Neural Networks with the “nnet” Package

Harry B. Rowe

June 16, 2018

Introduction

The document provides some simple examples of building and training neural networks using the “nnet” package.

Neural Networks, Briefly

The theory of neural networks is, as they say, “beyond the scope” of this treatise. Very briefly, however...

Neural networks are learning systems that attempt to mimic, in a simplified way, the function of a biological brain. They consist of artificial neurons, interconnected in a network.

An artificial neuron has one or more inputs which are summed and an activation function which is applied to the sum to generate an output. The activation function may be any one of a number of appropriately chosen functions including sigmoids, gaussians, and others.

Neural networks typically consist of “layers” of nodes, with a layer of input nodes connected to one or more layers of artificial neuron nodes, finally connected to a layer of output nodes.

The neural networks modeled in the “nnet” package have three layers, an input layer, an output layer, and a hidden layer of artificial neurons between the other two. The input layer consists of one node for each input, such as each column or factor in an input dataset. The output layer consists of one node for each output of the model (where multinomial classes are represented as one node per possible value or class of the variable). These networks are automatically generated. The number of artificial neurons in the hidden layer are specified in the call to the nnet function as the “size” parameter.

The connections between the nodes are automatically generated from the output of one node to an input of another: one connection from the output of each input node to the input of each hidden neuron, one bias connection to each hidden neuron, one bias connection to each output node, and one connection from each hidden neuron to each output node. Thus if there are ni input nodes, nh hidden neurons, and no output nodes, there are $(ni + 1)nh + (nh + 1)no$ connections.

Each connection has an associated “weight” or multiplier. The value passed from the output of one node to the input of the next is multiplied by the weight associated with the connection. An initial weight is assigned to each connection at random.

The process of “training” a neural network consists of a recursive process of adjusting the weights associated with each connection in such a way that the cumulative error at the outputs is minimized over the input dataset. When the cumulative error reaches a stable minimum, the model is said to have converged.

Continuous Function Approximation Example

We will first build a neural network to approximate a continuous function of a single variable.

We first create a data set of 100 x,y pairs representing the following equation:

$$y = 1 - e^{-}x$$

The x’s are chosen equally spaced over the interval from 0 to 5.

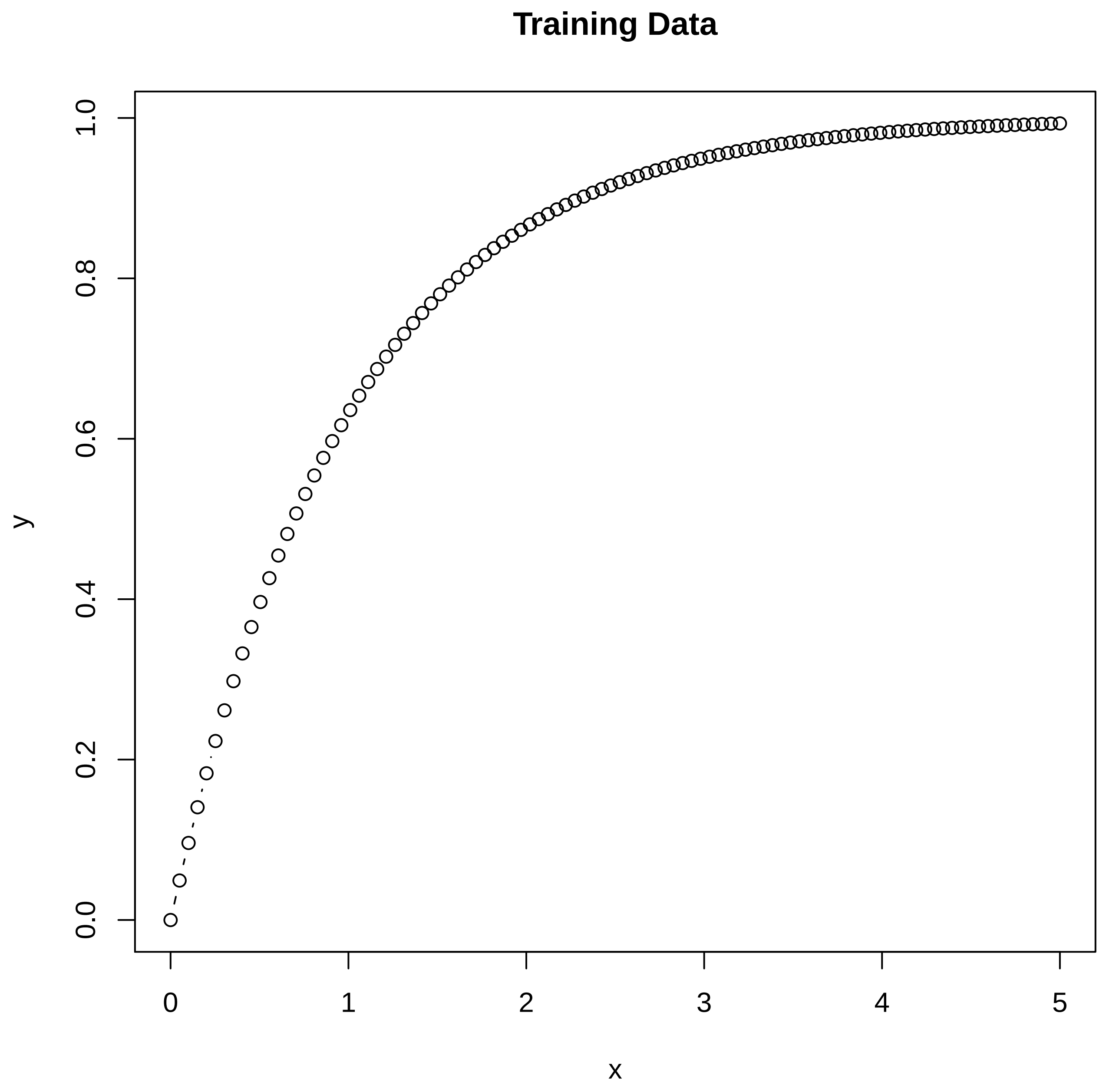
```r
# Load the xtable library and turn floating off
library( xtable )
options( xtable.floating=FALSE )

set.seed(1257)

x = seq( from=0, to=5, length.out=100 )
y = 1 - exp( -x )
```

And we plot the values.

```r
plot( x, y, type="b", main="Training Data" )
```



We now wish to create a neural network that is able to estimate y, given x. We will use the "nnet" function from the "nnet" package. This is quite simple to do.

Although not always necessary, it is recommended to scale the values of the input variables to the range (0,1). This speeds model convergence.

Since the "scale" function returns a matrix and we need a vector, we coerce the result to vector.

```
xs = as.vector( scale( x, center=0, scale=5 ) )
min( xs )
```

```
## [1] 0
```

```
max( xs )
```

```
## [1] 1
```

Then, we need to specify a formula for the model. In this case the formula is "y~xs", for "y is a function of xs".

Next, we need to choose a value for "size". As explained earlier, the nnet function builds a three layer neural network with

- an input layer with one node for each input
- a "hidden" layer with the number of artificial neurons specified by "size", and
- an output layer with one node for each output.

In this case we will arbitrarily choose three hidden neurons.

The "linout" argument specifies that the output of the model is to be continuous, rather than a binary (or logistic) value indicating a class.

Now we attempt to build the model.

```
library( nnet )
set.seed(123)
model = nnet( y~xs, size=3, linout=TRUE )
```

```
## # weights:  10
## initial  value 29.818719
## iter  10 value 1.130491
## iter  20 value 0.063700
## iter  30 value 0.017615
## iter  40 value 0.003120
## iter  50 value 0.001175
## iter  60 value 0.000910
## iter  70 value 0.000597
## iter  80 value 0.000543
## iter  90 value 0.000411
## iter 100 value 0.000363
## final  value 0.000363
## stopped after 100 iterations
```

The output traces the progress of the attempt to "train" the model. As can be seen, the process does not converge and is stopped after the default 100 iterations. This appears to be not at all unusual. Therefore, we increase the maximum number of iterations to 500 and repeat.

```
set.seed(123)
model = nnet( y~xs, size=3, linout=TRUE, maxit=500 )
```

```
## # weights:  10
## initial  value 29.818719
## iter  10 value 1.130491
## iter  20 value 0.063700
## iter  30 value 0.017615
## iter  40 value 0.003120
## iter  50 value 0.001175
## iter  60 value 0.000910
## iter  70 value 0.000597
## iter  80 value 0.000543
## iter  90 value 0.000411
## iter 100 value 0.000363
## iter 110 value 0.000266
## iter 120 value 0.000244
## iter 130 value 0.000189
## iter 140 value 0.000168
## final  value 0.000094
## converged
```

In this case, we see that convergence occurs in around 140 iterations.

We can examine the structure of the model and the final weights.

```
summary( model )
```

```
## a 1-3-1 network with 10 weights
## options were - linear output units
##  b->h1 i1->h1
##   1.62   6.57
##  b->h2 i1->h2
##  -0.58  -3.01
##  b->h3 i1->h3
##   2.27   0.90
##  b->o h1->o h2->o h3->o
## -2.76  4.74 -0.78 -1.01
```

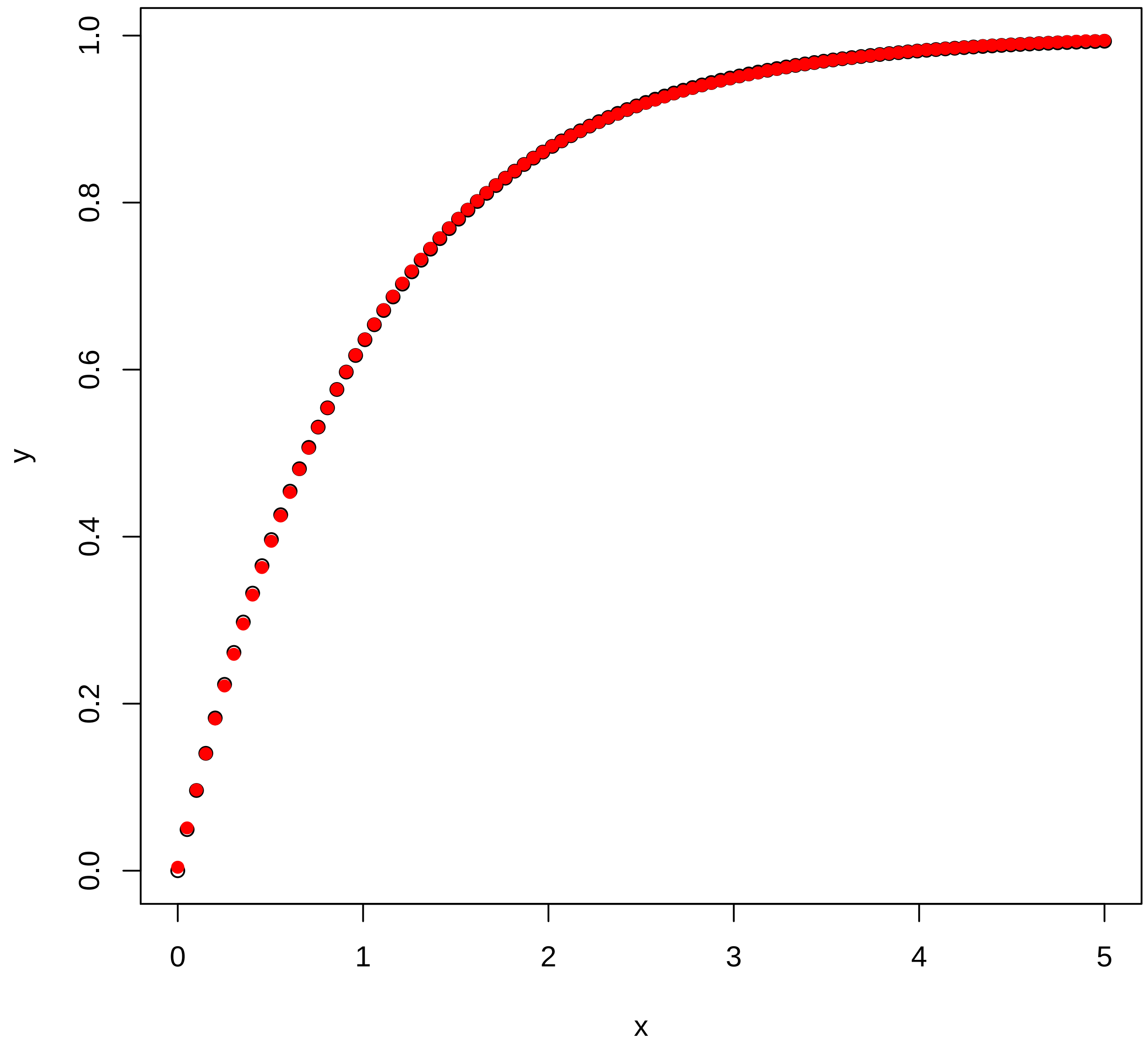
As can be seen, the neural network has one input, three hidden, and one output neurons. The weights are shown with the "from" and "to" nodes. For instance, the weight for the "b" (bias) node to the "h1" (first hidden) node is 1.62.

Now, let's see how closely the fitted model approximates the known function. We can use the "predict.nnet" function to test the model for the input values of x.

The black circles represent the training data and the red line is the response of the trained neural network to the same inputs.

Note that since we are using the "lines" function, the x values need to be sorted and the associated y estimates sorted in the same order.

```r
yhat = predict( model, type="raw" )
plot( x, y )
points( x, yhat, pch=16, col="red" )
```



We can see that the model fits the training data very well, as the solid red circles fit almost exactly within the open black circles.

We determine quantitatively how closely the predicted data matches the training data by calculating the "sum of squared error" or SSE and "mean squared error" or MSE.

```
SSE = sum( ( y - yhat )^2 )
SSE


## [1] 9.410624e-05


MSE = SSE/100
MSE


## [1] 9.410624e-07
```

This is very close agreement indeed, but should not be a surprise, since we are testing with training data. To better assess performance of the model, let's try it on a number of values of x which were not used in training the model.

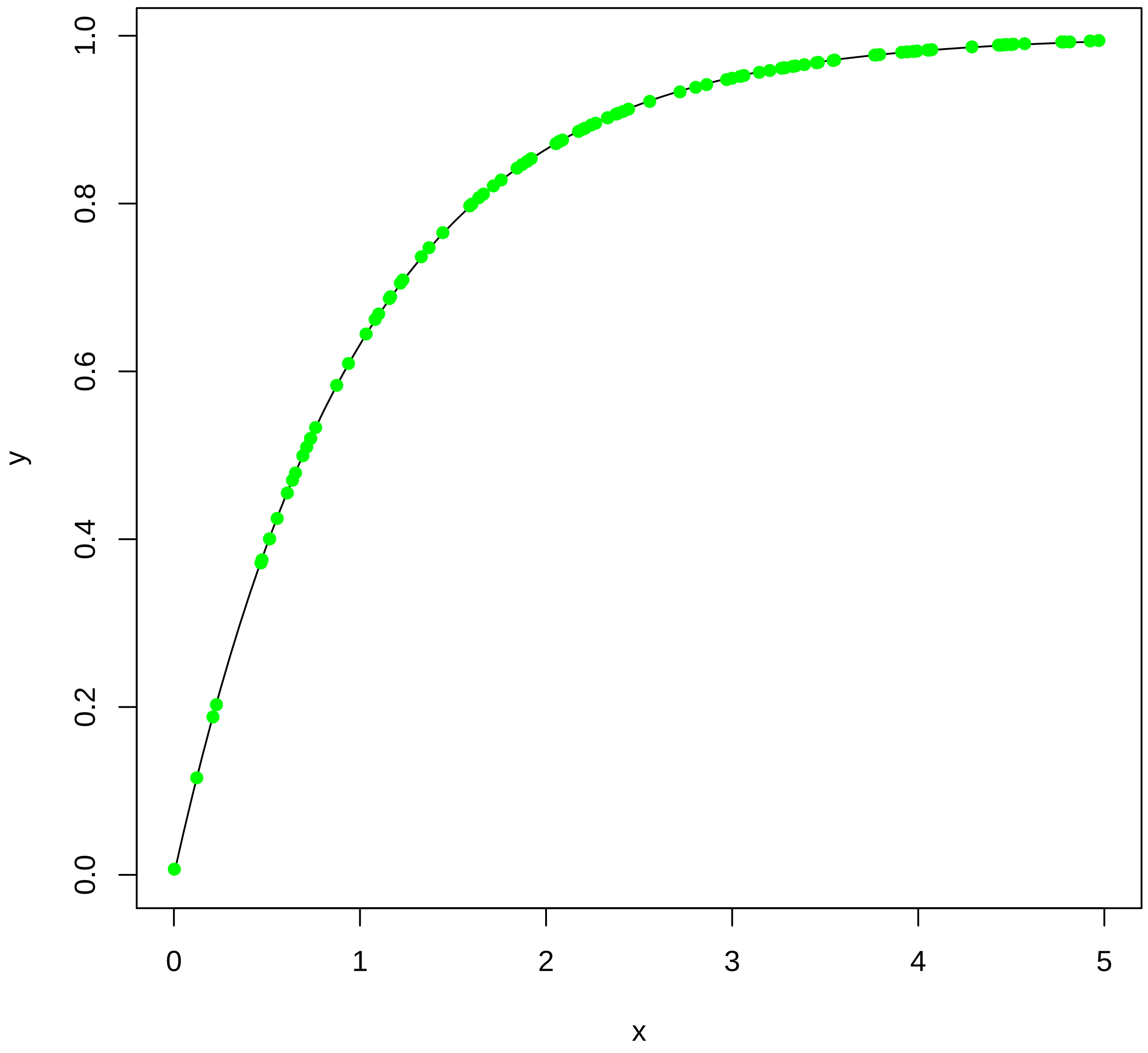
Note that when using the predict.nnet function with new data, the data must be in a data frame.

Note also that the new data must be scaled in the same way the input data was.

We select 100 new x values randomly on the interval (0,5) and use the model to predict the corresponding y values.

In this figure, the training data is shown as a black line, and the fitted values for the new data is shown as green circles.

```
newx = runif( 100, 0, 5 )
newxs = as.vector( scale( newx, center=0, scale=5 ) )
newy = predict( model, newdata=data.frame( xs=newxs ), type="raw" )
yhat = predict( model, type="raw" )
plot( x, y, type="l" )
points( newx, newy, pch=16, col="green" )
```



Again, visually, the fitted values associated with the random test data are very close to the line representing the training data.

We now calculate SSE and MSE values using the fitted values and the “true” values of the new y’s from the generating function.

```
SSE = sum( ( newy - ( 1 - exp( - newx ) ) )^2 )
SSE
```

```
## [1] 8.025341e-05
```

```
MSE = SSE/100
MSE
```

```
## [1] 8.025341e-07
```

Again, the MSE value is very small, in fact, even smaller than for the training data!

Simple Classifier Example

Next we will use “nnet” to build a neural network classifier using the iris dataset. The iris data contains 150 samples of measurements from three species of iris.

```
head( iris )
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

As can be seen, the four measurements are Sepal.Length, Sepal.Width, Petal.Length, and Petal.Width. The species is in the column Species.

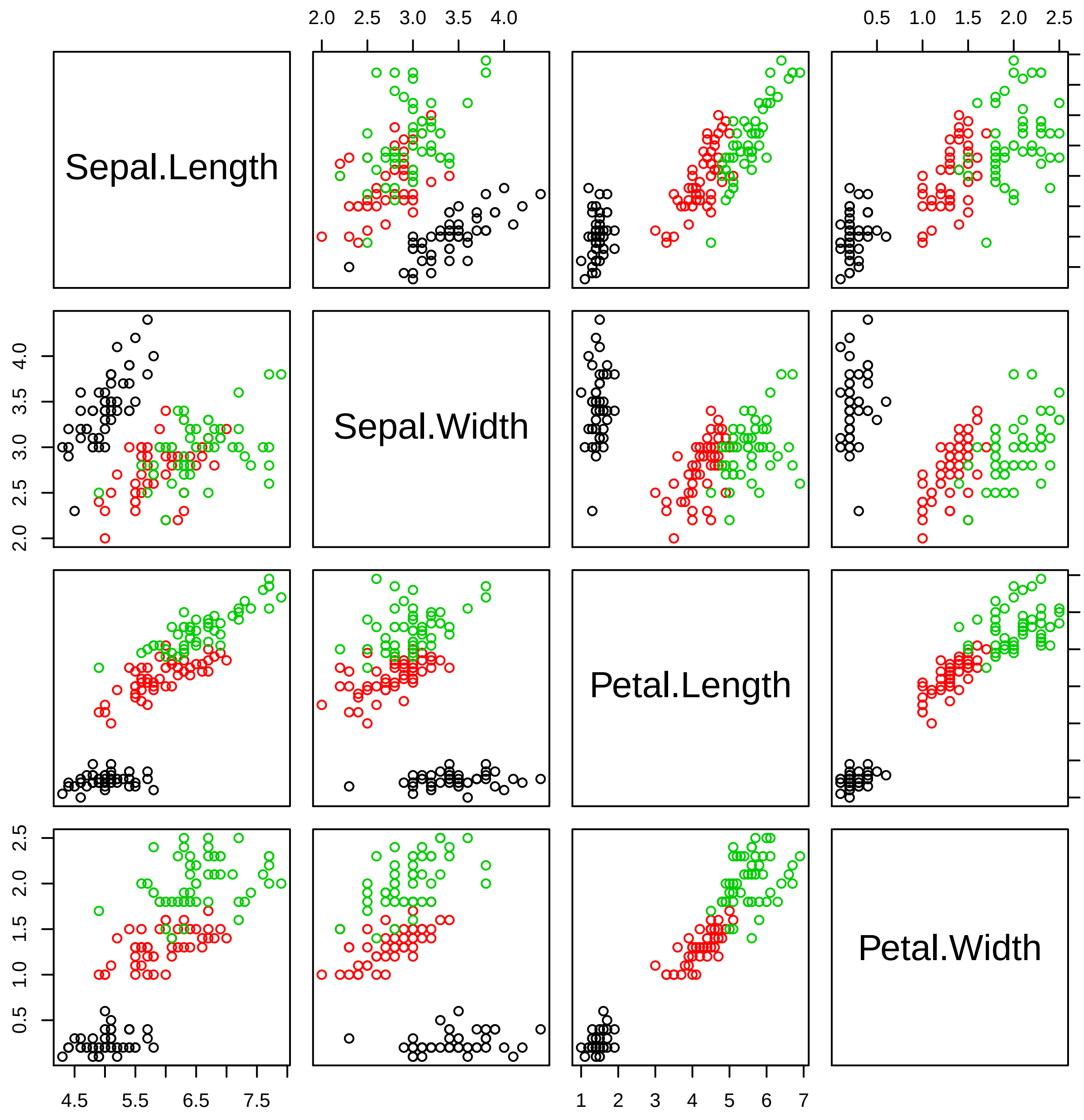
```r
print( xtable( table( iris$Species ) ) )
```

	V1
setosa	50
versicolor	50
virginica	50

There are fifty examples of each species of iris.

We will use a color coded pairs plot to examine the four variables relative to each other. Each species is shown in a different color. The black points represent setosa, the red, versicolor, and the green, virginica.

```r
pairs( iris[,1:4], col=iris$Species )
```



From the two variable plots, it appears that a simpler classifier, such as a decision tree, might perform quite well. But we want to experiment with neural networks, so...

Let us proceed with fitting a neural network to the data.

As with the continuous function example, the inputs must be scaled.

```r
# Isolate the numeric values in the first four columns
d = iris[ , 1:4 ]
# Find the maximum and minimum values in each column
maxv = apply( d, 2, max )
minv = apply( d, 2, min )
# Scale the values between 0 and 1
d = scale( d, center=minv, scale=maxv-minv )
# And add the class variable back as the fifth column
d = data.frame( d, Species=iris$Species )
```

We build the neural network with a hidden layer with eight artificial neurons. The number eight is arbitrary. The formula, "Species ." is the same notation as used in linear regression and implies "Species is a function of all the other variables". Since we intend to build a classifier, the default value of FALSE for linout need not be overridden.

```r
model = nnet( Species~., data=d, size=8, )


## # weights:  67
## initial  value 173.464647
## iter  10 value 18.737178
## iter  20 value 5.960674
## iter  30 value 5.754147
## iter  40 value 5.446756
## iter  50 value 4.695107
## iter  60 value 3.751339
## iter  70 value 2.156950
## iter  80 value 1.638691
## iter  90 value 1.440947
## iter 100 value 1.388004
## final  value 1.388004
## stopped after 100 iterations
```

As can be seen, the model did not converge, but the last several values printed were quite close together. Let's see what the implications of that are.

We print a summary of the model to see the weights.

```
summary( model )
```

```
## a 4-8-3 network with 67 weights
## options were - softmax modelling
##   b->h1  i1->h1  i2->h1  i3->h1  i4->h1
##  175.17  -16.48   91.78  -27.70  -41.46
##   b->h2  i1->h2  i2->h2  i3->h2  i4->h2
## -189.26   56.04   40.99   65.40  145.18
##   b->h3  i1->h3  i2->h3  i3->h3  i4->h3
##  200.32  227.37  -62.64 -428.29  -53.64
##   b->h4  i1->h4  i2->h4  i3->h4  i4->h4
## -115.56  117.56 -120.82  209.17  229.35
##   b->h5  i1->h5  i2->h5  i3->h5  i4->h5
## -196.61 -117.83 -129.38  269.73  137.33
##   b->h6  i1->h6  i2->h6  i3->h6  i4->h6
## -140.31  -55.12 -145.09  205.94  150.63
##   b->h7  i1->h7  i2->h7  i3->h7  i4->h7
##  -74.86   66.89  -92.01  134.87  129.07
##   b->h8  i1->h8  i2->h8  i3->h8  i4->h8
##   54.19    5.52  119.01 -105.86  -64.20
##   b->o1  h1->o1  h2->o1  h3->o1  h4->o1  h5->o1  h6->o1  h7->o1  h8->o1
##   83.52 -145.62   55.39  280.14   75.45  -92.39 -147.23 -116.03  114.62
##   b->o2  h1->o2  h2->o2  h3->o2  h4->o2  h5->o2  h6->o2  h7->o2  h8->o2
##   29.86  -90.39 -115.92  142.17  110.96 -124.28  -82.00  156.48   44.45
##   b->o3  h1->o3  h2->o3  h3->o3  h4->o3  h5->o3  h6->o3  h7->o3  h8->o3
## -113.42  235.68   60.35 -422.90 -186.61  216.66  229.61  -40.35 -159.71
```

And we print a so-called "confusion matrix" showing predicted versus actual species. The true classifications are in columns, the predicted classes in rows.

```r
pSpecies = predict( model, type="class" )
print( xtable( table( iris$Species, pSpecies ) ) )
```

	setosa	versicolor	virginica
setosa	50	0	0
versicolor	0	49	1
virginica	0	1	49

All the true setosa were predicted correctly, but one true versicolor was misclassified as a virginica and ont true virginica was misclassified as a versicolor.

So even without allowing the model training to run until it converged, the model is quite accurate **on training data**.

Notice that the network fitted is a "4-8-3 network". That implies four input, eight hidden, and **three** output neurons, while there is a single dependent variable in our formula. Why?

This is because a categorical variable that can take on more than two values is modeled as multiple outputs, one for each possible value of the categorical variable. Thus the species variable is modeled as three separate output variables.

Train/Test Evaluation

Our first model was trained using the full dataset. We would like to judge the performance of the model on data that it has not seen in training. To do so, we divide the dataset into test and training sets of roughly equal size. We use the training set to build a model, then test it with the training set.

We will allow up to 500 iterations for the model to converge.

Note that when predicting classes, we must provide the argument 'type="class"' to the predict.nnet function.

```r
# Create an index to partition the data
index = sample( 1:2, 150, replace=TRUE )
# Separate into training and test sets
train = d[ index==1, ]
```

```
test = d[ index!=1, ]
# Build and train the model
model = nnet( Species~., data=d, size=8, maxit=500 )


## # weights:  67
## initial  value 185.378231
## iter  10 value 35.467998
## iter  20 value 6.096249
## iter  30 value 5.878300
## iter  40 value 5.135710
## iter  50 value 3.158356
## iter  60 value 0.780305
## iter  70 value 0.050735
## final  value 0.000085
## converged


# Predict classes for the test set
pSpecies = predict( model, newdata=test, type="class" )
```

Then we compare the predicted classes for the test set to the true classes.

```
print( xtable( table( test$Species, pSpecies ) ) )
```

	setosa	versicolor	virginica
setosa	27	0	0
versicolor	0	20	0
virginica	0	0	22

In this case, the model converges quickly with the smaller set, and perfectly predicts the classes for the test set.

Had we called the predict function without the ‘type=”class”’ option, the output would have been an array of probabilities for each class for each row in the test dataset. This is seen below.

```
head( predict( model, newdata=test ) )
```

```
##    setosa   versicolor    virginica
## 3       1 1.593043e-61 5.668731e-87
## 4       1 1.012035e-61 5.660380e-86
## 5       1 1.774960e-61 3.275640e-87
## 6       1 1.024430e-61 6.035554e-86
## 8       1 9.921569e-62 6.259545e-86
## 13      1 1.014294e-61 5.596731e-86
```

Variance in the Model

Before we go on to try to determine the "best" number of artificial neurons for the hidden layer, we need to consider the potential for variance in the model.

When we divided the dataset into training and test sets, we used a random sample. This selection of training data results in a particular model. Had different instances been chosen for the training set, the model would have been different.

Less obviously, the model training algorithm used by the nnet function also has a random element. To begin optimization (back propagation), the function initializes the weights in the model with random values. So two successive runs with the same training set may result in different final weights (due to local optima).

Therefore, to estimate how well a particular model will perform with new data, these sources of variation must be taken into account.

Cross Validation

One technique for validating a predictive model while minimizing the effect of random selection of test and training sets is cross-validation. In cross validation the available training data is partitioned into multiple subsets, or "folds", of approximately equal size. The number of folds is frequently chosen as either five or ten.

Then the model is evaluated once for each fold in the partition. The data in the given fold is held out as test data, while the remaining data is used as training data. The performance of the model is evaluated for each fold, and the resulting metrics averaged or otherwise appropriately combined to evaluate overall performance.

Below we show five-fold cross-validation applied to evaluate the neural network model with eight neurons in the hidden layer. A composite confusion matrix is assembled along the way to provide an estimate of performance.

Note that we have supplied the option "trace=FALSE" to the nnet function to prevent generation of excessive output.

```
# get the list of possible classes
classes = unique( d$Species )

# get the number of classes
nclass = length( classes )

# Choose 8 hidden nodes
s = 8

# Create a zero composite confusion matrix of appropriate size
cmat = rep( 0, nclass*nclass )
dim( cmat ) = c( nclass, nclass )

# Partition the data into folds
fold = sample( 1:5, nrow( d ), replace=TRUE )

# Do five fold cross validation
for( f in 1:5 )
{
# Hold fold f back as test, use remainder as train
   train = d[ fold != f, ]
   test = d[ fold == f, ]
# Build and train a model
   model = nnet( Species~., data=train, size=s, maxit=500, trace=FALSE )
   pSpecies = predict( model, newdata=test, type="class" )
# Must ensure all classes accounted for in confusion matrix
   predSpecies = factor( pSpecies, levels=classes )
   trueSpecies = factor( test$Species, levels=classes )
# Confusion matrix for this fold
   cm = table( predSpecies, trueSpecies )
# Add to the cumulative confusion matrix
   cmat = cmat + cm
# Calculate classification accuracy
   acc = sum( diag( cm ) ) / sum( cm )
# Determine if the model converged
   text1 = "converged"
   if( model$convergence != 0 )
      text1 = "did not converge"
# Print fold results
   cat( paste( "Fold:", f, "Accuracy:", acc, "Model", text1, "\n" ) )

}
```

```
## Fold: 1 Accuracy: 0.931034482758621 Model converged
## Fold: 2 Accuracy: 0.939393939393939 Model converged
## Fold: 3 Accuracy: 0.962962962962963 Model converged
## Fold: 4 Accuracy: 1 Model converged
## Fold: 5 Accuracy: 0.891891891891892 Model converged
```

```
# Calculate overall classification accuracy
acc = sum( diag( cmat ) ) / sum( cmat )
cat( paste( "Overall accuracy:", acc, "\n\n" ) )
```

```
## Overall accuracy: 0.94
```

```
# Display confusion matrix
print( xtable( cmat, digits=0 ) )
```

	setosa	versicolor	virginica
setosa	50	0	1
versicolor	0	47	5
virginica	0	3	44

As can be seen, the overall classification accuracy of the model for cross validation is 94%, with nine misclassified instances.

Multiple Starts

When numerical methods depend on random starting values, it is common practice to repeat the procedure several times and compare the results. We will implement multiple starts going forward.

Selecting Number of Hidden Nodes

We next evaluate neural networks with varying numbers of hidden neurons to determine which is best to use. We incorporate cross-validation and multiple starts to deal with variance in the process.

In pseudo-code, these are the steps we will take:

```
for i from 1 to nstarts
for n from min size to max size
partition the dataset into 5 folds
for f from 1 to 5
use fold f as test data, remaining data as training data
fit a model with n hidden neurons to training data
evaluate model on test data
calculate performance metric for model size n
record performance of different sizes in a table
report on the variation in performance
```

```r
nstarts=100
minsize = 3
maxsize = 12

for( i in 1:nstarts )
{
   vsize = c()
   vacc = c()

   fold = sample( 1:5, nrow( d ), replace=TRUE )

   for( s in minsize:maxsize )
   {
# Create a zero composite confusion matrix of appropriate size
      cmat = rep( 0, nclass*nclass )
      dim( cmat ) = c( nclass, nclass )

# Partition the data into folds

# Do five fold cross validation
      for( f in 1:5 )
      {
# Hold fold f back as test, use remainder as train
         train = d[ fold != f, ]
         test = d[ fold == f, ]
# Build and train a model
         model = nnet( Species~., data=train, size=s, maxit=500, trace=FALSE )
         pSpecies = predict( model, newdata=test, type="class" )
# Must ensure all classes accounted for in confusion matrix
         predSpecies = factor( pSpecies, levels=classes )
         trueSpecies = factor( test$Species, levels=classes )
# Confusion matrix for this fold
         cm = table( predSpecies, trueSpecies )
# Add to the cumulative confusion matrix
         cmat = cmat + cm
# Calculate classification accuracy
         acc = sum( diag( cm ) ) / sum( cm )
# Determine if the model converged
```

```
         text1 = "converged"
         if( model$convergence != 0 )
            text1 = "did not converge"
# Print fold results
#          cat( paste( "Fold:", f, "Accuracy:", acc, "Model", text1, "\n" ) )

      }
# Calculate overall classification accuracy
      acc = sum( diag( cmat ) ) / sum( cmat )
#       cat( paste( "Overall accuracy:", acc, "\n\n" ) )

      vacc = c( vacc, acc )
      vsize = c( vsize, s )


   }
#    print( vsize )
#    print( vacc )

   if( i == 1 )
   {
      x = vacc
   } else
   {
      x = rbind( x, vacc )
   }
}
apply( x, 2, min )


##  [1] 0.8066667 0.8733333 0.8666667 0.8933333 0.8800000 0.9133333 0.9133333
##  [8] 0.9066667 0.8800000 0.9133333


apply( x, 2, max )


##  [1] 0.9866667 0.9866667 0.9733333 0.9733333 0.9733333 0.9800000 0.9800000
##  [8] 0.9733333 0.9733333 0.9733333
```

```
apply( x, 2, mean )
```

```
##  [1] 0.9416667 0.9474000 0.9485333 0.9492667 0.9460667 0.9488667 0.9468667
##  [8] 0.9444667 0.9444000 0.9438667
```

```
apply( x, 2, sd )
```

```
##  [1] 0.03510286 0.02135468 0.01671253 0.01356350 0.01655098 0.01275004
##  [7] 0.01447797 0.01373323 0.01393330 0.01393201
```

Based on this we see that there is not a great deal of difference in performance in the range of number of hidden nodes between four and twelve. We chose the model with six hidden nodes because it has the highest average accuracy with among the smallest variances.

Final Model

Finally we fit a model with six hidden nodes to the entire dataset. This is the model we would use to score new data.

```
model = nnet( Species~., data=d, size=6, maxit=500 )
```

```
## # weights:  51
## initial  value 209.917989
## iter  10 value 10.823016
## iter  20 value 5.836005
## iter  30 value 5.554466
## iter  40 value 5.138520
## iter  50 value 5.083642
## iter  60 value 4.986132
## iter  70 value 4.923178
## iter  80 value 4.921958
## iter  80 value 4.921958
## iter  80 value 4.921958
## final  value 4.921958
## converged
```

We print a summary of the model to see the weights.

```
summary( model )
```

```
## a 4-6-3 network with 51 weights
## options were - softmax modelling
##    b->h1   i1->h1   i2->h1   i3->h1   i4->h1
##  -297.33  -173.60    -2.64   485.89   130.72
##    b->h2   i1->h2   i2->h2   i3->h2   i4->h2
##   -80.01   -64.02    -3.01   -94.42   -93.70
##    b->h3   i1->h3   i2->h3   i3->h3   i4->h3
##   191.34  -101.15   315.05  -296.18  -323.15
##    b->h4   i1->h4   i2->h4   i3->h4   i4->h4
##   -40.95  -164.34    13.09  -238.24  -264.78
##    b->h5   i1->h5   i2->h5   i3->h5   i4->h5
##   229.57    47.76   -29.24   104.96    41.75
##    b->h6   i1->h6   i2->h6   i3->h6   i4->h6
##  -130.28   559.25 -1066.02   321.58   268.99
##    b->o1   h1->o1   h2->o1   h3->o1   h4->o1   h5->o1   h6->o1
##    61.03  -367.78   -44.81   583.17   -50.53   189.04  -269.98
##    b->o2   h1->o2   h2->o2   h3->o2   h4->o2   h5->o2   h6->o2
##   223.28   -25.85   -45.51  -116.12    45.68   419.26  -424.64
##    b->o3   h1->o3   h2->o3   h3->o3   h4->o3   h5->o3   h6->o3
##  -284.28   393.22    91.23  -466.71     4.96  -607.55   694.57
```

Deploying the Model

Having built and optimized a model, the next step is to use it to predict classes for new data. Obviously, one way to do this is in R, using the predict.nnet function to predict values for new data.

But in many cases, it will be necessary to implement the scoring on a different platform in a different language, for example C or C++. That is, as they say, “left as an exercise for the reader.” But it isn’t terribly difficult. It is a matter of implementing a series of loops to execute the calculations at each layer in sequence, performing the multiplication and summing, and applying the sigmoid function at the hidden nodes.