CS545: Assignment 5

Michael Yoensky

October 29, 2009

Contents

1 Introduction **1**

2 Creating the Neural Network **2**
2.1 Writing `makeNN` and `useNN` 2
2.2 Testing the Neural Network 4

3 Modeling the Noisy Sine Curve **5**
3.1 Training the Neural Network and Results 5
3.2 Effect of Hidden Values on RMSE 8

4 Modeling the Miles Per Gallon Data **10**
4.1 Training the Neural Network and Results 10
4.2 Effect of λ and Hidden Values on RMSE 14

5 Extra Credit **14**

6 Discussion **18**

7 Conclusions **21**

1 Introduction

This assignment was primarily intended to introduce the class to neural networks. Neural networks are a method of creating a network of artificial neurons and using the resulting network to obtain different results. The only thing that neural networks have to relate them to actual neurons are the fact that several different pieces are summed together to make a decision [1].

The assignment consisted of three main parts [2]. The first part was to generate functions to train a new neural network and a function to use that network. The second part was to use the functions written in part one to analyze a very noisy sine curve. The final part was to use the neural network to analyze some miles per gallon data provided by the machine learning repository.

As extra credit additional investigation of the built-in R function `optim` were also performed. This function implements a conjugate gradient algorithm. The final part will investigate a new algorithm implemented by `optim` that is not the same as others used for this assignment.

2 Creating the Neural Network

2.1 Writing makeNN and useNN

To write the makeNNand useNN code two functions were written to calculate the Mean Squared Error. This was written using tanh as the hidden function and used the two sets of weights V and W. The function to calculate the Mean Squared Error was called sqErrorF.

In addition to the Mean Squared Error, a function to calculate the gradient with respect to the weights V and W was also written. This function is called gradF. The gradF function will compute the current gradient considering the test data and weights.

Both the functions makeNN and useNN use matrix operations for efficiency. The makeNN code also use the SCG algorithm provided by Professor Anderson for completion of the previous assignment. The code for both makeNN and useNN can be found below.

```
##################################################
# makeNN
# Author: Michael Yoensky
#
# Inputs:
#     Xtrain - the training X values
#     Ttrain - The training Y values
#     numHiddenUnits - The number of hidden units
#     lambda - The weight decay (optional)
#     nIterations - Specify the number of iterations (optional)
#     xPrecision - Specify a limiting condition
#     fPrecision - Specify a limiting condition
makeNN <- function(Xtrain,
                   Ttrain,
                   numHiddenUnits,
                   lambda = 0,
                   nIterations = 10000,
                   xPrecision = 0,
                   fPrecision = 0)
{



   standardizeF <- makeStandardizeF(Xtrain)
   XtrainS <- standardizeF(Xtrain)

   standardizeFTtrain <- makeStandardizeF(Ttrain)
   unstandardizeFTtrain <- makeUnstandardizeF(Ttrain)
   TtrainS <- standardizeFTtrain(Ttrain)

   X <- XtrainS
   T <- TtrainS

   nh <- numHiddenUnits
   ni <- ncol(X)
   no <- ncol(T)
   V <- matrix((runif((ni+1)*nh, min=-0.01, max=0.01)), ni+1,nh)
   W <- matrix((runif((nh+1)*no, min=-0.01, max=0.01)), nh+1,no)

   X1 <- cbind(1,X)
   errorTrace <- NULL
```

```
N <- nrow(X)
K <- ncol(T)

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)
{
    unpacked_weights <- unpack(weights)
    Vtemp <- unpacked_weights$V
    Wtemp <- unpacked_weights$W

    Z <- tanh(X1 %*% Vtemp)
    Y <- cbind(1,Z) %*% Wtemp
    error <- Y - T
    mean(error^2) + (lambda * sum(t(Vtemp[-1,]) %*% Vtemp[-1,]))
}

gradF <- function(weights)
{
    unpacked_weights <- unpack(weights)
        Vtemp <- unpacked_weights$V
    Wtemp <- unpacked_weights$W

        Z <- tanh(X1 %*% Vtemp)
    Y <- cbind(1,Z) %*% Wtemp
    error <- Y - T
    Vtemp <- (1/N) * (1/K) * (t(X1) %*%
                 (error %*% t(Wtemp[-1,,drop=FALSE]) *
                 (1-Z^2))) +
                 (rbind(0,Vtemp[-1,,drop=FALSE]) * lambda)
    Wtemp <- (1/N) * (1/K) * t(cbind(1,Z)) %*% error
    pack(Vtemp,Wtemp)
}

scgResult <- scg( pack(V,W), sqErrorF, gradF,
                  nIterations=nIterations,
                  xPrecision=xPrecision,
                  fPrecision=fPrecision,
                  xtracep=FALSE, ftracep=TRUE)
VW <- unpack(scgResult$x)
V <- VW[[1]]
W <- VW[[2]]
```

```
    list(V=V,W=W, standardizeF=standardizeF,
         standardizeFTtrain=standardizeFTtrain,
                 unstandardizeFTtrain=unstandardizeFTtrain,
                 lambda=lambda, scgResult=scgResult)

}


###################################################
# useNN
# Author: Michael Yoensky
#
# Inputs:
#     nnet  - the neural network to use
#     Xtest - the test data to use
useNN <- function(nnet, Xtest)
{
    # Standardize Xtest
    XtestS <- nnet$standardizeF(Xtest)

    # Add constant 1
    XtestS1 <- cbind(1,XtestS)

    # Calculate output of neural network
    Z <- tanh(XtestS1 %*% nnet$V)
    Y <- cbind(1,Z) %*% nnet$W

    # Return the unstandardized results
    nnet$unstandardizeFTtrain(Y)
}
```

2.2 Testing the Neural Network

The neural network created using `makeNN` and was tested with three simple functions with `useNN`. The three functions were $f(x) = x$, $f(x) = x + noise$, $f(x) = -(x-5)^2 + 10 + noise$, and finally $f(x) = cos(x) + noise$. These three functions were then graphed to show the results of the test and training data. The code used to test the neural networks is below, and the resulting graphs are in Figure 1.

In Figure 1 it is observed that in all four cases, the test data is very close to the actual data. This shows that the neural network code is working as expected, and can be used for other functions and machine learning exercises.

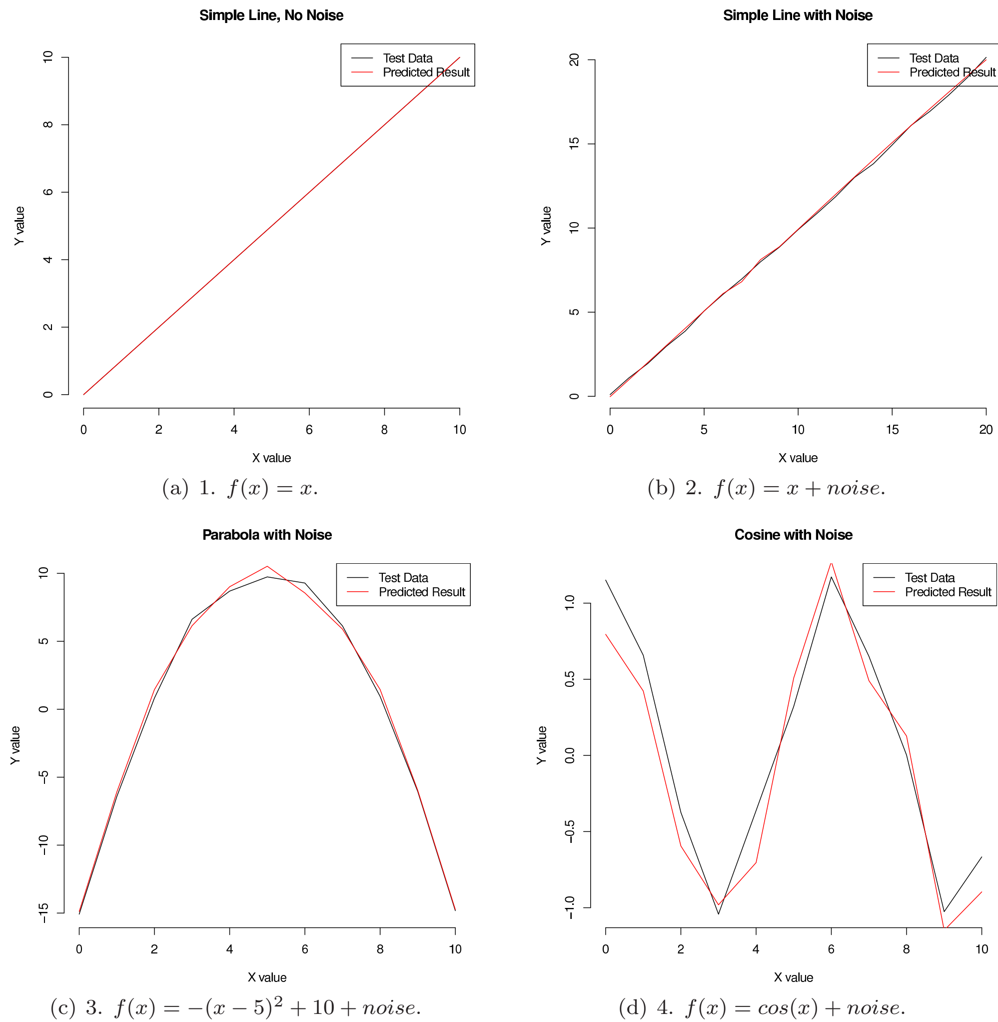
```
#####################################################################
# This is the simple data to check the Neural Network code
#####################################################################

checkNN <- function(f, Xtrain, nh=5, nReps=200000,
                    filename="outputFile.png",
                    description="X and Y Values with Neural Network")
{
  # Generate the data
  N <- nrow(Xtrain)
  xmax <- 20
  Ttrain <- f(Xtrain)
  Xtest <- Xtrain
```

```
  Ttest <- f(Xtest)

  # Train the neural network
  nnet <- makeNN(Xtrain,Ttrain,nh,nIterations=nReps)

  # Use the neural network
  testResult <- useNN(nnet,Xtest)


  pdf(filename)
  ptemp <- par(bty="n")

  # plot the data
  legendTags <- c("Test Data","Predicted Result")
  legendColors <- c("black","red")
  matplot(Xtest,Ttest,pch=1,lty=1,xlab="X value",ylab="Y value",
          main=description,type="l",
          col=legendColors[1])
  matplot(Xtest,testResult,pch=1,lty=1,add=TRUE,type="l",
          col=legendColors[2])
  legend(x="topright",legend = legendTags,col=legendColors,lty=1)

  # turn off the graphs and restore the defaults
  dummy <- dev.off()
  par(ptemp)
}

# Example 1, Use a simple line, no noise
f<-function(x) {x}
checkNN(f,matrix(seq(0,10)),filename="simple_ex1.pdf",
        description="Simple Line, No Noise")

# Example 2, Use a simple line
f<-function(x) {x+0.1*rnorm(length(x))}
checkNN(f,matrix(seq(0,20)),filename="simple_ex2.pdf",
        description="Simple Line with Noise")

# Example 3, Basic Parabola
f<-function(x) {-(x-5)^2+10+0.3*rnorm(length(x))}
checkNN(f,matrix(seq(0,10)),filename="simple_ex3.pdf",
        description="Parabola with Noise")

# Exame 4, Cosine
f<-function(x) {cos(x)+0.2*rnorm(length(x))}
checkNN(f,matrix(seq(0,10)),filename="simple_ex4.pdf",
        description="Cosine with Noise")
```

3 Modeling the Noisy Sine Curve

3.1 Training the Neural Network and Results

A function was given to generate a noisy sine curve for further testing of the neural network implementation. This function was used to generate a list of test and training data. The data was tested using the `makeNN`



(a) 1. $f(x) = x$.

(b) 2. $f(x) = x + noise$.

(c) 3. $f(x) = -(x-5)^2 + 10 + noise$.

(d) 4. $f(x) = cos(x) + noise$.

Figure 1: These are the graphs for Simple Data.

to train a neural network, and then executed on test data using the `useNN` function. The code used for this test is listed below along with the resulting graphs are in Figure 2.

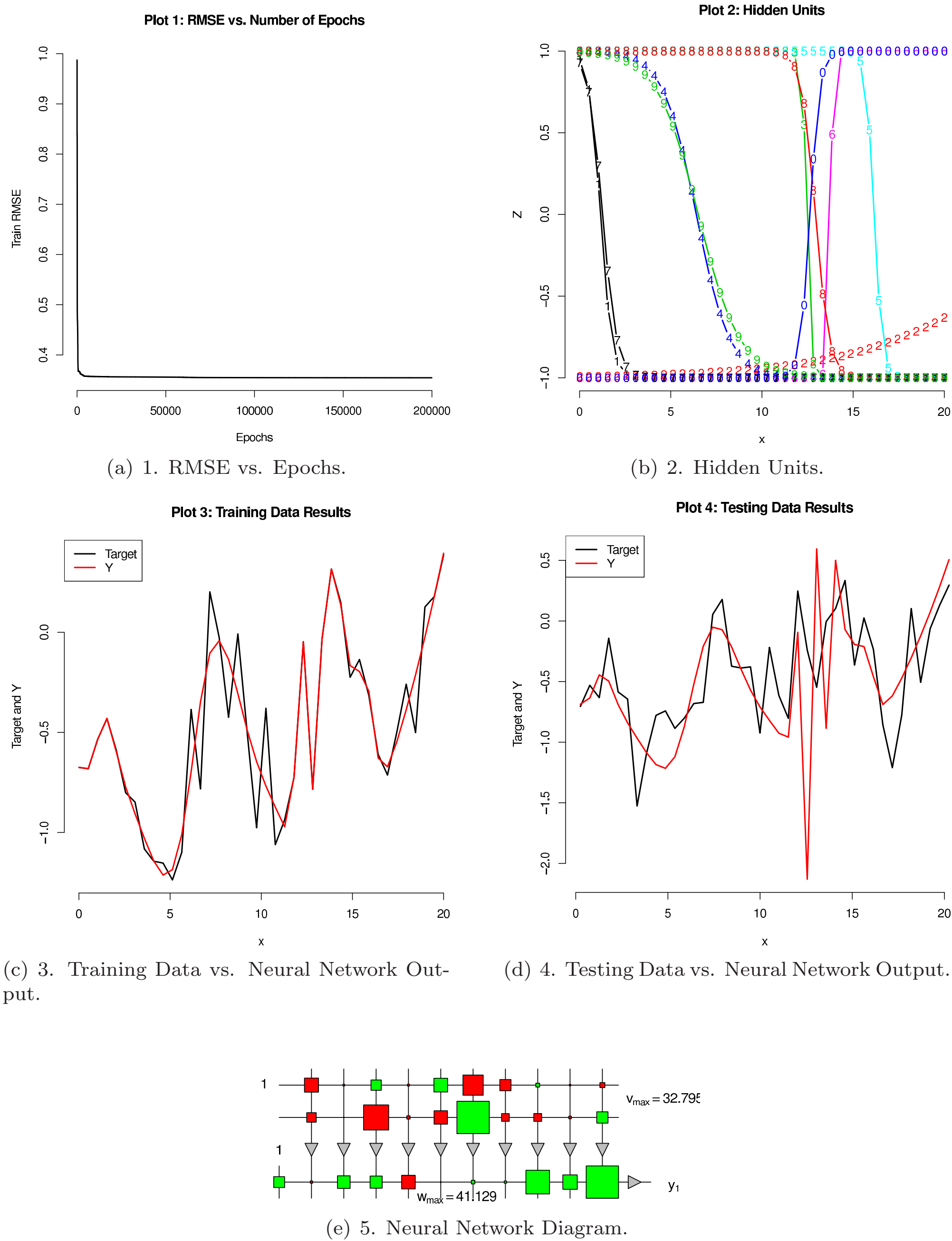
```
# Noisy sine curve init
f <- function (x)  -1 + 0.05 * x + 0.4 * sin(x) + 0.3 * rnorm(length(x))
nSamples <- 40
xmax <- 20
Xtrain <- matrix(seq(0,xmax,length=nSamples),nSamples,1)
Ttrain <- f(Xtrain)
Xtest <- Xtrain + xmax/nSamples/2
Ttest <- f(Xtest)

# Train the neural network
hiddenUnits <- 10
lambda <- 0
nReps <- 200000
nnet <- makeNN(Xtrain,Ttrain,hiddenUnits,lambda,nIterations=nReps)

# Use the neural network
trainResult <- useNN(nnet,Xtrain)
testResult <- useNN(nnet,Xtest)

####
# Plot 1
####
pdf("noisySine_plot1.pdf")
ptemp <- par(bty="n")

# plot the data
plot(sqrt(nnet$scgResult$ftrace),type="l",lwd=2,
      xlab="Epochs",ylab="Train RMSE",
      main="Plot 1: RMSE vs. Number of Epochs")

# turn off the graphs and restore the defaults
dummy <- dev.off()
par(ptemp)

####
# Plot 2
####
pdf("noisySine_plot2.pdf")
ptemp <- par(bty="n")

#standardize the Training Data
XtrainS <- nnet$standardizeF(Xtrain)
X1 <- cbind(1,XtrainS)
Z <- tanh(X1 %*% nnet$V)

# plot the data
matplot(Xtrain,Z,type="b",lwd=2,lty=1,xlab="x",ylab="Z",
         main="Plot 2: Hidden Units")

# turn off the graphs and restore the defaults
dummy <- dev.off()
par(ptemp)
```

```
####
# Plot 3
####
pdf("noisySine_plot3.pdf")
ptemp <- par(bty="n")

# plot the data
matplot(Xtrain,cbind(Ttrain,trainResult),lty=1,type="l",lwd=2,
        xlab="x",ylab="Target and Y",
        main="Plot 3: Training Data Results")
legend("topleft",c("Target","Y"),lty=1,lwd=2,col=1:2)

# turn off the graphs and restore the defaults
dummy <- dev.off()
par(ptemp)


####
# Plot 4
####
pdf("noisySine_plot4.pdf")
ptemp <- par(bty="n")

# plot the data
matplot(Xtest,cbind(Ttest,testResult),lty=1,type="l",lwd=2,
        xlab="x",ylab="Target and Y",
        main="Plot 4: Testing Data Results")
legend("topleft",c("Target","Y"),lty=1,lwd=2,col=1:2)

# turn off the graphs and restore the defaults
dummy <- dev.off()
par(ptemp)
####
# Plot 5
####
pdf("noisySine_plot5.pdf")

# plot the diagram of weights
drawNNet(nnet)

# turn off the graphs and restore the defaults
dummy <- dev.off()
```

A total of five graphs were generated in the code above. In Figure 2 it can be observed that the noise is very significant in the test data. There are a few small areas of error, but in general the trend of the test data is followed with the learned function. It is likely that if the data had a little less noise, that an even better fit could be made with the neural network, but since that is not the case, the accuracy still looks acceptable considering the noise.

3.2 Effect of Hidden Values on RMSE

After the initial analysis of the noisy sine data, an experiment with the number of hidden units was completed. In this experiment all parmaters were held constant except the number of hidden units was varied from 1 to 20. The code used to vary these units is listed below.



(a) 1. RMSE vs. Epochs.

(b) 2. Hidden Units.

(c) 3. Training Data vs. Neural Network Output.

(d) 4. Testing Data vs. Neural Network Output.

(e) 5. Neural Network Diagram.

Figure 2: These are the graphs for Noisy Sine Data.

```
rmseCumulative <- c()

for(hiddenUnits in c(1:20))
{
# Train the neural network
lambda <- 0
nReps <- 2000
nnet <- makeNN(Xtrain,Ttrain,hiddenUnits,lambda,nIterations=nReps)

# Use the neural network
trainResult <- useNN(nnet,Xtrain)
testResult <- useNN(nnet,Xtest)

rmseCumulative <- rbind(rmseCumulative, cbind(hiddenUnits,
                        sqrt(mean((trainResult-Ttrain)^2)),
                        sqrt(mean((testResult-Ttest)^2))))

}

####
# Graph Hidden Value vs. RMSE error
####
pdf("noisySine_hiddenUnits.pdf")
ptemp <- par(bty="n")

# plot the data
matplot(x=rmseCumulative[,1],y=rmseCumulative[,2],
        type="l",lwd=2,xlab="Number of Hidden Units",ylab="RMSE",
        main="Hidden Units vs. RMSE",col=1)
matplot(x=rmseCumulative[,1],y=rmseCumulative[,3], add=TRUE,col=2, type="l")
legend("topright",c("Train Data","Test Data"),lty=1,lwd=2,col=1:2)

# turn off the graphs and restore the defaults
dummy <- dev.off()
par(ptemp)
```

The result of the above code is shown in Figure 3. Looking at this graph you can see that the best count of the hidden units is around the middle of the range selected. There is also some risk of overfitting, which can be observed with the worsening test results with the higher hidden unit counts.

4 Modeling the Miles Per Gallon Data

4.1 Training the Neural Network and Results

The next data set that was modeled was some miles per gallon data gathered from [3]. This data can be used to train machine learning algorithms to predict the fuel economy from factors such as engine displacement and cylindar count.

Code was written to process the miles per gallon data. This code divided the data into a training set and a testing set. The training set was used to train a neural network. The code to complete these tasks is listed below.

```
mpg <-read.table("auto-mpg.data")
mpgnames <- c("mpg","cylinders","displacement","horsepower",
              "weight","acceleration","year","origin","name")
colnames(mpg) <- mpgnames
```

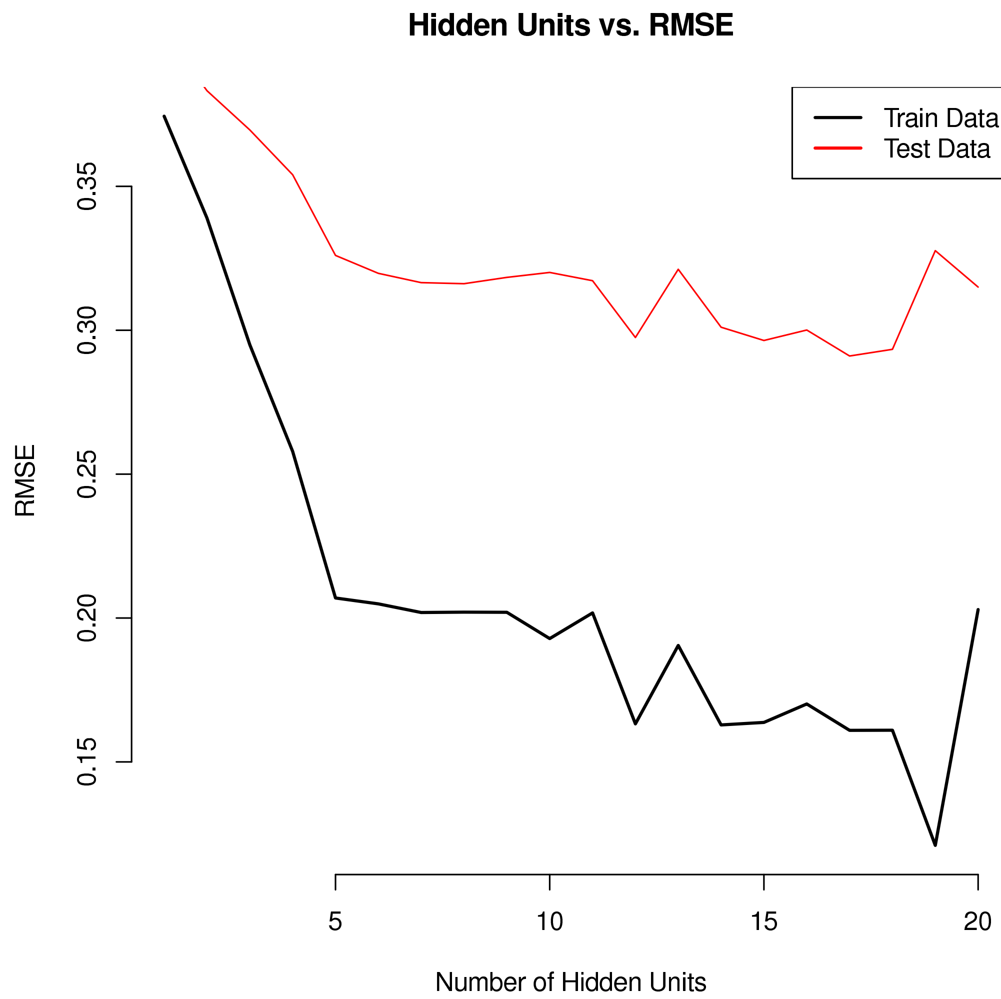
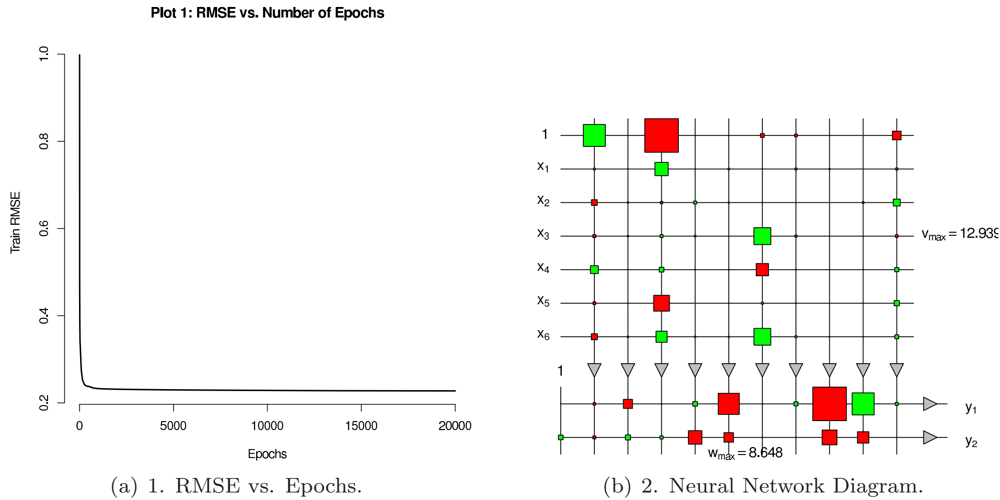


Figure 3: Noisy Sine Graph of Hidden Units vs. RMSE

```
### Remove all samples that have at least one "?"
keepRows <- apply(mpg != "?", 1, all)
mpg <- mpg[keepRows,1:8]


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

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

### Assemble Xtrain and Ttrain matrices
Xtrain <- as.matrix(mpg[trainRows, c(2,3,5:8)])
Ttrain <- as.matrix(mpg[trainRows, c(1,4), drop=FALSE])

### Assemble Xtest and Ttest matrices
Xtest <- as.matrix(mpg[testRows, c(2,3,5:8)])
Ttest <- as.matrix(mpg[testRows, c(1,4), drop=FALSE])


##########################################
# TRU NORMAL PLOTS
##########################################
hiddenUnits <- 10
lambda <- 0
nReps <- 20000
nnet <- makeNN(Xtrain,Ttrain,hiddenUnits,lambda,nIterations=nReps)

# Use the neural network
trainResult <- useNN(nnet,Xtrain)
testResult <- useNN(nnet,Xtest)


####
# Plot 1
####
pdf("mpgData_plot1.pdf")
ptemp <- par(bty="n")

# plot the data
plot(sqrt(nnet$scgResult$ftrace),type="l",lwd=2,xlab="Epochs",ylab="Train RMSE",
     main="Plot 1: RMSE vs. Number of Epochs")

# turn off the graphs and restore the defaults
dummy <- dev.off()
par(ptemp)


####
# Plot 2
```

```
####
pdf("mpgData_plot2.pdf")

# plot the diagram of weights
drawNNet(nnet)

# turn off the graphs and restore the defaults
dummy <- dev.off()


###############################################
# Plot the lambda and hidden units experiment
###############################################

pdf("mpg_plot1.pdf")
ptemp <- par(mfrow=c(3,2),bty="n")
# Train the neural network
nReps <- 10000
networkLambdasResults <- c()
for(lamb in c(0.01,0.10,0.25,0.5,1,1.5))
{
  print(paste("next lambda= ",lamb))
  resultsForHiddenUnits <- c()
  for(hiddenUnits in c(1,3,6,9,12,15))
  {
    nnet <- makeNN(Xtrain,Ttrain,hiddenUnits,lamb,nIterations=nReps)

    # Use the neural network
    trainResult <- useNN(nnet,Xtrain)
    testResult <- useNN(nnet,Xtest)

    resultsForHiddenUnits <- rbind(resultsForHiddenUnits, cbind(lamb,
                             hiddenUnits,
                             sqrt(mean((trainResult-Ttrain)^2)),
                             sqrt(mean((testResult-Ttest)^2))))


  }
  networkLambdasResults <- rbind(networkLambdasResults,
                                 resultsForHiddenUnits)

# plot the data
matplot(x=resultsForHiddenUnits[,2],y=resultsForHiddenUnits[,4],
        type="l",lwd=2,xlab="Number of Hidden Units",ylab="RMSE",
        main=paste("Hidden Units vs. RMSE for ",
        expression(lambda), " ", lamb),col=2,
        xlim=c(1,15), ylim=c(0,35))
matplot(x=resultsForHiddenUnits[,2],y=resultsForHiddenUnits[,3],
         add=TRUE, col=1, type="l")
legend("right",c("Train Data","Test Data"),lty=1,lwd=2,col=1:2)

}
```

The code above generates the two graphs shown in Figure 4. These graphs represent the RMSE of the base case of 10 hidden values and 20,000 iterations. The second graph shows a graphical representation of



(a) 1. RMSE vs. Epochs.

(b) 2. Neural Network Diagram.

Figure 4: These are the graphs for MPG Data.

the neural network itself. No graph is available to compare the training and test data with the output of the function due to the difficulty in displaying a function with mutiple input variables and multiple output variables.

4.2 Effect of λ and Hidden Values on RMSE

The graphs resulting from the code in the previous section can be seen in Figure 5. These graphs show that varying the lambda and hidden values can result in a fine tune of the machine learning algorithm. These graphs show even more of what was found when analyzing the noisy sine data, that larger numbers of hidden values help train the data, until some threshold where the algorithm starts to overfit.

5 Extra Credit

The `optim` function is a function built into R that can be used for optimization. In the case of the neural network, this can be used to replace the SCG code that was used earlier. The `optim` function supports a number of different types of optimization, but the one observed was the "CG" method, which should have very similar results to the "SCG" method we used previously, but with a different function. These results were generated with the code listed below.

```
##################################################
# makeNNoptim
# Author: Michael Yoensky
#
# Inputs:
#     Xtrain - the training X values
#     Ttrain - The training Y values
#     numHiddenUnits - The number of hidden units
#     lambda - The weight decay (optional)
#     nIterations - Specify the number of iterations (optional)
#     xPrecision - Specify a limiting condition
#     fPrecision - Specify a limiting condition
#     optimType - Specify the algorithm for optim to use
makeNNoptim <- function(Xtrain,
                        Ttrain,
                        numHiddenUnits,
```

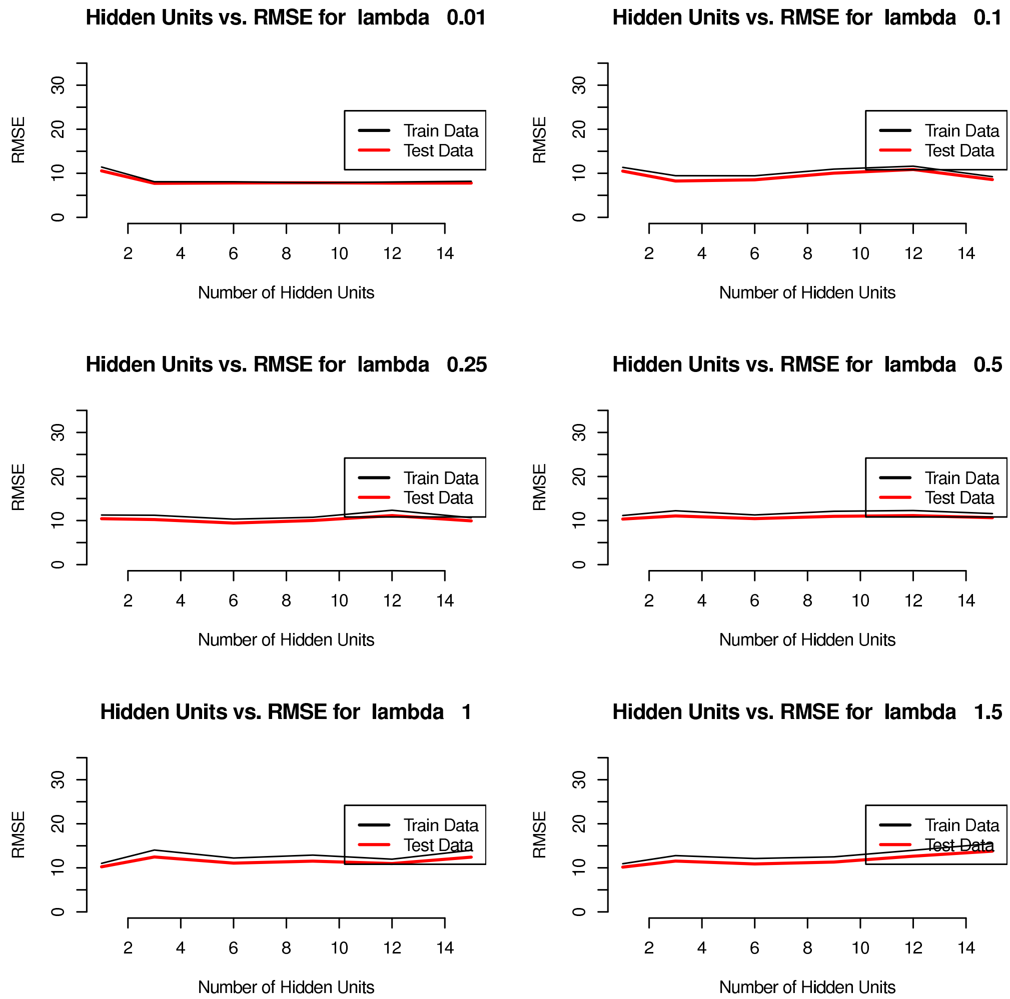


Figure 5: Graphs of Miles Per Gallon Data with Varying λ and Hidden Value Count.

```
                    lambda = 0,
                    nIterations = 10000,
                    xPrecision = 0,
                    fPrecision = 0,
                    method = "CG")
{



   standardizeF <- makeStandardizeF(Xtrain)
   XtrainS <- standardizeF(Xtrain)

   standardizeFTtrain <- makeStandardizeF(Ttrain)
   unstandardizeFTtrain <- makeUnstandardizeF(Ttrain)
   TtrainS <- standardizeFTtrain(Ttrain)

   X <- XtrainS
   T <- TtrainS

   nh <- numHiddenUnits
   ni <- ncol(X)
   no <- ncol(T)
   V <- matrix((runif((ni+1)*nh, min=-0.01, max=0.01)), ni+1,nh)
   W <- matrix((runif((nh+1)*no, min=-0.01, max=0.01)), nh+1,no)

   X1 <- cbind(1,X)
errorTrace <- NULL

   N <- nrow(X)
   K <- ncol(T)

   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)
   {
      unpacked_weights <- unpack(weights)
      Vtemp <- unpacked_weights$V
      Wtemp <- unpacked_weights$W

      Z <- tanh(X1 %*% Vtemp)
      Y <- cbind(1,Z) %*% Wtemp
      error <- Y - T
      mean(error^2) + (lambda * sum(t(Vtemp[-1,]) %*% Vtemp[-1,]))
   }
```

```
  gradF <- function(weights)
  {
     unpacked_weights <- unpack(weights)
         Vtemp <- unpacked_weights$V
     Wtemp <- unpacked_weights$W

         Z <- tanh(X1 %*% Vtemp)
     Y <- cbind(1,Z) %*% Wtemp
     error <- Y - T
     Vtemp <- (1/N) * (1/K) * (t(X1) %*%
               (error %*% t(Wtemp[-1,,drop=FALSE]) *
               (1-Z^2))) +
               (rbind(0,Vtemp[-1,,drop=FALSE]) * lambda)
     Wtemp <- (1/N) * (1/K) * t(cbind(1,Z)) %*% error
     pack(Vtemp,Wtemp)
 }

  control <- list(maxit=nIterations)
  scgResult <- optim( pack(V,W), sqErrorF, gradF,
                      control=control, method = method)
  VW <- unpack(scgResult$par)
  V <- VW[[1]]
  W <- VW[[2]]

  list(V=V,W=W, standardizeF=standardizeF,
       standardizeFTtrain=standardizeFTtrain,
              unstandardizeFTtrain=unstandardizeFTtrain,
              lambda=lambda, scgResult=scgResult)

}
checkECNN <- function(f, Xtrain, nh=5, nReps=200000,
                      filename="outputFile.png",
                      description="X and Y Values with Neural Network")
{
  # Generate the data
  Xtrain <- matrix(Xtrain)
  N <- nrow(Xtrain)
  xmax <- 20
  Ttrain <- f(Xtrain)
  Xtest <- Xtrain
  Ttest <- f(Xtest)

  # Train the neural network
  nnet <- makeNNoptim(Xtrain, Ttrain, nh, nIterations=nReps)

  # Use the neural network
  testResult <- useNN(nnet, Xtest)


  pdf(filename)
  ptemp <- par(bty="n")

  # plot the data
```

```
  legendTags <- c("Test Data","Predicted Result")
  legendColors <- c("black","red")
  matplot(Xtest,Ttest,pch=1,lty=1,xlab="X value",ylab="Y value",
          main=description,type="l",
          col=legendColors[1])
  matplot(Xtest,testResult,pch=1,lty=1,add=TRUE,type="l",
          col=legendColors[2])
  legend(x="topright",legend = legendTags,col=legendColors,lty=1)

  # turn off the graphs and restore the defaults
  dummy <- dev.off()
  par(ptemp)
}


# Example 1, Use a simple line, no noise
f<-function(x) {x}
checkECNN(f,seq(0,10),filename="simple_opt_ex1.pdf",
        description="Simple Line, No Noise")

# Example 2, Use a simple line
f<-function(x) {x+0.1*rnorm(length(x))}
checkECNN(f,seq(0,20),filename="simple_opt_ex2.pdf",
        description="Simple Line with Noise")

# Example 3, Basic Parabola
f<-function(x) {-(x-5)^2+10+0.3*rnorm(length(x))}
checkECNN(f,seq(0,10),filename="simple_opt_ex3.pdf",
        description="Parabola with Noise")

# Exame 4, Cosine
f<-function(x) {cos(x)+0.2*rnorm(length(x))}
checkECNN(f,seq(0,10),filename="simple_opt_ex4.pdf",
        description="Cosine with Noise")
```

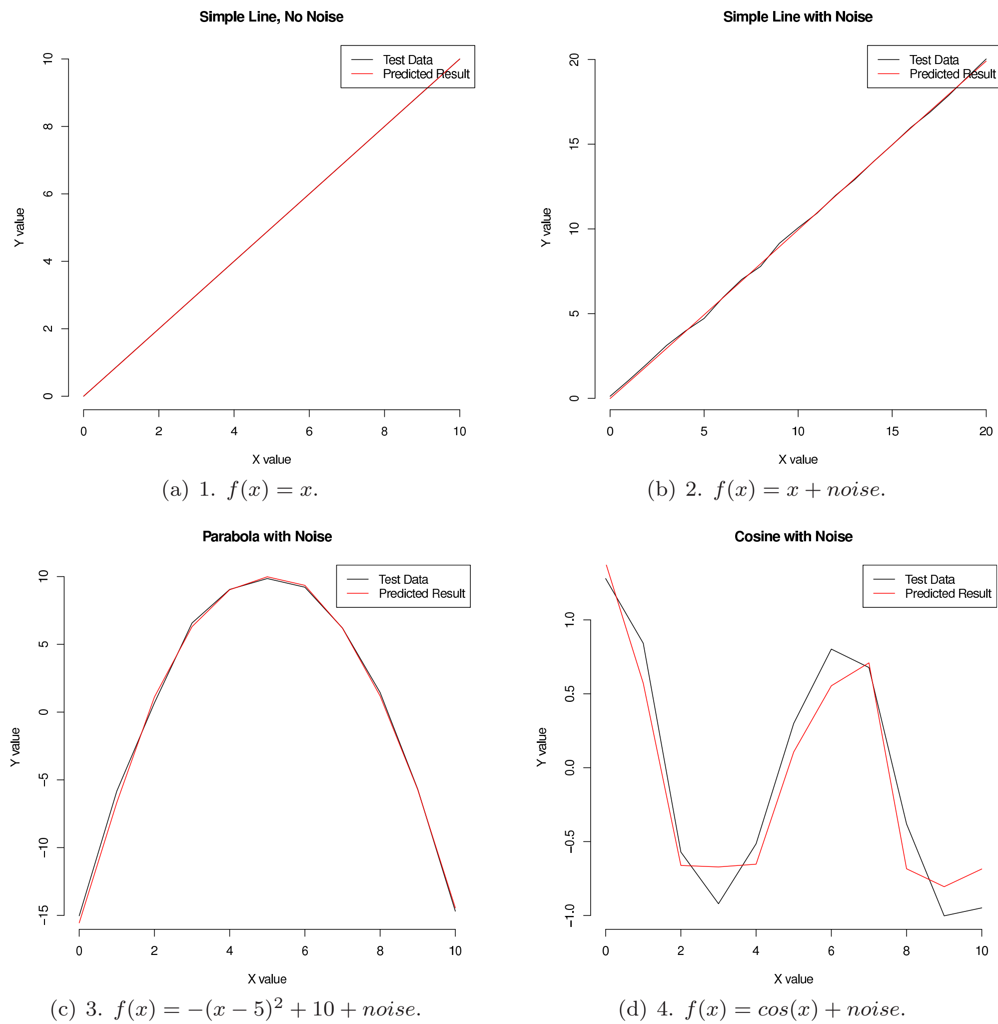
The graphs in Figure 6 were generated with the code above. These graphs show that the function `optim` using "CG" can have a similar effect as the "SCG" algorithm used in the other examples.

The graphs in Figure 7 were generated with the code above, but using a different algorithm of the `optim` function. These graphs show that the function `optim` using "CG" can have a similar effect as the "SCG" algorithm used in the other examples.

6 Discussion

The biggest challenge I had completing this project was to get the SCG algorithm correctly working with the rest of the neural network code. It seems that the results of the two `sqErrorF` and `gradF` are critical to have the correct format.

One problem that I had was that I used the wrong variable name for my training data. I had included two training data variables, one with the original training data and a second with the standardized training data. I accidentally used the original training data instead of the standardized data, which resulted in an output graphe with the values shifted up and flattened.



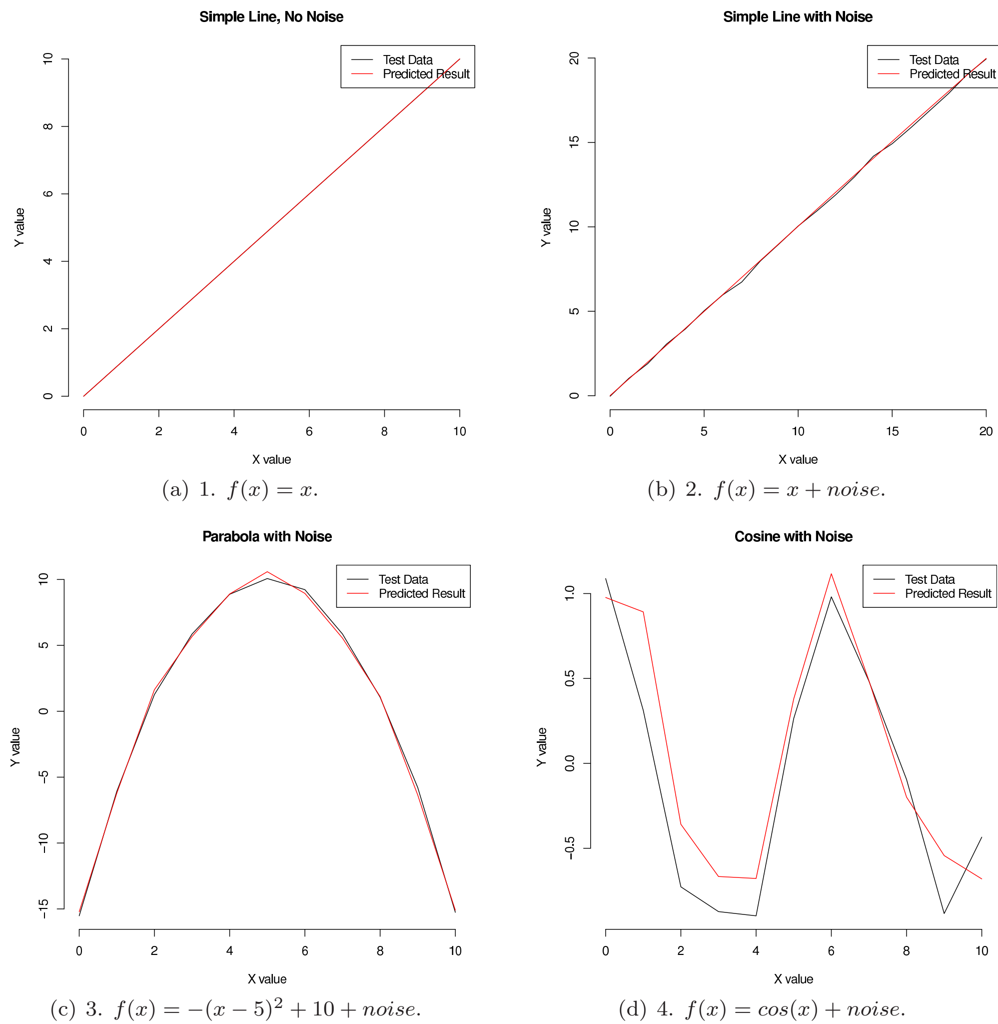
(a) 1. $f(x) = x$.

(b) 2. $f(x) = x + noise$.

(c) 3. $f(x) = -(x-5)^2 + 10 + noise$.

(d) 4. $f(x) = cos(x) + noise$.

Figure 6: Training with CG using `optim`.



(a) 1. $f(x) = x$.

(b) 2. $f(x) = x + noise$.

(c) 3. $f(x) = -(x-5)^2 + 10 + noise$.

(d) 4. $f(x) = cos(x) + noise$.

Figure 7: Training with BFGS using `optim`.

7 Conclusions

Overall the data was very closely matched with the neural networks using scaled conjugate gradient. This method of analyzing data appears to be very accurate, and very configurable for different types of data. It also appears to be vulnerable to the same types of errors that other methods have been, particularly overfitting.

With this method of analysis it seems that the best way to choose a count of hidden units to use, that the number of local maximums and minimums appears to be related to the best number of hidden units to use. When there are no true maximums (in the case of a linear value, a single hidden unit may be sufficient for a highly accurate

References

[1] Anderson, C., *Lecture Notes 1 through 20, 10/29/2009*, `http://www.cs.colostate.edu/~anderson/cs545/`, 2009.

[2] Anderson, C., *Homework Assignment 5, 10/23/2009*, `http://www.cs.colostate.edu/~anderson/cs545/assignments/assignment5.html`, 2009.

[3] Asuncion, A. & Newman, D.J. (2007). UCI Machine Learning Repository `http://www.ics.uci.edu/~mlearn/MLRepository.html`. Irvine, CA: University of California, School of Information and Computer Science.