CS545: Assignment 5

Christopher Mullins

October 27, 2009

Contents

1 Introduction **1**

2 Implementation and Testing of Neural Network **1**
2.1 Cost Function and its Gradient 1
2.2 `makeNN` and `useNN` 2
2.3 Application to Simple Data Set 5

3 Noisy Sine Curve **6**
3.1 Results 6

4 Miles Per Gallon Data **8**
4.1 Finding Good Values for λ and Hidden Unit Count 9
4.2 Applying Optimized Model 9

1 Introduction

The purpose of this assignment is to implement and experiment with a one-layer neural network. The implementation uses a version of the scaled conjugate gradient algorithm provided by Dr. Anderson [2], and allows for variation on the maximum number of iterations SCG is allowed, a value λ that controls the magnitude of the weights vector, and the number of hidden units in the neural network.

I apply my implementation of a neural network to a variety of data sets. Two contrived data sets, the automobile miles per gallon data explored in lecture, and the particle physics data used in the past two assignments.

2 Implementation and Testing of Neural Network

2.1 Cost Function and its Gradient

In order to implement a neural network, it is necessary to provide a function that calculates the error between predicted data and actual data (sometimes called “cost function”). Since our purpose is to estimate continuous data, the mean squared error can be used.

As we intend to parametrize our model with a value λ that indicates how much we want to penalize models with large weights, the implementation of the squared error function should also include this as a consideration. The implementation follows.

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

```
    # Unpack V and W
    V <- unpacked$V
    W <- unpacked$W

    # Forward pass through the neural network
    Z <- tanh(Xtrain %*% V)
    Y <- cbind(1, Z) %*% W

    # Penalize the V weights.
    weights <- V[-1, ]
    lt <- sum(lambda * weights^2)

    # Calculate mean squared error
    return( mean((Y - Ttrain)^2) + lt )
}
```

The gradient function is used by SCG to settle into a local minima of the mean squared error function. It calculates the derivatives of `sqErrorF` with respect to v and w within the neural network. The implementation of the gradient function follows.

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

    # Unpack V and W
    V <- unpacked$V
    W <- unpacked$W

    # Forward pass through the neural network
    Z <- tanh(Xtrain %*% V)
    Y <- cbind(1, Z) %*% W

    # Calculate errors
    error <- Y - Ttrain

    # Calculate lambda term for gradV
    lambdaV <- (lambda * rbind(0, V[-1, ]))

    # Calculate gradient w.r.t. V
    nk    <- (nrow(Xtrain) * ncol(Ttrain))
    gradV <- (1/nk) * t(Xtrain) %*% (error %*% t(W[-1, ]) * (1 - Z^2)) + lambdaV
    gradW <- (1/nk) * t(cbind(1,Z)) %*% error

    # Pack the results and return
    return( pack(gradV, gradW) )
}
```

2.2 makeNN and useNN

The "guts" of the code for `makeNN` are mostly comprised of `sqErrorF` and `gradF`. However, the full implementation is given below.

```
makeNN <- function(Xtrain, Ttrain, numHiddenUnits, lambda = 0, xPrecision = 0,
    fPrecision = 0, nIterations = 10000, ftrace = FALSE, updates = TRUE,
    huFn = tanh)
```

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

        # Unpack V and W
        V <- unpacked$V
        W <- unpacked$W

        # Forward pass through the neural network
        Z <- huFn(Xtrain %*% V)
        Y <- cbind(1, Z) %*% W

        # Penalize the V weights.
        weights <- V[-1, ]
        lt <- sum(lambda * weights^2)

        # Calculate mean squared error
        return( mean((Y - Ttrain)^2) + lt )
    }
    #################################################
    gradF <- function(weights)
    {
        unpacked <- unpack(weights)

        # Unpack V and W
        V <- unpacked$V
        W <- unpacked$W

        # Forward pass through the neural network
        Z <- huFn(Xtrain %*% V)
        Y <- cbind(1, Z) %*% W

        # Calculate errors
        error <- Y - Ttrain

        # Calculate lambda term for gradV
        lambdaV <- (lambda * rbind(0, V[-1, ]))
```

```
        # Calculate gradient w.r.t. V
        nk    <- (nrow(Xtrain) * ncol(Ttrain))
        gradV <- (1/nk) * t(Xtrain) %*% (error %*% t(W[-1, ]) * (1 - Z^2)) + lambdaV
        gradW <- (1/nk) * t(cbind(1,Z)) %*% error

        # Pack the results and return
        return( pack(gradV, gradW) )
    }
    ##################################################

    # Standardize Xtrain, Ttrain.
    XstdF <- makeStandardizeF(Xtrain)
    TstdF <- makeStandardizeF(Ttrain)
    TusdF <- makeUnstandardizeF(Ttrain)

    Xtrain <- XstdF(Xtrain)
    Ttrain <- TstdF(Ttrain)

    # Compute appropriate dimensions
    ni <- ncol(Xtrain)
    no <- ncol(Ttrain)
    nh <- numHiddenUnits

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

    # Add constant column of 1s to Xtrain
    Xtrain <- cbind(1, Xtrain)

    # Use SCG to find a local minima
    scgResult <- scg( pack(V,W), sqErrorF, gradF,
        nIterations = nIterations,
        xPrecision = xPrecision, fPrecision = fPrecision,
        ftracep = ftrace, updates = updates
    )

    # Unpack result.
    unpacked <- unpack(scgResult$x)
    V <- unpacked$V
    W <- unpacked$W

    return(list(
        V = V, W = W, XstandardizeF = XstdF, TstandardizeF = TstdF,
        TUnstandardizeF = TusdF, ftrace = scgResult$ftrace, huFn = huFn
    ))
}
```

The code for `useNN` is relatively simplistic, as it simply makes a forward pass and "unstandardizes" its predictions. My implementation is given below.

```
useNN <- function(model, Xtest, returnNN = FALSE)
{
    # Standardize and add constant 1 column to Xtest
```

```
    Xtest <- model$XstandardizeF(Xtest)
    Xtest <- cbind(1, Xtest)

    V <- model$V
    W <- model$W

    # 1. Calculate output of neural network
    Z <- model$huFn(Xtest %*% model$V)
    Y <- cbind(1, Z) %*% W

    # 2. Unstandardize the output and return the result
    return(model$TUnstandardizeF(Y))
}
```

2.3 Application to Simple Data Set

For the simple data set, I train the neural network on the function (defined in R)

`x^3 + 2000*sin(x) + 1000*rnorm(length(x)).`

This is superficially more complex than the noisy sine data examined in the next section, but the amount of noise with respect to the range of f is significantly lower in this example. The maximum number of iterations allowed was set as 500. I used $\lambda = 0$, and 6 hidden units. The test data is generated in the same way as suggested for the noisy sine curve data. The results of this application are shown in figure 1.

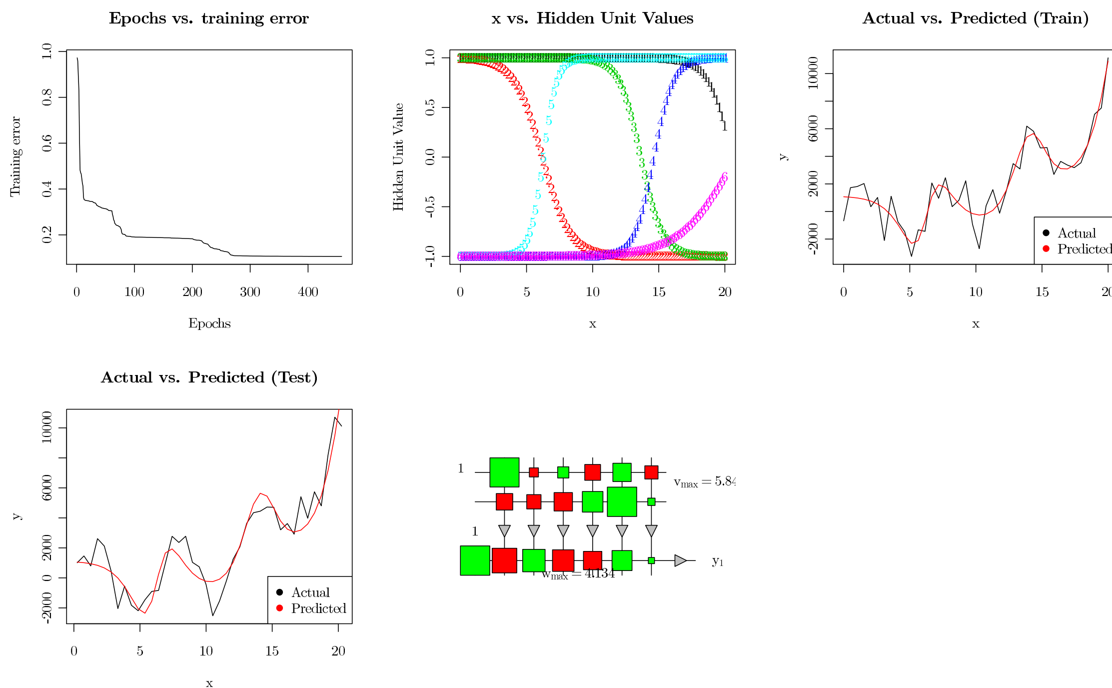

Figure 1: Results for simple data set.

The results suggest, first and foremost, that the neural network behaves as we hope it would. It fits a nice curve to the training data, and the curve fits nicely over the test data.

In order to test that λ is affecting the results as we want it to (intuitively, forcing the model to be more linear), I tested the neural network on the same data with $\lambda = 1$. The results from this test are shown in

figure 2. The results indicate that the effects of λ are what we expect them to be. With the same parameters (other than $\lambda = 1$), the resulting model is much more linear than the previous results.

A neural network is linear when the weights on the hidden units are small. Since a higher value of λ enforces this, we should expect to see small values for the hidden unit weights. As we can see in the last plot of the results, this is what happens.

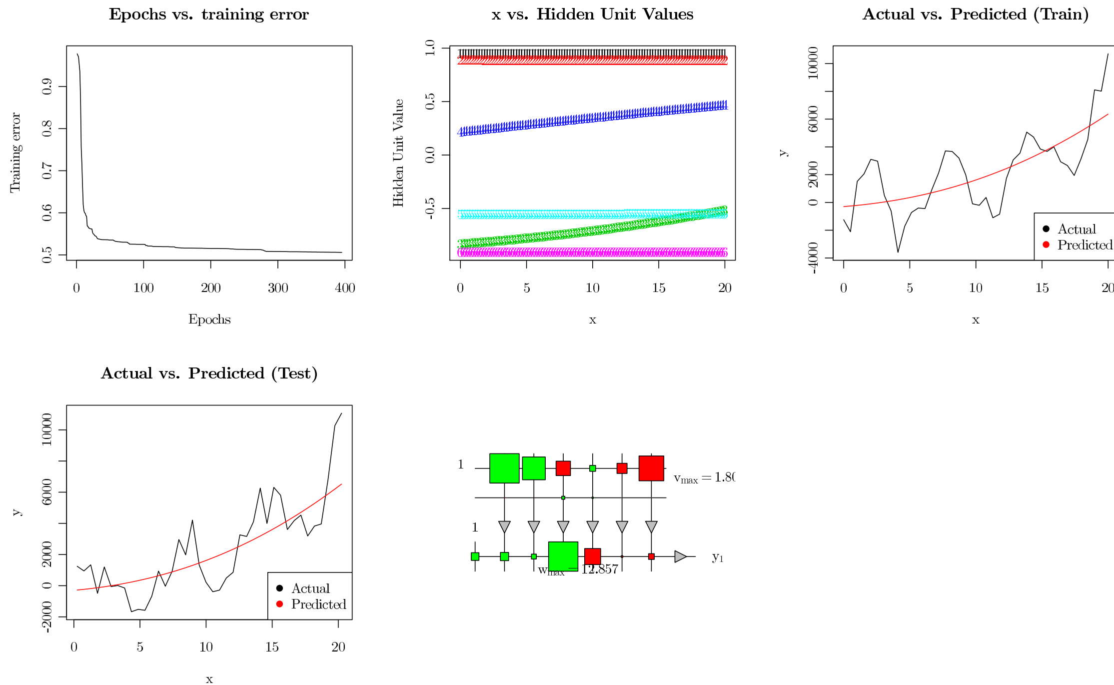

Figure 2: Results for simple data set with $\lambda = 1$.

3 Noisy Sine Curve

The noisy sine curve data is generated using the following expression (in R).

```
-1 + 0.05 * x + 0.4 * sin(x) + 0.3 * rnorm(length(x))
```

As shown in the results (mentioned soon), this data tends to be very noisy. The sine curve is barely noticeable. This helps to accentuate the effects of the number of hidden units on the resulting error. As suggested in the assignment specifications, I use 10 hidden units and $\lambda = 0$. I allow SCG to converge to a minima for no more than 500 iterations. This decision was made by changing the value and observing the effects on the test error.

3.1 Results

The results of applying the neural network to this data are shown in figure 3. As we can see, the model fits the training data well, but it looks slightly misfit when overlaid on the testing data. To some degree, this is expected given the amount of noise in the data, but it could suggest overfitting.

Varying the number of hidden units can have a significant effect on the test error. If there are too few, the model is too simplistic and is not fit the data well. If there are too many, the model can overfit the training data and introduce significant error when applied to future observations. I vary the number of hidden units from 1 to 100 and measure the RMSE. Because SCG is somewhat stochastic (since we start at a random position), there is some variation in the results. As such, it is a good idea to take several samples from the

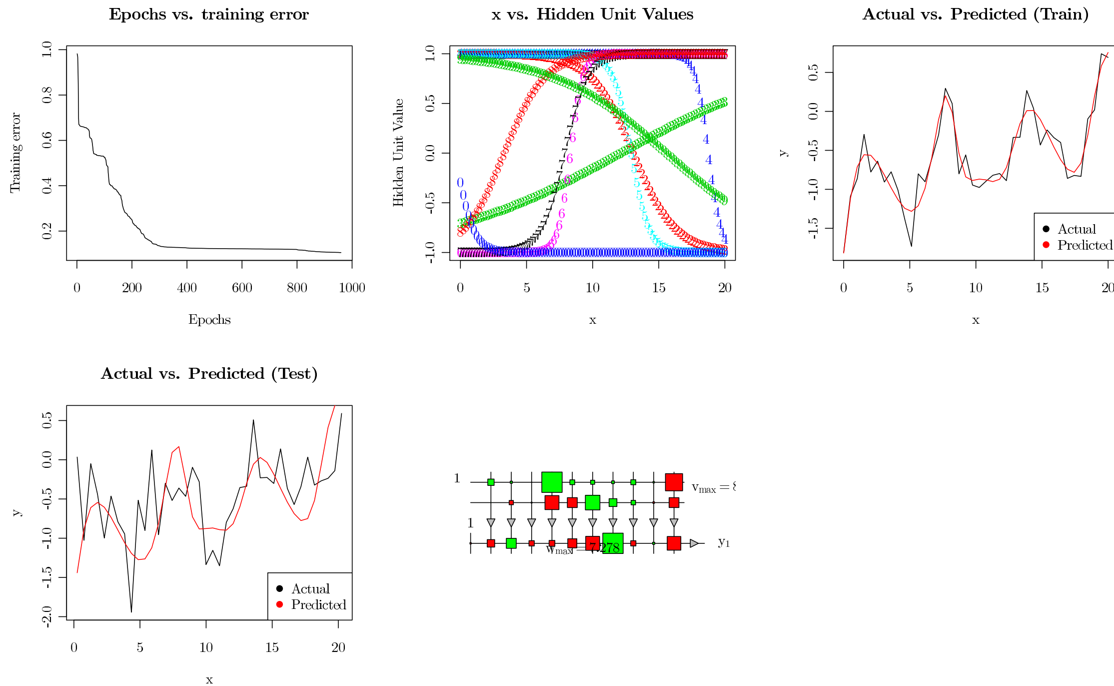

Figure 3: Results for noisy sine data.

same set of parameters and use the mean of these samples as our observation. I repeat each measurement 100 times.

The code I used to take these measurements is shown below. A plot of these results is shown in figure 4. As we might expect, the training error decreases as the number of hidden units increases. However, the test error begins to increase as the model begins to overfit the training data.

Initially, I only allowed SCG a maximum of 1,000 iterations to converge. This produced some odd results. The testing error reached a minimum at about 10 hidden units, and then climbs, as expected (as the model begins to overfit the testing data). However, after approximately 20 hidden units, it began to slowly trend downwards. Likewise, the training error reached a minimum, and slowly began to trend upwards. I hypothesized that this was because I was terminating SCG prematurely, and that as the model became more complex (more hidden units), it took a longer time to overfit the training data. As such, the testing error decreased as the model became less overfit, and the training error increased. In order to test this hypothesis, I increased the maximum number of iterations to 2,500 and repeated the same experiment. These are the results depicted in the figure. As we can see, both the testing and training errors plateau as we would expect (confirming that the hypothesis is likely correct).

```
testNH <- function(data, nIterations = 2500, nhMax = 20, reps = 10)
{
    errs <- c()
    terrs <- c()

    for (i in 1:nhMax)
    {
        print(sprintf("Running reps for nh = %d", i))
        runErrs <- c()
        runTerrs <- c()

        for (j in 1:reps)
```

```
        {
model <- makeNN(data$Xtrain, data$Ttrain, i, nIterations =
                      nIterations, updates = FALSE)
            p     <- useNN(model, data$Xtest)
            tp    <- useNN(model, data$Xtrain)

            runErrs <- c(runErrs, sqrt(mean((p - data$Ttest)^2)))
            runTerrs <- c(runTerrs, sqrt(mean((tp - data$Ttrain)^2)))
        }

        errs  <- c(errs, mean(runErrs))
        terrs <- c(terrs, mean(runTerrs))
    }

    plot(1:nhMax, errs, type = "l", main = "Hidden Unit Count vs. RMSE",
        xlab = "# Hidden Units", ylab = "Test RMSE")
    lines(1:nhMax, terrs, type = "l", col = "red")
    legend("bottomleft", legend = c("Test RMSE", "Train RMSE"),
       pch = 19, col = c(1, "red"))

    return(list(
        testErr = errs, trainErr = terrs
    ))
```

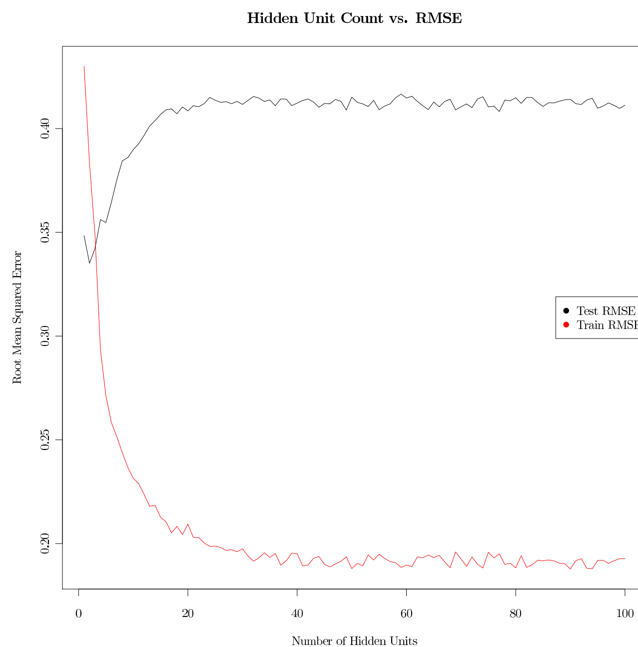

Figure 4: Results for varying the number of hidden units vs. test/train error in noisy sine curve data.

4 Miles Per Gallon Data

The data in this section are taken from the University of California, Irvine Machine learning repository[3]. The data are measurements from a collection of car models, including number of cylinders, displacement,

horsepower, weight, acceleration, model year, origin, and miles per gallon. The purpose of this section is to examine the effectiveness of our neural network in predicting the horsepower *and* the miles per gallon of a car given the other measurements. Particularly, we would like to find a good combination of values for λ and the number of hidden units.

Some of the measurements in the observations are missing (indicated by a `?`). I chose to ignore the observations where one or more measurement was missing, as we did with this data when we applied a linear model to it. The code to load, parse, and partition the data is given below. Some of the code is adapted from [1].

Some of the measurements in this data are multi-valued/discrete. As each of them is represented by a unique integer, however, I saw no need in converting these measurements to indicator variables. The standardization of the input data (done automatically in `makeNN`) will take care of the inconsistent ranges in the data.

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

### Remove all samples that have at least one "?"
keepRows <- apply(mpg != "?", 1, all)
mpg <- mpg[keepRows,1:8]
mpg <- apply(mpg, 2, as.numeric)

part <- partition(mpg, 0.8)

iCols <- c(2,3,5:8)
oCols <- c(1,4)

Xtrain <- part$trainRows[ , iCols]
Ttrain <- part$trainRows[ , oCols]

Xtest <- part$testRows[ , iCols]
Ttest <- part$testRows[ , oCols]
```

4.1 Finding Good Values for λ and Hidden Unit Count

As in the previous sections, I am allowing SCG to run for a maximum number of iterations. Because this is an entirely different dataset, I experimented with varying this and found that a value of approximately 3,700 seems to work best. A plot of the max number of iterations vs. test RMSE is shown in figure 5. This data was generated by sampling the same set of parameters (number of epochs, λ, etc.) 100 times and varying the maximum number of epochs. As the plot suggests, there was a large amount of variation in the data. Some of the smaller "dips" and "peaks" in the data can likely be attributed to this variation.

In order to test for a good value of λ, I fix the maximum number of epochs at 3,700 and vary λ from 1 to 10 in increments of approximately $\frac{1}{3}$. A plot of the results for this test is shown in figure 6. This data was just as noisy as the data in the previous experiment, but the results are obvious without running more repetitions. For $\lambda > 0$, the test RMSE sharply and significantly increases (from < 6 to > 8). From this, we can conclude that $\lambda > 0$ does not work well with this data.

The same procedure was applied in order to determine a good value for the number of hidden units. The results are shown in figure 7. From this plot, we can see that a good pick for the number of hidden units is around 20.

4.2 Applying Optimized Model

After running these tests, I generated the the same plots we used to test the effectiveness of a regression algorithm in assignment 2. On one axis lies the actual value, and on the other is the predicted value. The

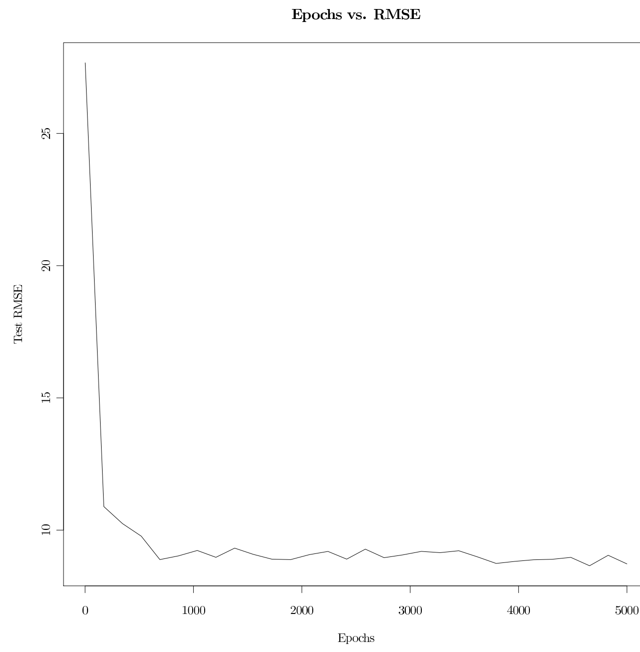

Figure 5: Results for varying the number of SCG epochs vs. testing RMSE.

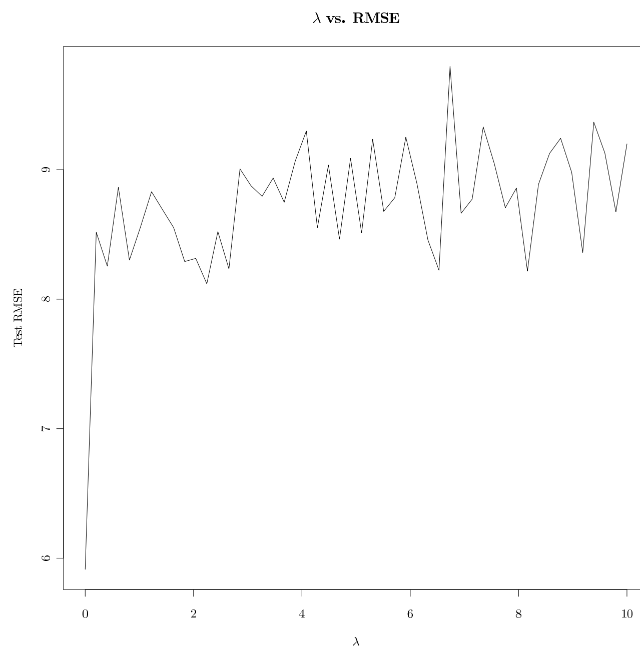

Figure 6: Results for varying λ vs. testing RMSE.

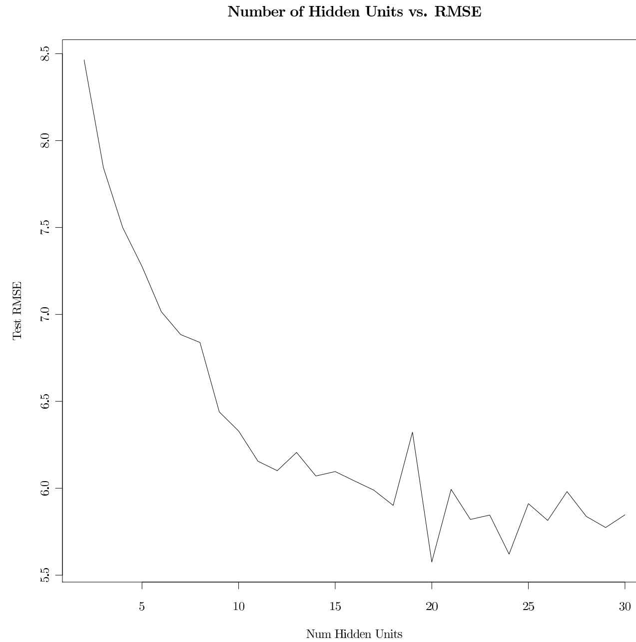

Figure 7: Results for varying number of hidden units vs. testing RMSE.

more linear these plots look, the more accurate the regression model is. The plots were generated separately for horsepower and miles per gallon, and each has a plot for the testing and training data. These plots are shown in figure 8.

We can see that, as we would expect, the model is very good at predicting the training data for both miles per gallon and horsepower. The model also applies very nicely to the test data for both. However, we can see that (especially for horsepower), the model is worse at predicting higher values for both horsepower and miles per gallon. This is evidenced by the data points that straggle further from the $y = x$ line closer to the top of the plots. Overall, though, this appears to be an excellent model for this data.

References

[1] C. Anderson. R code applying linear model to mpg data, September 2009. http://www.cs.colostate.edu/ anderson/cs545/Lectures/week3day1/mpg.R.

[2] C. Anderson. Scaled conjugate gradient algorithm implementation, October 2009. http://www.cs.colostate.edu/ anderson/cs545/assignments/gradientDescents.R.

[3] A. Asuncion and D.J. Newman. UCI machine learning repository, 2007.

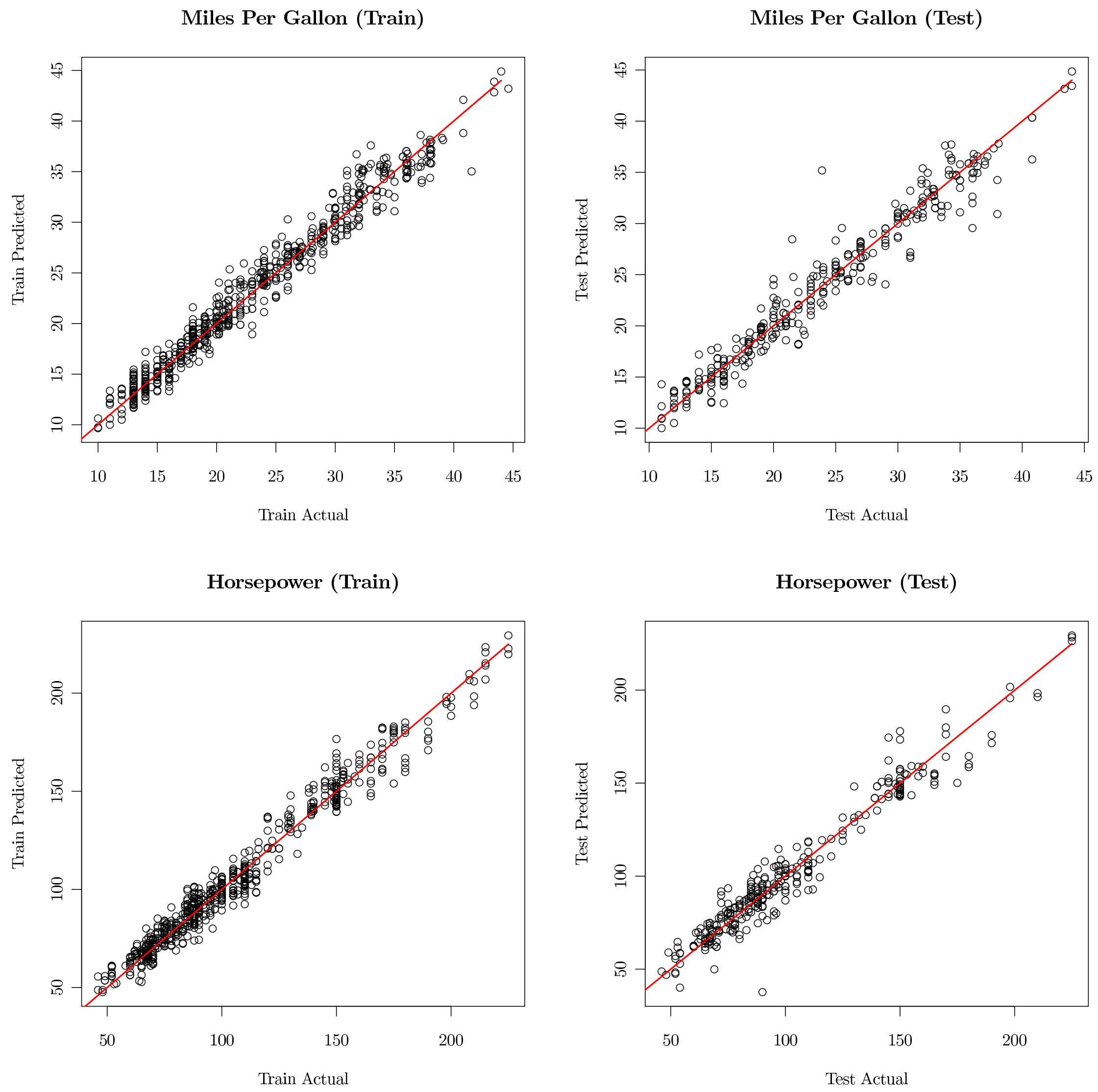

Figure 8: Results of applying "optimized" neural network to the miles per gallon data.