

CS545: Assignment 5 - Nonlinear Regression with Neural Networks

Carlos Oliveira

November 2, 2009

Contents

1	Introduction	2
2	Training a Neural Network with Scaled Conjugate Gradient	3
2.1	Functions and procedure - makeNN()	3
2.1.1	Functions pack() and unpack()	4
2.1.2	Function sqErrorF()	4
2.1.3	Function gradF()	5
2.2	Functions and procedure - unStdize() and useNN()	5
2.2.1	Function unStdize()	5
2.2.2	Function useNN()	6
2.3	Training with small samples	6
2.3.1	A parabola	6
2.3.2	A modified linear function	6
2.3.3	A sinusoidal	6
3	Noisy Sine Curve	9
3.1	<i>RMSE</i> for the training data vs. number of epochs	9
3.2	Outputs and weights	9
3.3	<i>RMSE</i> vs. hidden units	11
4	Automobile Miles-Per-Gallon Data	15
4.1	Automobile Miles-per-gallon and horsepower data	15
4.2	Lambdas and hidden units	16
5	Extra Questions	17
5.1	Computational time: steepest() versus scg()	17
5.2	Modeling with functions other than $\tanh(x)$	17
6	Conclusions	20
7	Notes and Special Thanks	21

1 Introduction

This assignment [1] of the CS545 - Machine Learning class [2] is divided in 4 parts. The first part has the modeling and training of a neural network, using the $\tanh()$ function for the hidden units (hu) and using “scg” - scaled conjugate gradient. Then I applied the neural network model (NN) to a couple of sample data functions such as an sinusoidal curve, a parabola and a modified linear function that I will explain later.

In the second part, I applied the NN to predict the values of a noisy sinusoidal curve. Then I calculated the *RMSE* as a function of the number of epochs. I also show the predictions and hidden units outputs and “(V,W)” weights in this part.

The third part has the analysis of predicting horsepower and miles-per-gallon from a given automobiles data base from the UCI Machine Learning data repository. The analysis here concentrated in observing the effects of λ and number of hidden units to find the minimum *RMSE*.

The last part addresses 2 of the extra questions proposed. The first one is to compare the execution time when using the $\text{scg}()$ or the $\text{steepest}()$ functions to train the neural network. The second question that I am addressing is about the use of a function other than $\tanh()$ in the hidden units. I tried 2 different functions: $\sin(x)$ and $\text{sigmoid}(x)$.

2 Training a Neural Network with Scaled Conjugate Gradient

The first section of this assignment is about Training a Neural Network with Scaled Conjugate Gradient. For that I had to create new functions and have the neural network to be trained using the “Scaled Conjugate Gradient algorithm” [3].

2.1 Functions and procedure - makeNN()

Basically this is how I will be training and using the neural network:

1. Train the neural network by using makeNN and passing Xtrain, Ttrain and number of hidden units to it. Other parameters are optional. The function signature is:

```
makeNN( Xtrain, Ttrain, numHiddenUnits, lambda, nIterations, xPrecision, fPrecision)
```

2. Make predictions by passing the neural networks weights (V and W) returned from the makeNN function and the test data to be predicted (Xtest) to the useNN function. The useNN signature is:

```
useNN( nnet, Xtest)
```

Following is the structure of the makeNN function, with its sub-functions. As mentioned, the function signature is:

```
makeNN( Xtrain, Ttrain, numHiddenUnits, lambda, nIterations, xPrecision, fPrecision)
```

The last four arguments to makeNN should be optional with default values of:

```
lambda=0, xPrecision=0, fPrecision=0 and nIterations=10000
```

Find below the structure of the makeNN function.

```
makeNN <- function(Xtrain, Ttrain, numHiddenUnits,
                  ..., lambda=0, nIterations=10000, xPrecision=0, fPrecision=0){
  ni <- ncol(Xtrain)
  no <- ncol(Ttrain)
  nh <- numHiddenUnits

  V <- matrix(0.1*(runif((ni+1)*nh)-0.5), ni+1,nh)
  W <- matrix(0.1*(runif((nh+1)*no)-0.5), nh+1,no)

  stdXtrain <- makeStandardizeF(Xtrain)
  Xtrain <- cbind(1,stdXtrain(Xtrain))
  unStd <- unStdize(Ttrain)

  stdTtrain <- makeStandardizeF(Ttrain)
  Ttrain <- stdTtrain(Ttrain)
```

The section above is the preparation to train the network. The initial matrices V and W are created, the data is standardized (Xtrain and Ttrain), but before standardizing Ttrain, I called the unStdize function in order to calculate the mean and sigma (μ and σ) of the Ttrain data to be used when unstandardizing the predictions for Xtest.

```
pack <- function(V,W){...}
unpack <- function(weights) {...}
sqErrorF <- function(weights) { ... }
gradF <- function(weights) { ... }
```

Still inside the makeNN function, three functions are defined. The pack(), unpack(), sqErrorF() and the gradF(). Above you see just the definition and signatures of each function. I will explain and detail them in the following sections.

```

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

VW <- unpack(scgResult$x)
list(V=VW$V,W=VW$W,unStd=unStd)

} # end of makeNN

```

Finally, in the last part of the makeNN function the call to scg will be make. The result will be unpacked and the makeNN function will return V and W. Also, makeNN will return the function to be used with the predictions results from the useNN().

2.1.1 Functions pack() and unpack()

These 2 functions below do not require a lot of explanation. The pack() function take the weights matrices V and W and return a columns vector. The unpack() function does the opposite. It converts the column vector weights into matrices and returns a list containing V and W. Notice that it converts the weights vector back to matrices by using ni, no and nh, which are number of inputs, outputs and hidden units respectively.

```

pack <- function(V,W){
  matrix(c(V,W))
} # end of pack

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

```

2.1.2 Function sqErrorF()

The sqErrorF() function unpacks the weights that is passed to it, calculates Z and Y via a forward pass through the network and returns the mean square error. This is implemented with the penalty. When we do not want to apply the penalty to the calculation, all we need to do is pass lambda=0 as argument (actually, lambda is default to 0).

```

sqErrorF <- function(weights) {
  # Calculate Y using the fwdPass() function
  # first unpacks V and W from weights
  VW <- unpack(weights)

  # Forward pass
  Z <- tanh(Xtrain %*% VW$V)
  Y <- cbind(1,Z) %*% VW$W
  # calculates and returns the mean square error
  Vmtx <- matrix(V,1)
  penalty <- lambda * (Vmtx%*%t(Vmtx))

  mean((Y - Ttrain)^2) + penalty
} # end of sqErrorF

```

2.1.3 Function gradF()

The gradF() function takes the weights column matrix as an argument, unpacks it, calculate the Y output of the network and errors calculate the gradients of V and W, returning them as a column matrix using the function pack().

```
gradF <- function(weights) {
  # first unpacks V and W from weights
  VW <- unpack(weights)

  # Forward pass
  Z <- tanh(Xtrain %*% VW$V)
  Y <- cbind(1,Z) %*% VW$W

  # calculates all errors
  errors <- (Y-Ttrain)

  # In the gradF function, N is nrow(Xtrain) and K is ncol(Ttrain).
  # - Chuck Anderson 2009/10/22 19:42
  NK <- nrow(Xtrain)*nrow(Ttrain)

  tmp <- V[-1,,drop=FALSE]
  V0 <- rbind(rep(0,ncol(tmp)),tmp)
  # gradients
  gradV <- 1/(NK) * t(Xtrain) %*% (errors %*% t(W[-1,,drop=FALSE]) * (1-Z^2)) + lambda * V0
  gradW <- 1/(NK) * t(cbind(1,Z)) %*% errors

  pack(gradV,gradW)
} # end of gradF
```

2.2 Functions and procedure - unStdize() and useNN()

2.2.1 Function unStdize()

The predictions output from the network is standardized. Therefore, for meaningful results, we need to unstandardize it. Given a dataset X, Z is the standardized function calculated with the following formula:

$$Z = \frac{(X - \mu)}{\sigma}$$

From that, we can calculate the unstandardized matrix by using:

$$X = Z * \sigma + \mu$$

This is implemented in the following R-code function:

```
unStdize <- function(X){
  # X is from where we will figure out the mu and sigma
  mu <- colMeans(X)
  sigma <- sd(X)

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

2.2.2 Function useNN()

Finally we got to the meat of all this: the predictions! I mean, all of the describe above was the hard work. Now it comes the fun part.

```
useNN <- function(nnet, X){
  V <- nnet$V
  W <- nnet$W

  standardizeX <- makeStandardizeF(X)

  X <- cbind(1,standardizeX(X))
  Z <- tanh(X %*% V)
  predY <- cbind(1,Z) %*% W # predictions

  nnet$unStd(predY)
}
```

The useNN() function above extracts the matrices V and W from the nnet model, standardize the input matrix X does a forward pass in it. The predictions are in the predY variable which will be unstandardized by the function unStd(). Notice that V, W and unStd come from the nnet model created with makeNN() function.

2.3 Training with small samples

In this section create some examples of Xtrain and Ttrain. I used 3 data sets and trained the neural network for each one of them. All the examples below were run with 3 hidden units and 10000 epochs. See the results in the following sections.

2.3.1 A parabola

The first data set is the parabola described by the following R function:

```
f <- function (x) x^2 - 20*x + 75
```

See on figure 1 that the predictions for the training and the testing set worked very well. For the training set, the errors are smaller than 1% and for the testing data, the error is less than 7%.

2.3.2 A modified linear function

The second example uses a modified linear function, that is composed by 2 linear functions, one ascendent and other descendent.

```
f <- function (x) abs(10-x)
```

Figure 2 shows the results. A quick analysis of this chart shows that the error is less than 5% in the Xtrain range.

2.3.3 A sinusoidal

I decided to use a sinus function for the third and last example.

```
f <- function (x) sin(x)
```

Initially I used just 3 hidden units as in all examples before. However, figure 3 shows that 3 hidden units do not give a very good prediction.

However, as expected, we will get much better results when using 5 hidden units. See figure 4 for the results. Eventhough the prediction is better, a quick analysis of this figure shows that the error is smaller than 10% for the training set and probably little bit greater than 30%.

My errors estimates in these sections are not accurate, but these are just some examples to find out whether or not the neural network model is behaving well or not. It seems that the model is working well.

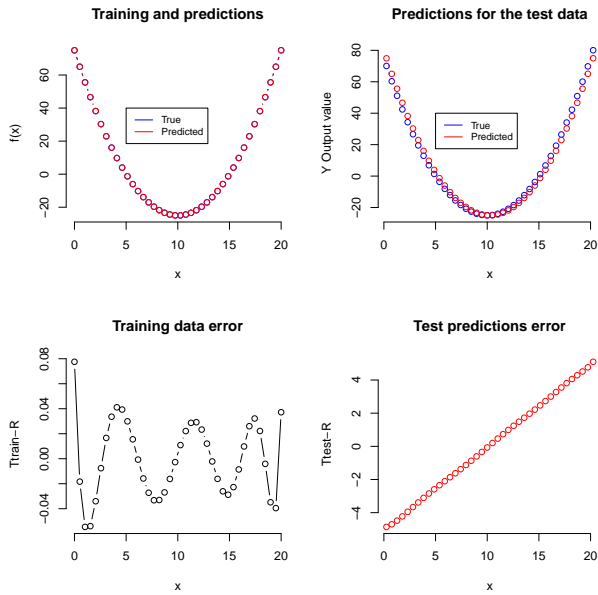


Figure 1: Training and Predictions for $f(x) = x^2 - 20 \cdot x + 75$, using 3 hidden units and 10000 iterations

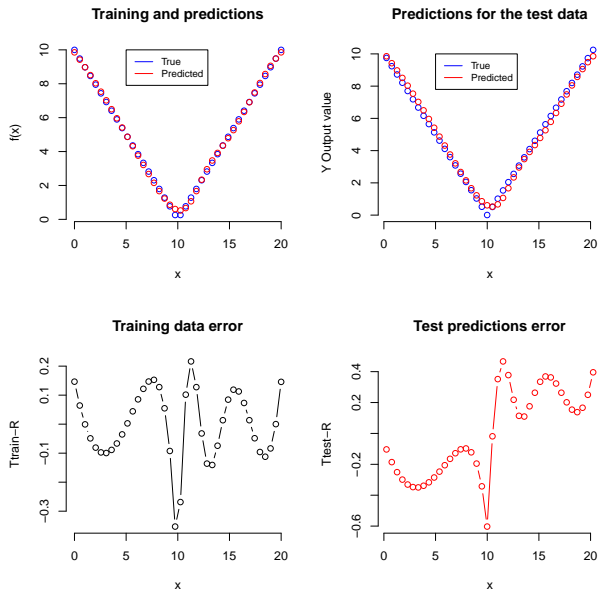


Figure 2: Training and Predictions for $f(x) = abs(10 - x)$, using 3 hidden units and 10000 iterations

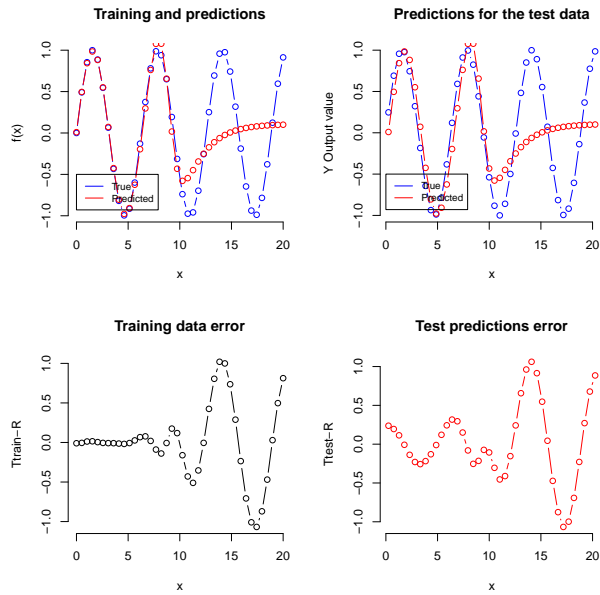


Figure 3: Training and Predictions for $f(x) = \sin x$, using 3 hidden units and 10000 iterations

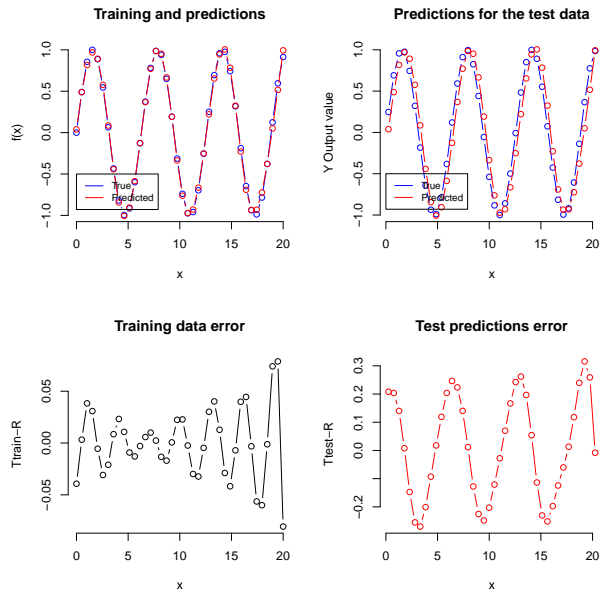


Figure 4: Training and Predictions for $f(x) = \sin x$, using 5 hidden units and 10000 iterations

3 Noisy Sine Curve

In this section I will model the neural network to predict the output of the following noisy function:

$$f(X) = -1 + 0.05 \cdot X + 0.4 \cdot \sin(X) + Noise$$

And Noise is given by:

$$Noise = 0.3 \cdot rnorm(length(X))$$

For this exercise, the neural network will have 10 hidden units and $\lambda=0$ so the model is applied with no magnitude penalty. See results in the following sections.

3.1 RMSE for the training data vs. number of epochs

The R-code used to calculate the *RMSE* and plot it is found 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)

errorTrace <- NULL
nReps <- 200000
for (reps in 1:nReps) {
  myNNet <- makeNN(Xtrain,Ttrain,10,nIterations=reps,lambda=0,ftracep=TRUE)
  results <- useNN(myNNet,Xtrain)
  error <- results - Ttrain
  errorTrace <- c(errorTrace, sqrt(mean(error^2)))
}
```

Figure 5 has the plot of final results.

Unfortunately the process of running the code below was taking too long. I decide to run 2000 points of 10 epochs each. Therefore figure 6 shows the *RMSE* for epochs from 10 to 20,000 in increments of 10. We can also observe that the *RMSE* values do not improve much as we increase the number of epochs beyond 500. It seems that there is a average (or probably median) value around 0.18.

3.2 Outputs and weights

In this section, I plotted the output of the hidden units (*Z*) as a function of the input *X*. I used the neural network with 10 hidden units and 10,000 epochs. In order to plot the intermediate values, I had to change the `useNN()` to have the values of *Z* returned with the predictions *Y*. The new function is called `useNN_ZY()` See the modified code below:

```
useNN_ZY <- function(nnet, X){
  V <- nnet$V
  W <- nnet$W

  standardizeX <- makeStandardizeF(X)

  X <- cbind(1,standardizeX(X))
  Z <- tanh(X %*% V)
  predY <- cbind(1,Z) %*% W # predictions

  list(Z=Z,Y=nnet$unStd(predY))
}
```

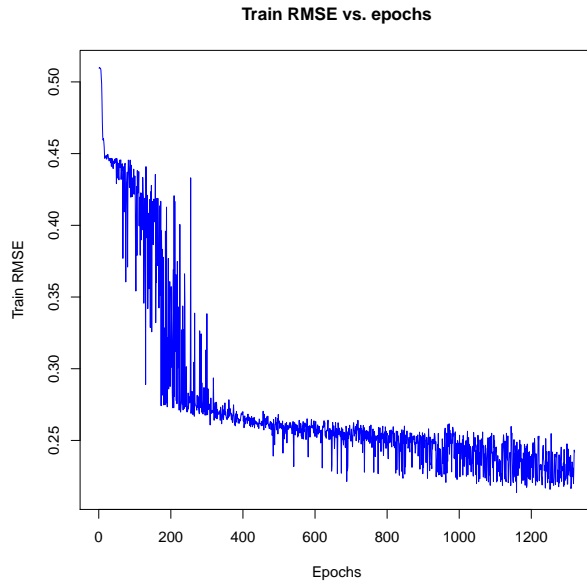


Figure 5: RMSE using a neural network with 10 hidden units and up to 1200 iterations

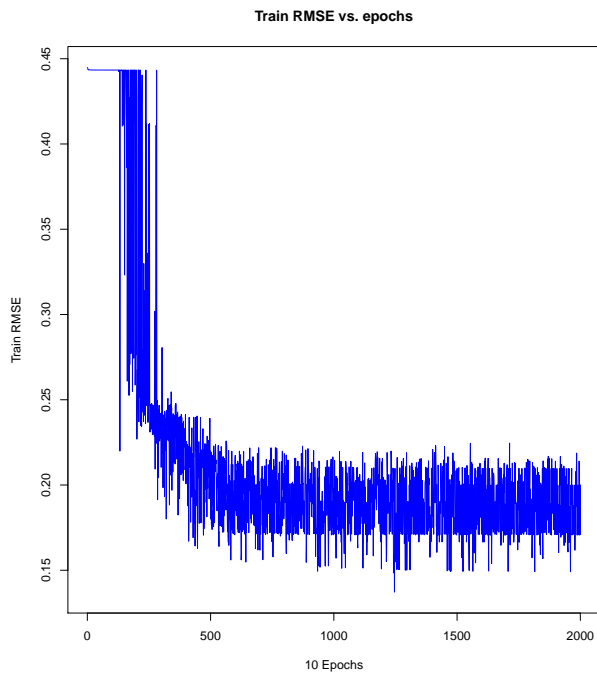


Figure 6: RMSE, 10 hidden units, from 10 to 20,000 epochs, increments of 10

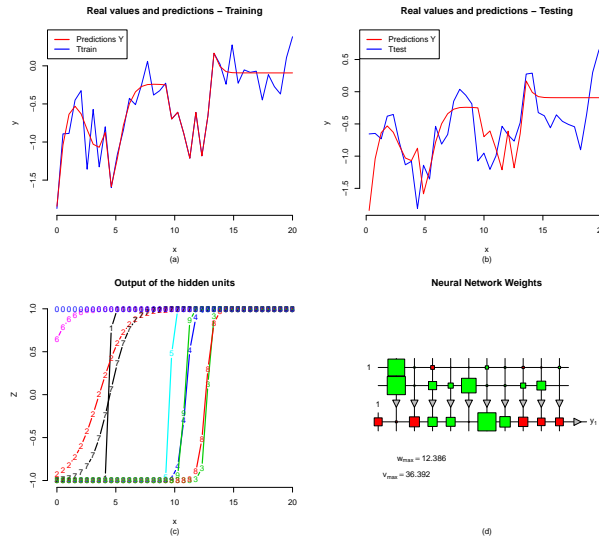


Figure 7: (a) Output of predictions and true values for the training data; (b) Output of predictions and true values for the testing data; (c) Output of 10 hidden units; (d) Diagram of neural network weights.

Figure 7 is divided in 4 parts as described below. It shows the results when using 10 hidden units.

- (a) Output of training data target with output of neural network given training data input.
- (b) Output of testing data target with output of neural network given testing data input.
- (c) Output of hidden units for range of x values from 0 to 20.
- (d) Diagram of neural network weights.

I was curious to see how the output would be for 5 and 20 units. See the results for 5 and 20 hidden units on figures 8 and 9. It is very interesting to observe that indeed increasing the number of hidden units improves our predictions. See how well the training data was predicted when we used 20 hidden units (figure 9).

3.3 RMSE vs. hidden units

In this section I collect train and test RMSE for different numbers of hidden units, from 1 to 40. Both plots can be found on figure 10.

One thing that I did different from what the assignment had asked is to average the error for 100 samples. I thought that using just one sample was not very robust. See the results using the average values on figure 11.

Find below the R-code used to generate the plots on figures 10 and 11.

```

errorTrain <- NULL
errorTest <- NULL
errorTrainMtx <- NULL
errorTestMtx <- NULL
averageIndex <- 100
hiddenUnits <- 40
parOri <- par(mfcol=c(1,2),mar=c(5,4,4,2),bty="n")

for (huNumber in 1:hiddenUnits) {
  myNNnet <- makeNN(Xtrain,Ttrain,huNumber,
                    nIterations=10000,lambda=0,ftracep=FALSE)
}

```

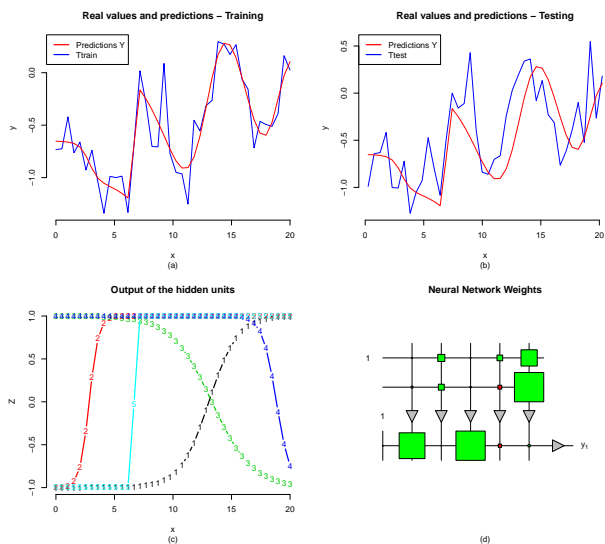


Figure 8: (a) Output of predictions and true values for the training data; (b) Output of predictions and true values for the testing data; (c) Output of 5 hidden units; (d) Diagram of neural network weights.

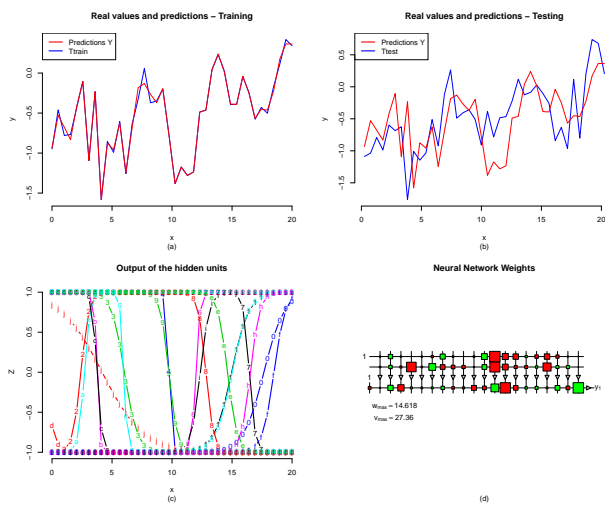


Figure 9: (a) Output of predictions and true values for the training data; (b) Output of predictions and true values for the testing data; (c) Output of 20 hidden units; (d) Diagram of neural network weights.

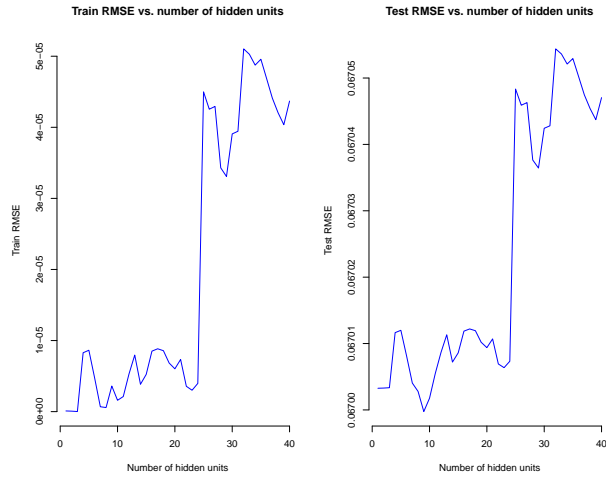


Figure 10: RMSE for the training and testing sets vs. number of hidden units

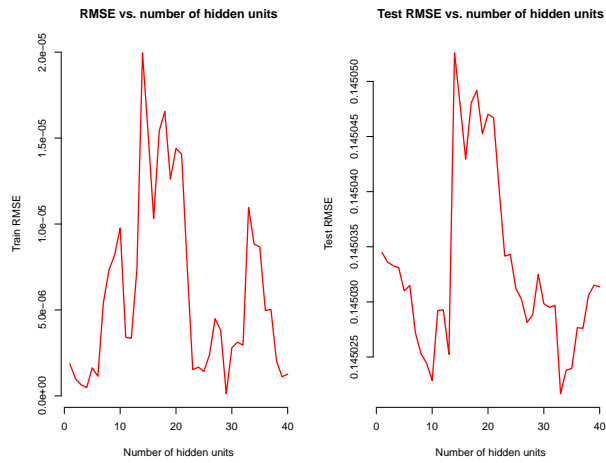


Figure 11: Average RMSE for the training and testing sets vs. number of hidden units

```

for (reps in 1:averageIndex){
# predicting train
results <- useNN(myNNet,Xtrain)
errorTrainMtx <- rbind(errorTrainMtx,results - Ttrain)
  results <- useNN(myNNet,Xtest)
errorTestMtx <- rbind(errorTestMtx,results - Ttest)
}

error <- mean(errorTrainMtx)
errorTrain <- c(errorTrain, sqrt(mean(error^2)))

# predicting train
error <- mean(errorTestMtx)
errorTest <- c(errorTest, sqrt(mean(error^2)))
}

```

Observe that, on figure 11 the minimum RMSE happens in between 30 to 40 units when we average the RMSE values. However, when running a single value, we would be inclined to believe that as we increase the number of hidden units much more beyond 20 units, the prediction errors would increase.

4 Automobile Miles-Per-Gallon Data

In this section, I used the neural network model created from the previous sections to predict miles-per-gallon (mpg) and horsepower (hp) for the cars from the automobile miles-per-gallon data set from the UCI Repository we worked with earlier in the semester. The R-code used for these predictions is described in the following sections.

4.1 Automobile Miles-per-gallon and horsepower data

First of all, I believe that for this assignment we are not interested on how the data was downloaded from the UCI repository, how the non-numeric arguments were removed or how the data was split, as this was part of previous assignment. I will report only the work specifically related to the neural networks models and algorithms, object of this report. The only thing I need to mention is that the automobile data base was split in 80% for the training set and 20% for the testing set, as in the previous assignment.

Find below the R-code used to find the lambda and number of hidden units that will give us the “best” errors and predictions.

```
1: hiddenUnits <- c(5,10,15,20)
2: Lambdas <- c(0,.25,1,5,10)
3: epochs <- 10000
4: for (lambda in Lambdas){
5:   colNumber <- colNumber + 1
6:   rowNumber <- 0
7:   for (huNumber in hiddenUnits){
8:     # Training the network
9:     rowNumber <- rowNumber + 1
10:
11:     cat("Calculating for hidden:",huNumber," and lambda:",lambda, ".\n")
12:
13:     myNNet <- makeNN(Xtrain,Ttrain,huNumber,nIterations=epochs,lambda=lambda,ftracep=FALSE)
14:
15:     for (avgs in 1:10){
16:       # predicting train
17:       results <- useNN(myNNet,Xtrain)
18:       errorTrainMtx <- rbind(errorTrainMtx,results - Ttrain)
19:
20:       # predicting test
21:       results <- useNN(myNNet,Xtest)
22:       errorTestMtx <- rbind(errorTestMtx,results - Ttest)
23:     }
24:
25:     # Avg Error Train
26:     error <- colMeans(errorTrainMtx)
27:     errorTrainMPG[rowNumber,colNumber] <- sqrt(mean(error["mpg"]^2))
28:     errorTrainHP[rowNumber,colNumber] <- sqrt(mean(error["horsepower"]^2))
29:
30:     # Avg Error test
31:     error <- colMeans(errorTestMtx)
32:     errorTestMPG[rowNumber,colNumber] <-sqrt(mean(error["mpg"]^2))
33:     errorTestHP[rowNumber,colNumber] <- sqrt(mean(error["horsepower"]^2))
34:
35:   } # end for huNumber
36: } # end for Lambda
```

This is the summary of the code presented above:

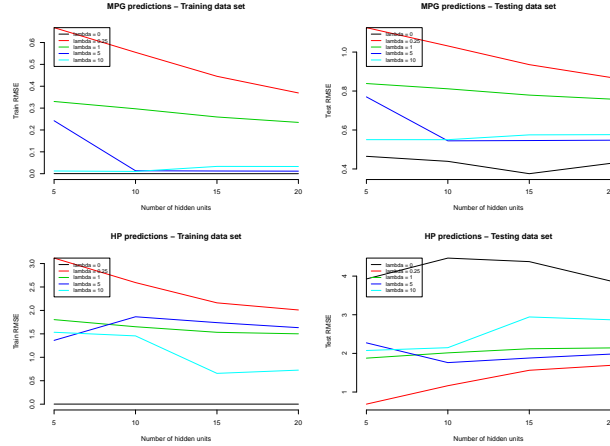


Figure 12: Training and Testing $RMSE$ for mpg and hp predictions for different values of λ and hidden units.

mpg	horsepower
Min. : 9.00	Min. : 46.0
Mean :23.26	Mean :105.9
Max. :46.60	Max. :230.0

Table 1: Minimum, Maximum and Mean values of horsepower and miles-per-gallon of the training data set.

- There is an outer for loop to generate the lambda values - line 4.
- The first inner for loop is responsible to generate the calculation - line 7.
- The network model (weights from the neural network) is created in line 13.
- Lines 15 to 23 show a for loop where I am averaging the predictions from 10 readings. I could have done 100, but this costs more computational time and therefore I leave this as a suggestion only.
- Finally, lines 25 to 33 show the calculation of the $RMSE$ for the training and testing set for the HP and MPG predictions. I stored these values in some matrix to later generate the charts found in figure 12
- It is also worth to notice that on lines 26 and 31 I used the `colMeans`, which is different from the previous sections, because now we have 2 outputs in our prediction model, Y_1 and Y_2 being mpg and hp .

4.2 Lambdas and hidden units

On figure 12 we have the $RMSE$ curves for different values of λ and different numbers of hidden units.

These charts seem to indicate that the best option is $\lambda = 0$ for all but the horsepower predictions in the testing set. $\lambda = 5$ or $\lambda = 10$ would be the next best choice, but again, the hp predictions on the testing set do not seem to agree either. Having said that, we could use $\lambda = 0$ with 15 hidden units for all predictions on testing and training but the hp predictions where we would use $\lambda = 5$ with 5 hidden units.

We have to be careful also to look at the magnitude of the errors. Notice that for the training sets, either HP or MPG, the errors $RMSE$ are in the order of 10^{-3} . It is also always a good idea to look at the absolute values that we are trying to predict to have a notion of the maximums and minimums of that data set. See table 1.

Epochs	Hidden Units	scg() time	steepest() time	Ratio
1000	10	0.92	2.25	2.45
10000	10	9.06	22.51	2.49
10000	20	15.34	39.63	2.58
10000	40	28.54	74.90	2.62
50000	10	46.86	110.57	2.40

Table 2: Computational time comparison between the `scg()` and `steepest()` functions

5 Extra Questions

5.1 Computational time: `steepest()` versus `scg()`

I have made some modifications to the `makeNN()` function signature in order to have it running either `scg()` or `steepest()`. See the modifications below:

```
makeNN <- function(Xtrain, Ttrain, numHiddenUnits,
                  ..., lambda=0, nIterations=10000, xPrecision=0, fPrecision=0,
                  ftracep=FALSE,
                  steepest = FALSE # flag to use either steepest or scg
                  )
```

Notice that the `makeNN()` signature now takes a *steepest* parameter which is set to “FALSE”. The I added the if statement to choose from `steepest()` or `scg()` as shown below:

```
if(steepest){
  cat("Running stp()\n")
  scgResult <- steepest(pack(V,W),sqErrorF,gradF,nIterations=nIterations,
                       xPrecision=xPrecision,fPrecision=fPrecision)
} else {
  cat("Running scg()\n")
  scgResult <- scg(pack(V,W),sqErrorF,gradF,nIterations=nIterations,
                  xPrecision=xPrecision,fPrecision=fPrecision)
}
```

Also, I did not use the `useNN()` function to predict the data, but I have just run the `makeNN()` using different numbers of epochs and hidden units. I used the same data from the previous section, i.e. the automobile mpg data set.

See table 2 for the results. The `steepest()` algorithm was around 2.5 times slower than the `scg()` one for all runnings. It did not matter if I used 1000 or 10000 or 5000 epochs with 10, 20, or 40 hidden units, the `scg()` algorithm always won the race.

5.2 Modeling with functions other than $\tanh(x)$

First, I created the function `sigmoid()` as in below:

```
sigmoidF <- function(X){
  1/(1+exp(-X))
}
```

The next step was to replace all Z calculations as in:

```
Z <- tanh(Xtrain %*% V)
```

with:

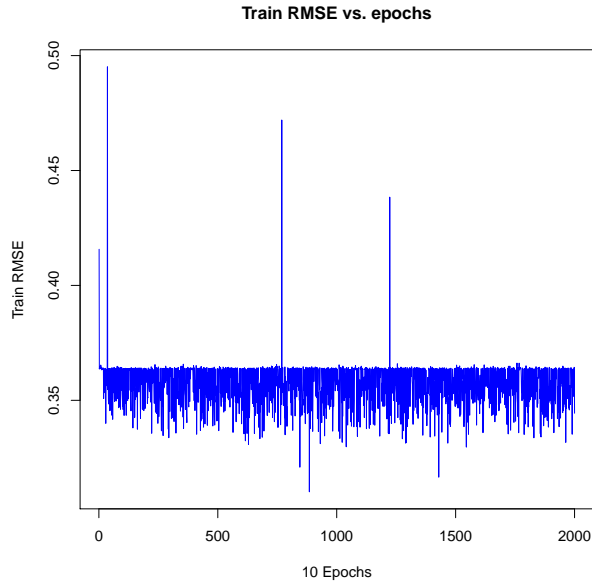


Figure 13: *RMSE* using a sigmoid function for predictions

sigmoid RMSE	
Min.	0.3102
1st Qu.	0.3509
Median	0.3589
Mean	0.3567
3rd Qu.	0.3638
Max.	0.4951

Table 3: Summary of the *RMSE* values when using a sigmoid() function in the hidden units

```
Z <- sigmoidF(Xtrain %*% V)
```

These are the only differences I have made to the original code. After that, I have just called the `makeNN()` functions using 10 hidden units and 10000 epochs. I applied the model on the “noisy sinus” function and compared the *RMSE* values. The results can be found on figure 13.

The *RMSE* values are around 0.35 and then do not vary a lot depending on the number of epochs. This is a very interesting result. There are some spikes in the chart but the average will be very close to this value. So, comparing with our previous usage of the $\tanh(x)$ function, it seems that the sigmoid does not vary as much. Just to verify what the limits are on the *RMSE* I printed the table ???. We can verify that *RMSE* is between 0.31 and 0.49. However, if we compare these results to the ones on figure 6, we can observe that the model using the $\tanh(x)$ function has smaller error and variance for epochs greater than 500.

I also plotted the results of using the sigmoid function against the number of hidden units.

And just for my curiosity, I created another model using $\sin(x)$, trained the network and plotted the error results against the number of epochs. The results can be found at figures 14 and 15. By comparing the charts on figure 13 and 15 we can say that although the model using the $\sin(x)$ function has a bigger error value, it is still doing very well as it has its values around 0.35. The reason for the line on figure 15 to be so straight is that I calculated only for 100, 500, 1000, 5000, 10000, 20000 epochs. On figure 13 I used values from 1 to 20000.

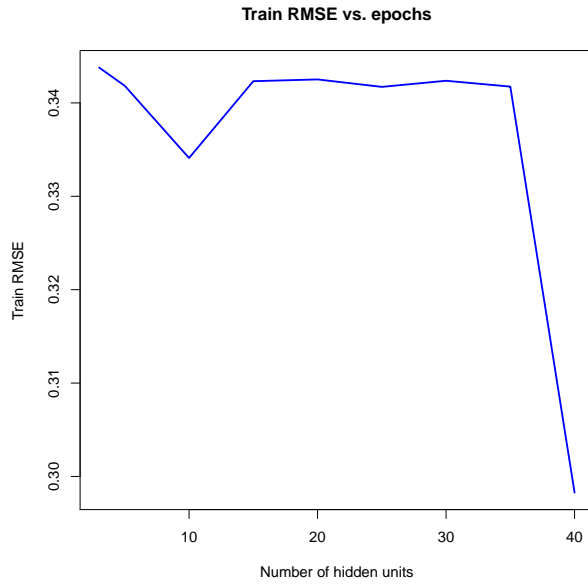


Figure 14: $RMSE$ as a function of the number of hidden units, using sigmoid()

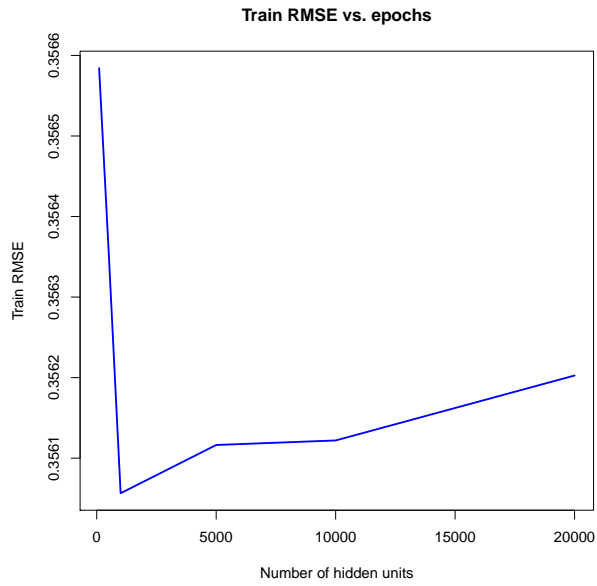


Figure 15: $RMSE$ versus epochs, using function $\sin(x)$ in the hidden units

6 Conclusions

Neural network is very interesting. I struggled a little bit with understanding how to put the `scg()` implementation to work with the `makeNN()` and `useNN()`. This is what consumed most of my time for this work. After this was done, using the network was fun. The downside is that after I started testing some data, training it for new functions, it was hard to stop. I always wanted to test with other ideas. What I mean to say here is that neural network is exciting and I did not want to stop after it was working well.

Regarding each previous section, all the conclusions and comments are embedded in them.

One thing that I would like to try is to print the output of the hidden functions when using $\sin(x)$ inside the hidden units. The reason would be to compare with the Fourier transformation analysis where we can describe any periodic signal (or data) as a Sum of $\sin(x)$ and $\cos(x)$. I do not know how related this would be but I leave this as a suggestion for future implementation for now.

7 Notes and Special Thanks

- The Training section was the most difficult one in terms of implementing the NN using the scg and writing the gradient function. After I was done, it became clear, but it took me a while to get there.
- The other sections were a lot of work, but fun. I could not avoid trying other possibilities as described in this report, such as using $\sin(x)$ inside the hidden units to see what happens.
- And again I must give special thanks to Dr. Anderson for extending the deadline for this assignment 5. Thanks Dr. Anderson.

References

- [1] Anderson, Charles, *Nonlinear Regression with Neural Networks*, <http://www.cs.colostate.edu/~anderson/cs545/assignments/assignment5.html>, Fall 2009.
- [2] Computer Science Department, *CS545 - Machine Learning*, <http://www.cs.colostate.edu/~anderson/cs545/overview.html>, Fall 2009.
- [3] Martin F. Moller, *A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning*, *Neural Networks*, vol. 6, pp. 525-533, 1993