

CS 545: Assignment 5

Nonlinear Regression with Neural Networks

Manaf Gharaibeh

October 28, 2009

Contents

1 Introduction	2
2 Implementing the Neural Network	2
2.1 Supporting functions	3
2.2 makeNN Function	3
2.3 useNN function	5
3 Small Set of Samples Test	6
3.1 Generating Samples	6
3.2 Creating the Neural Network	6
3.3 Results of Applying the Neural Network to the Simple Samples Set	6
4 Applying the NN to a noisy sine Curve	9
4.1 Generating the Training and Testing Data	9
4.2 Modeling the Neural Network to Fit the Noisy sin Curve and Testing Results	9
5 Applying the NN to MPG Data	17
5.1 Data Preparation	17
5.2 Results and Observations of Applying NN to mpg Data	18
6 more Results With Another Cost Functions	22
7 Conclusions	23

1 Introduction

Artificial Neural Networks (ANN), or simply Neural Network (NN) are non-linear statistical tools used to model data. A NN consists of a number of “neurons” which is term from the biology world, in ANN it can be simply called a node or a unit. There are usually a number of hidden units and output units as illustrated in figure 1[1], the hidden units produces their results to the output units, while the final outcome of a NN system comes from the output units. A NN can have many hidden layers and typically one output layer. The output of the hidden units is somehow hidden, it is typically an input for the next layer in the NN. During the learning phase of building a NN model, a series of forward and backward passes of results or weights between the hidden and output units take place in order to optimize the weights so that the errors of the model are minimized. This is actually a very attractive feature of the NN, that gives it the ability to adapt based on previously calculated outputs compared with some predefined expected output, usually through the use of root mean square error calculations. A cost function is needed in the hidden units to assess the inputs and produce results as inputs for next layers. Usually this function is a sigmoid function which takes the shape of the letter S, in this report experiments we used the hyperbolic tangent function \tanh which is defined by the following formula:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

The rest of this report discusses the implementation of a two layer NN, and testing the NN on a simple set of samples. The report also discusses applying the NN to a noisy curve sine, and finally the report presents some results of applying the NN model to some real data downloaded from the UCI data sets repository website.

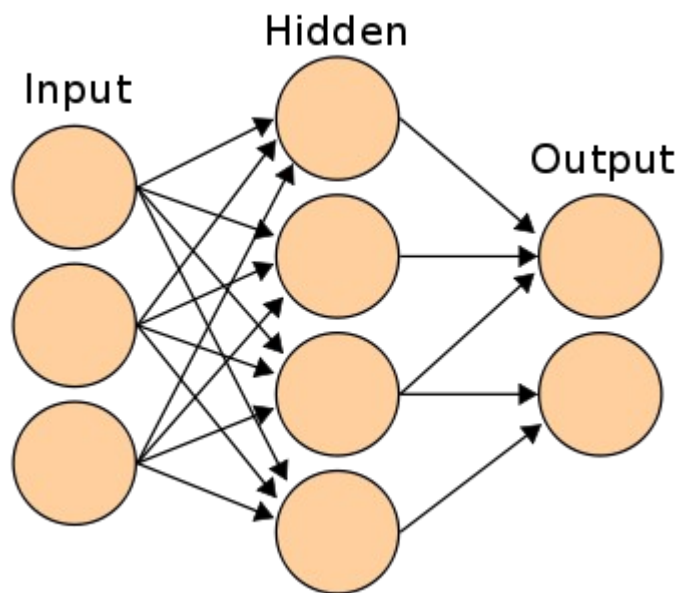


Figure 1: A Simple Neural Network (NN) representation

2 Implementing the Neural Network

This section talks about the functions used to implement the Neural Network including some supportive functions that are not specific to NNs.

2.1 Supporting functions

The *makeStandardizeF* function is used to standardize data samples, on the other hand another function is the *makeUnStandardizeF* is used to un-standardize the standardized results of the neural network by doing the reverse math of the *makeStandardizeF* function, this can be done using the following R code.

```
### make unstandardize
makeUnStandardizeF <- function(Ttrain) {
  mu <- colMeans(Ttrain)
  sigma <- sd(Ttrain) ##sd should be named colSds

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

The implementation of the *NNs* involves the process of finding better weights values for both, the hidden and output units, this is done with the assist of the Scaled Conjugate Gradient function *scg* adapted from "*gradientDescents.R*" example by professor Anderson. The function arguments are mainly the initial weights values, a function to be minimized, and the gradient of that function, which are explained in the talk about the *makeNN* function in section 2.2 of this report. *scg* returns a list of optimized weights to be used in the NN model. Another function used in this experiment is the *drawNNet* adapted from "*nnUtils.R*" example also given by professor Anderson. This function takes a list of weights from the hidden and output units and draw a diagram to show the impact of the weights on the NN model.

2.2 makeNN Function

This is the main function in this experiment and most of the work is done inside it. A number of functions are implemented within *makeNN*. Following is a brief description for each of them.

pack Function

This is a simple function used to pack to list of weight together using the *R* function *c* as it is shown in the following code.

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

This is useful for calling the *scg* function which expects a list of weights while we have two lists, one for the hidden units, and the other for the output units.

unpack Function

This function does exactly the opposite of the *pack* function.

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

```

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

```

where ni is the number of inputs, nh is the number of hidden units, and no is the number of output units.

sqErrorF Function

This function receives the weights of V and W as one list, it unpacks them and calculate and return the mean square error.

```

sqErrorF <- function(weights)
{
    VW <- unpack(weights)
    V <- VW[[1]]
    W <- VW[[2]]

    ### forward pass
    Z <- tanh(Xs1 %*% V)
    Y <- cbind(1,Z) %*% W

    ### calculating errors
    errors <- (Y - Ts)
    (mean(errors^2) + sum(lambda * (t(V[-1,]) %*% V[-1,])))
}

```

This function also implements a weight decay penalty on weight magnitudes for all hidden layer weights except for the constant 1 input weights, this penalty is added to the returned value of the mean squared errors.

gradF Function

This function makes a forward pass, and then it calculates the gradients of squared error for each V and W weights and return the resulted gradients values of V and W as one list.

```

gradF <- function(weights)
{
    VW <- unpack(weights)
    V <- VW[[1]]
    W <- VW[[2]]

    ### forward pass
    Z <- tanh(Xs1 %*% V)
    Y <- cbind(1,Z) %*% W

    ### calculating errors
    error <- (Y - Ts)

    N <- nrow(Xtrain)
    K <- ncol(Ttrain)

    ###gradients

    gradientV <- (1/(N+K)) * (t(Xs1) %*% (error %*% t(W[-1,])*(1-Z^2))) +

```

```

    rbind(0,V[-1,]) * lambda

    gradientW <- (1/(N+K)) * (t(cbind(1,Z)) %*% error)

    pack(gradientV,gradientW) #return V and W gradients
}

```

sqErrorF and *gadeF* are passed to the *scg* function which will use them to optimize V and W values so that they best fit the training data.

The rest of the function *makeNN* does the work of standardizing the train data and the target values through creating two standardizing functions. Also an un-standardize function is made to be returned at the end of this function to be used later by the *useNN* function later to un-standardize the output of the NN. Also a bias constant input is added to the training data samples. Initial values for V and W are created in *makeNN* using the following R code.

```

V <- matrix(runif((ni+1)*nh,-0.01,0.01), ni+1,nh)
W <- matrix(runif((nh+1)*no,-0.01,0.01), nh+1,no)

```

After setting V and W weights, and the *sqErrorF* and *gradF* functions, the *scg* function is called using the next statement.

```

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

```

Among the results of calling *scg* is a list *x* containing V and W weights after many iteration done by *scg*, through which it called the *sqErrorF* and *gradF* functions to do forward and backward passes based on the calculated mean squared errors in order to optimize the weights to minimize those errors values.

The returned results can be unpacked back to V and W using the following R code.

```

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

```

those are returned along with a list of standardizing functions for the training data and the targets, and the un-standardize function, lambda values, and the number of epochs returned by the *scg* function which represent the number of iteration *scg* used to find a local minimum value in the process of optimizing weights.

```

list(V=V, W=W, standardizeFXt=standardizeFXt, standardizeFTtr=standardizeFTtr,
     unstandardizeFTtr=unstandardizeFTtr, lambda=lambda,
     epochs=length(scgResult$ftrace))

```

2.3 useNN function

This function receives the NN model and the data to test, it first standardizes the data and then adds a constant 1 column to the left of that data, then it applies the NN model to calculate the output of the NN

as the following code shows.

```
useNN <- function(nnet,Xtest)
{
  XtestS <- nnet$standardizeFxt(Xtest)
  XtestS1 <- cbind(1,XtestS)

  Z <- tanh(XtestS1 %*% nnet$V)
  Y <- cbind(1,Z) %*% nnet$W

  list(preds=nnet$unstandardizeFTtr(Y),Z=Z)
}
```

useNN returns a list that contains un-standardized predictions, and the output of the hidden layers.

3 Small Set of Samples Test

This section describes the generation of a simple small set of samples to model and test the NN described in section 2.

3.1 Generating Samples

The following R code is used to generate the samples set including the training and targets data .

```
f <- function (x) x*2
nSamples <- 10
xmax <- 20
Xtrain <- matrix(seq(0,xmax,length=nSamples),nSamples,1)
Ttrain <- f(Xtrain)
```

3.2 Creating the Neural Network

The NN is created by making the following call to the *makeNN* function with the arguments corresponding to the train data, targets values, and the number of hidden units which is one in this example.

```
nn <- makeNN(Xtrain,Ttrain,1)
```

The returned model is stored in *nn* to be used later in a call to *useNN* function.

3.3 Results of Applying the Neural Network to the Simple Samples Set

The following statement calls the *useNN* function using the *nn* model created previously in the call to *makeNN*, also the *Xtrain* data is passed to *useNN* in order to test the model on the train data, the *useNN* function returns a list containing the output units output (i.e. Y), and the hidden units output (i.e. Z).

```
trainPred <- useNN(nn, Xtrain)
```

Figure 2 shows the output of the NN and the target values against the train set values. This figure is the result of the following R code.

```
matplot(Xtrain,cbind(Ttrain,trainPred$preds),lty=1,type="l",lwd=2,xlab="x",ylab="T
and Y", main="Training data Outputs(Y) and Targets(T)")
```

```
legend("topleft",c("T","Y"),col=c("Black","Red"),lty=c(1,2),lwd=2,bty="n")
```



Figure 2: Training data outputs

The following is a list of training data generated as described in section 3.1.

Xtrain

```
0.000000  
2.222222  
4.444444  
6.666667  
8.888889  
11.111111  
13.333333  
15.555556  
17.777778  
20.000000
```

And the following list represents the predictions returned from the NN, which is actually very close from the targets values

trainPred\$preds

```
0.007339855  
4.442199682  
8.882976436  
13.327998297  
17.775587083  
22.224060717
```

26.671735719
31.116929699
35.557963838
39.993165377

Figure 3 shows the output of the hidden units, we have only one output unit in this example, the line drawn in the figure matches the curve generated by the drawing the target values of the simple function used to generate the samples. The plot is generated using the following R code.

```
matplot(Xtrain,trainPred$Z,type="b",lwd=2,lty=1,xlab="x",ylab="Z",main="Output of hidden units (Z)")
```

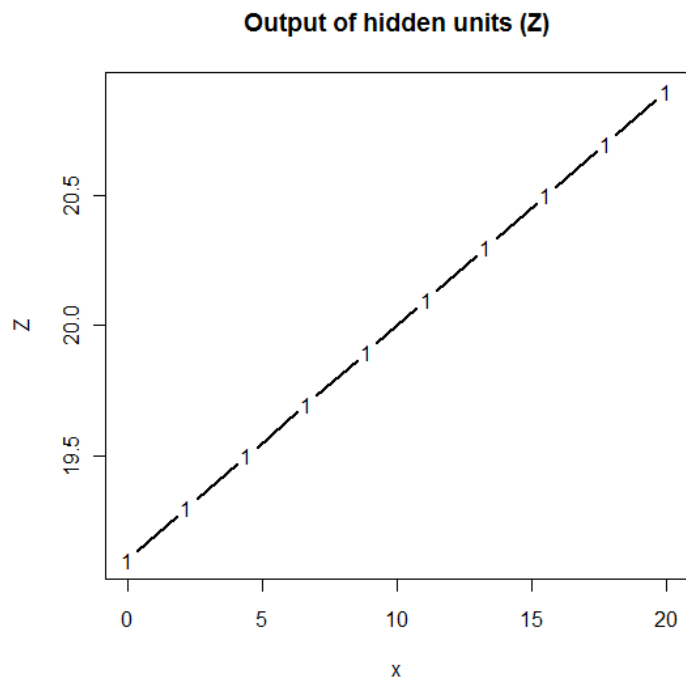


Figure 3: Output of hidden units

Figure 4 shows that the root mean square error RMSE declines sharply when *sgc* uses 1000 iterations to find an optimal local minimum instead of 500, after that the model couldn't improve anymore because RMSE is already at its best possible minimum value which is zero. Obviously this is due to the simple data set used in this example.

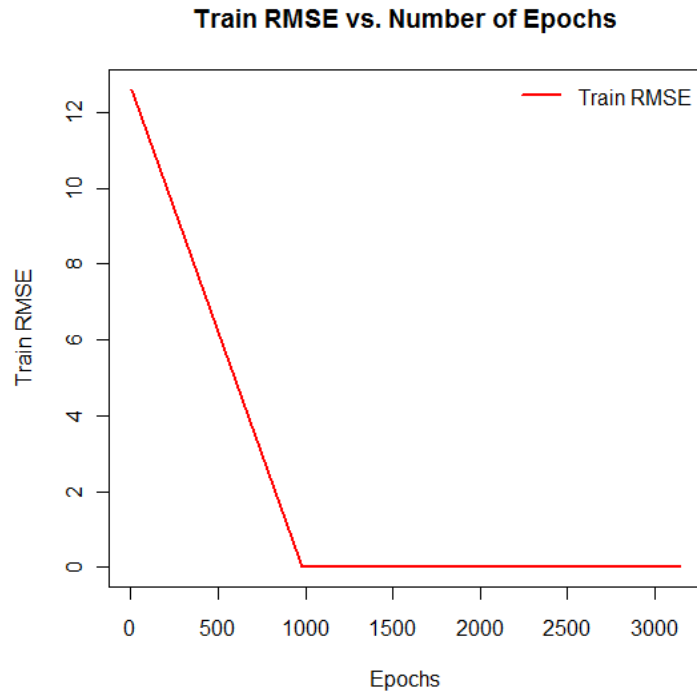


Figure 4: Train RMSE and number of Epochs

4 Applying the NN to a noisy sine Curve

This section describes the use of the NN we implemented to fit a noisy sine curve.

4.1 Generating the Training and Testing Data

The data samples for this experiment are generated using the following R code.

```
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)
```

4.2 Modeling the Neural Network to Fit the Noisy sin Curve and Testing Results

A call to *makeNN* function is made with arguments representing the training data, and target values, the number of hidden units used in this experiment is 10, also no weight decay is applied because lambda is set to zero. The test data is send to the *useNN* function along with already NN generated model. The following figures describes various results for training and testing data samples.

Figure 5 shows the RMSE for both the training and testing data versus the number of epochs returned from the *scg* function, the plot is generated using the following R code.

```

### train and test RMSE vs. epochs
epochs<-NULL
errorTraceTr<-NULL
errorTraceTs<-NULL
nReps <- 10000
for (reps in seq(1,nReps,1000))
{
  nn <- makeNN(Xtrain,Ttrain,10,0,reps)

  trainPred <- useNN(nn, Xtrain)
  testPred <- useNN(nn, Xtest)

  errorTr <- trainPred$preds - Ttrain
  errorTs <- testPred$preds - Ttest

  errorTraceTr <- c(errorTraceTr, sqrt(mean(errorTr^2)))
  errorTraceTs <- c(errorTraceTs, sqrt(mean(errorTs^2)))

  epochs <- c(epochs,nn$epochs)

  matplot(epochs,cbind(errorTraceTr,errorTraceTs),col=c("Red","Blue"),lty=1,type="l",lwd=2,x
    lab="Epochs",ylab="RMSE", main="Train and Test RMSE vs. Number of Epochs")
  legend("topright",c("Train RMSE","Test RMSE"),col=c("Red","Blue"),lty=1,lwd=2,bty="n")
}

```

The plot for the train RMSE could be generated by accumulating and then plotting the errors results calculated inside the *sqErrorF* function through the iterative calls made by the *scg* function during the learning phase.

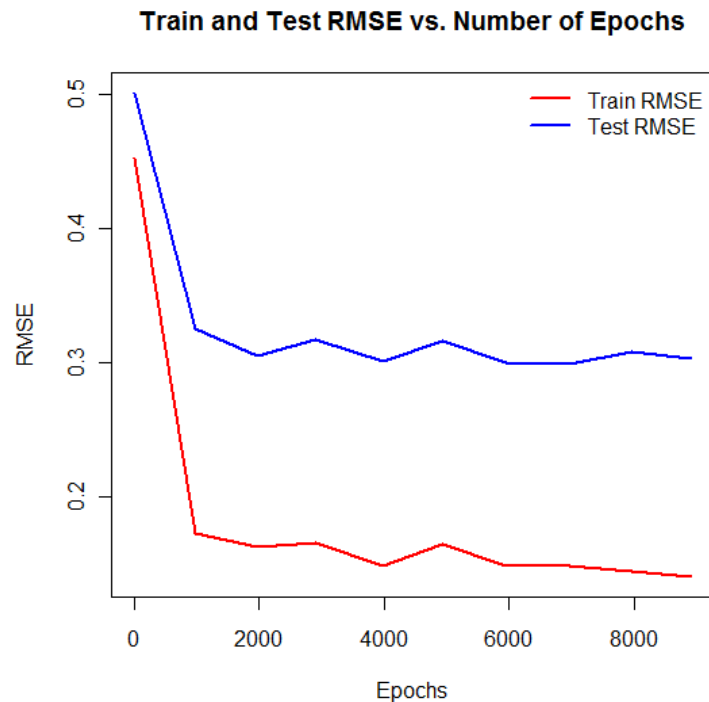


Figure 5: RMSE against epochs

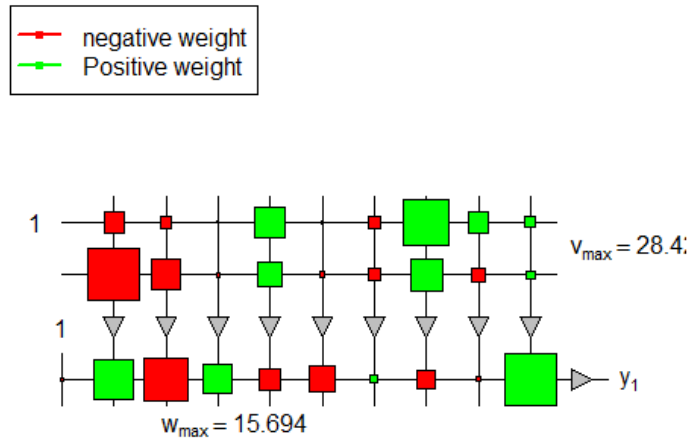


Figure 7: Hidden and output layers weights

Figure 8 shows a comparison between the predictions of the NN and the target values, The models seems to fit the noisy sin curve well with some exception for some part of the curve. The results are improved with a larger number of hidden units as it will be shown later. The plot of figure 8 is generated using the following R code.

```
###train output
trainPred <- useNN(nn, Xtrain)
matplot(Xtrain, cbind(Ttrain, trainPred$preds), lty=1, type="l", lwd=2, xlab="x", ylab="T
and Y", main="Training data Outputs(Y) and Targets(T)")
legend("topleft", c("T", "Y"), col=c("Black", "Red"), lty=1, lwd=2, bty="n")
```

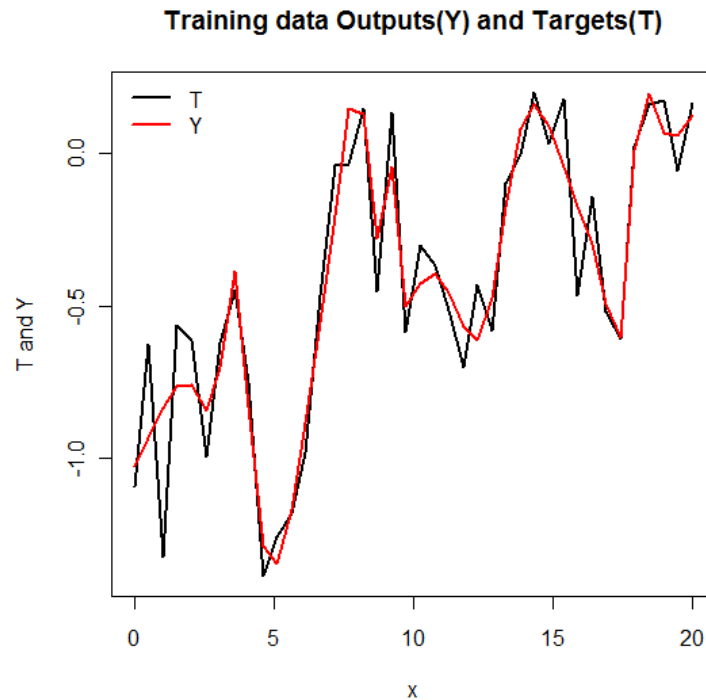


Figure 8: Comparison between NN output and target values

Figure 9 shows the results of comparing NN outputs to test data target values. Obviously the results are not as good as the results of the training data, which is expected because the model was build using the training data samples and it was contentiously trying to adapt to them. The plot is generated using the following code.

```
### test output
testPred <- useNN(nn, Xtest)
matplot(Xtest, cbind(Ttest, testPred$preds), lty=1, type="l", lwd=2, xlab="x", ylab="T
and Y", main="Testing data Outputs(Y) and Targets(T)")
legend("topleft", c("T", "Y"), col=c("Black", "Red"), lty=1, lwd=2, bty="n")
```

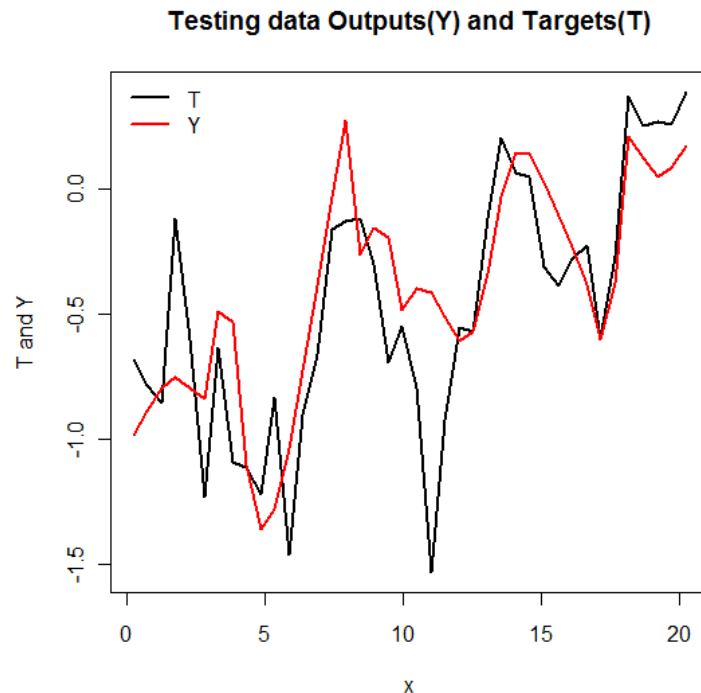


Figure 9: Comparison between NN test data predictions and test target values

Figure 10 shows the train and test data RMSE versus the number of hidden units. The RMSE decreases as the number of hidden units increases considering the train data set, however it is about the same after using 5 hidden units or more for the more important test data samples. Also figure shows how much the NN model performs better with training data than with testing data. The plot for figure 10 is generated using the following R code which involves a for loop to change the number of hidden units for each iteration.

```
errorTraceTr<-NULL
errorTraceTs<-NULL
nHidden <- 20
for (nh in 1:nHidden) {

  nn <- makeNN(Xtrain,Ttrain,nh) #nh is number of hidden units desired

  trainPred <- useNN(nn, Xtrain)
  testPred <- useNN(nn, Xtest)

  errorTr <- trainPred$preds - Ttrain
  errorTs <- testPred$preds - Ttest

  errorTraceTr <- c(errorTraceTr, sqrt(mean(errorTr^2)))
  errorTraceTs <- c(errorTraceTs, sqrt(mean(errorTs^2)))

  matplot(seq(1,nh),cbind(errorTraceTr,errorTraceTs),col=c("Red","Blue"),lty=1,ty
  pe="l",lwd=2,xlab="Number of hidden units",ylab="RMSE",
  main="Train and Test RMSE vs. Number of Hidden Units")
}
```

```

legend("topright",c("Train RMSE","Test
RMSE"),col=c("Red","Blue"),lty=1,lwd=2,bty="n")
}

```

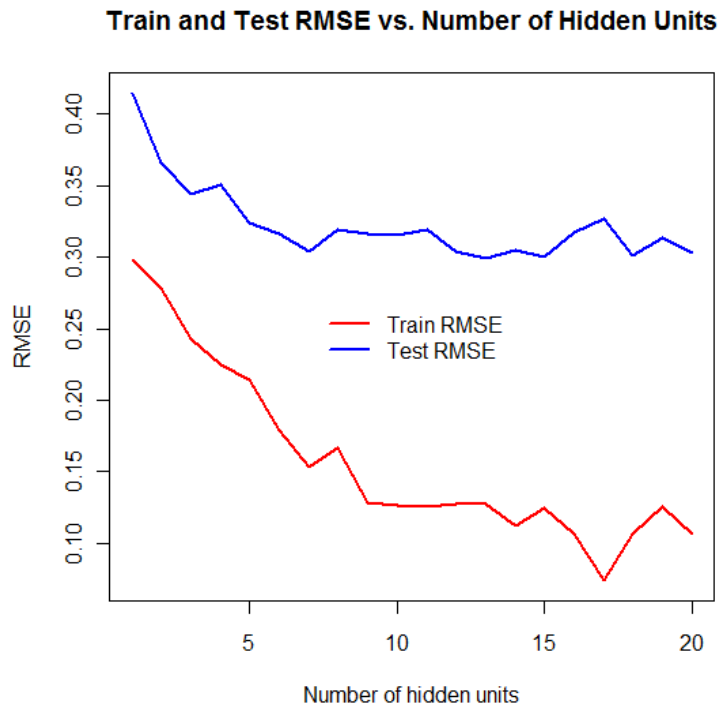


Figure 10: Train and test data RMSE versus number of hidden units

Figure 11 shows the results of NN for the training and testing data with 20 hidden units, again the figure shows how increasing the number of hidden units decreases the RMSE for training data, but doesn't with the testing data.

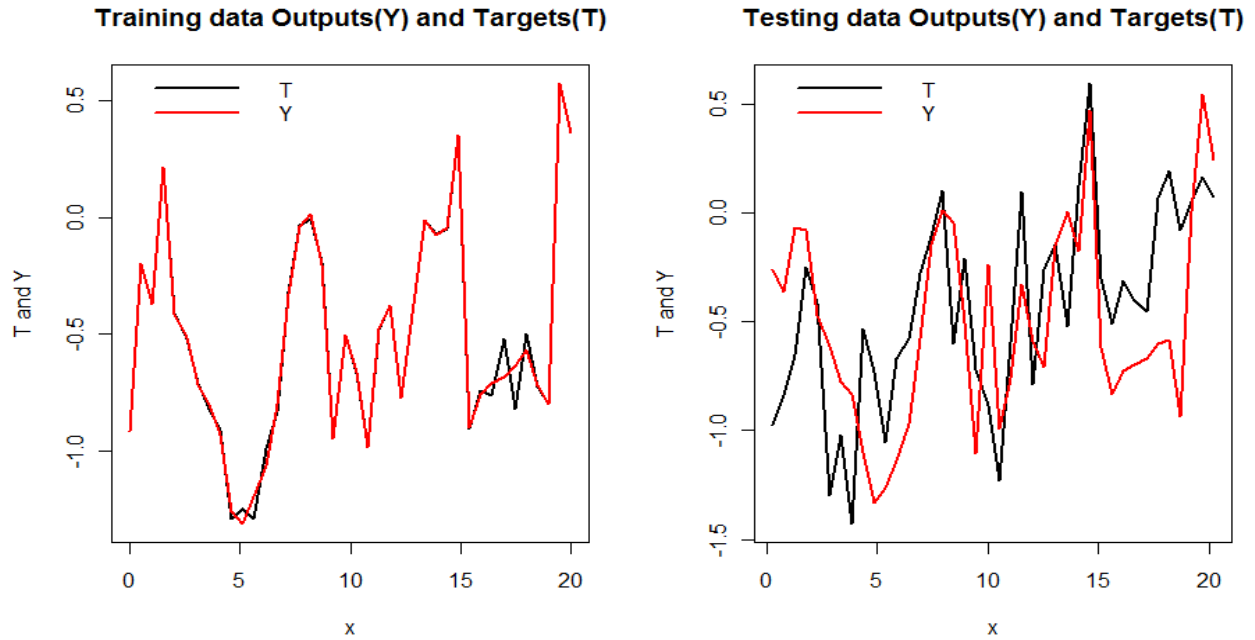


Figure 11: Training and Testing NN output with 20 hidden units

Figure 12 shows the results with a small number of hidden units, we have almost the same results for the testing data, which suggests that we don't need to have a large number of hidden units, because that in most of this report experiments only improved the results with training data.

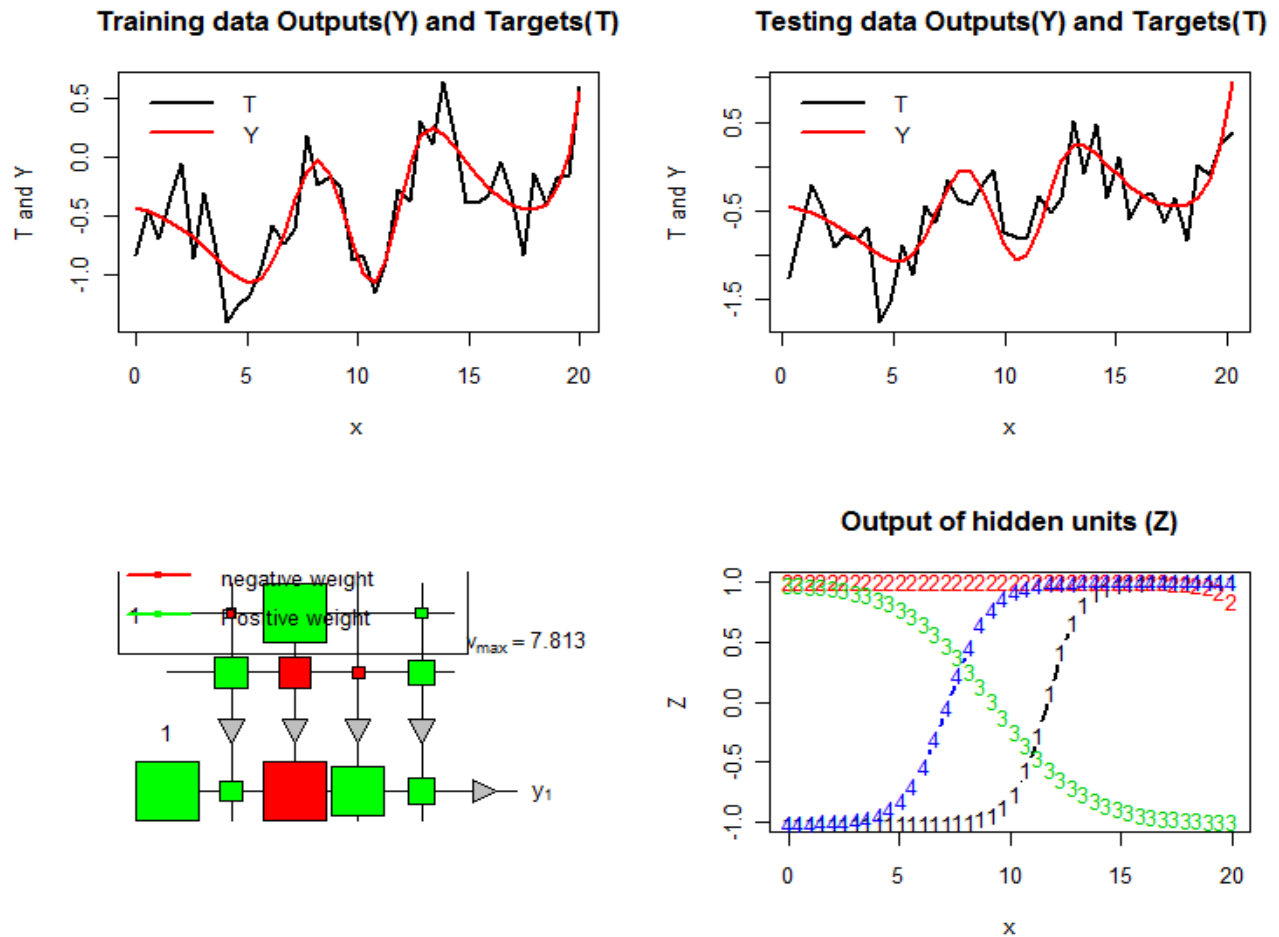


Figure 12: Results with small number of hidden units

5. Applying the NN to MPG Data

This section discusses applying the suggested NN model to a real data set, which is the mpg data set from the UCI repository website[2]. The experiment will try to predict the values of the *mpg* and *horsepower* columns, after training the NN with the rest columns values for each sample.

5.1 Data Preparation

Data for this experiment is read using the following R statement which directly bring the data from the UCI website.

```
mpg<-read.table("http://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/auto-mpg.data")
```

samples with missing data are removed, and then the remaining samples are portioned to training and testing samples with a ratios of 80% to 20% respectively.

```
randorder <- sample(nrow(mpg))
nTrain <- round(nrow(mpg)*0.8)
trainRows <- randorder[1:nTrain]
```

The following R statements are used to pick the training columns by excluding the columns to predict, while saving the columns to predict in T.

```
X <- mpg[trainRows, c(2,3,seq(5,8))]
T <- mpg[trainRows, c(1,4), drop=FALSE]
```

and finally the training data and targets are converted to numeric values using the following statements.

```
Xtrain <- apply(X,2,as.numeric)
Ttrain <- apply(T,2,as.numeric)
```

The testing data is prepared in similar way by taking the remaining 20% samples from the mpg set.

```
Xtest <- mpg[-trainRows, c(2,3,seq(5,8))]
Ttest <- mpg[-trainRows, c(1,4), drop=FALSE]
```

2 Results and Observations of Applying NN to mpg Data

The NN for the mpg data set is established with the following call to *makeNN*.

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

The results of the NN predictions applied to the *mpg* data set is compared to the target values for each of the mpg and Horsepower values and are presented in Figure 13 which is generated using the following code.

```
p <- par(mfrow=c(2,2))

###train output
trainPred <- useNN(nn, Xtrain) # useNN returns output of the NN output units
(preds should call it Y) and output of hidden units (Z)
matplot(cbind(Ttrain[,1],trainPred$preds[,1]),lty=1,col=c("Black","Red"),type="l",
lwd=1,xlab="x",ylab="T and Y",main="Training data (mpg)")
legend("topleft",c("T","Y"),col=c("Black","Red"),lty=1,lwd=1,bty="n")

matplot(cbind(Ttrain[,2],trainPred$preds[,2]),lty=seq(1,4),col=c("blue","pink"),ty
pe="l",lwd=1,xlab="x",ylab="T and Y",main="Training data (Horsepower)")
legend("topleft",c("T","Y"),col=c("blue","pink"),lty=1,lwd=1,bty="n")

### test output
testPred <- useNN(nn, Xtest)
matplot(cbind(Ttest[,1],testPred$preds[,1]),lty=1,col=c("Black","Red"),type="l",lw
d=1,xlab="x",ylab="T and Y",main="Testing data (mpg)")
legend("topleft",c("T","Y"),col=c("Black","Red"),lty=1,lwd=1,bty="n")

matplot(cbind(Ttest[,2],testPred$preds[,2]),lty=1,col=c("blue","pink"),type="l",lw
d=1,xlab="x",ylab="T and Y",main="Testing data (Horsepower)")
legend("topleft",c("T","Y"),col=c("blue","pink"),lty=1,lwd=1,bty="n")
```

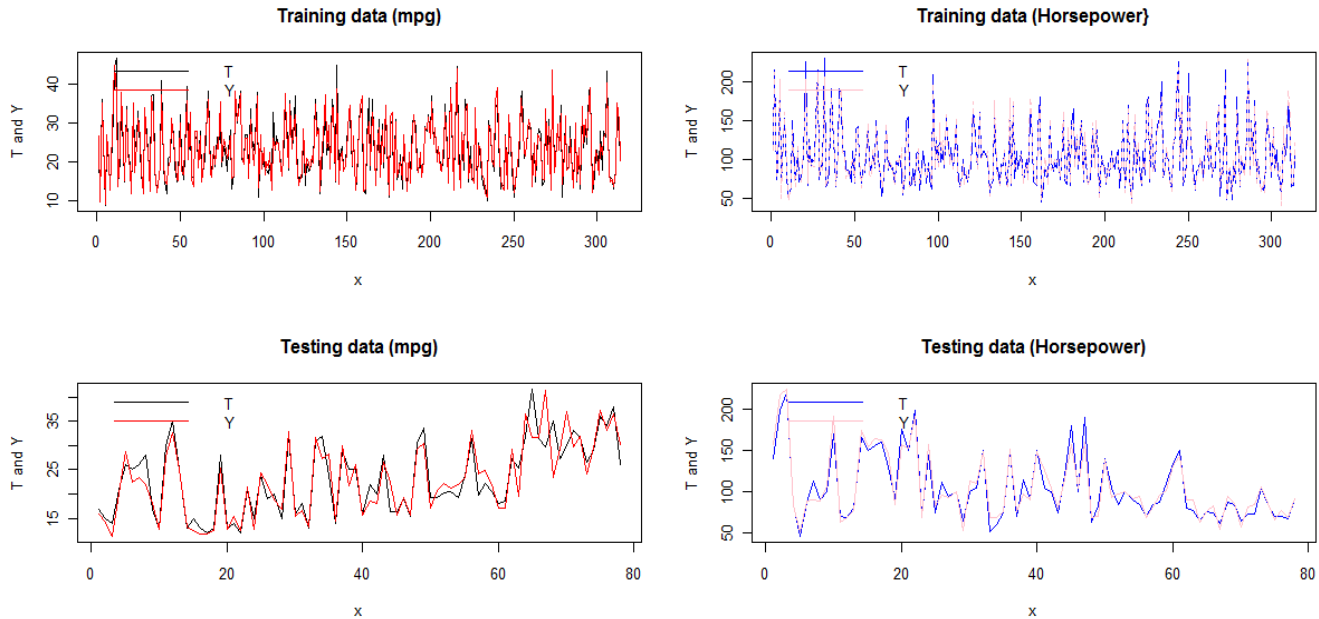


Figure 13: Target values against NN outputs predictions for training and testing data for both mpg and Horsepower values

As figure 13 shows, the NN model worked successfully with both train and test data and for both outputs, the mpg and the Horsepower values. Figure 14 also shows low RMSE values with respect to the margins of the values for the mpg and horsepower. Also figure 10 shows the relation between RMSE and the number of hidden units. The figure shows how again RMSE declines for training data when the number of hidden units increases, while it doesn't do the same for the testing data after using 11 hidden units for mpg and 8 for the horsepower attribute. The four plots are generated using the following R code.

```
errorTraceTrmpg <- NULL
errorTraceTsmpeg <- NULL
errorTraceTrhp <- NULL
errorTraceTshp <- NULL
p <- par(mfrow=c(1,2))
for (nh in seq(2,20,by=3)) {
  print(nh)

  nn <- makeNN(Xtrain,Ttrain,nh) #nh is number of hidden units desired

  trainPred <- useNN(nn, Xtrain)
  testPred <- useNN(nn, Xtest)

  errorTrmpg <- trainPred$preds[,1] - Ttrain[,1]
  errorTsmpeg <- testPred$preds[,1] - Ttest[,1]

  errorTraceTrmpg <- c(errorTraceTrmpg, sqrt(mean(errorTrmpg^2)))
  errorTraceTsmpeg <- c(errorTraceTsmpeg, sqrt(mean(errorTsmpeg^2)))

  errorTrhp <- trainPred$preds[,2] - Ttrain[,2]
```

```

errorTshp <- testPred$preds[,2] - Ttest[,2]

errorTraceTrhp <- c(errorTraceTrhp, sqrt(mean(errorTrhp^2)))
errorTraceTshp <- c(errorTraceTshp, sqrt(mean(errorTshp^2)))

matplot(seq(2,nh,by=3),cbind(errorTraceTrmpg,errorTraceTsmpeg),col=c("Red",
"Blue"),lty=1,type="l",lwd=2,xlab="Number of hidden units",ylab="RMSE",
main="Train and Test mpg RMSE vs. Number of Hidden Units")
legend("center",c("Train mpg RMSE","Test mpg
RMSE"),col=c("Red","Blue"),lty=1,lwd=2,bty="n")

matplot(seq(2,nh,by=3),cbind(errorTraceTrhp,errorTraceTshp),col=c("Red",
"Blue"),lty=1,type="l",lwd=2,xlab="Number of hidden units",ylab="RMSE",
main="Train and Test Horsepower RMSE vs. Number of Hidden Units")
legend("center",c("Train Horsepower RMSE","Test Horsepower
RMSE"),col=c("Red","Blue"),lty=1,lwd=2,bty="n")
}

```

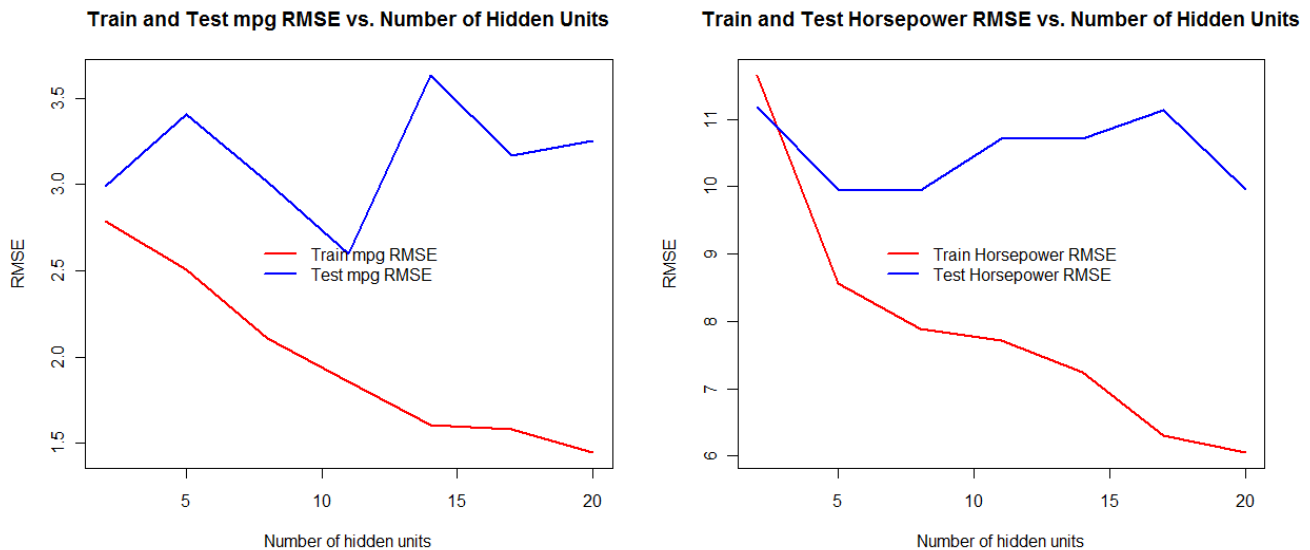


Figure 14: RMSE versus number of hidden units for mpg and horsepower for both training and testing data

The final experiment shows the effect of different lambdas on the RMSE, this done twice, the first time with small lambda values where the results are shown in figure 15, and the second time with larger lambda values with results shown in figure 16. Both figures show that the best results are with lambda equal or very close to zero. The code used to plot figures 15 or 16 is almost the same as the one used in figure 14, except that this time, lambda is changing and the number of hidden unit is chosen to be 10 where the testing data showed its best results around that number of hidden unit.

```

lams <- NULL
for (lambda in c(0, 0.1, 0.5, 1, 2)) {

  lams <- c(lams,lambda)
  nn <- makeNN(Xtrain,Ttrain,nh, lambda) #nh is number of hidden units desired

  trainPred <- useNN(nn, Xtrain)
}

```

```

testPred <- useNN(nn, Xtest)

errorTrmpg <- trainPred$preds[,1] - Ttrain[,1]
errorTsmgp <- testPred$preds[,1] - Ttest[,1]

errorTraceTrmpg <- c(errorTraceTrmpg, sqrt(mean(errorTrmpg^2)))
errorTraceTsmgp <- c(errorTraceTsmgp, sqrt(mean(errorTsmgp^2)))

errorTrhp <- trainPred$preds[,2] - Ttrain[,2]
errorTshp <- testPred$preds[,2] - Ttest[,2]

errorTraceTrhp <- c(errorTraceTrhp, sqrt(mean(errorTrhp^2)))
errorTraceTshp <- c(errorTraceTshp, sqrt(mean(errorTshp^2)))

matplot(lams,cbind(errorTraceTrmpg,errorTraceTsmgp),col=c("Red","Blue"),
        lty=1,type="l",lwd=2,xlab="lambda",ylab="RMSE", main= "Train and Test mpg
        RMSE vs. lambda")
legend("center",c("Train mpg RMSE","Test mpg
        RMSE"),col=c("Red","Blue"),lty=1,lwd=2,bty="n")

matplot(lams,cbind(errorTraceTrhp,errorTraceTshp),col=c("Red","Blue"),lty=1,
        type="l",lwd=2,xlab="lambda",ylab="RMSE", main="Train and Test Horsepower
        RMSE vs. lambda")
legend("center",c("Train Horsepower RMSE","Test Horsepower
        RMSE"),col=c("Red","Blue"),lty=1,lwd=2,bty="n")
}

```

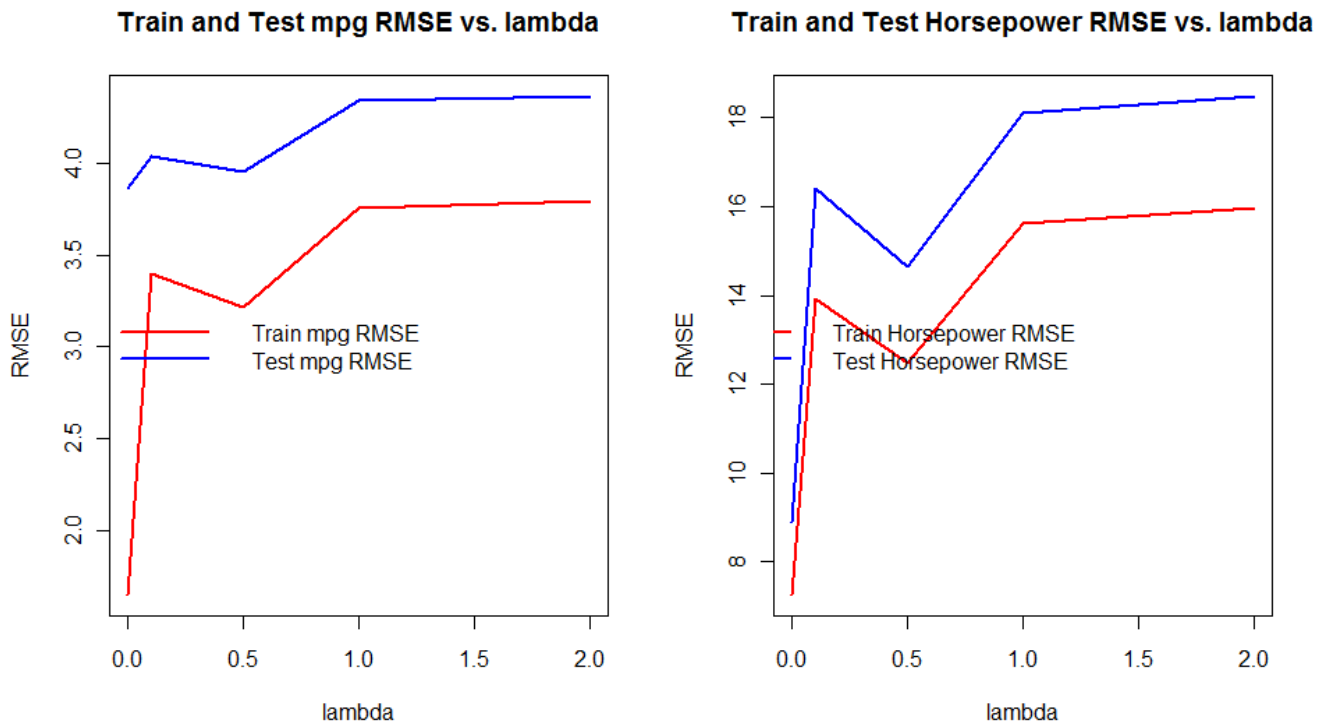


Figure 15: RMSE with small values of lambda

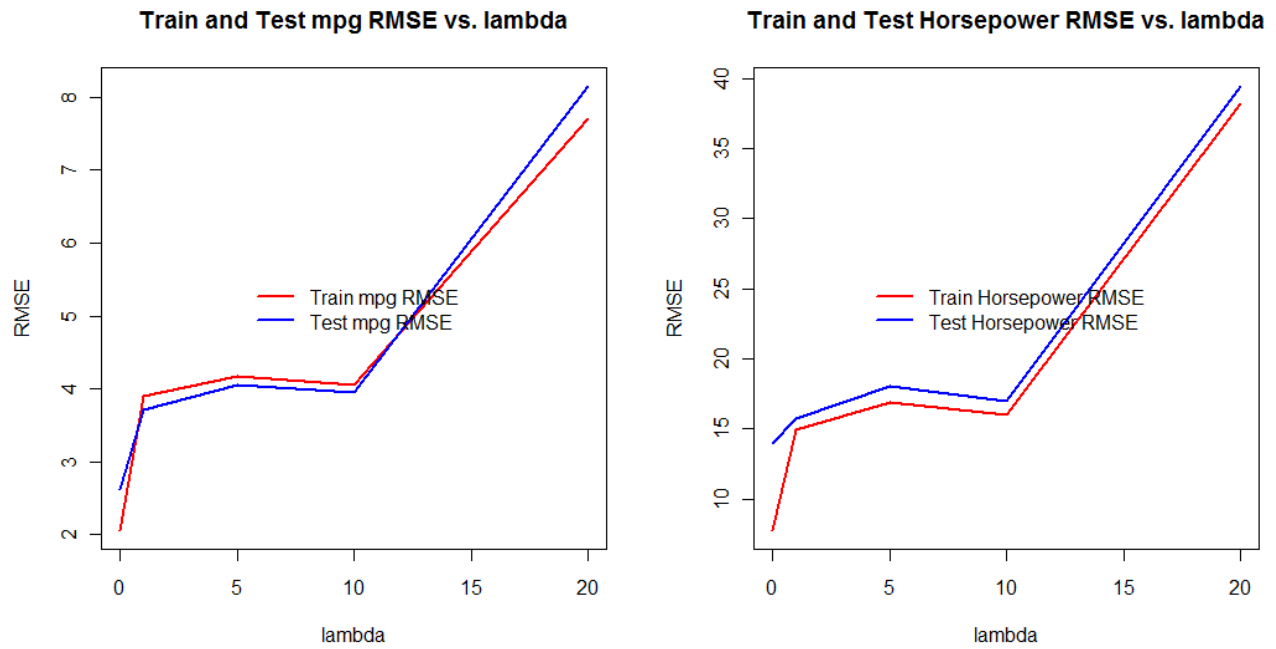


Figure 16: RMSE with larger values for lambda

6 more Results With Another Cost Functions

Another experiment using the function:

$$f(x) = x/\sqrt{1+x^2}$$

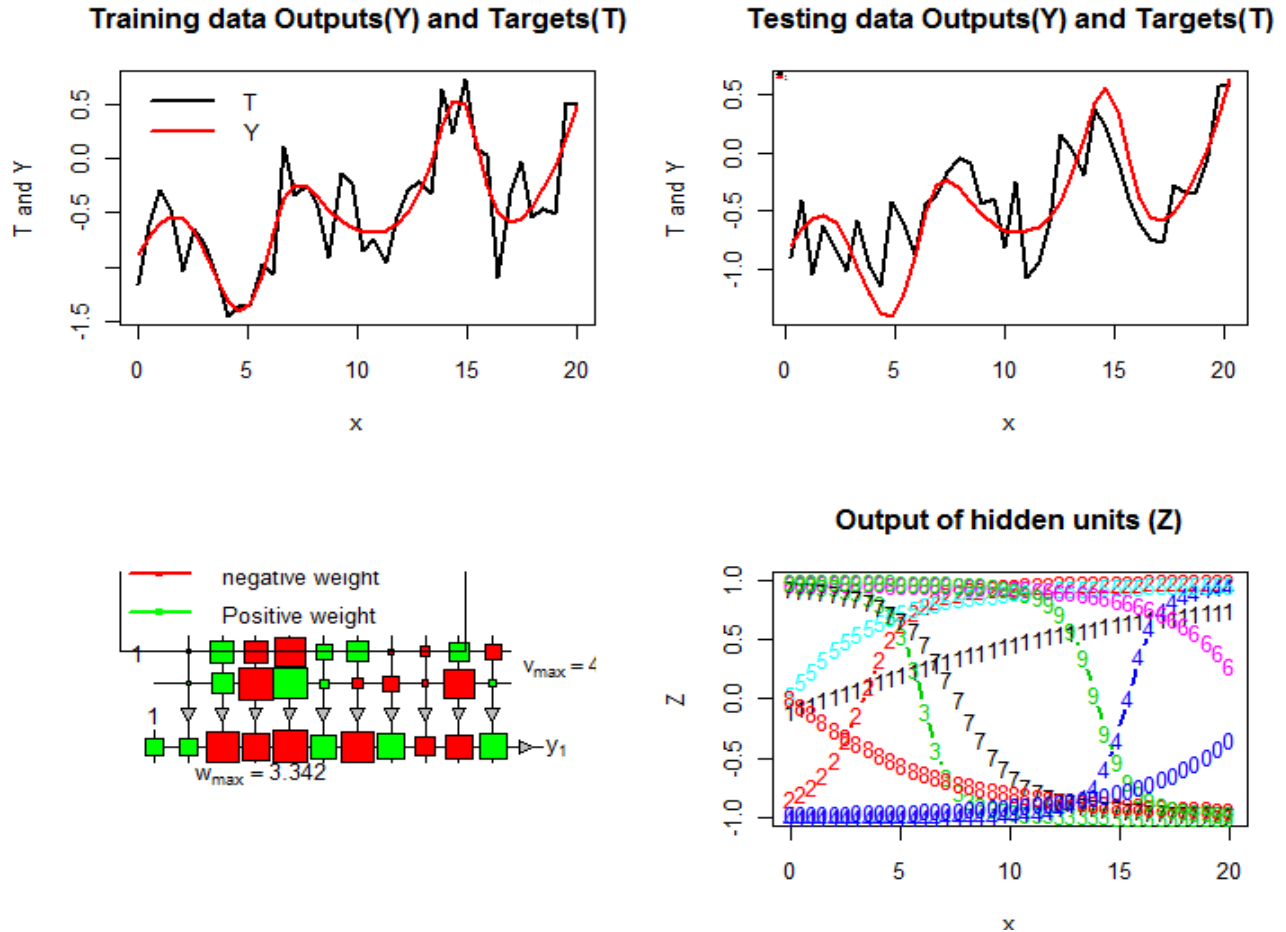


Figure 17: Results of using the function $f(x) = x/\sqrt{1+x^2}$ instead of \tanh

7 Conclusions

Neural Networks are adaptive tools that can provide complex models to fit data. The experiments on the the noisy sin curve show excellent results for the training data, but only fair results on the testing data. On the other hand the results on the mpg data set show very good results for both training and testing data. The number of hidden units can decrease the RMSE significantly, specially for the training data, however it has less effectiveness on the testing data, where most of the experiments show that using more than 5 hidden units did not really improve the prediction accuracy for the testing data of the noisy sine curve, and for the mpg data set the number was about 10 hidden units. Also the experiments on the noisy curve sine and mpg data set sample show that the best values for lambda are those that are very close to zero.

References:

- [1] Website: http://en.wikipedia.org/wiki/Hyperbolic_function
- [2] Website: <http://archive.ics.uci.edu/ml/>