CS545: Dr. Anderson's Online Section Assignment 5

John Cartmell

October 28, 2009

Contents

1 Problem Definition **1**

2 Artificial Neural Network Model **2**
2.1 Development . . . 3
2.2 Use . . . 6
2.3 Output . . . 7

3 Linear Non-Noisy Data Set **8**
3.1 Data Generation . . . 8
3.2 Execution . . . 8
3.3 Model Output and Analysis . . . 9

4 Noisy Sine Data Set **12**
4.1 Data Generation . . . 12
4.2 Execution . . . 13
4.3 Model Output and Analysis . . . 13

5 Miles per Gallon Data Set **15**
5.1 Data Generation . . . 16
5.2 Execution . . . 18
5.3 Model Output and Analysis . . . 18

6 Extra Credit **22**
6.1 Comparison of Gradient Descent Methods . . . 22
6.2 R `optim` functions . . . 22
6.3 Comparison of Activation Methods . . . 23

7 Conclusion **25**

1 Problem Definition

This assignment focused on performing nonlinear regression with artificial neural networks (ANN). The task is to define R functions that will allow the creation and use of several two-layered artificial neural networks with a variety of hidden units and weight decay parameters for several data sets of interest. The hidden layer activation function used is the `tanh` function. However, other activation functions will be explored to allow a comparison with the `tanh` activation results. The scaled conjugate gradient (scg) is primarily used to find the minimum of the error during the training phase of the neural network, however, the steepest descent algorithm is employed as well as the R built-in `optim` function for comparison purposes. Once the code for the model is developed, it will be tested on a simple data set to ensure that the implementation

is correct. Once this is shown, then more complicated data sets are used, such as a very noisy sine curve and then finally the automobile miles per gallon data set which has been used in past assignments. For each data set, a number of plots are made and included in this report which facilitate the discussion.

The code I developed for this assignment reuses several functions provided by Dr. Anderson during the semester. These are noted as needed in this paper but the source is not included for obvious reasons. For that code which I modified or created, it is included and comments have been removed to aide in the readability. For ease of grading, the extra credit sections are segregated from the required sections of this paper.

2 Artificial Neural Network Model

An artificial neural network is a method for emulating the function of neurons within the brain. A neuron receives many input signals and then determines an output based on some weighting combination of these inputs. The output of a particular neuron is then used as input to another neuron. This can continue through many layers until the final output is determined [1]. While this is a gross simplification of the biological process of a neuron, it is the exact process used in the development of an artificial neural network. Emulating a neuron, inputs are weighted, summed, and then used to develop an output. These outputs are fed into other neurons which do the same process.

In this assignment, the number of layers of artificial neurons is two and the task is to train an artificial neural network which will predict the target data based on input data. Each layer will have a number of neurons. The first, or hidden layer, has a variable number of neurons based on the problem definition. The second, or output layer, will have as many nodes as there are output variables. In the simple data sets we use, there is one output. For the miles per gallon data, I am attempting to predict two of the parameters, therefore, there are two output nodes. Each node within the artificial neural network will have a weight for each input to that neuron. The calculation of all these weights is an iterative process where a forward and reverse pass is made through the network many times. Each time, the weights used in each neuron are tweaked to minimize the error between the predicted outputs and the target outputs. In the forward pass, inputs from a data set are fed through the hidden layer and the output of these neurons are fed through the output layer. The outputs of the final layer are compared against the target data. The error is then backward passed through the model to adjust the weights in the hope of reducing the error. Once the error is reduced to within a desired tolerance, the model development phase is complete. The artificial neural network reminds me of a giant control loop. We are trying to get the control loop to converge to a position where the error is within a known tolerance. However, instead of knowing all the machinery within the control loop, the neural network can be thought of as a black box as all it requires is to be trained to go to certain positions (outputs) based on certain inputs.

The above begs certain questions, such as how many hidden nodes should be used? Can the developed artificial neural network over fit the data and falsely portray a level of precision not really in the training data? Conversely, can the developed network be too vague? Appropriately sizing the hidden node layer is critical. Should too few hidden nodes be used, the model will not be able to deal with complicated problems. Should too many hidden nodes be used, the data will be over fit and the computation time of the model will be larger [2]. A counterbalance to the number of hidden nodes is a technique called weight decay. Weight decay does exactly what is states, it decays the computed weights towards zero. When the total squared error is computed, the sum of the squares of the weight are included, thereby biasing the squared error to be larger. This will have the effect, hopefully, of tamping the weights towards zero. The decay term is also included in the gradient functions computed for the weights in each layer. Another benefit of weight decay is that it aides in weeding out noise from the data to be modeled [3].

Another question comes to mind concerning a system to be modeled where the number of outputs are greater than one. When the error between the target and predicted outputs for each output to be predicted are measured, what happens when the multiple outputs have very different ranges? If the number of outputs

to be predicted is greater than one and they have very different ranges, the outputs with a larger range can crowd-out the other outputs. To prevent this the target outputs are standardized, so they all have the same range. For the miles per gallon data, this is critical since the *mpg* and *horsepower* fields which we are attempting to predict have very different ranges. As a result, the standardization must be reversed when the model is used to predict the output values.

In order to develop and use this artificial neural network, two functions must be created, makeNN and useNN, which are described below.

2.1 Development

In order to develop the model, there are a number of steps that must be accomplished. The full source code for this function is shown below and described in the following paragraphs. This function accepts several inputs which are needed to perform the process of developing an artificial neural network:

- Input Training Data
- Predicted Target Data
- Number of Hidden Nodes
- Decay Weight parameter, λ
- Number of Iterations to be performed in minimum error search
- Two precision values used by the function performing minimum error search
- Logical value to determine which function is used to perform minimum error search

The training data presented to the makeNN must be standardized, both the input data and the target data. A standardization function is developed for each, using the standardization function we have used in all of our previous assignments and is not included.

A new function was defined to reverse the standardization of the target data. This will be used to unstandardize both the target data and the predicted data. The unstandardize function is presented with the mean and standard deviation of each target variable so that the original range can be restored.

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

Once the input and output data is standardized and the unstandardize function is defined for the target data, the initial weights that are used to kick-off the iterative process must be computed. This is done by selecting a random number for each weight between *-0.01* and *0.01*. These will be updated by the scg function so these initial values used as placeholders to start the iterative process. However, selecting correct values is important since selecting too large numbers seem to cause the scg function to not converge. Once the initial weights are seeded, a bias column must be added to the input data. Note that the bias column is added after standardization since were it to be added prior to standardization, the bias column would be zero and would provide no bias.

After this, the weights must be determined that minimize the error. This is done by iterating the model repeatedly, each time computing the squared error and modifying the weights as a function of the error to attempt to drive the error to within a tolerance. In the gradientDescents.R provided by Dr. Anderson, there are two techniques to perform these iterations. First is scg and the other is steepest descent. My

implementation of makeNN allows for the use of either scg or steepest descent. The function defaults to use scg if no preference is offered.

The scg (or steepest descent) function is called to find the local minimum of the squared error. The scg function will iterate the model until either the squared error has reached a minimum or has exceeded the number of iterations. Each time scg iterates, the weights will be changed in an attempt (hopefully) to minimize the error. The scg functionrequires many inputs, including the weights to be adjusted, the function that computes the value to be minimized, the function that computes the gradients of the weights, and the parameters that are used to determine when to stop the search for a minimum. Similar to what was required for the previous assignment, the weights need to be merged into a single list before they are input to the scg function. Also, after the scg function concludes its work, the weights need to be separated back into the hidden node weights and output node weights. This was accomplished by creating two helper functions called pack and unpack. The pack function concatenates the weights for the hidden node(s) and the weights for the output node(s). Not surprisingly, the unpack does the opposite.

The two functions provided to scg which compute the error and which compute the gradient are sqErrorF and gradF. The sqErrorF function will compute the squared error based on the current iteration of the model. The current weight values are provided and unpacked. The output of the hidden nodes are computed based on the V weights and standardized inputs (plus bias) using the tanh activation function:

$$z = h(vx) \tag{1}$$

where

$$h(a) = tanh(a) \tag{2}$$

The output of the hidden nodes are fed into the output nodes, multiplied by the output node weights, W, to yield the predicted standardized targets:

$$y_s = zw \tag{3}$$

To compute the error, the difference between the predicted standardized targets and the actual standardized targets is computed. This error is then squared and augmented with the weight decay parameter applied to the sum of the squares of the weight:

$$E = (y_s - t_s)^2 + \lambda weights^2 \tag{4}$$

The weights are the concatenation of all the weights except for the bias weight in both the hidden and output nodes and λ is the decay factor. Once E is computed, the RMSE is computed and returned.

The gradF function will compute the gradient for the weights in the current iteration of the model. In a similar fashion to the sqErrorF it is provided with the weights. The weights are unpacked into the hidden weights, V, and output weights, W. Again, in a similar fashion to sqErrorF, the weights are used along with the standardized inputs and targets to compute the predicted standardized targets as well as the error between these and the actual standardized targets. This error is then fed into the gradient computations. The variables with tildes are that parameter with the bias column added and the weight, v, is all the v weights except for the bias weight.

$$\triangledown V = \frac{2}{NK}(\tilde{x}_s)^t E(w)^t(1 - z^2) + \lambda v^2 \tag{5}$$

$$\triangledown W = \frac{2}{NK}(\tilde{z}_s)^t E \tag{6}$$

Once the scg function has converged it returns the hidden and output weights. If the steepest descent algorithm is used, it returns the same parameters as the scg function. Therefore, from an interface point of view, there is no difference between the functions. These are unpacked and, together with the two standardize functions, the unstandardize function, and the scg function results are returned to the calling routine. The makeNN R source code is shown below:

```
makeNN <- function(X, T, nh, lambda=0, nIterations=100000,
                   xPrecision=0, fPrecision=0.00001, usescg=TRUE) {

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

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

  sqErrorF <- function(weights) {
    weights <- unpack(weights)
    V <- weights$V
    W <- weights$W
    Z <- tanh(Xs %*% V)
    Y <- cbind(1,Z) %*% W

    error <- Y - Ts
    weightDecay <- lambda *
                   t(matrix(V[-1,], length(V[-1,]), 1)) %*%
                   matrix(V[-1,], length(V[-1,]), 1)

    mean((error)^2) + weightDecay
  }

  gradF <- function(weights) {
    VW <- unpack(weights)
    V <- VW$V
    W <- VW$W
    Z <- tanh(Xs %*% V)
    Y <- cbind(1,Z) %*% W

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

    error <- Y - Ts
    gradV <- 2 / N / K * t(Xs) %*% (error %*% t(W[-1,]) * (1-Z^2)) +
             lambda * rbind(0, matrix(V[-1,], nrow(V)-1, nh))
    gradW <- 2 / N / K * t(cbind(1,Z)) %*% error

    pack(gradV, gradW)
  }

  standardizeFx <- makeStandardizeF(X)
  Xs <- standardizeFx(X)
  standardizeFt <- makeStandardizeF(T)
  Ts <- standardizeFt(T)

  Tmu <- colMeans(T)
  Tsigma <- sd(T)

  unstandardizeFt <- makeUnstandardizeF(Ts, Tmu, Tsigma)
```

```
  ni <- ncol(Xs)
  no <- ncol(Ts)
  V <- matrix(0.02*(runif((ni+1)*nh)-0.0099), ni+1, nh)
  W <- matrix(0.02*(runif((nh+1)*no)-0.0099), nh+1, no)

  Xs <- cbind(1,Xs)

  if (usescg==TRUE) {
    scgResult <- scg(pack(V, W), sqErrorF, gradF,
                     nIterations = nIterations,
                     fPrecision = fPrecision,
                     ftracep=TRUE)
    VW <- unpack(scgResult$x)
    list(V=VW$V, W=VW$W, lambda=lambda, standardizeFx=standardizeFx,
         standardizeFt=standardizeFt, unstandardizeFt=unstandardizeFt,
         result=scgResult)
  }
  else {
    steepResult <- steepest(pack(V, W), sqErrorF, gradF,
                            nIterations = nIterations,
                            fPrecision = fPrecision,
                            ftracep=TRUE)
    VW <- unpack(steepResult$x)
    list(V=VW$V, W=VW$W, lambda=lambda, standardizeFx=standardizeFx,
         standardizeFt=standardizeFt, unstandardizeFt=unstandardizeFt,
         result=steepResult)
  }
}
```

With this function defined, we can now pass in a set of training data, target data, number of hidden nodes, decay constant, and the configuration for the scg function and create an artificial neural network. Currently, the code has no input testing. While not necessary for this assignment, if this function was to become part of a library or be used more in the future, I would invest the time and add in functionality to ensure that the input is proper and correct.

2.2 Use

Once the model is developed, it is natural to go to the next step, to actually use it to predict the results of a test set of data. To do this, I developed the useNN function. This function requires a number of inputs:

- Artificial Neural Network Model output by makeNN
- Input data to generate predictions

Once this function is called, it will start by standardizing the input using the standardize function developed for the input data during the creation of the model. After this, the bias column will be bound to the input data. Then a forward pass through the model will be executed, first through the activation function of the hidden layer and then through the output layer. This processing will result in a standardized predicted output value. Given that this function is supposed to return the predicted output values, the standardized output will have to be unstandardized. This is accomplished by using the unstandardize function developed for the output data during the creation of the model. Once the unstandardize function is executed, both the unstandardized predicted outputs and the hidden layer outputs are returned. The hidden layer outputs are returned to facilitate the analysis and discussion of the model found throughout this document. The code is as follows:

```
useNN <- function(nn, X) {

  Xs <- nn$standardizeFx(X)
  Xs <- cbind(1,Xs)
  Z <- tanh(Xs %*% nn$V)
  Ys <- cbind(1,Z) %*% nn$W
  Yu <- nn$unstandardizeFt(Ys)
  list(Yu=Yu, Z=Z)
}
```

This function can be invoked multiple times once the model is developed. Again, for the sake of time, no error processing has been included. If this were to be formally released and used long-term by myself or others, then I would add in that functionality.

2.3 Output

Five plots are generated to graphically depict what the model is doing. These plots also provide some guidance as to the next steps and facilitate the discussion about the results of the model. The five plots include the following information:

- RMSE for Training Data as a function of Epoch
- Hidden Nodes versus X values
- Training Data Targets and Model generated Predicted Targets
- Test Data Targets and Model generated Predicted Targets
- Weight Diagram of the Neural Network

This structure is used repeatedly in the generation of plots. This code is based in part on the code provided by Dr. Anderson in nn1.R and uses functions provided by nnUtils.R. The mean squared error for the training data as a function of the epoch is returned to the calling function, in this case makeNN. The makeNN function passes this along to the calling routine so this plot can be created. When the model is actually used, via the useNN function, it returns the value of hidden node outputs, Z, based on X. The training data targets and model generated predicted targets are each plotted against X. The same plot is generated for the test data sets. Finally, a weight diagram is generated that shows the neural network weights as boxes, size and color varying as a function of sign and size of the weight. For example, large positive weights are shown as large green boxes while small negative weights are shown as small red boxes. The R code to perform the plotting is as follows:

```
x11()
par.orig <- par(mfcol=c(3,2), mar=c(4,4,4,4), bty="n")

  plot(sqrt(nn$result$ftrace), type="l", lwd=2,
       xlab="Epochs", ylab="Train RMSE", main="RMSE")
  matplot(Xtrain, cbind(Ttrain,resultTr$Yu), lwd=2,
          type="l", lty=1, xlab="x", ylab="T and Y",
          main="Training Data")
  drawNNet(list(W=list(nn$V,nn$W)))
  matplot(Xtrain, resultTr$Z, type="b", lwd=2, lty=1,
          xlab="x", ylab="Z", main="Hidden Node Output")
  matplot(Xtest, cbind(Ttest, resultTe$Yu), lty=1,
          type="l", lwd=2, xlab="x", ylab="T and Y",
          main="Test Data")

par(par.orig)
```

```
dev.copy2eps(file=filename)
```

The parameters plotted and the filename are all set before the plotting is performed, so the same code can be reused instead of having multiple copies of the same code except for small differences.

3 Linear Non-Noisy Data Set

3.1 Data Generation

In order to create and then use the model, data sets are required. For this assignment, we need several data sets. First, we need a small set of easy data to determine whether or not the model is functioning properly. Once the models are developed and then used, we look at the error and determine whether or not the functions that develop and use the model are correct. This very simple data is linear with a small amount of noise added. The noise was added so that the actual and predicted graphs would be distinct in the plots. If the noise was not added, the actual and predicted would overlay and it wouldn't be apparent that two lines were on the plot. The equation is as follows:

$$f(x) = 1.0 + 0.5x + 0.1rnorm(length(x)) \tag{7}$$

and is generated by the following R code:

```
f <- function (x)  1.0 + 0.5 * x + 0.1 * rnorm(length(x))
N <- 40
xmax <- 20
Xtrain <- matrix(seq(0,xmax,length=N),N,1)
Ttrain <- f(Xtrain)
Xtest <- Xtrain
Ttest <- f(Xtest)

lambda <- 0.001
nh <- 1
filename <- "assign5part1.eps"
```

Also note that other values are also set, including the number of hidden nodes, the size of the weight decay parameter, as well as the filename which is used to uniquely store the plot. Additionally, in this simple case the training and test data sets are equivalent. The above structure is reused for all subsequent internally generated data sets with changes to whichever parameters require change.

3.2 Execution

In order to train and then use the model, the makeNN and useNN functions must be invoked with the data generated by the above code. The R code to create the model is as follows:

```
converge <- FALSE
counter <- 1

while (!converge && counter <= 10) {
  cat("Attempting to create model, attempt",counter,"\n")
  nn <- makeNN(Xtrain, Ttrain, nh, lambda=lambda, nIterations=25000,
               xPrecision=0, fPrecision=0, usescg=TRUE)
  counter <- counter + 1
  if (nn$result$reason != "did not converge") converge = TRUE
}
```

Originally, my realization of creating the model only included a call the makeNN as shown above. However, once in a while, the scg function does not converge. This resulted in errors when an attempt was made to use

the model which did not fully converge in the scg function. Therefore, to overcome the problem where the scg function does not converge, a loop was placed around the call to the makeNN function which will allow it to be called multiple times. Since the initial values of the weights are random, if the first attempt does not converge, another attempt, with different weight may converge. The above code will make ten attempts and will stop as soon as an attempt is successful. Other realizations of this are possible, such as if the first pass through scg is not successful, switch to steepest descent. I selected ten attempts as a guess. This is probably too large of a number.

Once the model is created it must be used but only if the scg function converged. Therefore, before the useNN can be employed, we must ensure that the scg function did converge. If it did not, an error is printed to the screen and no further processing is done on this data set. If the model did converge, then the model generated predicted target values are needed for both the training and test data to create the plots and to compute the RMSE error. The trained neural network, *nn*, that was returned from the makeNN function is used along with either the training or test data as input to useNN. As previously mentioned, it returns both the predicted targets and the hidden node outputs for use in the plotting routines. The R code to effectuate this is as follows:

```
if (converge) {
  cat("Model did converge\n")

  resultTr <- useNN(nn, Xtrain)
  resultTe <- useNN(nn, Xtest)

  rmseTrain <- sqrt(mean((resultTr$Yu - Ttrain)^2))
  rmseTest <- sqrt(mean((resultTe$Yu - Ttest)^2))

  cat("RMSE Training Error ", rmseTrain, "\n")
  cat("RMSE Test Error ", rmseTest, "\n")

  ## Plotting routines removed as previously discussed
} else {
  cat("Model did not converge\n")
}
```

3.3 Model Output and Analysis

To determine the fidelity of the implementation of the artificial neural network, the simple linear data set with a small amount of noise added was used. Four different instances were executed, three with varying number of hidden units and λ set to *0* and the fourth with a large number of hidden units (which should not be needed for this data set) and λ set to 0.01.

Figure 1 shows the five plots as described in the previous section that describe the created model for the simple input data using only one hidden unit. The RMSE of the predicted and actual training targets decays as the number of forward and backward passes through the model increases. This is to be expected since the weights are being adjusted to better fit the data on every complete cycle through the model. The hidden node output for the one hidden unit appears to follow the tanh function curve, albeit slightly since the data being modeled is linear. Since the training and test data are the same data sets, the plots where the predicted and actual targets are plotted for the training and test data is identical. The predicted outputs closely follow the actual outputs which is to be expected. I did add some noise to the target output so that both curves would be evident on the plot. In both plots, the predicted and actual track each other closely. The last plot on this figure is the weight diagram that shows the magnitude and sign of the weights for all the neurons in this artificial neural network. This gives a quick glance of the entire neural network and the characteristics of the learned weights.

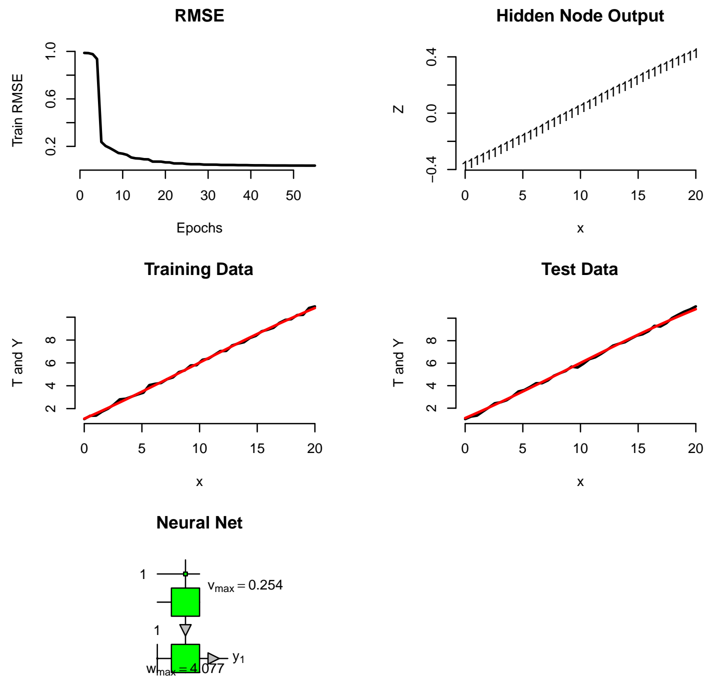


Figure 1: Artificial Neural Network with Hidden Units = 1, $\lambda = 0$

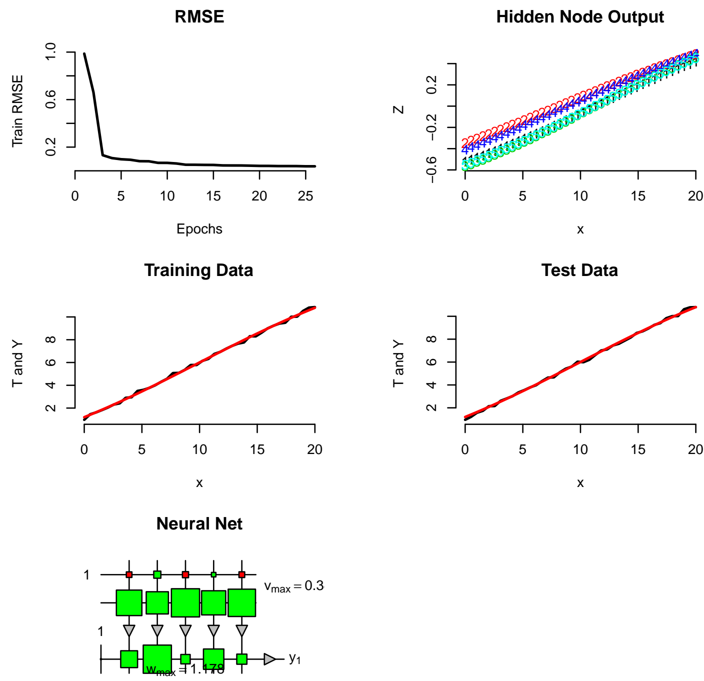


Figure 2: Artificial Neural Network with Hidden Units = 5, $\lambda = 0$

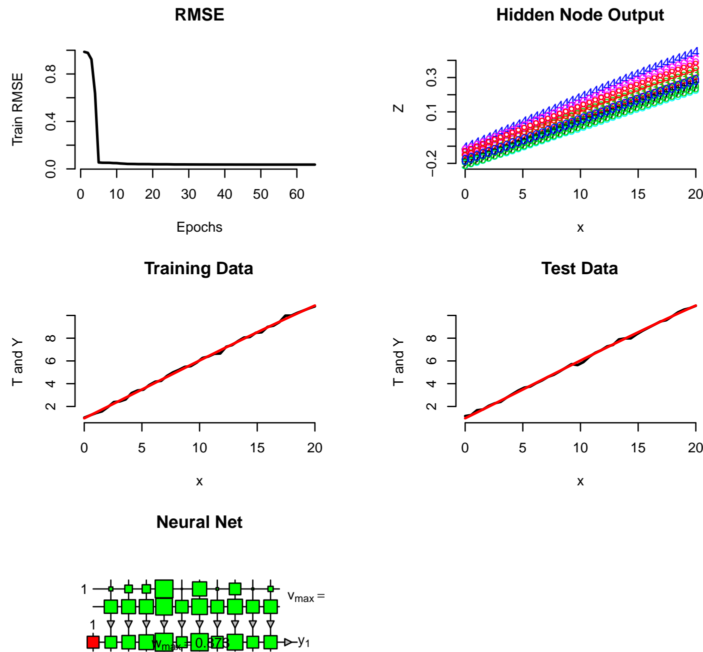


Figure 3: Artificial Neural Network with Hidden Units = 10, $\lambda = 0$

Figure 2 shows the same five plots as the previous figure, however, the number of hidden units has been increased to five. Given the nature of the input data, five is not needed but I am attempting to test the implementation. The same comments from the previous figure apply with a few items to note. First, the RMSE of the predicted and actual targets decays faster than when one one hidden unit was used. I believe this is because with five hidden units, there are more weights which can allow it to converge faster. I don't think this will always apply, but in the case of this simple data it does. Another difference is the hidden node output plot now has five curves as well as the weight diagram has five hidden nodes. This too is to be expected since there are five nodes. The five hidden node curves are almost the same and the weights for each node are very similar. I think this lends itself to support my point above, that with this simple linear system, the hidden nodes are working together to allow the model to converge faster during the training period. However, this is one execution of the model. To firm up this reasoning, the model would have to be executed many times. However, since the point of this section is to test the implementation, I will not perform that test.

Figure 3 shows the same five plots as the previous figures, however, the number of hidden units has been increased to ten. Again, ten is overkill, but I am attempting to test the implementation. This plot reveals a few items of interest. First, the RMSE of the predicted and actual targets decays at about the same rate and with the same characteristics as when one one hidden unit was used, but not as good as for five. Is this just a random variation or is there a limit to how much fitting can be done with the model before it becomes an exercise in silliness? I believe there is a limit as more and more hidden nodes lends itself to over fitting the data. Not necessarily in this case since the data is linear, however, for more complicated data sets, more hidden nodes is not necessarily a good thing. Additionally, I note that the hidden node output plot now has ten curves as well as the weight diagram has ten hidden nodes. This is also expected since there are ten nodes.

Figure 4 shows the same five plots as the previous figures, however, λ has been introduced, being set to *0.01*. Given the nature of the data, I didn't expect the introduction of weight decay to make much of a

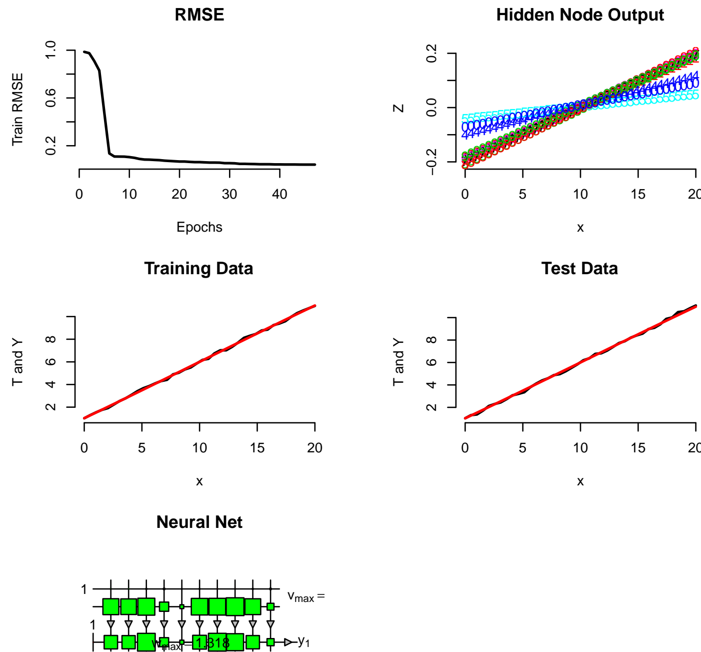


Figure 4: Artificial Neural Network with Hidden Units = 10, $\lambda = 0.01$

difference and it didn't, however, there were some small changes. For the hidden node plot, the curves are a little more distinct than the previous plot. This is because, I believe, the training epochs required to get to the desired target error were less. This prevented the weights from all trending to the same place, thereby generating the same hidden node outputs.

I believe based on the four figures presented in this section, my realization of makeNN and useNN is correct.

4 Noisy Sine Data Set

4.1 Data Generation

The noisy sine curve data was generated using the methods described above, the model was created and used in the manner previously described and the results plotted using the same plotting functions as mentioned above. This assignment called for presenting this plot for ten hidden nodes. This is presented below. Additional curves are given that show the training and test RMSE as a function of the number of hidden nodes in the model. Given the noisy nature of this data, I would expect that as the number of hidden nodes increases, the artificial neural network will be over fitting the data. This will be apparent if the training RMSE decreases while the test RMSE increases and may be compensated via weight decay. We shall see below if that is indeed what occurred. Here is the equation that was used to generate the noisy sine data:

$$f(x) = -1.0 + 0.05x + 0.4sin(x) + 0.3rnorm(length(x)) \tag{8}$$

and this function is realized by the following R code:

```
f <- function (x)  -1.0 + 0.05 * x + 0.4 * sin(x) + 0.3 * rnorm(length(x))
N <- 40
xmax <- 20
```

```
Xtrain <- matrix(seq(0,xmax,length=N),N,1)
Ttrain <- f(Xtrain)
Xtest <- Xtrain + xmax/N/2
Ttest <- f(Xtest)

lambda <- 0.0
nh <- 1
filename <- "assign5part10.eps"
```

4.2 Execution

The artificial neural network was trained and then used in the same manner as was done for the non-noisy linear data set. This was done with the exact same code, therefore, it is not discussed nor shown here. However, this section is included for symmetry and to indicate that the model did indeed need to be trained and then used.

4.3 Model Output and Analysis

Figure 5 shows the same five plots as the previous figures, however, there are many differences that require discussion. First, the number of epochs required to get to the desired accuracy is much larger. Instead of less than 100 for the previous easy data set, many thousands of epochs are required to train the model. This is to be expected since the data used for training in this example is very noisy. Also, the error is much higher than for the previous data set. This too is expected given the level of noise in the system, the model can only do so much to fit itself to the data. The hidden node plot and weight diagram are also telling. Instead of each hidden node working in concert to determine the output, there is a struggle between each of the hidden nodes to model this noisy data. Each of the hidden node outputs is very different and the final weights for this model are sprinkled with both positive and negative weights. There is also information to be gleaned from the training and test plot, actual versus predicted. For the training data, the model did a good job of limiting the error and closely following the noisy training data. However, for the test data, the model does not do nearly as good a job fitting the data. My suspicion is that the developed model has over fit the training data and the accuracy of the model for predicting test targets has suffered as a result.

Figure 6 shows the RMSE for both the training and test data sets with models developed using varying number of hidden nodes. The number of hidden nodes went from one to 20, by steps of one. Using the noisy sine data, a model was trained using the training data and then evaluated on both the training and test data. Given the randomness in the input data and the initial selection of weights, I created the model 25 times for each number of hidden nodes. The RMSE data over all 25 instances for each hidden node configuration was averaged to subdue variations resulting from randomness. The technique to loop through these is obvious from an implementation point of view and is similar to the techniques I have used in previous assignments. As the number of hidden nodes is increased, the training RMSE decreases as expected. For this noisy data, more hidden nodes yields a better fit to the training data and as a result a lower RMSE. The opposite is true for the test data. As the number of nodes is increased, the test error actually increases. This is a result of the developed neural network being over fit to the training data. Given the data presented, the number of hidden nodes that yielded the smallest average RMSE error in the test data is approximately six. However, to determine this we would need a third set of data, a validation set.

In an attempt to prove that the data is being over fit, I reran the above scenario with a small change. Let us assume that the training data was recorded using a faulty sensor, which caused the data to be noisy. Perhaps a wire was loose, or some other failure was fixed resulting in test data that was not noisy. What happens to the RMSE of the test data when the training data is noisy and the test data is not. To model this, I created another data set. The training data is the noisy data generated above. For the test data, I removed the noise. The hope is that the RMSE of the training data will decrease as noted above and the RMSE of the test data will increase, albeit more dramatically than for the previous figure.

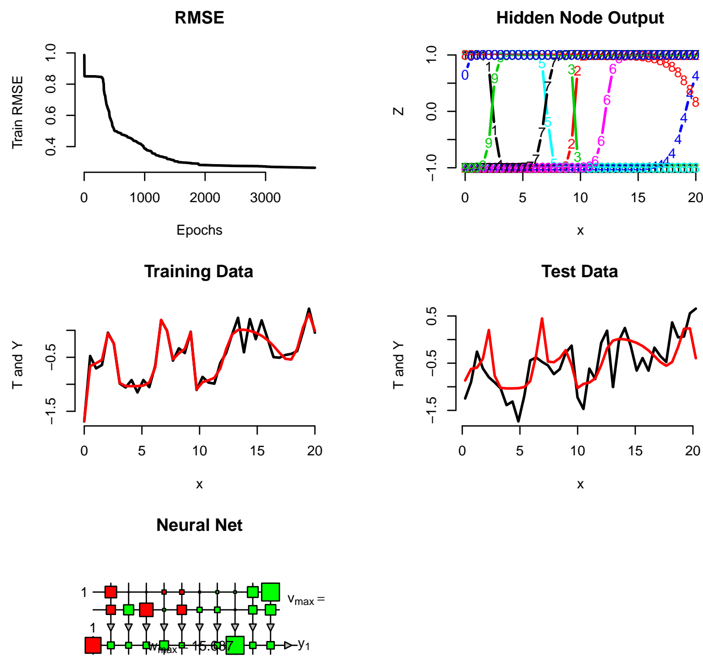


Figure 5: Artificial Neural Network with Hidden Units = 10, $\lambda = 0.0$

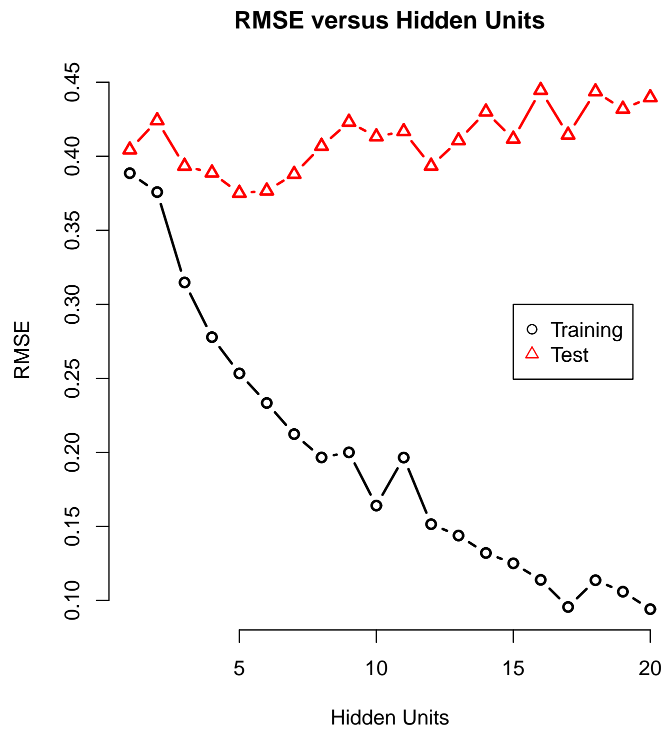


Figure 6: RMSE for noisy Training and Test data

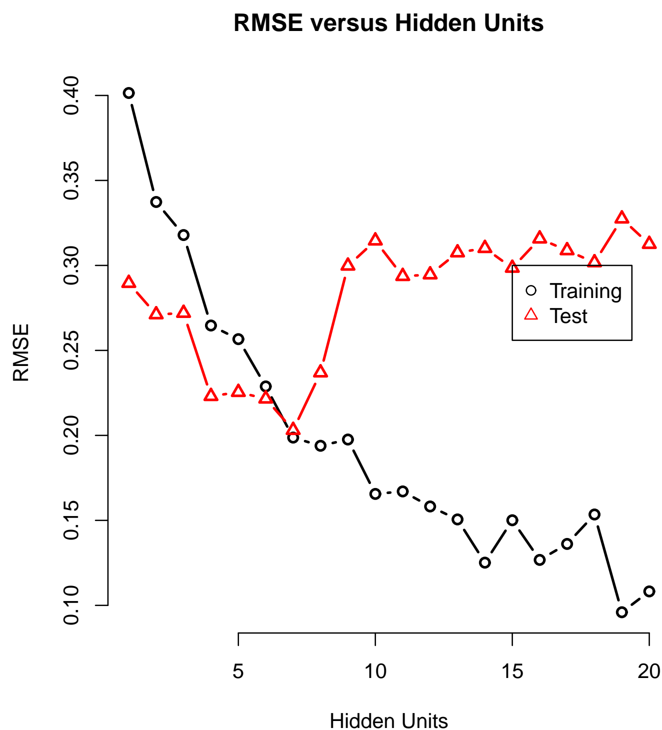


Figure 7: RMSE for noisy Training data and not-noisy Test data

Figure 7 shows the RMSE for both the training and test data sets with models developed using varying number of hidden nodes with the training data being very noisy and the test data having no noise. This figure absolutely shows the over fitting in a more profound manner! The training data error is reduced as more hidden nodes are introduced which is consistent with the previous figure. For the test data, which now has no noise, the RMSE decreases as hidden nodes are introduced up to a point. After about six hidden nodes are included, the RMSE of the test data increases much more sharply and pronounced than for the RMSE of the noisy test data. I definitely believe that the developed model is over fitting the training data as more and more hidden units are included in the model. In order to prevent this over fitting, there are several techniques that can be used [4] [5] [6]. In an attempt to prevent the model from over fitting the data, the iterative process of adjusting the weights can be stopped prior to reaching the desired level of precision in the fit of the error in the modeling of the outputs based on the training data. This is known as early stopping and may be beneficial in this case. Additionally, weight decay can be used to help prevent the over fitting. In fact, since the model I implemented supports weight decay, let us see what happens when weight decay is included. Is the over fitting evident in the previous two figures overcome by the use of weight decay?

Figure 8 was generated the same as the previous two plots, except, λ was set to *0.01*. With the incorporation of weight decay, the trend where the test RMSE increases as hidden nodes are added has been stopped. In fact, the model doesn't perform much better by adding the complexity of additional hidden units, the RMSE of both the test and training data is fairly constant despite the added model complexity.

In summary, this data set provided a good illustration of the effects of over fitting the training data, to the detriment of the model fidelity, and how weight decay can be used to prevent this from occurring. While I did not show how early stopping is also of benefit, references are cited and were it to be modeled, I believe it would show the same benefits.

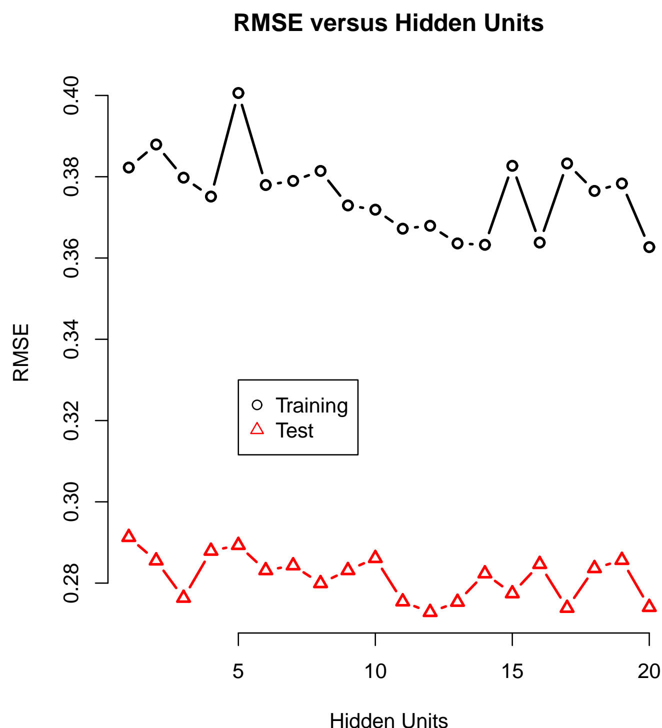


Figure 8: RMSE for noisy Training data and not-noisy Test data with $\lambda = 0.01$

5 Miles per Gallon Data Set

The MPG data that was used in class and needed for this assignment is available at the UCI Machine Learning Repository [7]. This data contains nine attributes, including the name of the vehicle. For purposes of this exercise, the name attribute will not be used. Of the eight other attributes, five are continuous valued and three are multi-valued discretes. Of the multi-valued discretes, I am using number of cylinders and year as is, therefore, I am not converting them to indicator variables. Even though they are discrete valued attributes, they are not boolean in nature as is, for example, sex. However, country of origin is a discrete value, similar to the sex attribute in the Abalone data set from Assignment 2. Therefore, it must be converted. Since it has values one through three, it would translate into three indicator variables. However, there would be a correlation between the three fields if all were used, hence only two are needed to express the three possible values for origin.

5.1 Data Generation

The following code was mostly reused from Dr. Anderson's R code provided during the lecture that used this data set. It downloads the data from the UCI website and removes non-numeric rows in the data. Next, the origin attribute is converted into two discretes. Given that origin has a range of three, two attributes are sufficient to represent this field. Once this conversion occurs, the data is then partitioned between the input parameters and the output parameters. For this exercise, the MPG and horsepower attributes are those we wish to predict. All the other attributes are used as input to the neural network to be trained except name. Once the data is read in and conditioned, the data must be split between the training and test data sets. This is done in a similar fashion to how we have performed this function in previous assignments and during class. The code to effectuate reading in the data, conditioning it, and then segregating the data is as follows:

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

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

keepRows <- apply(mpg != "?", 1, all)
mpg <- mpg[keepRows,]

mpg1 <- cbind(mpg[,1:7], mpg[,8]==1, mpg[,8]==2)
colnames(mpg1)[8] <- "orig-1"
colnames(mpg1)[9] <- "orig-2"

mode(mpg1[,8]) <- "numeric"
mode(mpg1[,9]) <- "numeric"
mpg1 <- apply(mpg1,2,as.numeric)

rmseErrorMPGCompTr <- NULL
rmseErrorHPCompTr <- NULL
rmseErrorMPGCompTe <- NULL
rmseErrorHPCompTe <- NULL
hiddenUnits <- seq(2, 20, by=1)
for (units in hiddenUnits) {
  rmseErrorMPGAllTr <- NULL
  rmseErrorHPAllTr <- NULL
  rmseErrorMPGAllTe <- NULL
  rmseErrorHPAllTe <- NULL
  for (loopy in 1:25) {
    lambda = 0.005
    filename <- "ass5part51.eps"

    randorder <- sample(nrow(mpg1))
    nTrain <- round(nrow(mpg1)*0.8)
    trainRows <- randorder[1:nTrain]

    X <- cbind(mpg1[trainRows, 2:3], mpg1[trainRows, 5:9])
    T <- cbind(mpg1[trainRows, 1, drop=FALSE], mpg1[trainRows, 4, drop=FALSE])

    Xtest <- cbind(mpg1[-trainRows, 2:3], mpg1[-trainRows, 5:9])
    Ttest <- cbind(mpg1[-trainRows, 1, drop=FALSE], mpg1[-trainRows, 4, drop=FALSE])

    nn <- makeNN(X, T, units, lambda=lambda, nIterations=25000,
                 xPrecision=0, fPrecision=0, usescg=TRUE)

    resultTr <- useNN(nn, X)
    errorTr <- resultTr$Yu - T
    resultTe <- useNN(nn, Xtest)
    errorTe <- resultTe$Yu - Ttest

    rmseErrorMPGTr <- sqrt(mean(errorTr[,1]^2))
    rmseErrorHPTr <- sqrt(mean(errorTr[,2]^2))
    rmseErrorMPGTe <- sqrt(mean(errorTe[,1]^2))
    rmseErrorHPTe <- sqrt(mean(errorTe[,2]^2))

    rmseErrorMPGAllTr <- rbind(rmseErrorMPGAllTr, rmseErrorMPGTr)
```

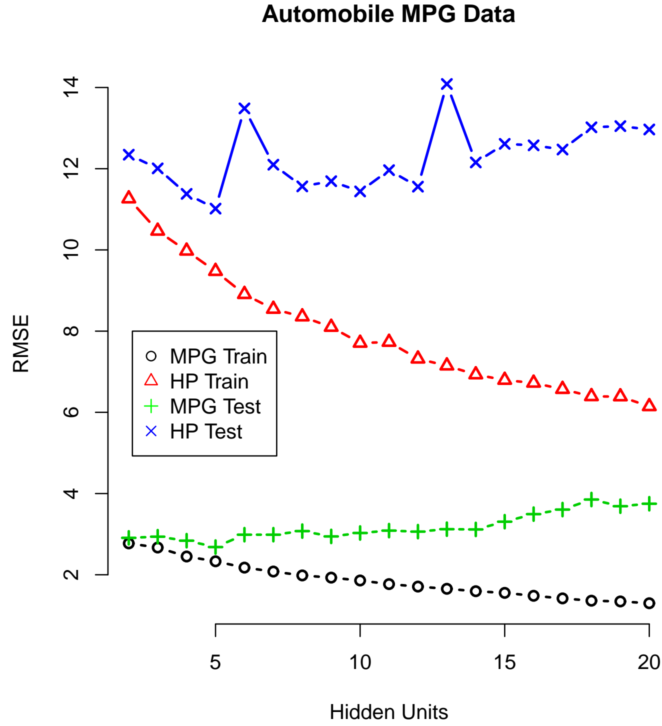


Figure 9: RMSE Error for MPG Data with varying Hidden Units and $\lambda = 0.0$

```
    rmseErrorHPAllTr <- rbind(rmseErrorHPAllTr, rmseErrorHPTr)
    rmseErrorMPGAllTe <- rbind(rmseErrorMPGAllTe, rmseErrorMPGTe)
    rmseErrorHPAllTe <- rbind(rmseErrorHPAllTe, rmseErrorHPTe)

  }

  rmseErrorMPGCompTr <- rbind(rmseErrorMPGCompTr, mean(rmseErrorMPGAllTr))
  rmseErrorHPCompTr <- rbind(rmseErrorHPCompTr, mean(rmseErrorHPAllTr))
  rmseErrorMPGCompTe <- rbind(rmseErrorMPGCompTe, mean(rmseErrorMPGAllTe))
  rmseErrorHPCompTe <- rbind(rmseErrorHPCompTe, mean(rmseErrorHPAllTe))
}
```

Once the data was retrieved, conditioned, and segregated, the model must be trained and then used to compute the RMSE of both the horsepower and MPG for both the training and test sets. In this first instance, the number of hidden nodes varies from two to 20 and λ is set to *0.0*. Each configuration of model parameters (number of hidden nodes) is executed 25 times. The RMSE is calculated for each instance and then averaged. For each instance of the model being created, the data set is re-partioned into a new training and new test data set to promote some randomness into the averaging. The results are shown in Figure 9. In an attempt to gather more data to provide further analysis, λ was modified to be *0.005* and *0.010* as shown in Figures 10 and 11 respectively.

5.2 Execution

The artificial neural network was trained and then used in the same manner as was done for the non-noisy linear data set. This was done with the exact same code, therefore, it is not discussed nor shown here. However, this section is included for symmetry and to indicate that the model did indeed need to be trained

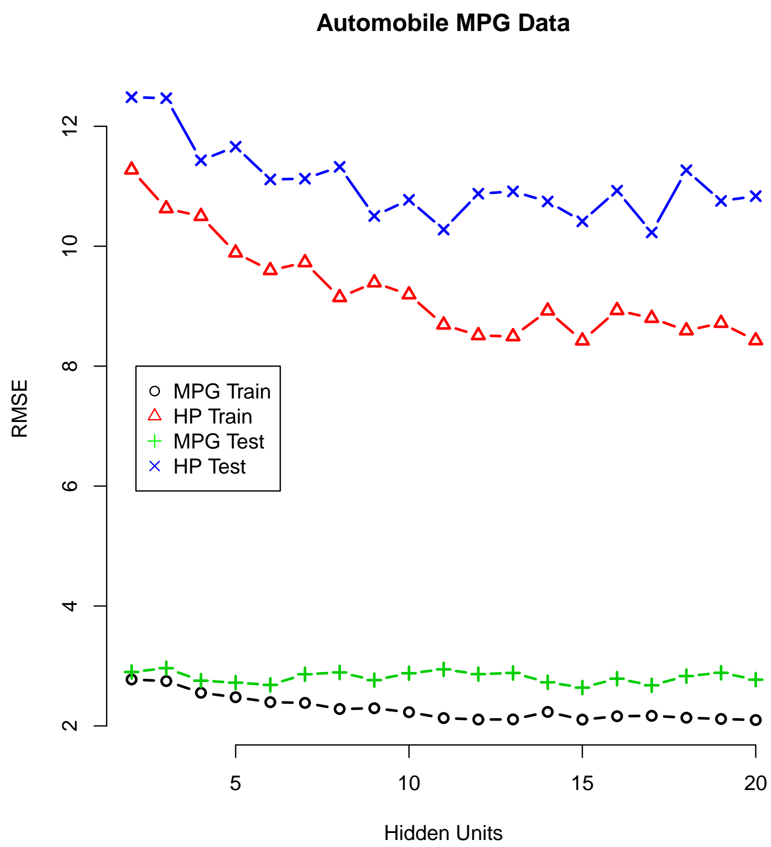


Figure 10: RMSE Error for MPG Data with varying Hidden Units and $\lambda = 0.005$

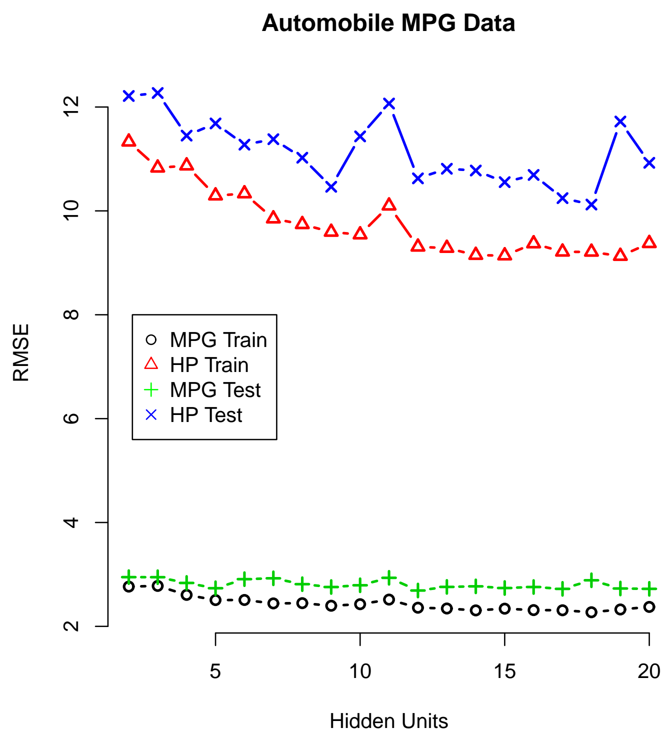


Figure 11: RMSE Error for MPG Data with varying Hidden Units and $\lambda = 0.010$

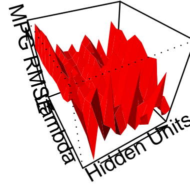
and then used.

5.3 Model Output and Analysis

Figure 9 shows the artificial neural network performance against both the training and test data sets for both of the predicted outputs, MPG and horsepower. As was noted earlier for the noisy sine data, as the number of hidden nodes increases, the model fidelity as viewed from the training data set perspective improves. The RMSE for both MPG and horsepower decrease as the number of hidden units increases. However, the RMSE between the predicted and target outputs for the test data set increases as the number of hidden nodes increases. This indicates the model is being trained to over fit the data as hidden nodes are introduced. As noted above, there are two ways to correct this over fitting. We could use early stopping, preventing the scg function from driving the errors to the lowest of levels. Or we could introduce weight decay by setting λ to greater than zero. This approach led to the plots in Figures 10 and 11. As we can see, the training data output RMSE errors are decreasing, but the model is not being over fit since the RMSE of both test data outputs is also decreasing. With the introduction of a small λ, the over fitting is prevented and the fidelity of the predictability of the model is improved for both outputs. The code to generate these plots is as follows:

```
x11()
par.orig <- par(mfcol=c(2,1),mar=c(4,4,4,4),bty="n")

matplot(hiddenUnits, cbind(rmseErrorMPGCompTr, rmseErrorHPCompTr,
          rmseErrorMPGCompTe, rmseErrorHPCompTe), pch=c(1, 2, 3, 4),
          lty=1, type="b", lwd=2, xlab="Hidden Units", ylab="RMSE",
          main="Automobile MPG Data")
```



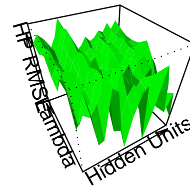


Figure 12: RMSE Error for MPG Data with varying Hidden Units and varying λ

```
legend(x=2.125, y=8,legend=c("MPG Train", "HP Train", "MPG Test",
         "HP Test"),col=c("black", "red", "green", "blue"),
         pch=c(1, 2, 3, 4))

par(par.orig)

dev.copy2eps(file=filename)
```

For the purposes of this data, it appears that the selection of λ is as critical as the selection of the number of hidden nodes. I researched whether or not there were any algorithms to select the best number of hidden units and the value of λ. Many papers stated that the selection of the number of hidden nodes is not well developed and an area of study [8] [9]. In fact, my frustration at not knowing how many hidden units to use or the size of λ prompted me to generate Figure 12. This plot is interesting as it shows the test RMSE for both data outputs, MPG and horsepower. The number of hidden nodes was varied from two to 20 while the λ was slewed from *0.005* to *0.05*. Given the amount of points in this plot, it is difficult to gauge what the best number of hidden nodes and λ minimize the errors, however, I thought this plot was interesting so it is included. The code to generate these three-dimensional plots is as follows:

```
x11()
par.orig <- par(mfcol=c(2,1),mar=c(4,4,4,4),bty="n")

persp(hiddenUnits,lambdas, matrix(rmseErrorMPGCompTe,length(hiddenUnits),
       length(lambdas),byrow=TRUE),xlab="Hidden Units",ylab="Lambda",zlab="MPG RMSE",
       col="red",theta=-30,phi=40,border=NA,shade=0.3)

persp(hiddenUnits,lambdas, matrix(rmseErrorHPCompTe,length(hiddenUnits),
       length(lambdas),byrow=TRUE),xlab="Hidden Units",ylab="Lambda",zlab="HP RMSE",
```

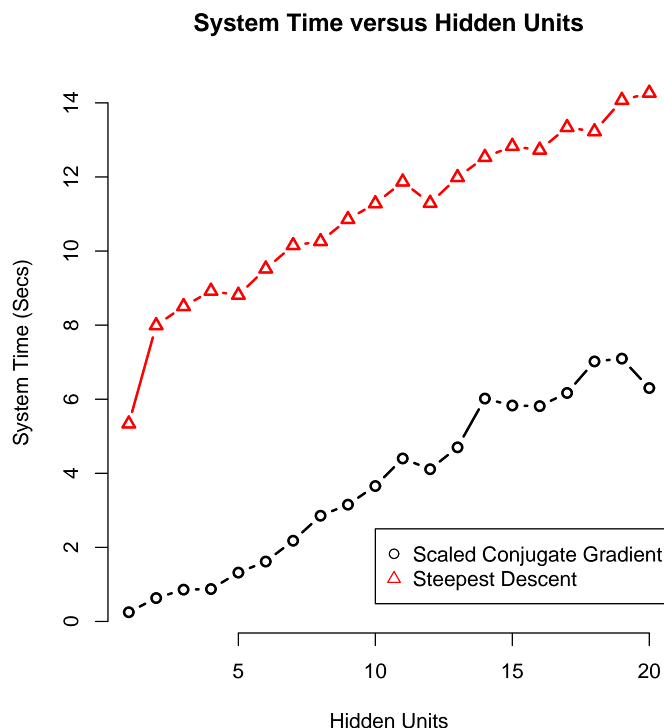


Figure 13: Execution Time for scg and steepest descent as a function of Hidden Units

```
    col="green",theta=-30,phi=40,border=NA,shade=0.3)
par(par.orig)
dev.copy2eps(file=filename)
```

6 Extra Credit

6.1 Comparison of Gradient Descent Methods

For this portion of the extra credit, we need to measure the time required to iterate the weights to within the required output tolerance by different methods. I will reuse the code presented above, with the exception of measuring and saving the time it takes for the scg function and the steepest descent function to iterate and adjust the model weights. I will reuse the noisy sine data and will vary the number of hidden nodes. For each of these, the time required to train the model will be measured and the results plotted for a comparison. The results are shown in Figure 13. Steepest Descent takes much longer than scg to converge the error to within the desired tolerance. However, the slope of each line is similar. For this figure, both the scg function and steepest descent were executed 25 times for each number of hidden units. The time required to perform each were saved, then averaged, and then plotted.

6.2 R optim functions

I explored the R optim function. I tried to use this for training and then running an artificial neural network for the noisy sine data used earlier in this assignment. The results were less than stellar and are shown in Figure 14. I invoked the optim function as follows:

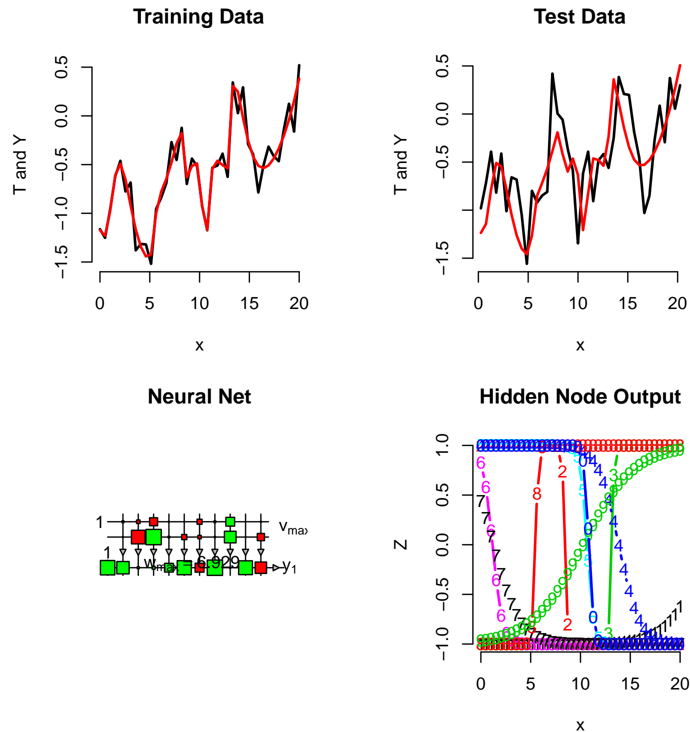


Figure 14: Noisy sine data model generated with optim function

```
result <- optim(pack(V, W), sqErrorF, gradF, method="CG",
                control = list(maxit = 25000, trace = 1, abstol = 0.00001))
```

The training data fit is fair, with an RMSE of slightly more than *0.10*. The test data is not fit very well and the RMSE error is substantial at about *0.40* however, the scg and steepest descent didn't do much better. Therefore, I conclude that is is possible to create a neural network with the optim function. However, the help page on optim indicated that the default method did not work well. Research did not yield any results as to why this statement is included, but I stand by my conclusion that it can be used.

6.3 Comparison of Activation Methods

The activation function that was used in this assignment so far was the tanh function. However, there are several other activation functions that can be employed [10]. A favorite is the sigmoid function [11]:

$$h(a) = \frac{1}{1 + e^{-x}} \tag{9}$$

which has a derivative of:

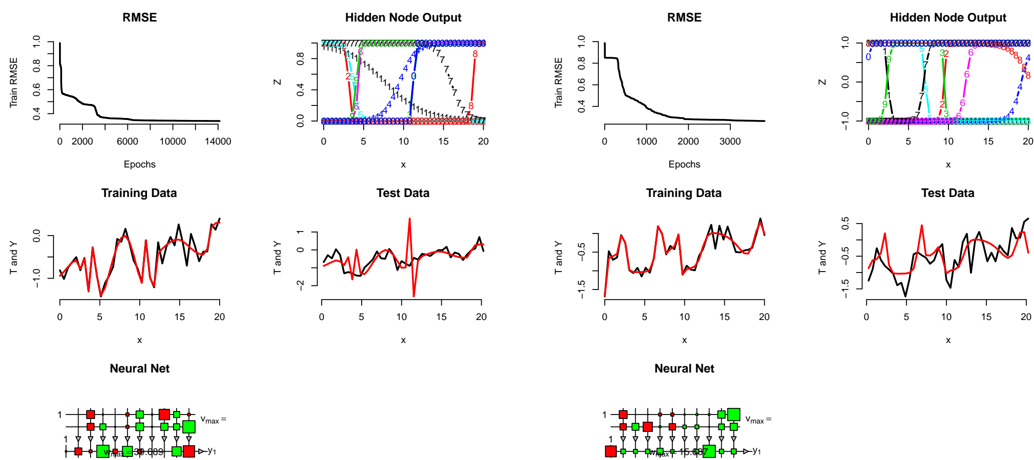
$$h'(a) = a(1 - a) \tag{10}$$

The noisy sine data was reused again and the sqErrorF, gradF, and useNN were modified to use this activation function. The changes to sqErrorF and useNN are to replace the Z equation as follows:

```
Z <- 1/(1+exp(-(Xs %*% V)))
```

and change to gradF is to replace the *gradV* equation as follows:

```
gradV <- 2 / N / K * t(Xs) %*% (error %*% t(W[-1,]) * Z * (1-Z)) +
         lambda * rbind(0, matrix(V[-1,], nrow(V)-1, nh))
```



(a) Noisy sine data model with sigmoid function

(b) Noisy sine data model with tanh function

Figure 15: Activation function model comparison, hidden units = 10, $\lambda = 0.0$

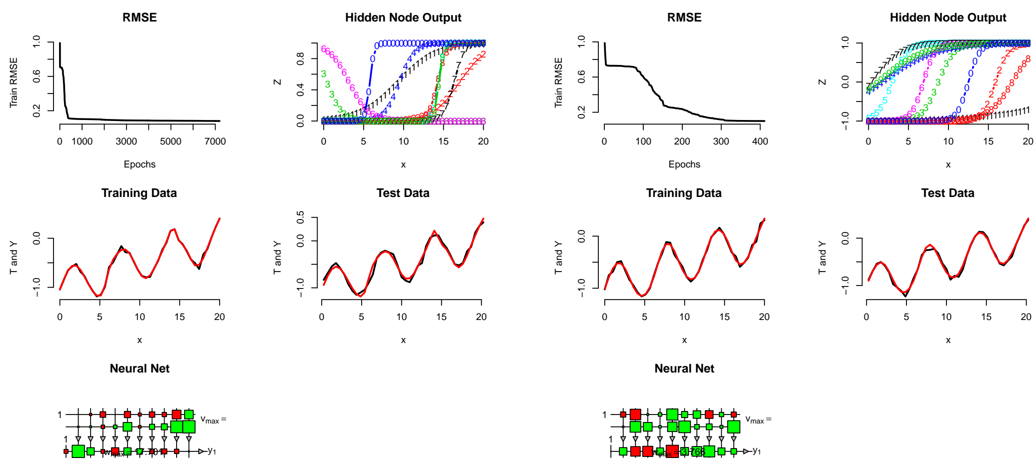
The result for the noisy cosine data is shown in Figure 15. The figure on the left shows the results with the sigmoid activation function while the figure on the right uses the tanh activation function. These results are very similar, but there are some differences to note. First, the hidden node functions have a different range. This is to be expected since the tanh has a range of *-1* to *1* while the sigmoid function has a range of *0* to *1*. Both models trained to about the same RMSE for the training data set, about *0.2*. However, the tanh function took less epochs to converge than the sigmoid function. Another run was performed where the noise on the sine data was reduced. The results are shown in Figure 16. The same differences noted above for the noisy data are seen in this figure as well.

7 Conclusion

I enjoyed working on this assignment as it allowed me to understand (albeit not completely) neural networks. I am fascinated by the degrees of freedom in developing the model and the techniques which can be used to not over fit the data. I did have problems including the weight decay into the equations, but I think that is a result of my poor math skills rather than anything else.

I think I developed an interesting approach to handle the problem when the scg does not converge by allowing makeNN it to try again several times with new random weight values. After a number of attempts, my code could be easily modified to try the steepest descent implementation. I begin to question the wisdom of scg when it seems to not converge on every attempt. Even though it may be a quicker algorithm, if it does not converge all the time, I question the value of the algorithm. However, it usually converges and is much faster than steepest descent, even if it has to be executed several times.

I also used one other technique to provide some level of insight as to the model varying the weights and their effects. If in the errorF the hidden node outputs, Z, are plotted for every iteration of the model, one can see the curves move radiply in the beginning and then each hone in on their final value. I think this graph, while easy to incorporate, allowed for an even greater level of understanding of the iterative process of creating an artificial neural network. This idea was based on the technique used by Dr. Anderson in the provided nn1.R code.



(a) Quiet sine data model with sigmoid function

(b) Quiet sine data model with tanh function

Figure 16: Activation function model comparison, hidden units = 10, $\lambda = 0.0$

References

[1] *So, what exactly is a Neural Network?*, `http://www.ai-junkie.com/ann/evolved/nnt2.html`, Retrieved October 25th, 2009.

[2] *Optimization of hidden nodes and training times in ANN-QSAR model*, `http://www.iseis.org/EIA/pdfstart.asp?no=07051`, Retrieved October 25th, 2009.

[3] *What is weight decay?*, `http://www.faqs.org/faqs/ai-faq/neural-nets/part3/section-6.html`, Retrieved October 25th, 2009.

[4] *Lecture 6: Artificial Neural Networks*, `http://ocw.mit.edu/NR/rdonlyres/Sloan-School-of-Management/15-062Data-MiningSpring2003/650A194A-828C-4990-98CE-7EB966628437/0/NeuralNet2002.pdf`, Retrieved October 26th, 2009.

[5] *Artificial Neural Network*, `http://psychology.wikia.com/wiki/Artificial_neural_network`, Retrieved October 26th, 2009.

[6] *Artificial Neural Networks*, `http://issuu.com/fanizzi/docs/ann-1`, Retrieved October 25th, 2009.

[7] *Auto MPG Data Set*, `http://archive.ics.uci.edu/ml/datasets/Auto+MPG`, Retrieved October 26th, 2009.

[8] *Back Propagation Neural Networks*, `http://murphylab.web.cmu.edu/publications/boland/boland_node17.html`, Retrieved October 26th, 2009.

[9] *3.17 Development to a Well-Designed Network Model*, `http://www.sasenterpriseminer.com/documents/Chapter3-17.pdf`, Retrieved October 27th, 2009.

[10] *A Heuristic Neural Network Initialization Scheme for Modeling Nonlinear Functions in Engineering Mechanics*, `http://students.ou.edu/M/Eric.C.Mai-1/SPIE_NN_Final.pdf`, Retrieved October 26th, 2009.

[11] *Lecture 5: Regression II: Adaptive Basis Networks Supervised Mixtures*, `http://www.cs.toronto.edu/~roweis/csc2515-2006/notes/lec6x.pdf`, Retrieved October 27th, 2009.