results <- useNN(myNeuralNetwork, Xtest)
Ttest <- myNeuralNetwork$standardizeOutputs(Ttest)
matplot(y=cbind(Ttest[,1], results[,1]), lty=1,type="l",lwd=2,xlab="x",ylab="T and Y",
        main="MPG Test")
matplot(y=cbind(Ttest[,2], results[,2]), lty=1,type="l",lwd=2,xlab="x",ylab="T and Y", main=

### CODE CHANGED WITHIN makeNN
matplot(y=cbind(Ttrain[,1], Y[,1]), lty=1,type="l",lwd=2,xlab="x",
        ylab="T and Y", main="MPG Train")
matplot(y=cbind(Ttrain[,2], Y[,2]), lty=1,type="l",lwd=2,xlab="x",
        ylab="T and Y", main="Horsepower Train")

The results of this code based on lambda =0, 10 hidden units and 10,000 iterations is displayed at Figure
10. As anyone can see, the data tends to be very noisy, yet the neural network does a very good job at
"superimposing" predicted data on top of actual data. This is even more evident in the test graphs (on the
right) - the two lines appear to be following a same general pattern.

Then, I experimented with different values of lambda and number of hidden units. I selected five of them
that appeared interesting to me (no other reason), programmed the following code (10,000 iterations) and
launched the analysis.

TestRMSEMPG <- c()
TestRMSEHorse <- c()
TestRMSEMPG2 <- c()
TestRMSEHorse2 <- c()

hiddenUnits <- c(5,10,15,20,25)
lambdas <- c(0.1, 0.5, 1, 2, 5)

par(mfcol=c(2,2), bty="n")

print("Entering neural network")
for(i in hiddenUnits)
{
    myNeuralNetwork <- makeNN(Xtrain, Ttrain, i, lambda =0, nIterations=10000)
    results <- useNN(myNeuralNetwork, Xtest)
    Ttest <- myNeuralNetwork$standardizeOutputs(Ttest)

```

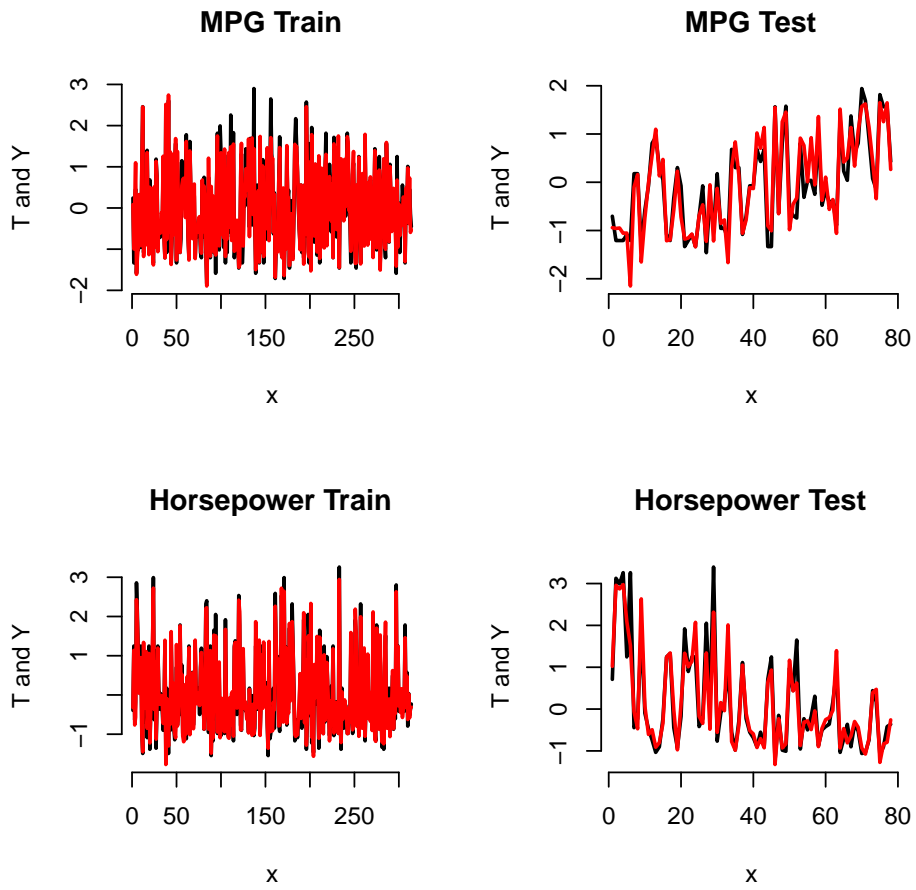


Figure 10: Neural network results for MPG data - actual values in black, predicted in red

```

matplot(y=cbind(Ttest[,1], results[,1]), lty=1, type="l", lwd=2, xlab="x", ylab="T and Y",
          main="MPG Test")
matplot(y=cbind(Ttest[,2], results[,2]), lty=1, type="l", lwd=2, xlab="x", ylab="T and Y",
          main="Horsepower Test")
TestRMSEMPG <- rbind(TestRMSEMPG, mean((Ttest[,1] - results[,1])^2))
TestRMSEHorse <- rbind(TestRMSEHorse, mean((Ttest[,2] - results[,2])^2))
}

for(j in lambdas)
{
  myNeuralNetwork <- makeNN(Xtrain, Ttrain, 10, lambda=j, nIterations=10000)
  results <- useNN(myNeuralNetwork, Xtest)
  Ttest <- myNeuralNetwork$standardizeOutputs(Ttest)
  matplot(y=cbind(Ttest[,1], results[,1]), lty=1, type="l", lwd=2, xlab="x", ylab="T and Y",
            main="MPG Test")
  matplot(y=cbind(Ttest[,2], results[,2]), lty=1, type="l", lwd=2, xlab="x", ylab="T and Y",
            main="Horsepower Test")
  TestRMSEMPG2 <- rbind(TestRMSEMPG2, mean((Ttest[,1] - results[,1])^2))
  TestRMSEHorse2 <- rbind(TestRMSEHorse2, mean((Ttest[,2] - results[,2])^2))
}

par(mfcol=c(2,1), bty="n")
matplot(hiddenUnits, cbind(TestRMSEMPG, TestRMSEHorse), lty=1, type="l", lwd=2,
          xlab="# hidden units", ylab="RMSE", main="Test RMSE & # hidden units")
legend("topleft", c("MPG", "Horse"), lty=1, lwd=2, col=1:2)

matplot(lambdas, cbind(TestRMSEMPG2, TestRMSEHorse2), lty=1, type="l", lwd=2,
          xlab="lambda value", ylab="RMSE", main="Test RMSE & lambda values")
legend("topleft", c("MPG", "Horse"), lty=1, lwd=2, col=1:2)

```

I then retreated to my bed - that would take a while to run!

When I came back to my analysis the morning after, the results I got were surprising to say the least. It seemed that augmenting the number of units by five from 5 to 25 actually /textitincreased RMSE (the line stabilized around 10) for the test data ( 11). This was surprising because I expected the predicted line behavior to become increasingly complex with more hidden units. But, I also reminded myself that in the earlier phases of that project, maximum efficiency was achieved at around five hidden units.

The behavior for lambda values is a real mystery to me. I would have expected the underlying predicted values to become smoother (with less ridges), but overall the RMSE barely budged - negatively in the case of MPG and positively in the case of horsepower. I went back to the code and looked long and hard to find any bug in my code that could explain this behavior, but I couldn't find any. I also remember that in earlier phases, using different values of lambda had an effect, and that the lambda value did not need to be large in order to generate an effect (values as low as 0.05). I hypothesized that for such noisy data any gain you make by reducing spikes is contreracted by increasing RMSE elsewhere - in other words, both the target and predicted lines are sufficiently noisy that lambda effect is limited.

## 5 The sigmoid function

Finally, I tried to use another activation function. Instead of using the tangente *tanh*, I tried to use a sigmoid function. The function is provided below:

$$h(a) = \frac{1}{1 + e^{-a}}$$

In order to do this, I extracted the *tanh* function out of my *makeNN* function, and make it an argument to that same function. I then run both the *tanh* and sigmoid functions against our noisy sine curve.

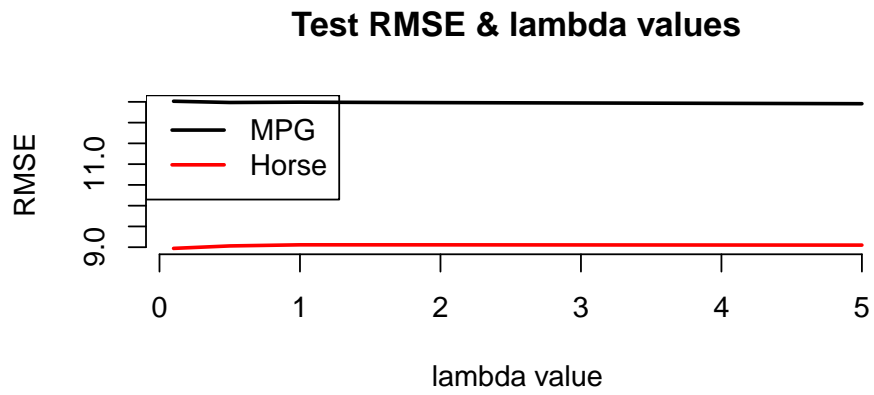
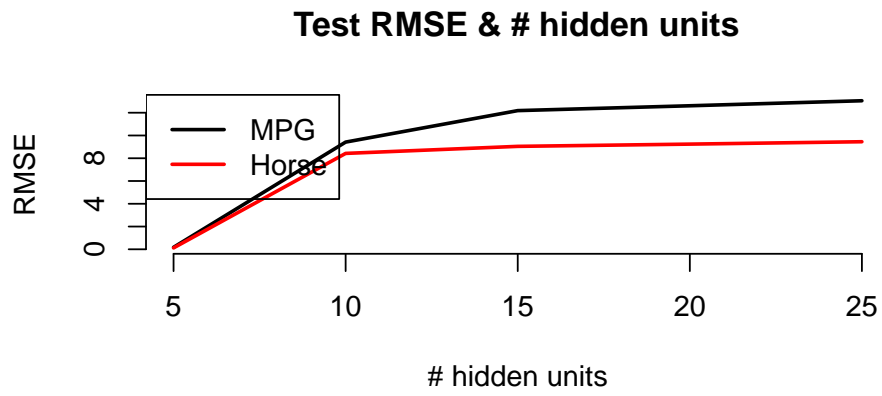


Figure 11: Neural network mean RMSE results for MPG data for different hidden units and lambda values

Changes in the code are as follows:

```
sine <- function(X,V)
{
  tanh(X \%*\% V)
}
```

```
sigmoid <- function(X,V)
{
  (1 / (1 + exp(-(X \%*\% V))))
}
```

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

```
(...)
```

```
  sqErrorF <- function(weights)
  {
    items <- unpack(weights)
    V <- items$V
    W <- items$W
    Z <- nnFunction(Xtrain,V)
    Y <- cbind(1,Z) \%*\% W
    weight <- matrix(V[-1,])

    error <- mean((Y - Ttrain)^2) + (lambda * (t(weight) \%*\% weight))
  }
```

```
  gradF <- function(weights)
  {
    items <- unpack(weights)
    V <- items$V
    W <- items$W
    Z <- nnFunction(Xtrain,V)
    Y <- cbind(1,Z) \%*\% W
    error <- (Y - Ttrain)
    # 1/N 1/K divider
    divider <- nrow(Xtrain) * ncol(Ttrain)

    gradV <- ((t(Xtrain)/divider) \%*\% (error \%*\% t(W[-1,]) * (1-Z^2)) + (la
      rbind(0,V[-1,])))
    gradW <- ((t(cbind(1,Z))/divider) \%*\% error)
    result <- pack(gradV,gradW)
  }
```

```
(...)
```

```
## Graphs
```

```
  Z <- nnFunction(Xtrain,V)
  Y <- cbind(1,Z) \%*\% W
  YU <- unstandardizeTtrain(Y)
```

```

error <- mean((TtrainU - YU)^2)
matplot(XtrainU, cbind(TtrainU, YU), lty=1, type="l", lwd=2, xlab="x",
        ylab="T and Y", main="Train - actual/predicted")
legend("topleft", c("Target", "Y"), lty=1, lwd=2, col=1:2)

```

(...) END OF makeNN

I also modified my *useNN* function accordingly:

```

useNN <- function( nnet, Xtest, nnFunction )
{
  ## STEP 1
  Xtests <- nnet$standardizeInputs(Xtest)

  ## STEP 2
  X1 <- cbind(1, Xtests)

  ## STEP 3
  V <- nnet$V
  W <- nnet$W
  Z <- nnFunction(X1, V)
  Y <- cbind(1, Z) \%*\% W

  ## STEP 4
  results <- nnet$unstandardizeOutputs(Y)
}

```

And, finally, the neural network generation calls:

```

myNeuralNetwork <- makeNN(Xtrain, Ttrain, hiddenUnits, sine, lambda =0, nIterations=10000)
results <- useNN(myNeuralNetwork, Xtest, sine)

```

```

matplot(Xtest, cbind(Ttest, results), lty=1, type="l", lwd=2, xlab="x", ylab="T and Y",
        main="Test - actual/predicted")
legend("topleft", c("Target", "Y"), lty=1, lwd=2, col=1:2)
matplot(Xtest, ((Ttest-results)^2), type="l", ylab="RMSE", xlab="values of X",
        main="RMSE test - actual/predicted")

```

```

myNeuralNetwork2 <- makeNN(Xtrain, Ttrain, hiddenUnits, sigmoid, lambda =0,
        nIterations=10000)
results2 <- useNN(myNeuralNetwork, Xtest, sigmoid)

```

```

matplot(Xtest, cbind(Ttest, results2), lty=1, type="l", lwd=2, xlab="x", ylab="T and Y",
        main="Test - actual/predicted")
legend("topleft", c("Target", "Y"), lty=1, lwd=2, col=1:2)
matplot(Xtest, ((Ttest-results2)^2), type="l", ylab="RMSE", xlab="values of X",
        main="RMSE test - actual/predicted")

```

The results of this investigation are displayed in Figure 12. The results are surprising - overall, *tanh* did a lot better than the sigmoid function. The sigmoid function produced a far smoother curve, which might indicate it could be advisable to use the sigmoid function on data that is not too noisy (as our sine curve is). The performance on both the test data and RMSE indicates it is definitely not a good candidate to analyze the sine curve function.

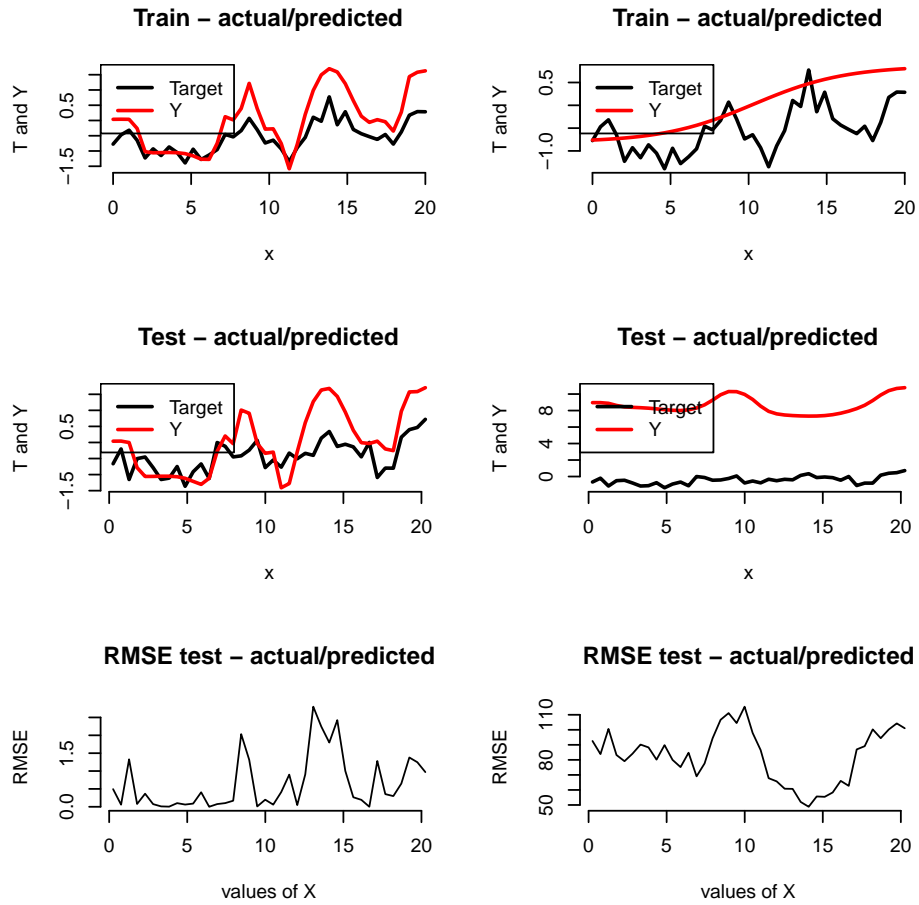


Figure 12: Neural network results -  $\tanh$  (left) vs  $\text{sigmoid}$  (right)

## 6 Conclusion

The goal of this assignment was to learn about neural networks by implementing a few of them. Overall, I was very impressed by their performance - predicted values can be made to follow the general pattern of even very noisy data sets, such as the sine curve and the MPG data. On the other hand, when a neural network prediction is wrong, it can be way off, which explains the spiking behavior of my RMSE curves... and why I sometimes receive calls from my credit card company indicating possible unauthorized usage even if I did not change my consumption pattern a bit!

Just like in the case of Assignment 3, I would like to thank the Professor Charles Anderson for his invaluable help troubleshooting my code - that did help me better understand and enabled me to complete the assignment.

## 7 References

### References

- [1] M. F. Moller, Scaled Conjugate Gradient algorithm from "A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning", *Neural Networks* vol. 6, pp. 525-533, 1993, Adapted by Chuck Anderson from the Matlab implementation by Nabney as part of the netlab library.