

CS545: Assignment 5

Greg Carter

October 29, 2009

Contents

Contents	1
1. Introduction	2
2. A Neural Network Implementation	3
2.1. Structure of the Model.....	3
2.2. Using the Neural Network Model	5
2.3. Results of the Neural Network from some Simple Generated Data	5
3. Predicting “Noisy” Data.....	7
3.1. Some Details from the 10-Hidden Unit Model	10
4. Predicting MPG and Horse Power with a Neural Network	12
5. Conclusion.....	16
References	21

1. Introduction

Machine learning models that rely upon linear combinations of fixed basis functions have uses that are generally limited to data with smaller dimensionality. For data with larger dimensionality, we would like the basis functions to be adjusted to fit the data. To achieve this objective, one technique is to augment our single-layer logistic regression model with a second, “hidden” layer, in which these basis functions (known as “activation” functions in this hidden layer) are weighted with values that are determined at training time. Figure 1 provides a visual reference for this two-layer logistic regression model known as a Neural Network.

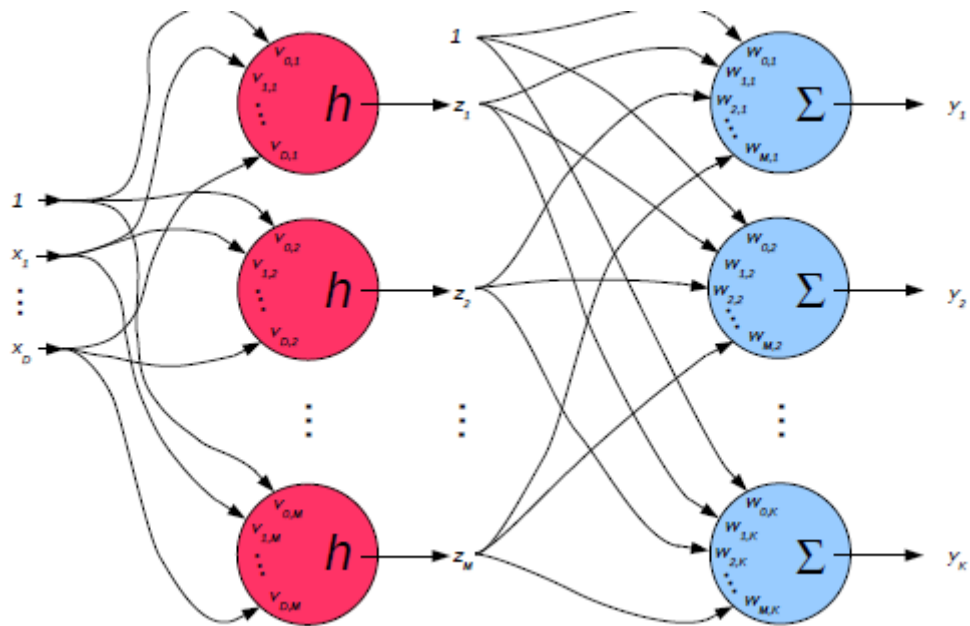


Figure 1. A logistic regression model, in blue, augmented with another, similar model known as the “hidden layer”. The blue layer is known as the “output” layer. In this model the basis functions, h , are parameterized with weights, v , whose values are determined at training time. Such a model, known as a Neural Network, extends the applicability of the simple logistic regression model to data with larger dimensionalities of input since the basis functions are adapted to the data.[1]

While neural networks typically provide for faster model evaluation than other “data-adaptive” techniques, the likelihood function exhibits multiple local maximums so that the model created, depending upon the techniques used to create it, may not be optimal.[2]

As before, the choice of the basis function, h , in the hidden layer (also known as the “activation” function) is an interesting question. Considering the original desire in the development of this model to simulate a neuron, we might consider the observation that a neuron either fires an electrical impulse down its axon or not according to its weighted inputs and some “firing threshold”. But such a discrete-valued function, not being continuous, does not lend itself to mathematical derivation. Instead, two popular choices for h , since they exhibit properties similar to the discrete “no fire/fire” function while being continuous, are the hyperbolic tangent and the so-called sigmoid function given by

$$\sigma(a) = \frac{1}{1 + e^{(-a)}}.$$

In this paper we explore the development of a neural network in R, and its success in application to various generated and non-generated data sets.

2. A Neural Network Implementation

2.1. Structure of the Model

For this example, we will use the hyperbolic tangent for our hidden layer activation function. In R:

```
h <- function(a)
{
  tanh(a)
}
```

We will use the Scaled Conjugate Gradient, SCG, algorithm[3] to adjust our hidden and output layer weights so that our model settles into a local minimum of mean squared error. Therefore, the function to minimize will calculate the mean squared error, and we will use a weight-decay penalty of the hidden layer weights (less the constant 1 input weight).¹

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

  a = Xtrains1 %**% upw$V

  Z <- h(a)
  Y <- cbind(1,Z) %**% upw$W

  error <- Y - Ttrains

  #
  # Implement weight decay (penalty on weight magnitudes)
  # for all hidden layer weights except the constant 1 input weights.
  #
  # Make sure we the right dimensioned matrix in V - a D*M x 1 matrix
  # in order to get a scaler added to our mean square error.
  #
  return(mean(error^2) + lambda*(t(matrix(upw$V[-1,]))%**matrix(upw$V[-
1,])))
}
```

Given the function to minimize, the gradient function as required by SCG is given by

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

  a = Xtrains1 %**% upw$V

  Z <- h(a)
  Y <- cbind(1,Z) %**% upw$W

  error <- Y - Ttrains
```

¹ Utility functions *pack* and *unpack* manage the hidden layer and output layer weights *V* and *W* respectively as a single-column matrix for convenience in passing between functions.

```

# Calculate the gradient of squared error for hidden and output layer
weights.
# Make sure we have a matrix.
#
if (!is.matrix(upw$W[-1,]))
{
  if (ni == 1)
  {
    WwoC <- (matrix(upw$W[-1,]))
  } else
  {
    WwoC <- t(matrix(upw$W[-1,]))
  }
} else
{
  WwoC <- upw$W[-1,]
}

if (!is.matrix(upw$V[-1,]))
{
  if (ni == 1)
  {
    VwoC <- t(matrix(upw$V[-1,]))
  } else
  {
    VwoC <- (matrix(upw$V[-1,]))
  }
} else
{
  VwoC <- upw$V[-1,]
}

gradEwrtV <- (2*(1/N)*(1/K)) * t(Xtrains1) %*% (error %*% t(WwoC) * (1-
Z^2)) + (lambda * rbind(0,VwoC))

gradEwrtW <- (2*(1/N)*(1/K)) * t(cbind(1,Z)) %*% (error) # No output layer
penalty per the assignment.

return(pack(gradEwrtV, gradEwrtW))
}

```

Here we must take pains to ensure that our weight matrices are of proper dimension. This is done not only by considering whether the weights after unpacking are considered matrices by R, but also by considering the number of dimensions, ni , in the input. Our constant bias input is row 1 in both the V and W weights.

Data and targets will be normalized before submitting to SCG, requiring us to unnormalize the model output for comparisons to our true targets. Initial weights for V and W are determined as random values uniformly distributed between -0.01 and 0.01. Taken all together, our setup and call to SCG is as follows:

```

# Obtain data characteristics.
#
ni <- ncol(Xtrain)
nh <- numHiddenUnits
no <- ncol(Ttrain)

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

# Standardize inputs
#
standardizeXF <- makeStandardizeF(Xtrain)
Xtrains <- standardizeXF(Xtrain)

standardizeTF <- makeStandardizeF(Ttrain)
Ttrains <- standardizeTF(Ttrain)

# Create our unstandardize function for the training data.

```

```

# We'll use it in a bit.
#
unStandardizeTF <- makeUnStandardizeF(Ttrain)

#
# Create the initial V (hidden layer weights) and
# W (output layer weights) as random values uniformly
# distributed between -0.01 and 0.01.
#
V <- matrix(runif((ni)*nh, min=-0.01, max=0.01), ni, nh)
V <- rbind(1,V)
W <- matrix(runif((nh)*no, min=-0.01, max=0.01), nh, no)
W <- rbind(1,W)

#
# Add bias column to input data.
#
Xtrains1 <- cbind(1,Xtrains)

#
# Use Scaled Conjugate Gradient to find the "refined" weights V and W
#
scgResult <- scg(pack(V,W), sqErrorF, gradF,
                 nIterations = nIterations,
                 xPrecision = xPrecision, fPrecision = fPrecision,
                 xtracep = FALSE, ftracep = TRUE )

```

2.2. Using the Neural Network Model

After creating the model for a given set of training data and their targets as described in section 2.1, the model is applied for a given set of inputs, X , with the following:

```

#
# Standardize input data
#
Xs <- model$standardizeXF(X)

#
# Add bias column to input data.
#
Xs1 <- cbind(1,Xs)

#
# Calculate output of neural network.
#
a = Xs1 %*% model$V

Z <- h(a)
Y <- cbind(1,Z) %*% model$W

#
# Return unstandardized target predictions for the input data.
#
list(Y = model$unStandardizeTF(Y), Z = Z)

```

Here, Z is the output of the hidden layer with Y being the standardized predicted targets (refer to Figure 1).

2.3. Results of the Neural Network from some Simple Generated Data

As an initial test of the neural network, we will generate a small amount of quadratic data. Using the following:

```

f <- function (x) -(x^2) + x + 30
nSamples <- 40
xmax <- 10
Xtrain <- matrix(seq(-xmax,xmax,length=nSamples),nSamples,1)
Ttrain <- f(Xtrain)
Xtest <- Xtrain + xmax/nSamples/2
Ttest <- f(Xtest)

```

Using 1 hidden unit initially, a neural network model was created with the training data and applied to both the training data and the testing data. The output predictions of the model indicate a deficiency in accurately predicting over half of the targets. See Figure 2.

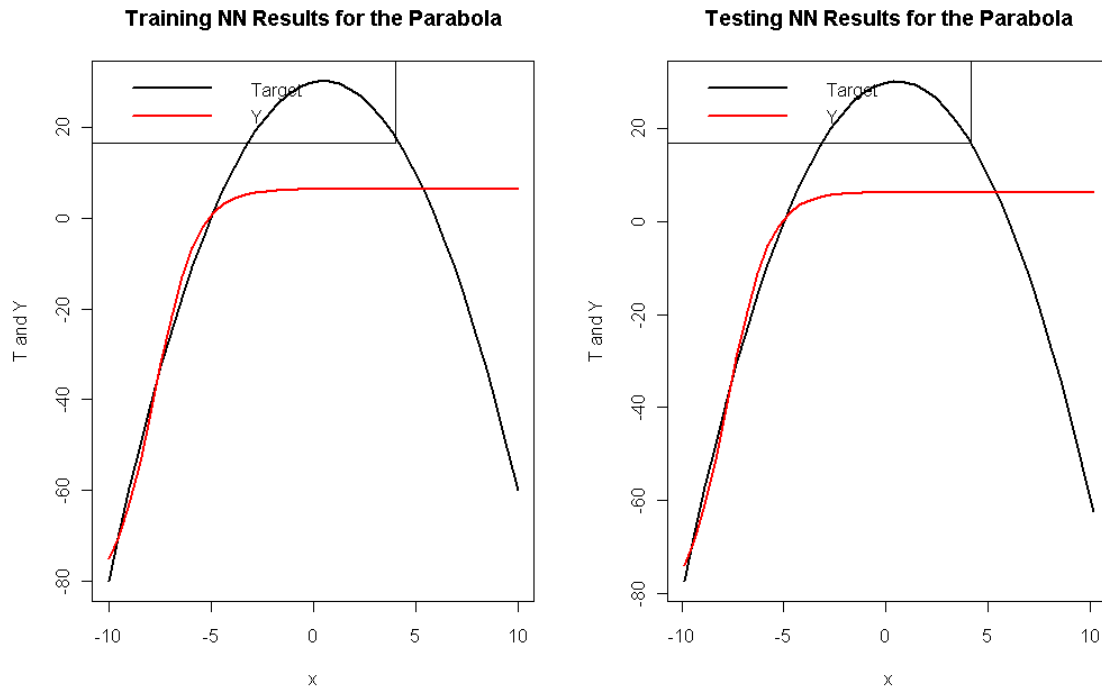


Figure 2. The neural network using 1 hidden unit applied to some generated quadratic data.

Running the same set of data through a neural network with 2 hidden units yields much better results as shown in Figure 3.

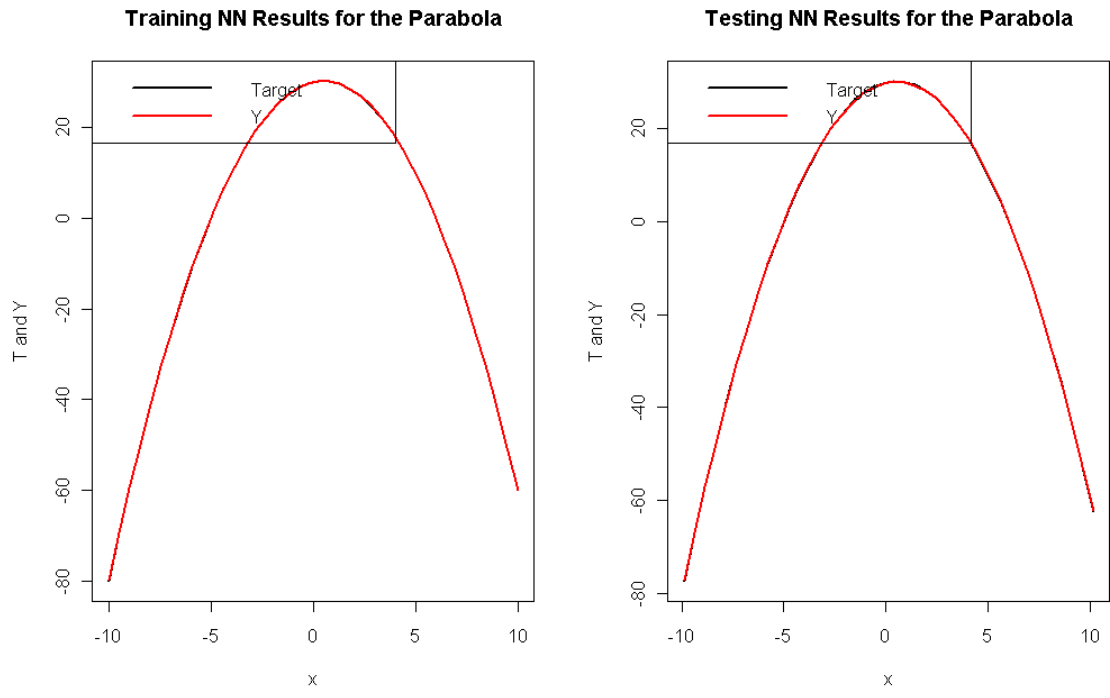


Figure 3. The neural network using 2 hidden units applied to the same data used in Figure 2.

The impact of the number of hidden units on the success of a neural network becomes immediately obvious. But the deficiency of the model with one hidden unit may also be an indication of the deficiency of the model in using the hyperbolic tangent as the activation function for this set of data. It may be that a periodic activation function such as sine may have produced results with a single hidden unit comparable to those of a 2-hidden unit network does with the hyperbolic tangent.

3. Predicting “Noisy” Data

A set of data was generated using a sine-based function in which normally distributed random value were introduced to create “noise”.

```
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)
```

Using 10 hidden units, a neural network model was created and applied to both the training and testing data. For this data, the model success was somewhat mixed as given in Figure 4.

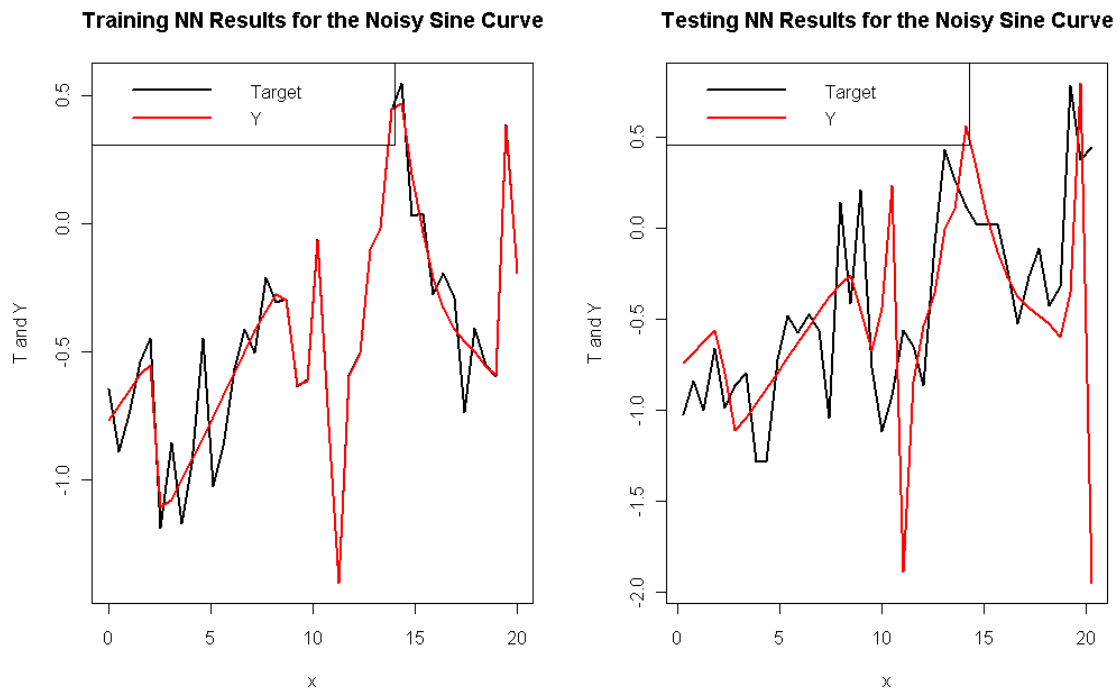


Figure 4. Predictions for the training and testing data generated with a “noisy” sine curve.

An examination of the RMSE for the training data in this case indicates that the model has trained fairly well. See Figure 5.

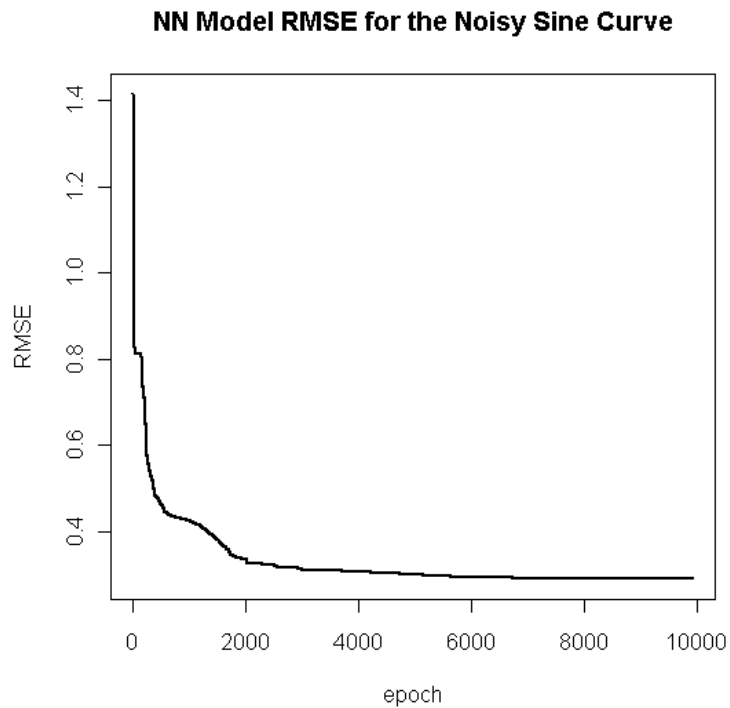


Figure 5. RMSE for the standardized noisy sine curve data using the neural network model with 10 hidden units.

However, the difficulties that the model has in applying its training to the test data is revealed by considering the RMSE for both the training and testing data while varying the number of hidden units between 1 and 20. See Figure 6. The code used to produce the results and draw the graph follows:

```
TrainRMSE <- NULL
TestRMSE <- NULL
for (hu in 1:20)
{
  model <- makeNN(Xtrain, Ttrain, hu)
  nsineTrainNNPred <- useNN(model, Xtrain)
  nsineTestNNPred <- useNN(model, Xtest)

  TrainRMSE <- rbind(TrainRMSE, sqrt(mean((Ttrain - nsineTrainNNPred$Y)^2)))
  TestRMSE <- rbind(TestRMSE, sqrt(mean((Ttest - nsineTestNNPred$Y)^2)))
}

dev.new()
matplot(matrix(1:20), cbind(TrainRMSE, TestRMSE), lty=1, type="l", lwd=2, xlab="Hidden
Units", ylab="RMSE",
        main="NN RMSE for the Noisy Sine Curve")
legend("topleft", c("Training", "Testing"), lty=1, lwd=2, col=1:2)
```

NN RMSE for the Noisy Sine Curve

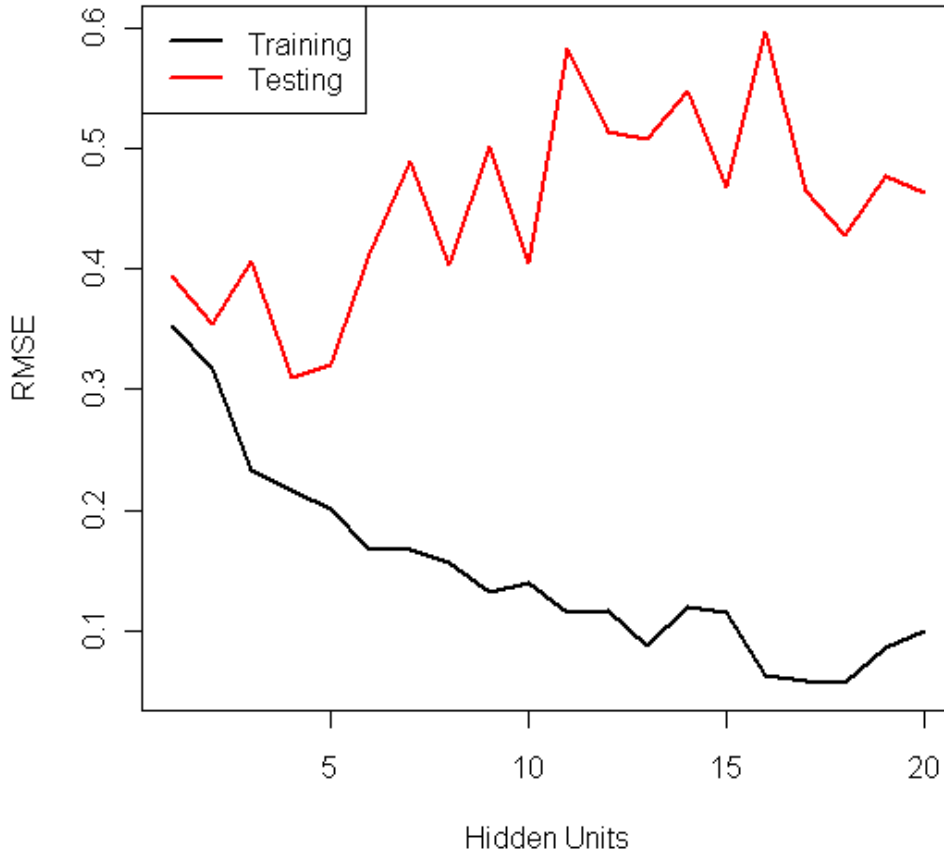


Figure 6. A comparison of RMSE between the training and testing data over a varying number of hidden units between 1 and 20.

While the general tendency as the number of hidden units increases is for the model to predict better for the training data, the testing data noticeably tends to the opposite direction, getting increasingly worse. This is an example of the non-linear neural network over fitting the training data as the number of hidden units increase.

While we might be tempted to put the lowest RMSE on the testing data at about 4 hidden units, a more prudent approach, since we can not, in general, assume that new data will be like the testing data, is to partition the data into three parts, 60% training, 20% validation and 20% testing. Then select the number of hidden units given by the validation data as yielding the lowest RMSE while reporting the RMSE yielded by the testing data for that number of hidden units as the expected RMSE. This approach tends to neutralize differences between the validation and testing data.

3.1. Some Details from the 10-Hidden Unit Model

Going back to the original 10-hidden unit model for this data, we may observe the output of the hidden units for the first 20 samples in the input as given in

NN Model Hidden Unit Output for the Noisy Sine Curve

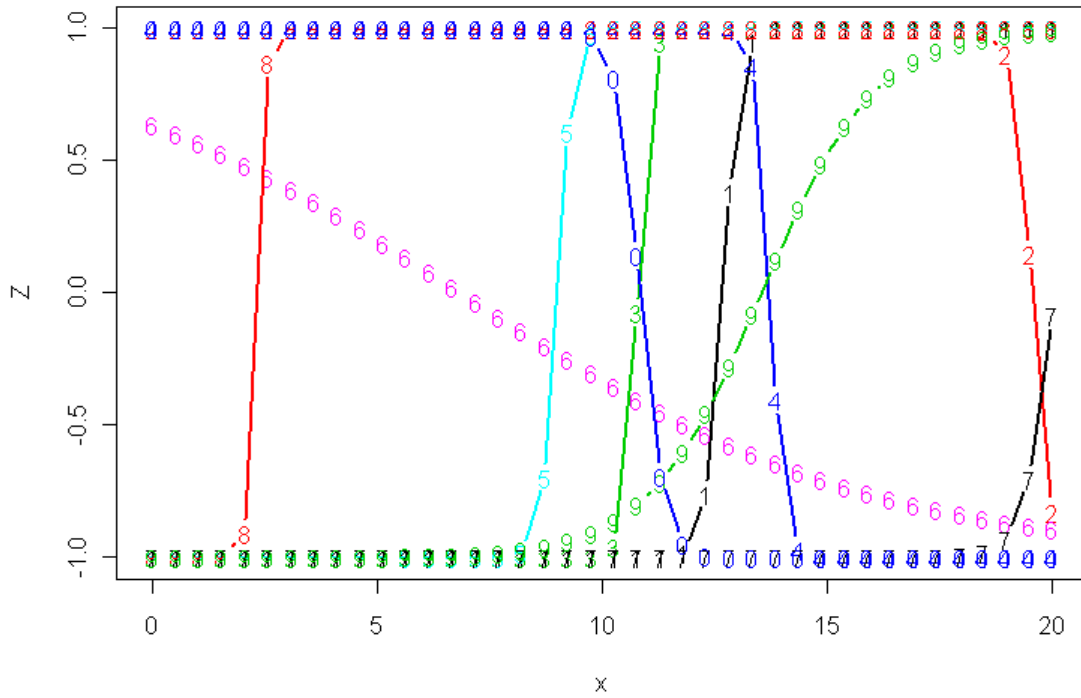


Figure 7. Output of the 10 hidden units in the neural network model for the noisy sine data.

The presentation of this graph is probably best considered along side a graph of the neural network weights as given in Figure 8.

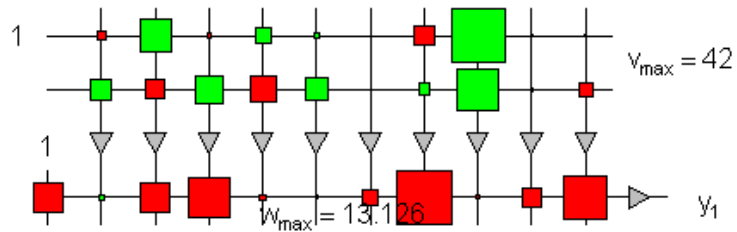


Figure 8. A visualization of the hidden layer and output layer weights in the 10-hidden unit neural network trained on the noisy sine data.

From these graphs it appears that this network is a “rough” fit of the data. That is, as x changes, the weights on the activation functions deviate sharply. Hidden units 6 and 9 are the only ones that maintain a relatively smooth fit to the data in terms of their weight transitions. But the sharp changes in weights on the other hidden units indicate that their respective inputs are sharply contrasting with each other indicating that a reduction in hidden units may be warranted.

4. Predicting MPG and Horse Power with a Neural Network

A set of automobile data was acquired from the UCI Machine learning repository for testing the neural network.[5] This data set includes measures of automobile attributes MPG, Cylinders, Displacement, Horsepower, weight, acceleration, model year and origin. With this data we will train a neural network to predict MPG and Horsepower given the other attributes.

The data is first read and cleansed using the following R code:

```
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

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

#
# Convert all values to numeric, excluding the name column.
#
mpg <- apply(mpg[, 1:8], 2, as.numeric)
```

Then an 80/20 training/testing partition was made.

```
#
# Randomly pick 80% of samples for training partition.
#
randorder <- sample(nrow(mpg))
nSamples <- round(nrow(mpg)*0.8)
trainRows <- randorder[1:nSamples]

Xtrain <- mpg[trainRows, c(2, 3, 5, 6, 7, 8)]
Ttrain <- mpg[trainRows, c(1, 4), drop=FALSE]

Xtest <- mpg[-trainRows, c(2, 3, 5, 6, 7, 8)]
Ttest <- mpg[-trainRows, c(1, 4), drop=FALSE]
```

Using 3 hidden units, a neural network model was trained and applied to both the training and testing partitions. The results were rather impressive. First, consider the accuracy of predictions as given in Figure 9 and Figure 10.

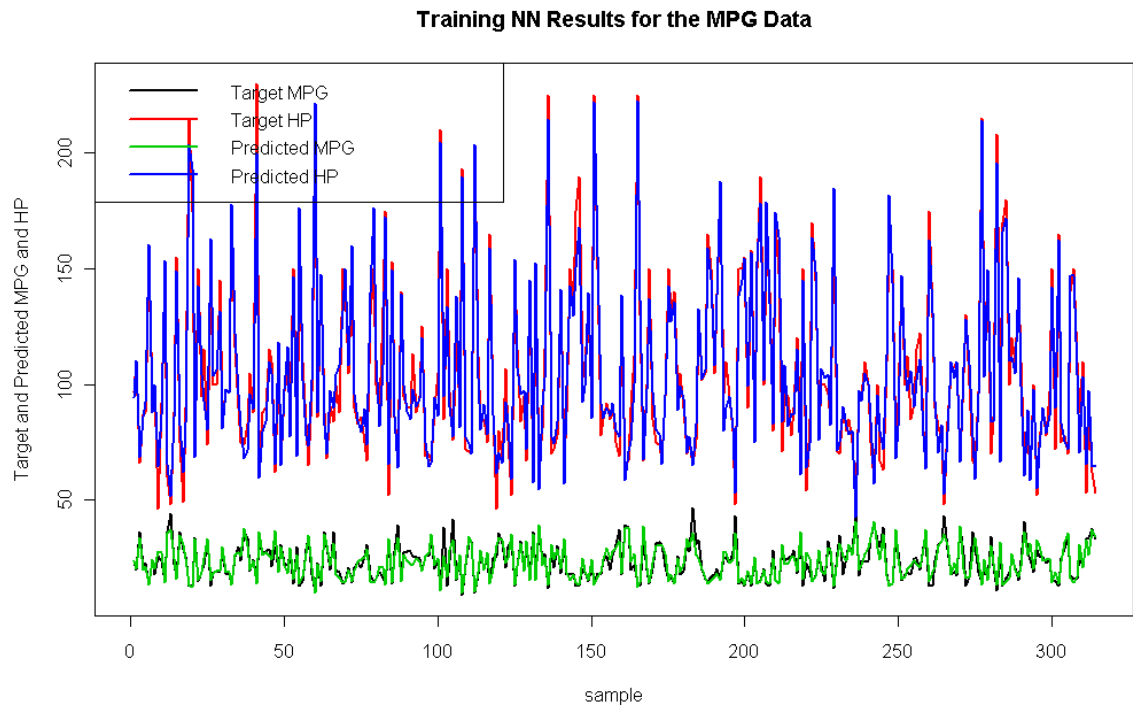


Figure 9. MPG and Horsepower predictions of the 3-hidden unit neural network against the training data.

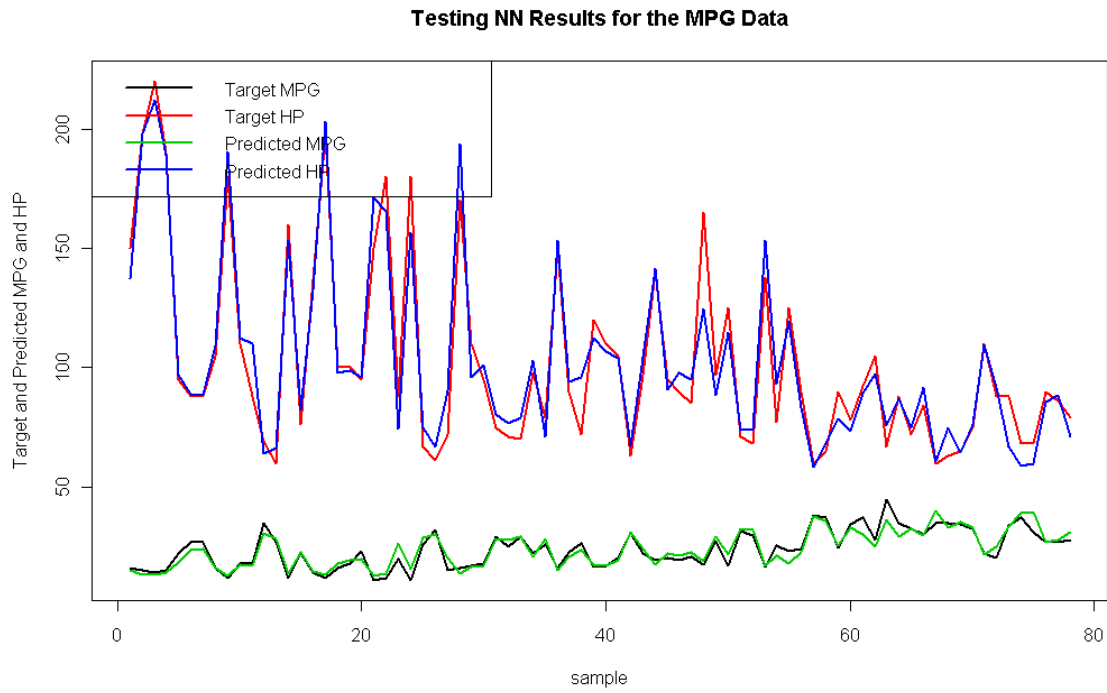


Figure 10. MPG and Horsepower predictions of the 3-hidden unit neural network against the testing data.

The standardized RMSE from the training data indicates a highly accurate model in this case. See Figure 11.

NN Model RMSE for the MPG Data

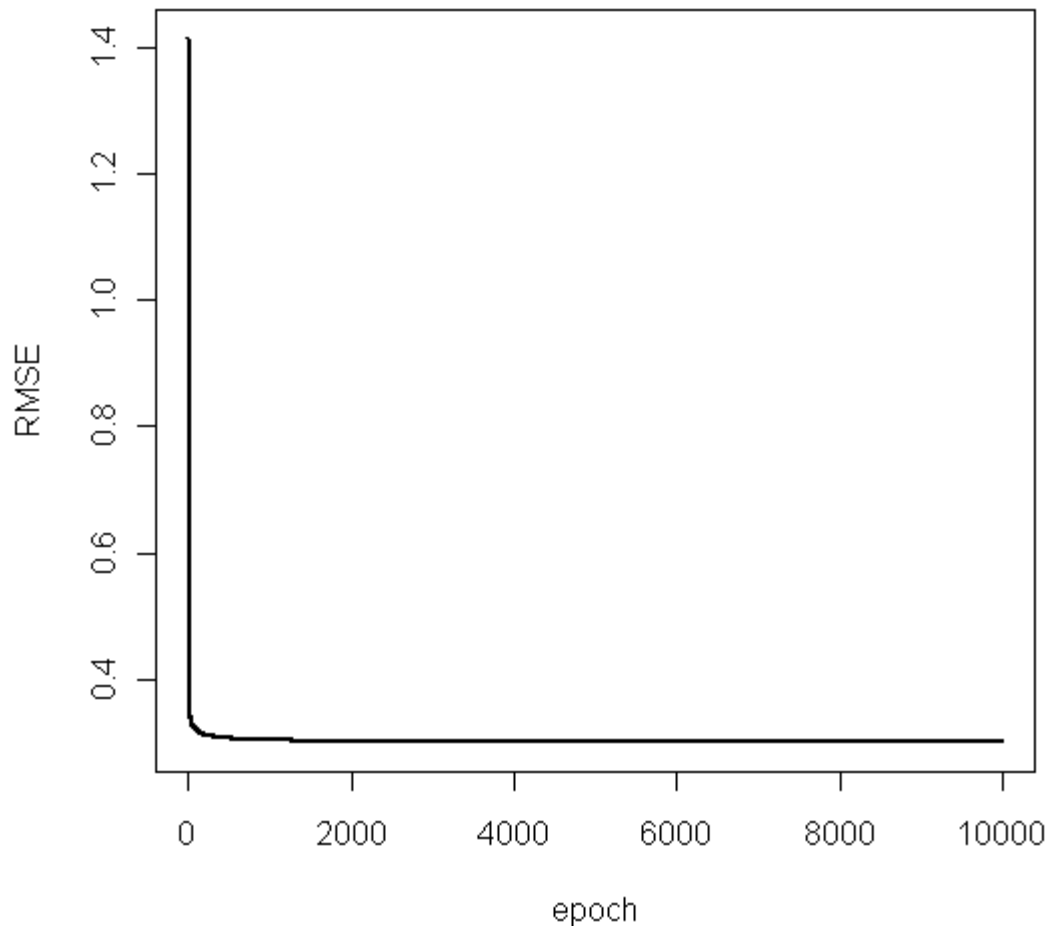


Figure 11. RMSE for predicting MPG and Horsepower from the standardized automobile training data.

By way of determining the best number of hidden units and the best value of lambda to use in our neural network, we varied the hidden units from 1 to 9 by 2 and the lambda values from 0.01 to 0.09 by 0.02 and collected RMSE results using the following code:

```
TrainRMSE <- NULL
TestRMSE <- NULL
for (hu in seq(1, 9, by=2))
{
  lTrainRMSE <- NULL
  lTestRMSE <- NULL
  for (lambda in seq(0.01, 0.09, by=0.02))
  {
    model <- makeNN(Xtrain, Ttrain, hu, lambda = lambda)
    nsineTrainNNPred <- useNN(model, Xtrain)
    nsineTestNNPred <- useNN(model, Xtest)

    lTrainRMSE <- cbind(lTrainRMSE, sqrt(mean((Ttrain -
nsineTrainNNPred$Y)^2)))

    lTestRMSE <- cbind(lTestRMSE, sqrt(mean((Ttest -
nsineTestNNPred$Y)^2)))
  }

  TrainRMSE <- rbind(TrainRMSE, lTrainRMSE)
  TestRMSE <- rbind(TestRMSE, lTestRMSE)
}
```

A graph of the result for the testing data is given in Figure 12.

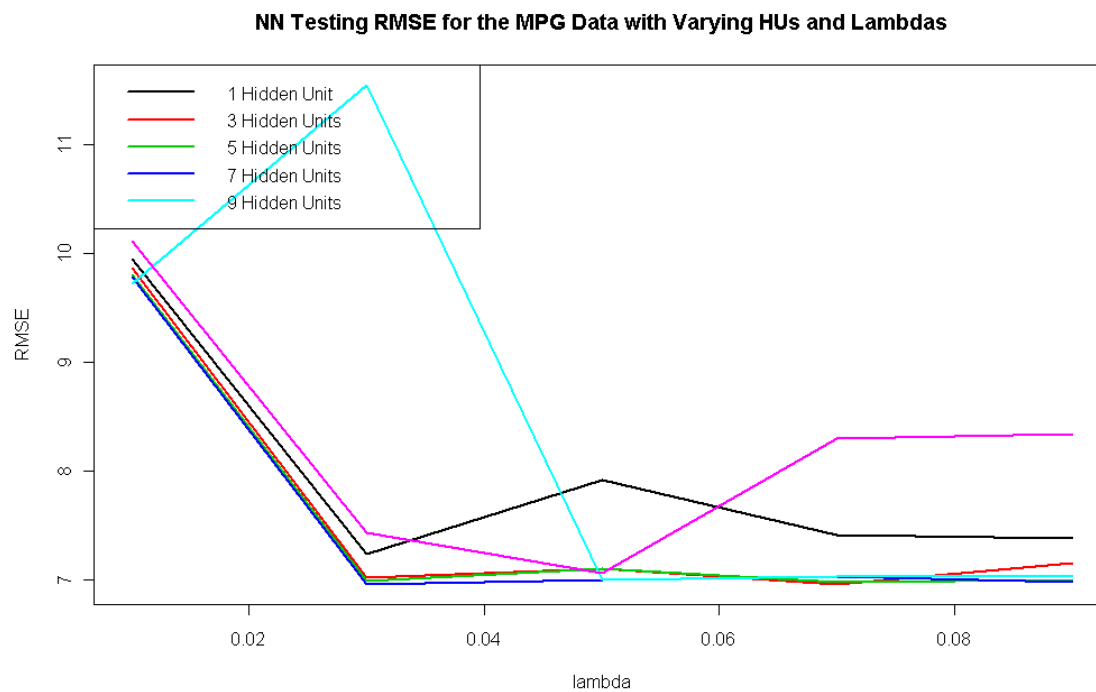


Figure 12. A comparison of RMSE for various numbers of hidden units over various choices of lambda.

For the most part, each combination of hidden unit and lambda seemed to do equally well. The one instance of the 9-hidden unit poor accuracy may be due to the SCG putting the model into a poor choice for a local minimum of the likelihood function.

5. Conclusion

The neural network model seems to provide a good mechanism for fitting the basis functions to the data. However, after this investigation there remain several outstanding questions including:

1. What techniques may be used to determine an appropriate number of hidden units?
2. What is the best way to determine the initial weights so that SCG does not put the model into “shallow” local minimums?
3. How does the choice of activation function impact the accuracy and performance of the model?

An attempt was made to answer the last question, at least the accuracy part, by using the logistic sigmoid function and the sine function in place of the hyperbolic tangent as the hidden unit activation function. We tested these different activation functions on the quadratic data from section 2.3.

To implement the neural network with the sigmoid function, activation function, h , and its gradient function were redefined as follows:

```
h <- function(a)
{
  1/(1 + exp(-a))
}

gradEwrtV <- (2*(1/N)*(1/K)) * t(Xtrains1) %*% (error %*% t(WwoC) * (Z*(1-Z))) +
(lambda * rbind(0,VwoC))

gradEwrtW <- (2*(1/N)*(1/K)) * t(cbind(1,Z)) %*% (error) # No output layer
penalty per the assignment.
```

Similarly, using the sine function was accomplished with the following modifications:

```
h <- function(a)
{
  sin(a)
}

gradEwrtV <- (2*(1/N)*(1/K)) * t(Xtrains1) %*% (error %*% t(WwoC) * cos(Z)) +
(lambda * rbind(0,VwoC))

gradEwrtW <- (2*(1/N)*(1/K)) * t(cbind(1,Z)) %*% (error) # No output layer
penalty per the assignment.
```

The sigmoid function implementation seemed to yield precisely the same results as the hyperbolic tangent as shown in the following figures:

aining NN Results for the Parabola with the Sigmoid Furesting NN Results for the Parabola with the Sigmoid Fun

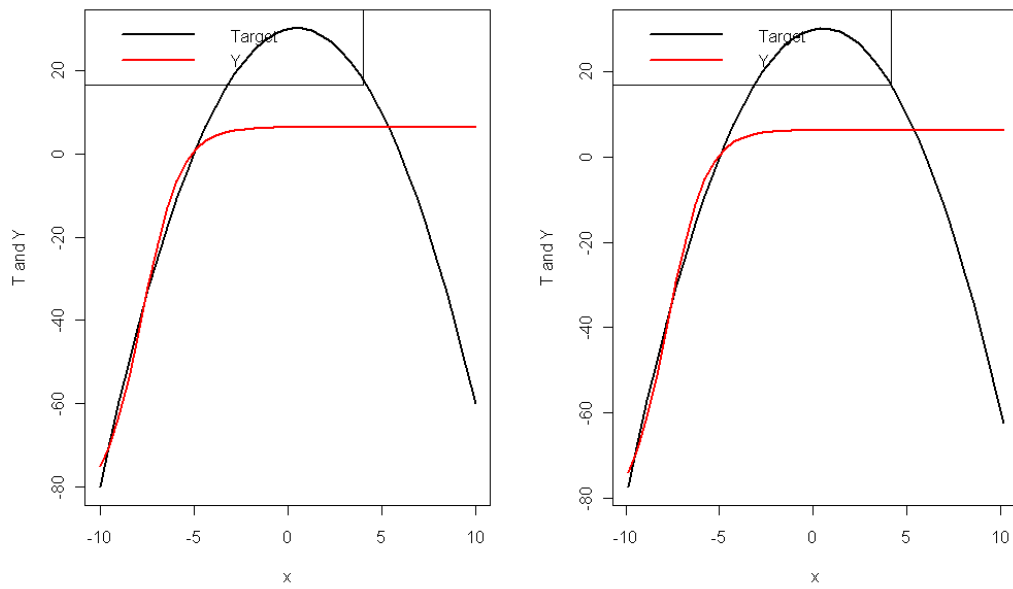


Figure 13. Using the sigmoid function as the activation function in 1 hidden unit with the quadratic data.

Training NN Results for the Parabola with the Sigmoid Function Testing NN Results for the Parabola with the Sigmoid Function

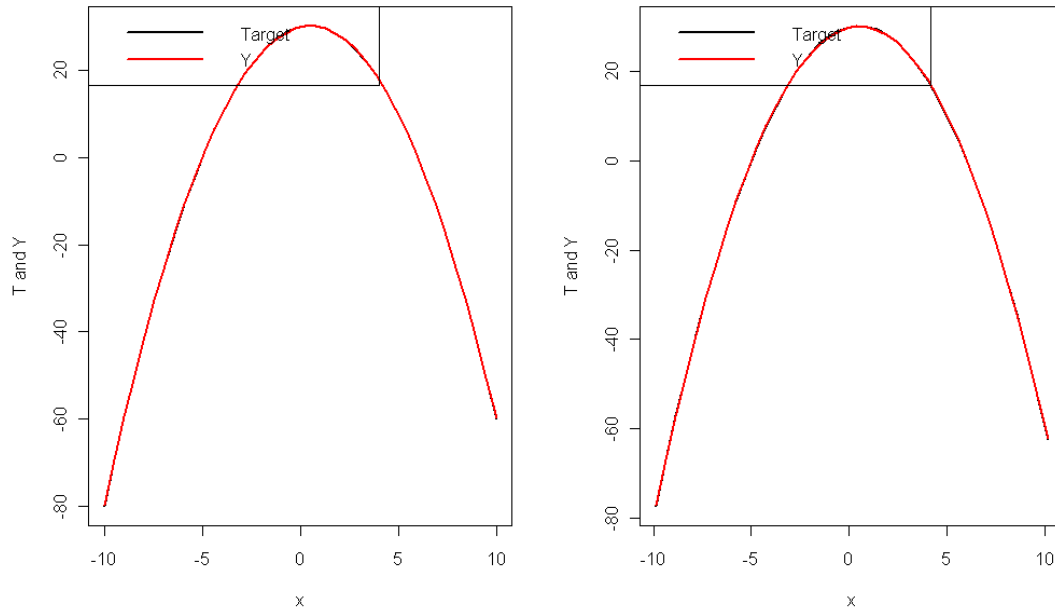


Figure 14. Using the sigmoid function as the activation function in 2 hidden units with the quadratic data.

On the other hand, the sine function implementation failed miserably as shown below.

Training NN Results for the Parabola with the Sine Func Testing NN Results for the Parabola with the Sine Func

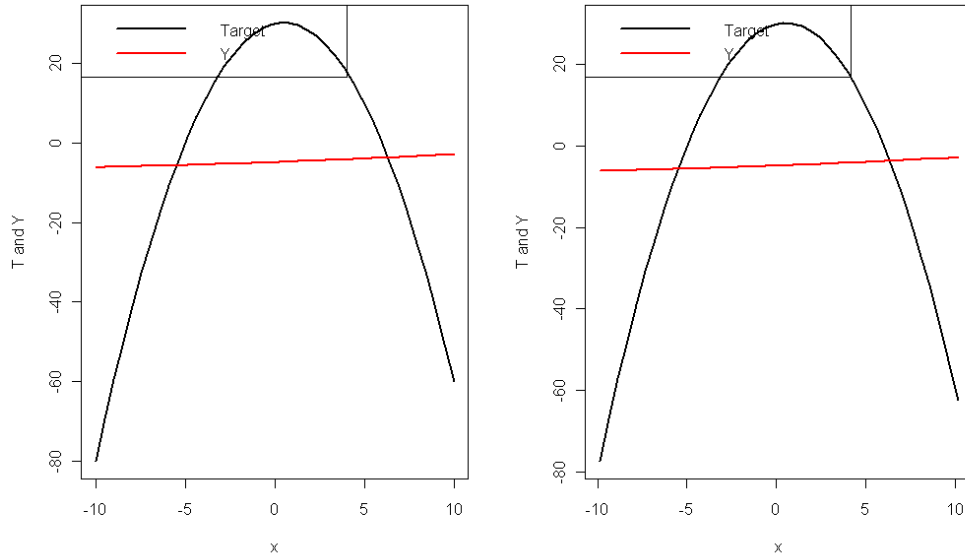


Figure 15. Using the sine function as the activation function in 1 hidden unit with the quadratic data.

Training NN Results for the Parabola with the Sine Func Testing NN Results for the Parabola with the Sine Func

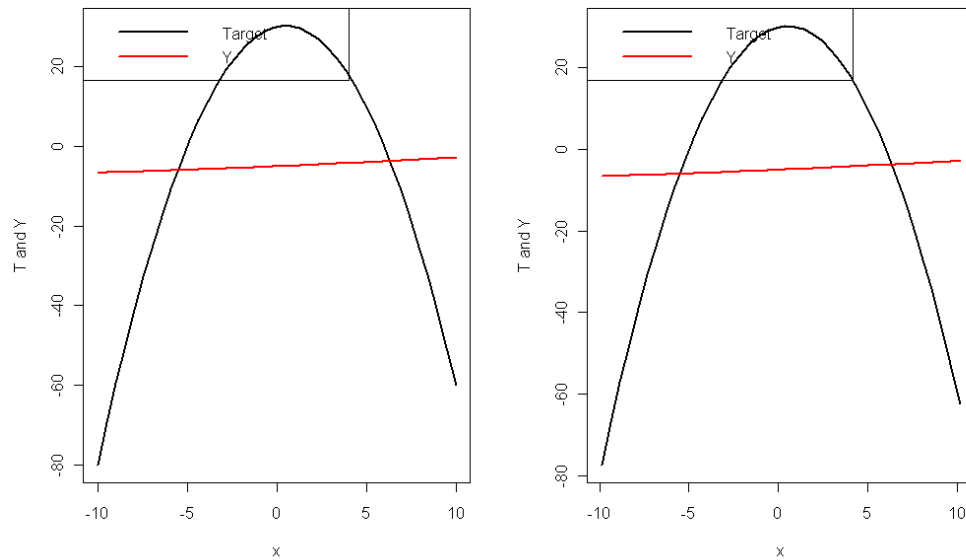


Figure 16. Using the sine function as the activation function in 2 hidden units with the quadratic data.

References

- [1] "**Anderson, Charles.**" Artificial Neural Networks. Course Notes, CS545, Department of Computer Science, Colorado State University. Fall 2009.
- [2] "**Bishop, Christopher M.**" Pattern Recognition and Machine Learning. Springer Science + Business Media, LLC, New York, NY. 2007.
- [3] "**Moller, Martin F.**" A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning. Neural Networks, vol. 6, pp. 525-533, 1993.
- [4] "**Anderson, Charles.**" drawNNet R Function. Sample R Code, CS545, Department of Computer Science, Colorado State University. Fall 2009.
- [5] "**Auto MPG Data Set**", StatLib library. Carnegie Mellon University. 1983 American Statistical Association Exposition. UCI Machine Learning Repository, July 1993.

This document was created with Win2PDF available at <http://www.win2pdf.com>.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.
This page will not be added after purchasing Win2PDF.