

Nonlinear Regression with Neural Networks

Christie Williams

October 27, 2009

Contents

1	Introduction	1
2	Building a Neural Network	2
2.1	Parts of the Neural Network	2
2.1.1	Helper Functions	2
2.1.2	Mean Square Error	3
2.1.3	Error Gradient	3
2.2	Putting it All Together	4
2.2.1	makeNN	4
2.2.2	useNN	5
2.3	Does the Neural Network Work?	5
3	Applying a Neural Network to a Noisy Sine Curve	6
3.1	The Neural Network in Action	6
3.2	Affects of Varying the Number of Hidden Units	10
3.3	Results	10
4	Applying a Neural Network to the Automobile MPG Dataset	12
4.1	Handling the Data	12
4.2	How Many Hidden Units is Best?	13
4.3	What is the Best Lambda Value?	15
5	Conclusion	16

1 Introduction

This paper explores the building of neural networks using the Scaled Conjugate Gradient method of calculating the neural net weights. The examination of this topic is broken into three parts; first, the methodology for constructing and testing the neural net is detailed, then a neural net created by this process is applied to a data set representing a noisy sine curve, and finally a second neural net is applied to a dataset dealing with automobiles' miles per gallon and horsepower attributes. As a part of the noisy sine analysis, the performance of the neural network is evaluated for various numbers of hidden units in order to examine the effect that the number of hidden units has on the predictions made by the net. In the automobile dataset section, an examination is made of the “best” number of hidden units and the “best” lambda value to be used in creating a neural network to predict the automobile dataset.

2 Building a Neural Network

The first section of this report deals with the underlying structure of a simple neural network using the Scaled Conjugate Gradient method of gradient descent to learn the weights. The Scaled Conjugate Gradient method requires two functions: a calculation of the mean square error generated by a forward pass through the neural net, and a gradient function, used to control how the weights are modified at each training iteration. In order to make it simpler to create and use a neural net generated by this code, construction of the neural net and evaluation of test datasets was explicitly broken out into two sections: a `makeNN` section builds and trains the neural net on a training dataset, and a `useNN` section takes a set of testing data and returns the neural net's predictions about the outputs.

2.1 Parts of the Neural Network

2.1.1 Helper Functions

In order to more easily pass the `V` and `W` matrices of neural network weights between methods, a couple of helper methods (`pack` and `unpack`) that combined the two matrices into a single matrix and separated them back apart, respectively, were created. `pack` takes as input the two weight matrices, and returns as single matrix using the R code

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

The `unpack` function is a little bit more complex; it needs not only the matrix to be split, but also the dimensions of the two weight matrices. The code to do so is below (this code was provided by Dr. Anderson [6]):

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

`ni`, `nh`, and `no` are the number of inputs, number of hidden units, and number of outputs, respectively.

Another set of functions created were to standardize and “unstandardize” attributes and results. Standardization is necessary in order to properly handle inputs where different attributes cover widely different ranges, but if the results are left in standardized form, the output of the net is not particularly useful. To remedy this, the outputs are “unstandardized” to return them to the ranges that they were in the input data. Code to create both functions is below; `makeStandardize` and `standardize` were provided by Dr. Anderson [1]).

```
makeStandardize <- function(X)
{
  X <- as.matrix(X)

  mu <- colMeans(X)
  sigma <- sd(X) ##sd should be named colSDs

  standardize <- function(newX)
  {
    newX <- as.matrix(newX)
    nr <- nrow(newX)
    nc <- ncol(newX)
    (newX - matrix(mu, nr, nc, byrow=TRUE)) / matrix(sigma, nr, nc, byrow=TRUE)
  }

  unstandardize <- function(standard_mat)
```

```

{
  standard_mat <- as.matrix(standard_mat)
  nr <- nrow(standard_mat)
  nc <- ncol(standard_mat)

  return((standard_mat * matrix(sigma, nr, nc, byrow=TRUE)) + matrix(mu,
    nr, nc, byrow=TRUE))
}

return(list(standardize=standardize, unstandardize=unstandardize))
}

```

2.1.2 Mean Square Error

The first of the two functions required by the Scaled Conjugate Gradient is the mean square error function. This function calculates the error between the results of a single forward pass of the neural net being trained and the training results. It is used to evaluate how “well” the net is approximating the actual results desired. The code for this function is as follows:

```

sqErrorF <- function(weights)
{
  u_weights <- unpack(weights, length(XtrainS[1,]) - 1, numHiddenUnits,
    length(TtrainS[1,]))

  V <- u_weights$V
  W <- u_weights$W

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

  Vmod <- rbind(0, V[-1, , drop=FALSE])
  return(mean((Y - TtrainS)^2) + c(lambda * sum(Vmod^2)))
}

```

First, the method splits the single weight matrix apart into the V and W matrices for the sets of weights on the inputs to the hidden units and the inputs to the final output nodes, respectively. It then calculates the Z matrix, or the weighting on the outputs from the hidden nodes. In this neural net implementation, the nonlinear function employed by the hidden nodes in `tanh(XtrainS` is the training attributes, which have been standardized and have a constant 1 column prepended to them). Once the Z matrix has been calculated, the Y matrix, which is the final outputs of the forward pass through the network, are calculated. At this point, a constant 1 column is added to Z to represent the constant hidden unit with a value of 1. Finally, the mean square error is calculated. This value also has a weight decay calculated for it in order to prevent the weights from becoming too large. This is desirable as otherwise the net may oscillate and never settle on a “better” value if that value is smaller than the amount the weight is changed each iteration.

2.1.3 Error Gradient

The second function needed by the Scaled Conjugate Gradient algorithm is the error gradient function itself. This function determines how the gradient descent progresses each iteration by specifying how the weights are updated at each step. The code to do this is below; this is a modified version of a gradient function provided by Dr. Anderson [4] (the original code was for a steepest-descent algorithm).

```

gradF <- function(weights)
{
  u_weights <- unpack(weights, length(XtrainS[1,]) - 1, numHiddenUnits,
    length(TtrainS[1,]))

```

```

V <- u_weights$V
W <- u_weights$W

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

error <- Y - TtrainS

Vmod <- rbind(0, V[-1,,drop=FALSE])
Wmod <- rbind(0, W[-1,,drop=FALSE])

Vgrad <- const * (t(XtrainS) %*% (error %*% t(W[-1,,drop=FALSE]) * (1-
  Z^2)))+ (lambda * sum(Vmod^2))
Wgrad <- const * (t(cbind(1,Z)) %*% error)

return(pack(Vgrad, Wgrad))
}

```

Similar to the mean square error function, this function first unpacks V and W , then calculates the Y matrix from a forward pass through the neural net. It then calculates the V and W weight gradients, taking into account the lambda weight decay values for V (this is done for the non-constant nodes only; hence, the first row of each matrix is replaced with 0s so that it is not penalized). A packed matrix containing both error gradients is then returned.

2.2 Putting it All Together

Once the functions required for the Scaled Conjugate Gradient algorithm were created, the next step is to put them all together to build and use the neural network. In order to facilitate reuse of the network, the functionality to build and train it was placed in a `makeNN` function, and the code to use the network to approximate a function was placed in a `useNN` function. These two functions are outlined below.

2.2.1 makeNN

The code involved in making the neural network primarily deals with getting all of the required matrices created and in the proper format. First, it standardizes the training matrices (both the attribute and result matrices are standardized, so as to deal with cases where more than one output is desired and the outputs cover different ranges). Then, a constant 1 bias column is added to the training attributes. The V and W matrices are also initialized with small random numbers; this is important because if they were initialized to 0, there would be no weights and hence no change could ever occur.

Once the matrices are all in the correct form, the Scaled Conjugate Gradient algorithm is used to train the net. The two error gradient functions described above, along with the packed version of the V and W matrices are passed into the `scg` function (provided by Dr. Anderson [2]), which runs for the specified number of iterations (or terminates sooner if one of the levels of precision is reached) and returns the final optimized weights. The final lines of the method unpack the weight matrices and return a list of the various parts of the calculation.

One other note about the `makeNN` function is that the two functions required by `scg` are defined within the `makeNN` method itself; they have been snipped in the below code for clarity.

The full code can be seen below; all code past the “Dr. Anderson” comment was provided (in slightly different form) by Dr. Anderson [6].

```

makeNN <- function(Xtrain, Ttrain, numHiddenUnits, lambda=0, nIterations
  =10000, xPrecision=0, fPrecision=0)
{
  # sqErrorF function code

  # gradF function code

```

```

standardizeX <- makeStandardize(Xtrain)
XtrainS <- standardizeX$standardize(Xtrain)
standardizeT <- makeStandardize(Ttrain)
TtrainS <- standardizeT$standardize(Ttrain)

XtrainS <- cbind(1, XtrainS)

V <- matrix(rnorm((length(XtrainS[1,])) * numHiddenUnits, mean=0, sd
  =0.01), length(XtrainS[1,]), numHiddenUnits)
W <- matrix(rnorm((numHiddenUnits+1) * length(TtrainS[1,]), mean=0, sd
  =0.01), numHiddenUnits+1, length(TtrainS[1,]))

# Dr. Anderson - assignment 5 writeup
scgResult <- scg(pack(V, W), sqErrorF, gradF, nIterations=nIterations,
  xPrecision=xPrecision, fPrecision=fPrecision, ftracep=TRUE)

VW <- unpack(scgResult$x, length(Xtrain[1,]), numHiddenUnits, length(
  Ttrain[1,]))
V <- VW$V
W <- VW$W

return(list(V=V, W=W, standardizeX=standardizeX, standardizeT=
  standardizeT, lambda=lambda, ftracep=scgResult$ftrace, XtrainS=
  XtrainS, TtrainS=TtrainS))
}

```

2.2.2 useNN

Once the neural network has been built, the next step is to use it. In order to make this simple, the `useNN` function was defined. This function is fairly straightforward; it first standardizes the test input and adds a constant 1 bias column, then runs a single forward pass through the trained neural network. It then returns the “unstandardized” version of the output of the neural net. Full code can be seen below:

```

useNN <- function(nnet, Xtest)
{
  XtestS <- nnet$standardizeX$standardize(Xtest)
  XtestS <- cbind(1, XtestS)

  neural_out <- cbind(1, (tanh(XtestS %*% nnet$V))) %*% nnet$W

  return(nnet$standardizeT$unstandardize(neural_out))
}

```

2.3 Does the Neural Network Work?

After the code to build the neural network is complete, it is necessary to test it to make sure it functions properly. In order to do this, a simple neural network was constructed on data from the function $y = x - 1$. This neural network had one input node, one output node, and one hidden unit (additionally, there was a constant 1 input node and a constant 1 hidden node). The code to generate the data and train and run the neural network over it was:

```

numHiddenUnits <- 1
Xtrain <- matrix(c(1, 2, 3))
Ttrain <- matrix(c(0, 1, 2))

```

```
Xttest <- matrix(c(-1, 2, 5, 3, 7)) # result should be -2, 1, 4, 2, 6
```

```
nnet <- makeNN(Xtrain, Ttrain, numHiddenUnits)
print(useNN(nnet, Xttest))
```

One interesting feature of this net that became immediately apparent was the affect that the number of input samples had on the output. In the example above, where three inputs are provided, the results generated by the network bear little resemblance to the targets. However, as more input values are added, the results rapidly become much closer. The result of running the same experiment with three and five inputs is:

	3 inputs	5 inputs	target
[1,]	-0.1570159	-1.982707	-2.0000000
[2,]	1.0000000	0.999173	1.0000000
[3,]	2.1570158	3.999580	4.0000000
[4,]	2.0000000	2.000004	2.0000000
[5,]	2.1579183	5.982647	6.0000000

The reason for this discrepancy is that, when there are a very small number of samples, the dataset is undersampled to the point where the neural network is unable to properly model the underlying function. The simplest case of this would be a single input and single output sample; in this case, the model would always predict the same value, as it would only be able to model a constant function for the single input value. This is not a failing of the network, and as such a larger data sampling is necessary to properly test the network.

Aside from the issue of making sure to use enough training data, the simple neural network does a good job of modeling the target function. However, it does not usually converge (particularly as more data samples are used), meaning that it ran for the maximum number of iterations allowed (by default 10,000) without reaching a precision value that was lower than the specified threshold (by default 0). This does not mean that the resulting network is unusable, but rather that, had it been allowed to train for more iterations, that a slightly better set of weights may have been attainable. However, the precision values are still very small and thus the lack of convergence is not a major drawback to the net.

3 Applying a Neural Network to a Noisy Sine Curve

After verifying that the neural network training and testing algorithm works on trivial datasets, we turn to building and using a neural net on a more meaningful dataset, in this case a set of data taken from a “noisy” sine curve. The “noisy sine” data was generated with the following code (provided by Dr. Anderson [6])

```
# Dr. Anderson - assignment5 writeup
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)
```

3.1 The Neural Network in Action

The first part of this experiment was simply to get a feel for how the neural net works on a “real” dataset. To this end, several different parts of the net’s functionality are visualized in Figures 1 and 2. These graphs were all generated on a single neural network, built using 10 hidden units and a lambda value of 0 (so that weight decay was not applied).

The first graph in Figure 1 shows the root mean square error over the training epochs and was generated while training the neural net. At first, the RMSE is quite high (as this is before the weights have had much

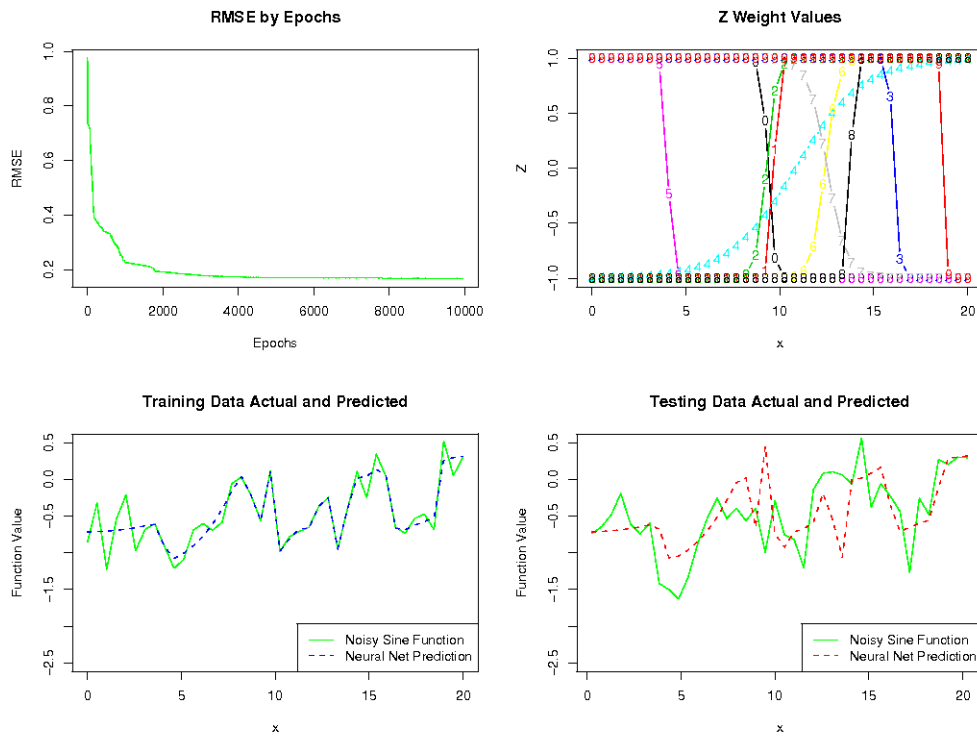


Figure 1: Neural network trained on “noisy sine” dataset: Graph 1: RMSE over epochs; Graph 2: Hidden unit values over data range; Graphs 3 & 4: Training and testing data (actual and predicted) respectively

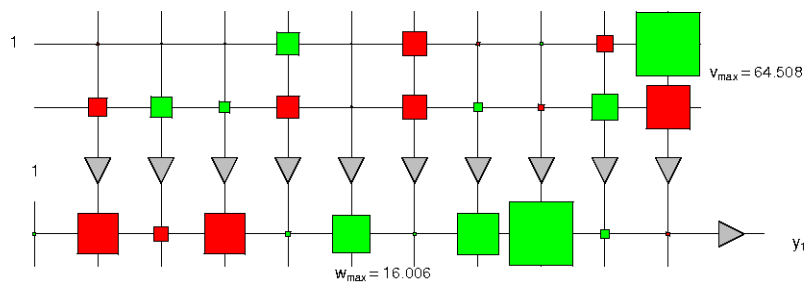


Figure 2: Neural network trained on “noisy sine” dataset: neural network weights. The larger a weight’s box, the more influential it is; green is positive influence, red is negative influence.

chance to change based on what the neural net “learns” and are still essentially random), but it drops off quickly after a few epochs. However, once the RMSE has dropped significantly (in this case at around 1000-2000 epochs), little further progress is made. This is because at this point, the error being fed back into the net is small enough that it does not greatly affect the weights, so each further iteration barely updates the weight values. The second graph in Figure 1 illustrates what each hidden unit is contributing to the overall output; each line represents the *tanh* function for that hidden unit. The shift (left or right of center) is determined by whether the output weights are positive or negative (negative shifts the curve right), and the magnitude of the shift is greater the greater the absolute value of the weight is. The steepness of the curves is determined by the magnitude of the input weighting for that node (steeper means a larger weight), with the direction (sloping “down” or “up”) determined by whether the weight is positive or negative (positive means an “upward” slope). A visual representation of the input and output weights can be seen in Figure 2, where the top boxes are the input weights and the bottom boxes are the output weights. A green box means a positive weight and a red means a negative weight, and the magnitude of the weight corresponds to the size of the box.

The third and fourth graphs in Figure 1 represent the results given by the neural network on training and testing datasets from the “noisy sine”. The third graph corresponds to the results returned by the net for the same data that it was trained on, whereas the fourth graph shows the results of using the same network to predict a second (test) dataset. The solid green lines represent the actual data, while the blue and red dotted lines represent the neural network’s predictions.

The code to generate the five graphs is shown below; the graphical representation of the neural network weights (2) was provided by Dr. Anderson [5].

```
nnet <- makeNN(Xtrain , Ttrain , numHiddenUnits=10, lambda=0)

# plot1: RMSE
matplot(seq(0,length(nnet$ftracep)-1), nnet$ftracep , type="l" , col="green" ,
        xlab="Epochs" , ylab="RMSE" , main="RMSE by Epochs")

# plot2: output of hidden units
matplot(Xtrain , tanh(nnet$XtrainS %*% nnet$V) , type="b" , lty=1, pch=c("0" ,"1" ,
        "2" ,"3" ,"4" ,"5" ,"6" ,"7" ,"8" ,"9") , col=seq(1,10) , xlab="x" , ylab="Z" , main=
        "Z Weight Values")

# plot3: training data
matplot(matrix(c(Xtrain , Xtrain) , length(Xtrain[,1]) , 2) , matrix(c(Ttrain ,
        useNN(nnet , Xtrain)) , length(Xtrain[,1]) , 2) , ylim=c(-2.5, 0.5) , type="l" ,
        col=c("green" , "blue") , xlab="x" , ylab="Function Value" , main="Training
        Data Actual and Predicted")
legend("bottomright" , c("Noisy Sine Function" , "Neural Net Prediction") , lty=c
        (1,2) , col=c("green" , "blue"))

# plot4: test data
matplot(matrix(c(Xtest , Xtest) , length(Xtest[,1]) , 2) , matrix(c(Ttest , useNN(
        nnet , Xtest)) , length(Xtest[,1]) , 2) , ylim=c(-2.5, 0.5) , type="l" , col=c("
        green" , "red") , xlab="x" , ylab="Function Value" , main="Testing Data Actual
        and Predicted")
legend("bottomright" , c("Noisy Sine Function" , "Neural Net Prediction") , lty=c
        (1,2) , col=c("green" , "red"))

# plot5: neural network diagram
drawNNet(nnet) # Dr. Anderson code - drawNNet.R
```

The first plot in Figure 1 is generated by graphing the values returned by `ftracep` in the `scg` function. This vector records the RMSE for each epoch, which is what the first graph is meant to display. The second plot is a graph of the Z values for each hidden unit (10 in this case), which for this neural net is a simple

tanh function that operates on the input values and the V weights coming into each hidden unit. The third and fourth graphs are generated from the actual function values (represented by `Ttrain` and `Ttest`) and the predictions the neural net makes on the corresponding set of data (generated by calls to `useNN` with the appropriate dataset). The last graph, shown in Figure 2, is generated by a call to the provided `drawNNet` code; the full code of this function is long and can be found in `drawNNet.R` [5].

3.2 Affects of Varying the Number of Hidden Units

For the previous test, the number of hidden units was arbitrarily chosen to be 10. However, there is no reason to assume that 10 is the “best” number of hidden units, so we now explore what the affects of varying the number of hidden weights are. For this experiment, the same “noisy sine” data set was used to train and test the networks, which were built with a number of hidden units varying from 1 to 20. For each network, the RMSEs for the training dataset and a testing dataset were recorded; the results can be seen in Figure 3. The code to do the test and generate the graphs can be seen below:

```

results <- c()
for (iter in 1:20)
{
  nnet <- makeNN(Xtrain, Ttrain, numHiddenUnits=iter, lambda=0)
  trainRMSE <- mean((useNN(nnet, Xtrain) - Ttrain)^2)
  testRMSE <- mean((useNN(nnet, Xtest) - Ttest)^2)

  newRow <- c(trainRMSE, testRMSE, iter)
  results <- rbind(results, newRow)
}

matplot(results[,3], results[,1:2], type="l", lty=1, col=c("blue", "red"),
        xlab="Number of Hidden Units", ylab="RMSE", main="RMSE by Number of Hidden
        Units")
legend("bottomleft", c("Training Data", "Test Data"), lty=1, col=c("blue", "
red"))

```

The RMSE values are calculated based on the difference between the actual output values and those predicted by the neural net. At this stage, both the actual outputs and those from the neural network are “unstandardized” which is why the RMSE values are large.

3.3 Results

One interesting aspect of the simple test (the one that uses a 10-hidden-unit neural network to predict the “noisy sine” training and testing data) is that the training data does a decent job of fitting the training data, but when the same network is applied to the testing dataset, it does not do nearly as well. Instead, it generates a curve of a very similar shape to that of the one generated for the training data, but this curve does not match up very well to the testing data. In fact, near the center of the testing graph (graph 4 in Figure 1), there are a couple of places where the neural net predicts a “spike” that goes in the opposite direction from the actual test data. Considering how closely the two predicted curves match each other, this is likely due to overfitting; there are likely too many hidden units, which allows the neural net to fit the training data with enough detail to pick up the noise in the data. This does not then transfer well to a different dataset with different noise.

The behavior of the neural networks created in the experiment that varied the number of hidden units seems to back this conclusion up. The RMSE values (see Figure 2) for the training and testing data behave quite differently from each other; while the training data RMSE continues to drop as more hidden units are added (although the RMSE decrease becomes less after around 10-11 hidden units), the testing data RMSE begins to increase at around the 4-5 hidden unit mark. This increase illustrates the tendency of a neural network with too many hidden units to overfit data, as can be seen in the simple 10-hidden-unit network. Several hidden units are necessary to properly model complex data such as a sine function (at least when

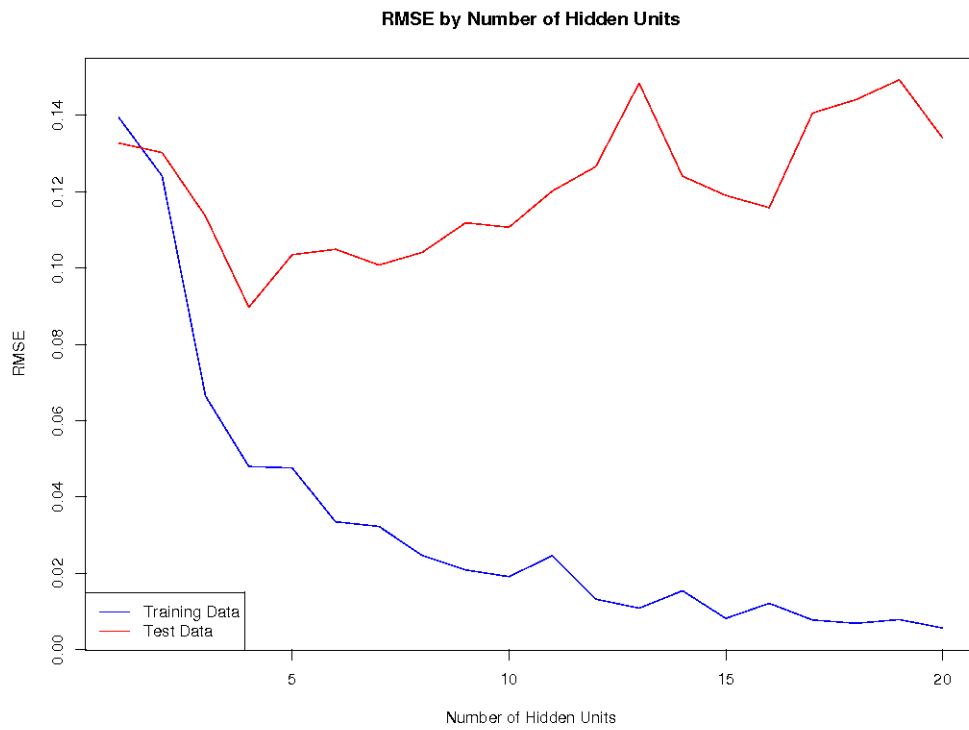


Figure 3: RMSE values for neural network predictions on training and testing data for the “noisy sine” dataset versus number of hidden units.

the nonlinear function is *tanh*), but past a certain point this advantage starts to become a liability. In this particular case, the RMSE dips again at around the 15 hidden unit mark, but it shortly climbs again. This dip is not seen in repetitions of the same experiment, which says that in this case it was likely just a “lucky” prediction by the neural network (in this particular graph, the decrease also looks unusually large as the neural network at 14 had an unusually high RMSE). Clearly, 10 hidden units is well past the point where adding more hidden units leads to a higher RMSE for the testing data set due to overfitting, which reinforces the behavior seen in the simple test.

4 Applying a Neural Network to the Automobile MPG Dataset

The final experiment done in this report is applying Scaled Conjugate Gradient neural nets to a “real” dataset, in this case a dataset dealing with automobile miles per gallon data. For this experiment, two values, miles per gallon and horsepower, were predicted by the neural nets. This experiment also sought to determine what the “best” number of hidden units and the best value of lambda are for this dataset.

4.1 Handling the Data

The first challenge of working with the automobile mpg dataset is getting the data into a format that can be used by the neural network. The automobile mpg dataset is a little more complex than many datasets in that it has a few samples that do not have all data values. Instead, these values are represented with a ‘?’ value, which R is unable to interpret numerically. The code below reads in the dataset and cleans it up by removing the incomplete datasets, as well as removing the last column, which is a text name for each car (which is not a useful attribute to be used by the neural net). This code is a slightly modified version of code presented by Dr. Anderson [3].

```
readData <- function()
{
  auto <- read.table("auto-mpg.data")
  autonames <- c("mpg", "cylinders", "displacement", "horsepower", "
    weight", "acceleration", "year", "origin", "name")
  colnames(auto) <- autonames
  auto <- as.matrix(auto)

  # remove names column
  auto <- auto[,1:8]

  # strip out values that contain a ?
  auto <- auto[apply(auto != "?", 1, all),]

  auto <- apply(auto, 2, as.numeric)
  return(auto)
}
```

After cleaning up the data, the next step is to partition it into training and testing sample sets. This is done with the following code (the first part of the code is a slightly modified version of code provided by Dr. Anderson [3]):

```
partitionData <- function(perTrain, perTest, auto)
{
  ## DR. ANDERSON CODE ## - day1.pdf
  auto <- auto[sample(length(auto[,1])),] ## randomly rearrange data
  train <- auto[1:floor(perTrain * length(auto[,1])),]
  test <- auto[-(1:floor(perTrain * length(auto[,1])),)]

  # results fields are MPG and horsepower
}
```

```

Xtrain <- train[,-c(1,4)]
Ttrain <- train[,c(1,4)]
Xtest <- test[,-c(1,4)]
Ttest <- test[,c(1,4)]

return(list(Xtrain=Xtrain, Ttrain=Ttrain, Xtest=Xtest, Ttest=Ttest))
}

```

Once the data has been correctly partitioned, it can be used to build, train, and evaluate neural networks.

4.2 How Many Hidden Units is Best?

The first part of this experiment deals with determining what number of hidden units is the “best” for the automobile mpg dataset. In this case, “best” is determined as the number of hidden units that leads to the lowest RMSE values for the testing data. The reason that the RMSE values for training data do not enter into the “best” definition is that at some point, increasing accuracy in predictions on training data comes at the expense of accuracy on testing data as the neural net increasingly overfits to the training data.

The R code to determine the “best” number of hidden units is very similar to the code described in a previous section that showed the affect of changing the number of hidden units for the “noisy sine” dataset. The only major change is that there are two outputs, mpg and horsepower, that need RMSE values to be calculated. A sequence of values for number of hidden units from 1 to 20, with a step of 4 was used, and the lambda value was fixed at 0 (the “best” lambda value was analyzed in a later experiment). The train and test RMSE values for both outputs can be seen in Figure 4, and the code that calculated the RMSE values and generated the figure is shown below:

```

results <- c()
for (iter in seq(1, 20, by=4))
{
  print(c("on iter:", iter))
  nnet <- makeNN(data$Xtrain, data$Ttrain, iter)

  trainRes <- useNN(nnet, data$Xtrain)

  TRmpgRMSE <- mean((trainRes[,1] - data$Ttrain[,1])^2)
  TRhpRMSE <- mean((trainRes[,2] - data$Ttrain[,2])^2)

  testRes <- useNN(nnet, data$Xtest)

  TEmpgRMSE <- mean((testRes[,1] - data$Ttest[,1])^2)
  TEhpRMSE <- mean((testRes[,2] - data$Ttest[,2])^2)

  newRow <- c(TRmpgRMSE, TRhpRMSE, TEmpgRMSE, TEhpRMSE, iter)
  results <- rbind(results, newRow)
}

matplot(results[,5], results[,1:4], type="l", lty=1, col=c("blue", "cyan", "red", "magenta"), xlab="Number of Hidden Units", ylab="RMSE", main="RMSE by Number of Hidden Units")
legend(4, 250, c("Training Data (MPG)", "Test Data (MPG)", "Training Data (Horsepower)", "Test Data (Horsepower)"), lty=1, col=c("blue", "cyan", "red", "magenta"))

```

The results for this experiment are interesting, although not as conclusive as might be expected. The test RMSE for horsepower behaves somewhat similarly to that of the “noisy sine” data, in that it at first drops and then rises as more hidden units are added. However, the increase is much less pronounced than with the “noisy sine” data, only showing a clear increase after 14 iterations, and even by 19, the RMSE is

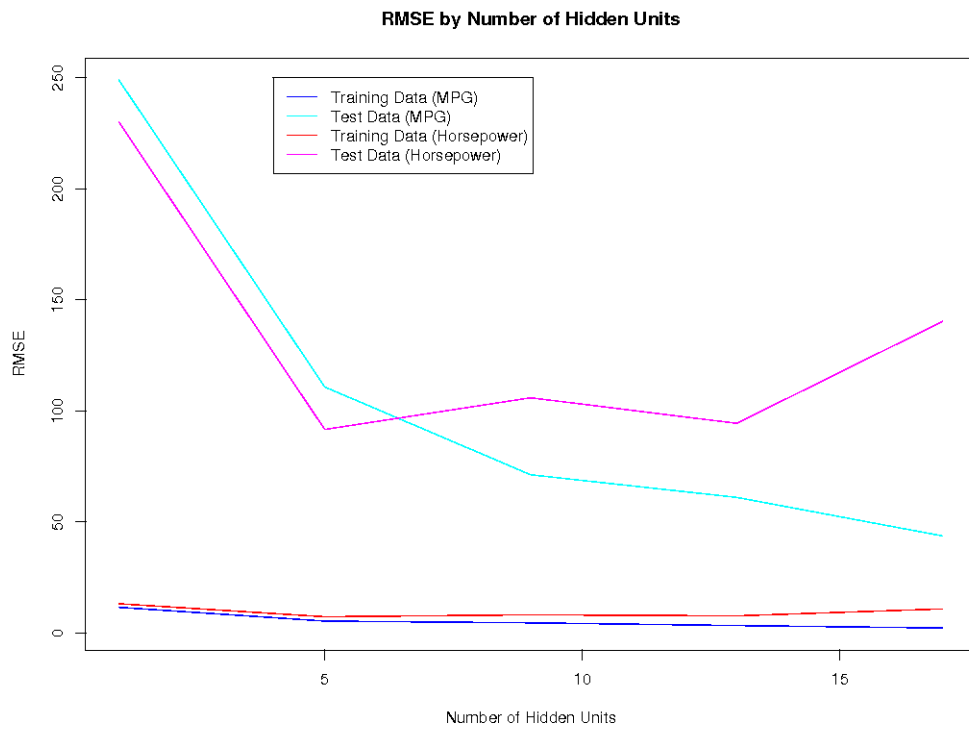


Figure 4: RMSE values for neural network predictions on training and testing data for the automobile mpg dataset versus number of hidden units.

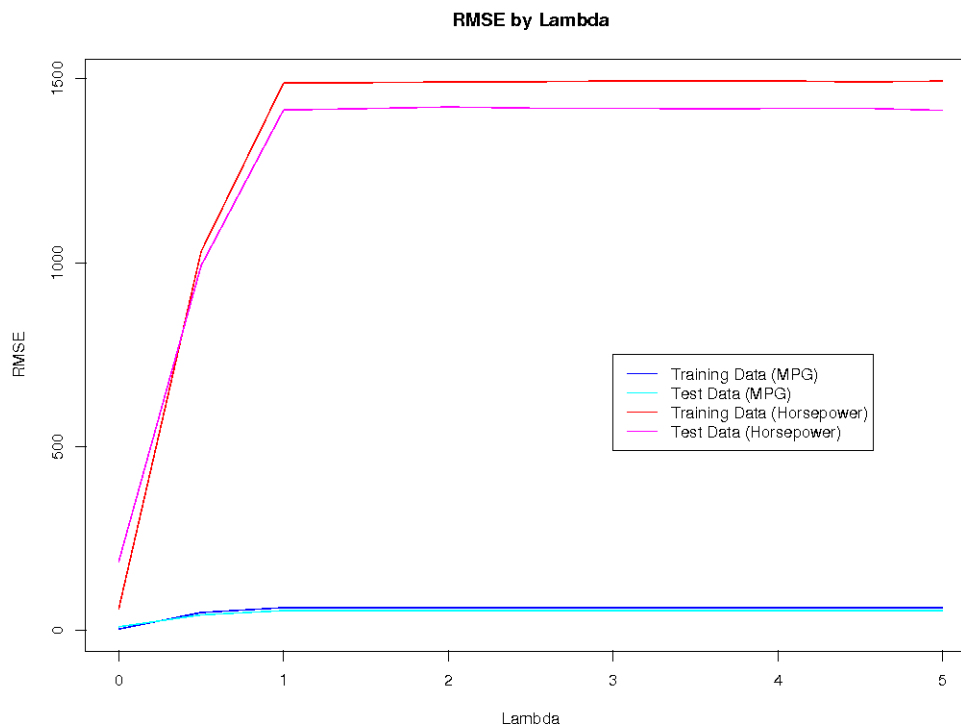


Figure 5: RMSE values for neural network predictions on training and testing data for the automobile mpg dataset versus lambda value.

less than the initial testing RMSE value. The mpg value, however, does not show the increase in RMSE that would be expected; instead, it continues to decline as more hidden units are added. This would seem to imply that the neural network for the automobile mpg dataset is less prone to overfitting than it is with the “noisy sine” dataset. This may be due to there being less noise, or the data being for fully sampled, in the automobile mpg dataset. If either of these is the case, particularly the less noise hypothesis, then there would not be much for the net to overfit to.

From the results, it would appear that the “best” number of hidden units for this dataset is somewhere around 5-6. This is the range where the horsepower testing RMSE is the lowest, and it is also the point where the mpg testing RMSE begins to level off. Ideally, the number of hidden units would be one where both RMSE values are the lowest, but that is unlikely to happen on a real dataset. In this case, this appears to be a good compromise as the mpg’s RMSE does not increase a huge amount more as more hidden units are added, and the horsepower’s RMSE begins to climb, albeit modestly, after this point. It would probably be worth running this same experiment multiple times, and determining whether the “best” point falls in roughly the same area of the curves each time.

4.3 What is the Best Lambda Value?

The experiment to determine the “best” lambda value is quite similar to the one to determine the “best” number of hidden units detailed above. The R code is virtually identical; the only change is that the number of hidden units was set to 5 (as this was determined by the previous experiment to be close to optimal) and kept fixed, while the lambda value was varied from 0 to 5, with a step of 0.5. The train and test RMSE values for horsepower and mpg can be seen in Figure 5.

For both the mpg and horsepower outputs, increasing the lambda value increased the RMSE, but the affects on the two outputs was very different. While both showed increases, the mpg was barely affected at all compared to the horsepower. This may be partly an issue of scales; since horsepower has a much larger

range, its “unstandardized” RMSE values are quite a bit larger than those of mpg. However, this is not the entire story, and the affect of an increased lambda on mpg was still significantly less. Another interesting feature of this graph is that once lambda reached a certain point (around 1), increasing it further made virtually no difference. This is likely the point at which the RMSE is unable to increase any more; this could be caused by the predicted value being forced all the way to zero, at which point RMSE simply becomes the training value squared.

Clearly, at least for the automobiles mpg dataset, the best lambda value is 0. This is not entirely surprising given the results from the hidden unit experiment above; on a dataset with low noise, which the previous experiment indicates might be the case for the automobile mpg dataset, lambda is not particularly necessary. This is because weight decay is primarily used to dampen noise; in a “quiet” dataset, all that it will accomplish is penalizing useful weights, eventually driving them all towards 0 as the lambda increases. However, in a dataset such as the “noisy sine” one, weight decay would likely be much more useful as it would correctly eliminate at least some of the noise that the neural network would otherwise learn.

5 Conclusion

One of the most interesting things about this project to me was the ability of an incredibly simple algorithm to predict a fairly complex dataset. Ultimately, a few simple matrix multiplications and the *tanh* function were able to model even complex functions with a high degree of accuracy. While the neural nets are by no means perfect, they are significantly better than the classifiers that we have used in previous projects. It is amazing to me how much data can be represented with these few data structures and simple algorithms.

Another thing that I learned from this project was about the ways in which these neural nets fall short. Whereas the previous classifiers were more of an “all-or-nothing” failure (either the classifier fit the data or it did not), with the neural networks, it is easy to see where the predictions fail. Another aspect of this is how clear the overfitting problem is; for example, on the “noisy sine” data, it is easy to compare the predictions the neural nets make on the training and testing data, and see what went wrong in predicting the test data. To me this is instructive as it provides a nice visual representation of what the neural net is actually learning internally.

The hardest part of this project was keeping the dimensionality of the various weight, input, and output matrices straight. While a correctly-built neural net is quite powerful, it is easy to get the dimensionality wrong, which leads to unusual results. For example, flipping the dimensions on the input and output matrices lead to the neural net learning to weight the input values at essentially 0, and use the W matrix in conjunction with the constant 1 W weight to drive the outputs. This created a neural net that would always produce the training data, regardless of what test data was fed into it. However, it was not immediately obvious what the problem was, which led to a significant amount of debugging effort in trying to track it down.

References

- [1] Anderson, C., *Lecture Notes - CS545: Linear Modeling*, <http://www.cs.colostate.edu/~anderson/cs545/Lectures/week3day2/week3day2.pdf>, 2009.
- [2] Anderson, C., *gradientDescents.R*, <http://www.cs.colostate.edu/~anderson/cs545/assignments/gradientDescents.R>, 2009.
- [3] Anderson, C., *Lecture Notes - CS545: Bishop (2.3-2.5, 3.1)*, <http://www.cs.colostate.edu/~anderson/cs545/Lectures/week3day1/day1.pdf>, 2009.
- [4] Anderson, C., *Lecture Notes - CS545: Artificial Neural Networks*, <http://www.cs.colostate.edu/~anderson/cs545/Lectures/week8day1/week8day1Handouts-2x2.pdf>, 2009.
- [5] Anderson, C., *drawNNet.R*, <http://www.cs.colostate.edu/~anderson/cs545/assignments/drawNNet.R>, 2009.
- [6] Anderson, C., *Assignment 5 Writeup: Nonlinear Regression with Neural Networks*, <http://www.cs.colostate.edu/~anderson/cs545/assignments/assignment5.html>, 2009.

- [7] UCI Machine Learning Repository, *Automobile Miles Per Gallon Dataset*, <http://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/auto-mpg.data>, 1993.