Linear Regression 202

Alfonso R. Reyes

2019-09-16

Contents

Prerequisites

This is a *sample* book written in **Markdown**. You can use anything that Pandoc's Markdown supports, e.g., a math equation $a^2 + b^2 = c^2$.

The **bookdown** package can be installed from CRAN or Github:

```
install.packages("bookdown")
# or the development version
# devtools::install_github("rstudio/bookdown")
```

Remember each Rmd file contains one and only one chapter, and a chapter is defined by the first-level heading `#`.

To compile this example to PDF, you need XeLaTeX. You are recommended to install TinyTeX (which includes XeLaTeX): https://yihui.name/tinytex/.

Chapter 1

Regression - cereals dataset

1.1 Introduction

Source: https://www.analyticsvidhya.com/blog/2017/09/creating-visualizing-neural-network-in-r/

Neural network is an information-processing machine and can be viewed as analogous to human nervous system. Just like human nervous system, which is made up of interconnected neurons, a neural network is made up of interconnected information processing units. The information processing units do not work in a linear manner. In fact, neural network draws its strength from parallel processing of information, which allows it to deal with non-linearity. Neural network becomes handy to infer meaning and detect patterns from complex data sets.

Neural network is considered as one of the most useful technique in the world of data analytics. However, it is complex and is often regarded as a black box, i.e. users view the input and output of a neural network but remain clueless about the knowledge generating process. We hope that the article will help readers learn about the internal mechanism of a neural network and get hands-on experience to implement it in R.

1.2 The Basics of Neural Network

A neural network is a model characterized by an activation function, which is used by interconnected information processing units to transform input into output. A neural network has always been compared to human nervous system. Information in passed through interconnected units analogous to information passage through neurons in humans. The first layer of the neural network receives the raw input, processes it and passes the processed information to the hidden layers. The hidden layer passes the information to the last layer, which produces the output. The advantage of neural network is that it is adaptive in nature. It learns from the information provided, i.e. trains itself from the data, which has a known outcome and optimizes its weights for a better prediction in situations with unknown outcome.

A perceptron, viz. single layer neural network, is the most basic form of a neural network. A perceptron receives multidimensional input and processes it using a weighted summation and an activation function. It is trained using a labeled data and learning algorithm that optimize the weights in the summation processor. A major limitation of perceptron model is its inability to deal with non-linearity. A multilayered neural network overcomes this limitation and helps solve non-linear problems. The input layer connects with hidden layer, which in turn connects to the output layer. The connections are weighted and weights are optimized using a learning rule.

There are many learning rules that are used with neural network:

a) least mean square;

b) gradient descent;
c) newton's rule;
d) conjugate gradient etc.

The learning rules can be used in conjunction with backpropgation error method. The learning rule is used to calculate the error at the output unit. This error is backpropagated to all the units such that the error at each unit is proportional to the contribution of that unit towards total error at the output unit. The errors at each unit are then used to optimize the weight at each connection. Figure 1 displays the structure of a simple neural network model for better understanding.

1.3 Fitting Neural Network in R

Now we will fit a neural network model in R. In this article, we use a subset of cereal dataset shared by Carnegie Mellon University (CMU). The details of the dataset are on the following link: http://lib.stat.cmu.edu/DASL/Datafiles/Cereals.html. The objective is to predict rating of the cereals variables such as calories, proteins, fat etc. The R script is provided side by side and is commented for better understanding of the user. . The data is in .csv format and can be downloaded by clicking: cereals.

Please set working directory in R using setwd( ) function, and keep cereal.csv in the working directory. We use rating as the dependent variable and calories, proteins, fat, sodium and fiber as the independent variables. We divide the data into training and test set. Training set is used to find the relationship between dependent and independent variables while the test set assesses the performance of the model. We use 60% of the dataset as training set. The assignment of the data to training and test set is done using random sampling. We perform random sampling on R using sample ( ) function. We have used set.seed( ) to generate same random sample everytime and maintain consistency. We will use the index variable while fitting neural network to create training and test data sets. The R script is as follows:

```
## Creating index variable

# Read the Data
data = read.csv(file.path(data_raw_dir, "cereals.csv"), header=T)

# Random sampling
samplesize = 0.60 * nrow(data)
set.seed(80)
index = sample( seq_len ( nrow ( data ) ), size = samplesize )

# Create training and test set
datatrain = data[ index, ]
datatest = data[ -index, ]
```

```
dplyr::glimpse(data)
#> Observations: 75
#> Variables: 6
#> $ calories <int> 70, 120, 70, 50, 110, 110, 130, 90, 90, 120, 110, 120...
#> $ protein  <int> 4, 3, 4, 4, 2, 2, 3, 2, 3, 1, 6, 1, 3, 1, 2, 2, 1, 1,...
#> $ fat      <int> 1, 5, 1, 0, 2, 0, 2, 1, 0, 2, 2, 3, 2, 1, 0, 0, 0, 1,...
#> $ sodium   <int> 130, 15, 260, 140, 180, 125, 210, 200, 210, 220, 290,...
#> $ fiber    <dbl> 10.0, 2.0, 9.0, 14.0, 1.5, 1.0, 2.0, 4.0, 5.0, 0.0, 2...
#> $ rating   <dbl> 68.4, 34.0, 59.4, 93.7, 29.5, 33.2, 37.0, 49.1, 53.3,...
```

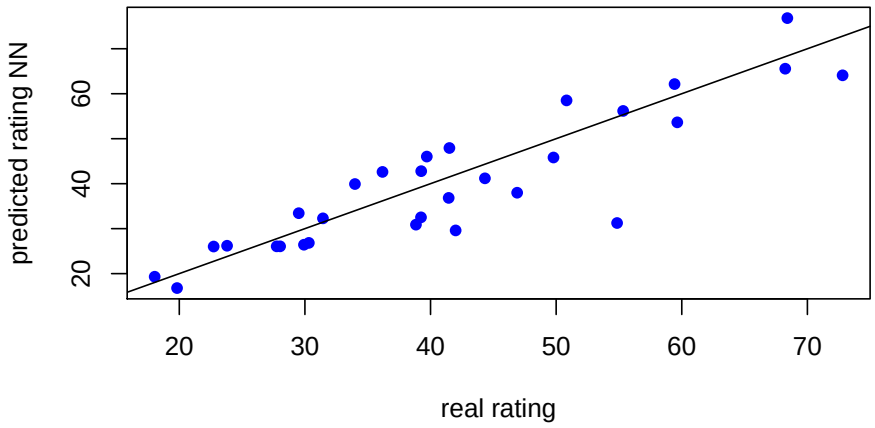
Now we fit a neural network on our data. We use neuralnet library for the analysis. The first step is to scale the cereal dataset. The scaling of data is essential because otherwise a variable may have large impact on the prediction variable only because of its scale. Using unscaled may lead to meaningless results. The

common techniques to scale data are: min-max normalization, Z-score normalization, median and MAD, and tan-h estimators. The min-max normalization transforms the data into a common range, thus removing the scaling effect from all the variables. Unlike Z-score normalization and median and MAD method, the min-max method retains the original distribution of the variables. We use min-max normalization to scale the data. The R script for scaling the data is as follows.

```
## Scale data for neural network

max = apply(data , 2 , max)
min = apply(data, 2 , min)
scaled = as.data.frame(scale(data, center = min, scale = max - min))
```

```
## Fit neural network

# install library
# install.packages("neuralnet ")

# load library
library(neuralnet)

# creating training and test set
trainNN = scaled[index , ]
testNN = scaled[-index , ]

# fit neural network
set.seed(2)
NN = neuralnet(rating ~ calories + protein + fat + sodium + fiber,
               trainNN, hidden = 3 , linear.output = T )

# plot neural network
plot(NN)
```

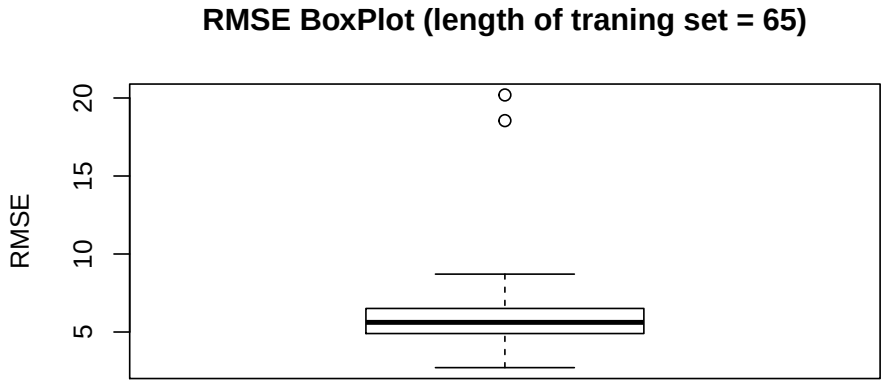
```
## Prediction using neural network

predict_testNN = compute(NN, testNN[,c(1:5)])
predict_testNN = (predict_testNN$net.result * (max(data$rating) - min(data$rating))) + min(data$rating)

plot(datatest$rating, predict_testNN, col='blue', pch=16, ylab = "predicted rating NN", xlab = "real ra

abline(0,1)

# Calculate Root Mean Square Error (RMSE)
RMSE.NN = (sum((datatest$rating - predict_testNN)^2) / nrow(datatest)) ^ 0.5
```




```
## Cross validation of neural network model

# install relevant libraries
# install.packages("boot")
# install.packages("plyr")

# Load libraries
library(boot)
library(plyr)

# Initialize variables
set.seed(50)
k = 100
RMSE.NN = NULL

List = list( )

# Fit neural network model within nested for loop
for(j in 10:65){
    for (i in 1:k) {
        index = sample(1:nrow(data),j )

        trainNN = scaled[index,]
        testNN = scaled[-index,]
        datatest = data[-index,]

        NN = neuralnet(rating ~ calories + protein + fat + sodium + fiber, trainNN, hidden = 3, linear.
        predict_testNN = compute(NN,testNN[,c(1:5)])
        predict_testNN = (predict_testNN$net.result*(max(data$rating)-min(data$rating)))+min(data$ratin

        RMSE.NN [i]<- (sum((datatest$rating - predict_testNN)^2)/nrow(datatest))^0.5
    }
    List[[j]] = RMSE.NN
}

Matrix.RMSE = do.call(cbind, List)
```

```
## Prepare boxplot
boxplot(Matrix.RMSE[,56], ylab = "RMSE", main = "RMSE BoxPlot (length of traning set = 65)")
```

Figure 1.1: Variation of RMSE




```
## Variation of median RMSE
# install.packages("matrixStats")
library(matrixStats)
#>
#> Attaching package: 'matrixStats'
#> The following object is masked from 'package:plyr':
#>
#>     count

med = colMedians(Matrix.RMSE)

X = seq(10,65)

plot (med~X, type = "l", xlab = "length of training set", ylab = "median RMSE", main = "Variation of RMS
```

Figure 1.1) shows that the median RMSE of our model decreases as the length of the training the set. This is an important result. The reader must remember that the model accuracy is dependent on the length of training set. The performance of neural network model is sensitive to training-test split.

1.4 End Notes

The article discusses the theoretical aspects of a neural network, its implementation in R and post training evaluation. Neural network is inspired from biological nervous system. Similar to nervous system the information is passed through layers of processors. The significance of variables is represented by weights of each connection. The article provides basic understanding of back propagation algorithm, which is used to assign these weights. In this article we also implement neural network on R. We use a publically available dataset shared by CMU. The aim is to predict the rating of cereals using information such as calories, fat, protein etc. After constructing the neural network we evaluate the model for accuracy and robustness. We compute RMSE and perform cross-validation analysis. In cross validation, we check the variation in model accuracy as the length of training set is changed. We consider training sets with length 10 to 65. For each length a 100 samples are random picked and median RMSE is calculated. We show that model accuracy increases when training set is large. Before using the model for prediction, it is important to check the robustness of performance through cross validation.

The article provides a quick review neural network and is a useful reference for data enthusiasts. We have provided commented R code throughout the article to help readers with hands on experience of using neural networks.

Chapter 2

Fitting a neural network

2.1 Introduction

https://www.r-bloggers.com/fitting-a-neural-network-in-r-neuralnet-package/

https://datascienceplus.com/fitting-neural-network-in-r/

Neural networks have always been one of the fascinating machine learning models in my opinion, not only because of the fancy backpropagation algorithm but also because of their complexity (think of deep learning with many hidden layers) and structure inspired by the brain.

Neural networks have not always been popular, partly because they were, and still are in some cases, computationally expensive and partly because they did not seem to yield better results when compared with simpler methods such as support vector machines (SVMs). Nevertheless, Neural Networks have, once again, raised attention and become popular.

Update: We published another post about Network analysis at DataScience+ Network analysis of Game of Thrones

In this post, we are going to fit a simple neural network using the neuralnet package and fit a linear model as a comparison.

2.2 The dataset

We are going to use the Boston dataset in the MASS package. The Boston dataset is a collection of data about housing values in the suburbs of Boston. Our goal is to predict the median value of owner-occupied homes (medv) using all the other continuous variables available.

```
set.seed(500)
library(MASS)
data <- Boston
```

```
dplyr::glimpse(data)
#> Observations: 506
#> Variables: 14
#> $ crim    <dbl> 0.00632, 0.02731, 0.02729, 0.03237, 0.06905, 0.02985, ...
#> $ zn      <dbl> 18.0, 0.0, 0.0, 0.0, 0.0, 0.0, 12.5, 12.5, 12.5, 12.5,...
#> $ indus   <dbl> 2.31, 7.07, 7.07, 2.18, 2.18, 2.18, 7.87, 7.87, 7.87, ...
#> $ chas    <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
#> $ nox     <dbl> 0.538, 0.469, 0.469, 0.458, 0.458, 0.458, 0.524, 0.524...
```

```
#> $ rm      <dbl> 6.58, 6.42, 7.18, 7.00, 7.15, 6.43, 6.01, 6.17, 5.63, ...
#> $ age     <dbl> 65.2, 78.9, 61.1, 45.8, 54.2, 58.7, 66.6, 96.1, 100.0,...
#> $ dis     <dbl> 4.09, 4.97, 4.97, 6.06, 6.06, 6.06, 5.56, 5.95, 6.08, ...
#> $ rad     <int> 1, 2, 2, 3, 3, 3, 5, 5, 5, 5, 5, 5, 5, 4, 4, 4, 4, 4, ...
#> $ tax     <dbl> 296, 242, 242, 222, 222, 222, 311, 311, 311, 311, 311,...
#> $ ptratio <dbl> 15.3, 17.8, 17.8, 18.7, 18.7, 18.7, 15.2, 15.2, 15.2, ...
#> $ black   <dbl> 397, 397, 393, 395, 397, 394, 396, 397, 387, 387, 393,...
#> $ lstat   <dbl> 4.98, 9.14, 4.03, 2.94, 5.33, 5.21, 12.43, 19.15, 29.9...
#> $ medv    <dbl> 24.0, 21.6, 34.7, 33.4, 36.2, 28.7, 22.9, 27.1, 16.5, ...
```

First we need to check that no datapoint is missing, otherwise we need to fix the dataset.

```
apply(data,2,function(x) sum(is.na(x)))
#>    crim      zn   indus    chas     nox      rm     age     dis     rad
#>       0       0       0       0       0       0       0       0       0
#>     tax ptratio   black   lstat    medv
#>       0       0       0       0       0
```

There is no missing data, good. We proceed by randomly splitting the data into a train and a test set, then we fit a linear regression model and test it on the test set. Note that I am using the gml() function instead of the lm() this will become useful later when cross validating the linear model.

```
index <- sample(1:nrow(data),round(0.75*nrow(data)))
train <- data[index,]
test <- data[-index,]
lm.fit <- glm(medv~., data=train)
summary(lm.fit)
#>
#> Call:
#> glm(formula = medv ~ ., data = train)
#>
#> Deviance Residuals:
#>     Min       1Q   Median       3Q      Max
#> -15.211   -2.559   -0.655    1.828   29.711
#>
#> Coefficients:
#>              Estimate Std. Error t value Pr(>|t|)
#> (Intercept)  31.11170    5.45981    5.70  2.5e-08 ***
#> crim         -0.11137    0.03326   -3.35  0.00090 ***
#> zn            0.04263    0.01431    2.98  0.00308 **
#> indus         0.00148    0.06745    0.02  0.98247
#> chas          1.75684    0.98109    1.79  0.07417 .
#> nox         -18.18485    4.47157   -4.07  5.8e-05 ***
#> rm            4.76034    0.48047    9.91  < 2e-16 ***
#> age          -0.01344    0.01410   -0.95  0.34119
#> dis          -1.55375    0.21893   -7.10  6.7e-12 ***
#> rad           0.28818    0.07202    4.00  7.6e-05 ***
#> tax          -0.01374    0.00406   -3.38  0.00079 ***
#> ptratio      -0.94755    0.14012   -6.76  5.4e-11 ***
#> black         0.00950    0.00290    3.28  0.00115 **
#> lstat        -0.38890    0.05973   -6.51  2.5e-10 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> (Dispersion parameter for gaussian family taken to be 20.2)
```

```
#>
#>     Null deviance: 32463.5  on 379  degrees of freedom
#> Residual deviance:  7407.1  on 366  degrees of freedom
#> AIC: 2237
#>
#> Number of Fisher Scoring iterations: 2
pr.lm <- predict(lm.fit,test)
MSE.lm <- sum((pr.lm - test$medv)^2)/nrow(test)
```

The sample(x,size) function simply outputs a vector of the specified size of randomly selected samples from the vector x. By default the sampling is without replacement: index is essentially a random vector of indeces. Since we are dealing with a regression problem, we are going to use the mean squared error (MSE) as a measure of how much our predictions are far away from the real data.

2.3 Preparing to fit the neural network

Before fitting a neural network, some preparation need to be done. Neural networks are not that easy to train and tune.

As a first step, we are going to address data preprocessing. It is good practice to normalize your data before training a neural network. I cannot emphasize enough how important this step is: depending on your dataset, avoiding normalization may lead to useless results or to a very difficult training process (most of the times the algorithm will not converge before the number of maximum iterations allowed). You can choose different methods to scale the data (z-normalization, min-max scale, etc…). I chose to use the min-max method and scale the data in the interval [0,1]. Usually scaling in the intervals [0,1] or [-1,1] tends to give better results. We therefore scale and split the data before moving on:

```
maxs <- apply(data, 2, max)
mins <- apply(data, 2, min)

scaled <- as.data.frame(scale(data, center = mins, scale = maxs - mins))

train_ <- scaled[index,]
test_ <- scaled[-index,]
```

Note that scale returns a matrix that needs to be coerced into a data.frame.

2.4 Parameters

As far as I know there is no fixed rule as to how many layers and neurons to use although there are several more or less accepted rules of thumb. Usually, if at all necessary, one hidden layer is enough for a vast numbers of applications. As far as the number of neurons is concerned, it should be between the input layer size and the output layer size, usually 2/3 of the input size. At least in my brief experience testing again and again is the best solution since there is no guarantee that any of these rules will fit your model best. Since this is a toy example, we are going to use 2 hidden layers with this configuration: 13:5:3:1. The input layer has 13 inputs, the two hidden layers have 5 and 3 neurons and the output layer has, of course, a single output since we are doing regression. Let's fit the net:

```
library(neuralnet)
n <- names(train_)
f <- as.formula(paste("medv ~", paste(n[!n %in% "medv"], collapse = " + ")))
nn <- neuralnet(f,data=train_,hidden=c(5,3),linear.output=T)
```

A couple of notes:

- For some reason the formula y~. is not accepted in the neuralnet() function. You need to first write the formula and then pass it as an argument in the fitting function.
- The hidden argument accepts a vector with the number of neurons for each hidden layer, while the argument linear.output is used to specify whether we want to do regression linear.output=TRUE or classification linear.output=FALSE

The `neuralnet` package provides a nice tool to plot the model:

This is the graphical representation of the model with the weights on each connection:

```
plot(nn)
```

The black lines show the connections between each layer and the weights on each connection while the blue lines show the bias term added in each step. The bias can be thought as the intercept of a linear model. The net is essentially a black box so we cannot say that much about the fitting, the weights and the model. Suffice to say that the training algorithm has converged and therefore the model is ready to be used.

2.5 Predicting medv using the neural network

Now we can try to predict the values for the test set and calculate the MSE. Remember that the net will output a normalized prediction, so we need to scale it back in order to make a meaningful comparison (or just a simple prediction).

```
pr.nn <- compute(nn,test_[,1:13])

pr.nn_ <- pr.nn$net.result*(max(data$medv)-min(data$medv))+min(data$medv)
test.r <- (test_$medv)*(max(data$medv)-min(data$medv))+min(data$medv)

MSE.nn <- sum((test.r - pr.nn_)^2)/nrow(test_)
```

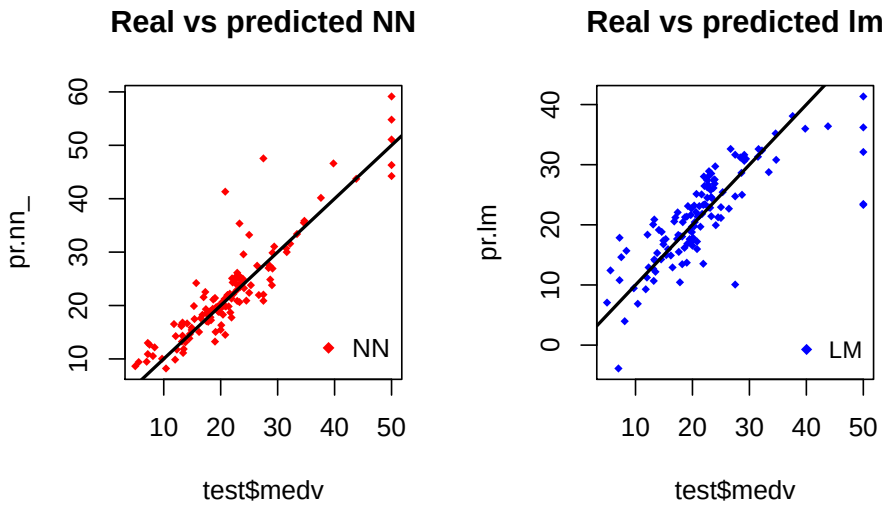
we then compare the two MSEs

```
print(paste(MSE.lm,MSE.nn))
#> [1] "31.2630222372615 16.4595537665717"
```

Apparently, the net is doing a better work than the linear model at predicting medv. Once again, be careful because this result depends on the train-test split performed above. Below, after the visual plot, we are going to perform a fast cross validation in order to be more confident about the results.

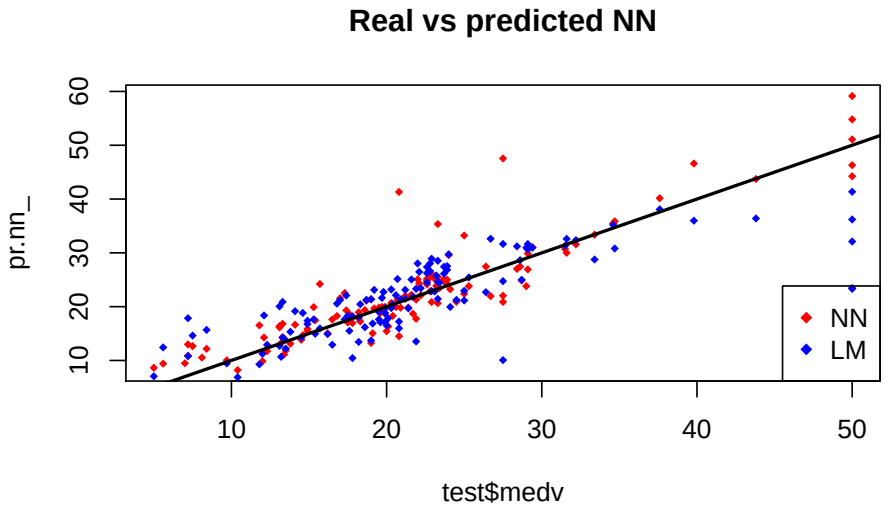
A first visual approach to the performance of the network and the linear model on the test set is plotted below

```
par(mfrow=c(1,2))

plot(test$medv,pr.nn_,col='red',main='Real vs predicted NN',pch=18,cex=0.7)
abline(0,1,lwd=2)
legend('bottomright',legend='NN',pch=18,col='red', bty='n')

plot(test$medv,pr.lm,col='blue',main='Real vs predicted lm',pch=18, cex=0.7)
abline(0,1,lwd=2)
legend('bottomright',legend='LM',pch=18,col='blue', bty='n', cex=.95)
```



By visually inspecting the plot we can see that the predictions made by the neural network are (in general) more concetrated around the line (a perfect alignment with the line would indicate a MSE of 0 and thus an ideal perfect prediction) than those made by the linear model.

```
plot(test$medv,pr.nn_,col='red',main='Real vs predicted NN',pch=18,cex=0.7)
points(test$medv,pr.lm,col='blue',pch=18,cex=0.7)
abline(0,1,lwd=2)
legend('bottomright',legend=c('NN','LM'),pch=18,col=c('red','blue'))
```



2.6 A (fast) cross validation

Cross validation is another very important step of building predictive models. While there are different kind of cross validation methods, the basic idea is repeating the following process a number of time:

train-test split

- Do the train-test split
- Fit the model to the train set
- Test the model on the test set
- Calculate the prediction error
- Repeat the process K times

Then by calculating the average error we can get a grasp of how the model is doing.

We are going to implement a fast cross validation using a for loop for the neural network and the cv.glm() function in the boot package for the linear model. As far as I know, there is no built-in function in R to perform cross-validation on this kind of neural network, if you do know such a function, please let me know in the comments. Here is the 10 fold cross-validated MSE for the linear model:

```
library(boot)
set.seed(200)
lm.fit <- glm(medv~.,data=data)
cv.glm(data,lm.fit,K=10)$delta[1]
#> [1] 23.2
```

Now the net. Note that I am splitting the data in this way: 90% train set and 10% test set in a random way for 10 times. I am also initializing a progress bar using the plyr library because I want to keep an eye on the status of the process since the fitting of the neural network may take a while.

```
set.seed(450)
cv.error <- NULL
k <- 10

library(plyr)
pbar <- create_progress_bar('text')
pbar$init(k)
#>
  |
  |                                                                      |   0%

for(i in 1:k){
    index <- sample(1:nrow(data),round(0.9*nrow(data)))
    train.cv <- scaled[index,]
    test.cv <- scaled[-index,]

    nn <- neuralnet(f,data=train.cv,hidden=c(5,2),linear.output=T)

    pr.nn <- compute(nn,test.cv[,1:13])
    pr.nn <- pr.nn$net.result*(max(data$medv)-min(data$medv))+min(data$medv)

    test.cv.r <- (test.cv$medv)*(max(data$medv)-min(data$medv))+min(data$medv)

    cv.error[i] <- sum((test.cv.r - pr.nn)^2)/nrow(test.cv)

    pbar$step()
}
#>
  |
  |======                                                                |  10%
  |
  |=============                                                         |  20%
  |
  |====================                                                  |  30%
  |
  |==========================                                            |  40%
  |
  |================================                                      |  50%
  |
```

```
  |======================================                |  60%
  |
  |=========================================            |  70%
  |
  |=============================================        |  80%
  |
  |=================================================    |  90%
  |
  |=====================================================| 100%
```

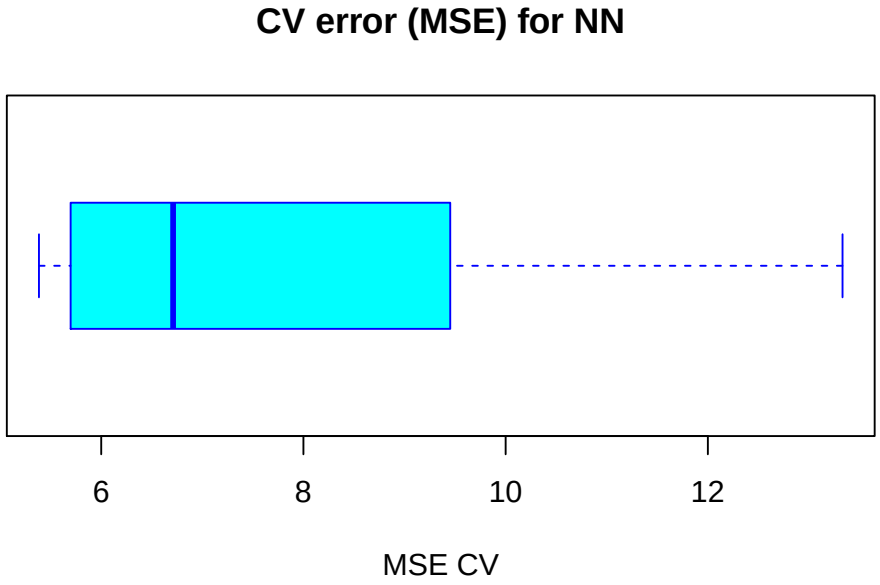
After a while, the process is done, we calculate the average MSE and plot the results as a boxplot

```
mean(cv.error)
#> [1] 7.64
```

```
cv.error
#>  [1] 13.33  7.10  6.58  5.70  6.84  5.77 10.75  5.38  9.45  5.50
```

The code for the box plot: The code above outputs the following boxplot:

```
boxplot(cv.error,xlab='MSE CV',col='cyan',
        border='blue',names='CV error (MSE)',
        main='CV error (MSE) for NN',horizontal=TRUE)
```



As you can see, the average MSE for the neural network (10.33) is lower than the one of the linear model although there seems to be a certain degree of variation in the MSEs of the cross validation. This may depend on the splitting of the data or the random initialization of the weights in the net. By running the simulation different times with different seeds you can get a more precise point estimate for the average MSE.

2.7 A final note on model interpretability

Neural networks resemble black boxes a lot: explaining their outcome is much more difficult than explaining the outcome of simpler model such as a linear model. Therefore, depending on the kind of application you need, you might want to take into account this factor too. Furthermore, as you have seen above, extra care is needed to fit a neural network and small changes can lead to different results.

A gist with the full code for this post can be found here.

Thank you for reading this post, leave a comment below if you have any question.

Chapter 3

Visualization of neural networks

https://beckmw.wordpress.com/tag/neuralnet/

In my last post I said I wasn't going to write anymore about neural networks (i.e., multilayer feedforward perceptron, supervised ANN, etc.). That was a lie. I've received several requests to update the neural network plotting function described in the original post. As previously explained, R does not provide a lot of options for visualizing neural networks. The only option I know of is a plotting method for objects from the neuralnet package. This may be my opinion, but I think this plot leaves much to be desired (see below). Also, no plotting methods exist for neural networks created in other packages, i.e., nnet and RSNNS. These packages are the only ones listed on the CRAN task view, so I've updated my original plotting function to work with all three. Additionally, I've added a new option for plotting a raw weight vector to allow use with neural networks created elsewhere. This blog describes these changes, as well as some new arguments added to the original function.

As usual, I'll simulate some data to use for creating the neural networks. The dataset contains eight input variables and two output variables. The final dataset is a data frame with all variables, as well as separate data frames for the input and output variables. I've retained separate datasets based on the syntax for each package.

```
library(clusterGeneration)
#> Loading required package: MASS
library(tictoc)

seed.val<- 12345
set.seed(seed.val)

num.vars<-8
num.obs<-1000

# input variables
cov.mat <-genPositiveDefMat(num.vars,covMethod=c("unifcorrmat"))$Sigma
rand.vars <-mvrnorm(num.obs,rep(0,num.vars),Sigma=cov.mat)

# output variables
parms <-runif(num.vars,-10,10)
y1 <- rand.vars %*% matrix(parms) + rnorm(num.obs,sd=20)
parms2 <- runif(num.vars,-10,10)
y2 <- rand.vars %*% matrix(parms2) + rnorm(num.obs,sd=20)

# final datasets
```

```
rand.vars <- data.frame(rand.vars)
resp <- data.frame(y1,y2)
names(resp) <- c('Y1','Y2')
dat.in <- data.frame(resp, rand.vars)
```

```
dplyr::glimpse(dat.in)
#> Observations: 1,000
#> Variables: 10
#> $ Y1 <dbl> 25.442, -14.578, -36.214, 15.216, -6.393, -20.849, -28.665,...
#> $ Y2 <dbl> 16.9, 38.8, 31.2, -31.2, 93.3, 11.7, 59.7, -103.5, -49.8, 5...
#> $ X1 <dbl> 3.138, -0.705, -4.373, 0.837, 0.787, 1.923, -1.419, 1.121, ...
#> $ X2 <dbl> 0.195, -0.302, 0.773, 1.311, 3.506, 1.245, 3.800, -0.165, 0...
#> $ X3 <dbl> -1.795, -2.596, 2.308, 4.081, -3.921, 1.473, -0.926, 7.101,...
#> $ X4 <dbl> -2.7216, 3.0589, 1.2455, 3.4607, 2.3775, -2.9833, 2.6669, -...
#> $ X5 <dbl> 0.0407, 0.7602, -3.0217, -4.2799, 2.0859, 1.4765, 0.0561, 2...
#> $ X6 <dbl> -1.4820, -0.5014, 0.0603, -1.8551, 2.2817, 1.7386, 1.7450, ...
#> $ X7 <dbl> -0.7169, -0.3618, -1.5283, 4.2026, -6.1548, -0.3545, -6.028...
#> $ X8 <dbl> 1.152, 1.810, -1.357, 0.598, -1.425, -1.210, -1.004, 2.494,...
```

The various neural network packages are used to create separate models for plotting.

```
# first model with nnet
#nnet function from nnet package
library(nnet)
set.seed(seed.val)
tic()
mod1 <- nnet(rand.vars, resp, data = dat.in, size = 10, linout = T)
#> # weights:  112
#> initial  value 4784162.893260
#> iter  10 value 1794537.980652
#> iter  20 value 1577753.498759
#> iter  30 value 1485254.945755
#> iter  40 value 1449238.248788
#> iter  50 value 1427720.291804
#> iter  60 value 1416977.236373
#> iter  70 value 1405167.753521
#> iter  80 value 1395046.792257
#> iter  90 value 1370522.267277
#> iter 100 value 1363709.540981
#> final  value 1363709.540981
#> stopped after 100 iterations
toc()
#> 0.195 sec elapsed
```

```
# nn <- neuralnet(form.in,
#                 data = dat.sc,
#                 # hidden = c(13, 10, 3),
#                 hidden = c(5),
#                 act.fct = "tanh",
#                 linear.output = FALSE,
#                 lifesign = "minimal")
```

```
# 2nd model with neuralnet
# neuralnet function from neuralnet package, notice use of only one response
library(neuralnet)
```

```
softplus <- function(x) log(1 + exp(x))
sigmoid  <- function(x) log(1 + exp(-x))

dat.sc <- scale(dat.in)
form.in <- as.formula('Y1 ~ X1+X2+X3+X4+X5+X6+X7+X8')
set.seed(seed.val)
tic()
mod2 <- neuralnet(form.in, data = dat.sc, hidden = 10, lifesign = "minimal",
                  linear.output = FALSE,
                  act.fct = "tanh")
#> hidden: 10    thresh: 0.01    rep: 1/1    steps:   26361 error: 160.06372    time: 58.25 secs
toc()
#> 58.265 sec elapsed
```

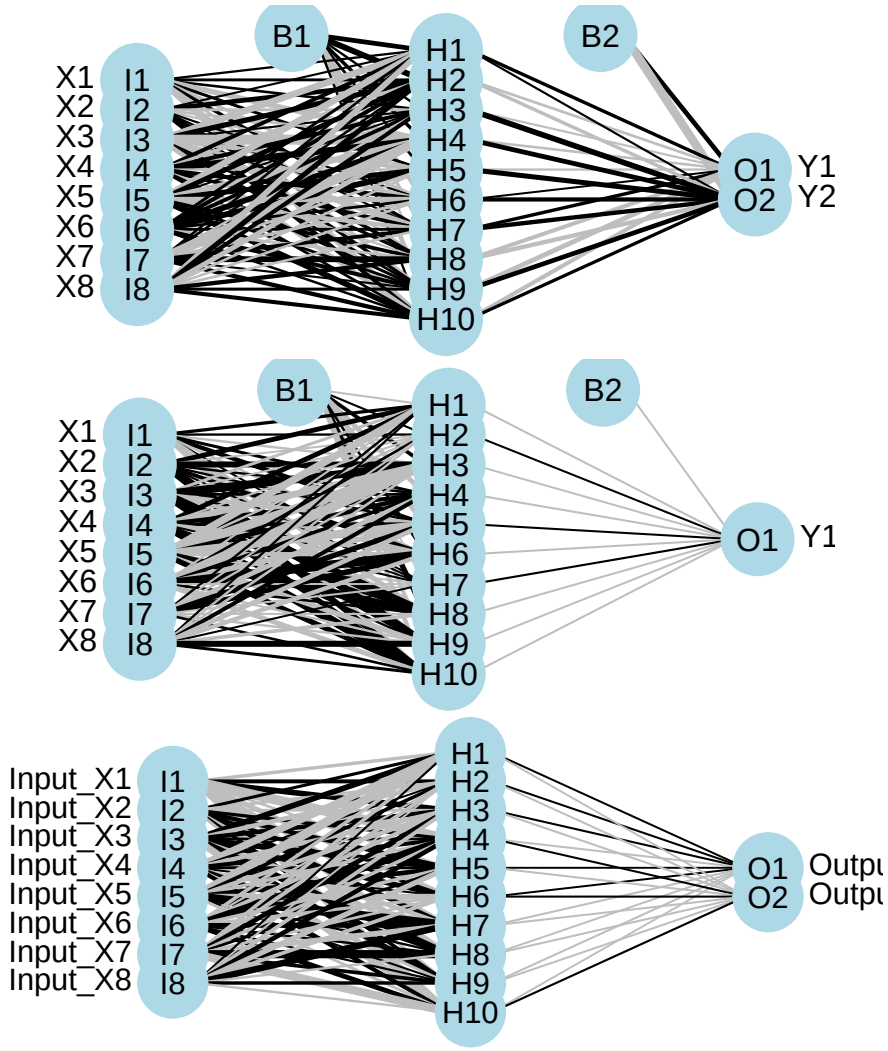
```
# third model with RSNNS
# mlp function from RSNNS package
library(RSNNS)
#> Loading required package: Rcpp
set.seed(seed.val)
tic()
mod3 <- mlp(rand.vars, resp, size = 10, linOut = T)
toc()
#> 0.406 sec elapsed
```

I've noticed some differences between the functions that could lead to some confusion. For simplicity, the above code represents my interpretation of the most direct way to create a neural network in each package. Be very aware that direct comparison of results is not advised given that the default arguments differ between the packages. A few key differences are as follows, although many others should be noted. First, the functions differ in the methods for passing the primary input variables.

The `nnet` function can take separate (or combined) x and y inputs as data frames or as a formula, the `neuralnet` function can only use a formula as input, and the mlp function can only take a data frame as combined or separate variables as input. As far as I know, the neuralnet function is not capable of modelling multiple response variables, unless the response is a categorical variable that uses one node for each outcome. Additionally, the default output for the neuralnet function is linear, whereas the opposite is true for the other two functions.

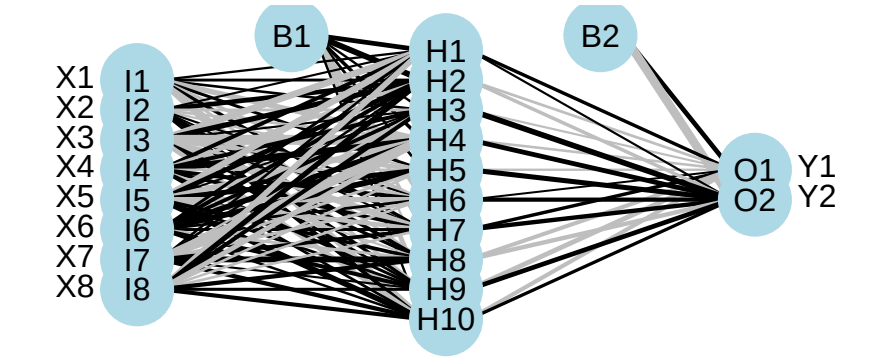
Specifics aside, here's how to use the updated plot function. Note that the same syntax is used to plot each model

```
# import the function from Github
library(devtools)
source_url('https://gist.githubusercontent.com/fawda123/7471137/raw/466c1474d0a505ff044412703516c34f1a4
#> SHA-1 hash of file is 74c80bd5ddbc17ab3ae5ece9c0ed9beb612e87ef

# plot each model
plot.nnet(mod1)
#> Loading required package: scales
#> Loading required package: reshape
plot.nnet(mod2)
plot.nnet(mod3)
#> Warning in plot.nnet(mod3): Bias layer not applicable for rsnns object
```



The plotting function can also now be used with an arbitrary weight vector, rather than a specific model object. The struct argument must also be included if this option is used. I thought the easiest way to use the plotting function with your own weights was to have the input weights as a numeric vector, including bias layers. I've shown how this can be done using the weights directly from mod1 for simplicity.

```
wts.in <- mod1$wts
struct <- mod1$n
plot.nnet(wts.in,struct=struct)
```



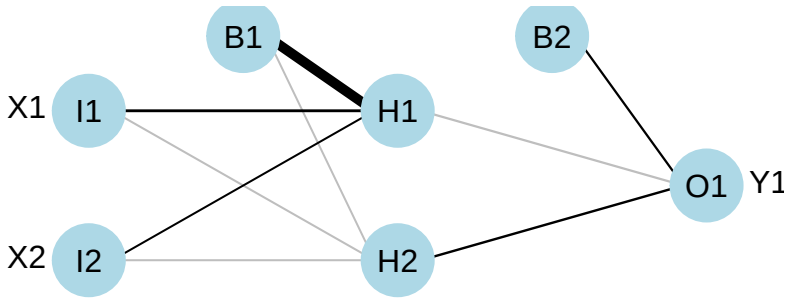
Note that `wts.in` is a numeric vector with length equal to the expected given the architecture (i.e., for 8 10

2 network, 100 connection weights plus 12 bias weights). The plot should look the same as the plot for the neural network from nnet.

The weights in the input vector need to be in a specific order for correct plotting. I realize this is not clear by looking directly at wt.in but this was the simplest approach I could think of. The weight vector shows the weights for each hidden node in sequence, starting with the bias input for each node, then the weights for each output node in sequence, starting with the bias input for each output node. Note that the bias layer has to be included even if the network was not created with biases. If this is the case, simply input a random number where the bias values should go and use the argument bias=F. I'll show the correct order of the weights using an example with plot.nn from the neuralnet package since the weights are included directly on the plot.

If we pretend that the above figure wasn't created in R, we would input the mod.in argument for the updated plotting function as follows. Also note that struct must be included if using this approach.

```
mod.in<-c(13.12,1.49,0.16,-0.11,-0.19,-0.16,0.56,-0.52,0.81)
struct<-c(2,2,1) #two inputs, two hidden, one output
plot.nnet(mod.in, struct=struct)
```



Note the comparability with the figure created using the neuralnet package. That is, larger weights have thicker lines and color indicates sign (+ black, – grey).

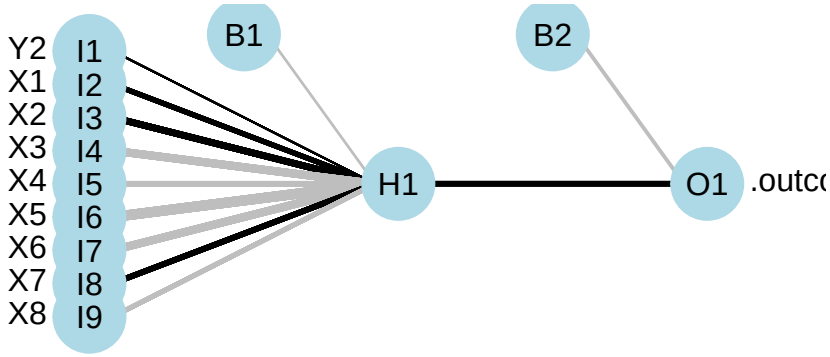
One of these days I'll actually put these functions in a package. In the meantime, please let me know if any bugs are encountered.

3.1 caret and plot NN

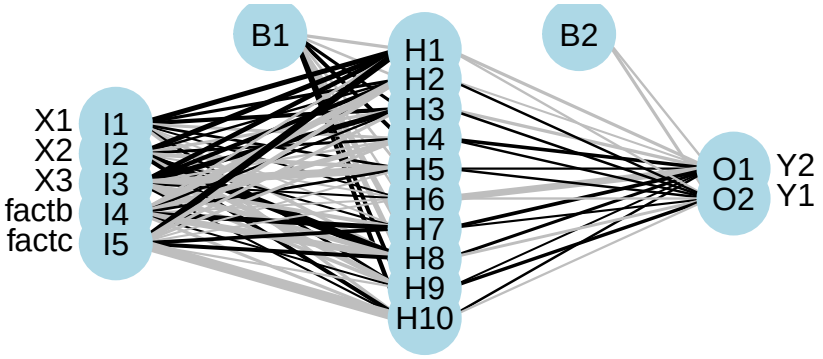
I've changed the function to work with neural networks created using the train function from the caret package. The link above is updated but you can also grab it here.

```
library(caret)
#> Loading required package: lattice
#> Loading required package: ggplot2
#> Registered S3 methods overwritten by 'ggplot2':
#>   method         from
#>   [.quosures     rlang
#>   c.quosures     rlang
#>   print.quosures rlang
#>
#> Attaching package: 'caret'
#> The following objects are masked from 'package:RSNNS':
#>
#>     confusionMatrix, train
mod4 <- train(Y1 ~., method='nnet', data=dat.in, linout=T)
```

```
plot.nnet(mod4,nid=T)
#> Warning in plot.nnet(mod4, nid = T): Using best nnet model from train
#> output
```




```
fact<-factor(sample(c('a','b','c'),size=num.obs,replace=T))
form.in<-formula('cbind(Y2,Y1)~X1+X2+X3+fact')
mod5<-nnet(form.in,data=cbind(dat.in,fact),size=10,linout=T)
#> # weights:  82
#> initial  value 4799569.423556
#> iter  10 value 2864553.218126
#> iter  20 value 2595828.194160
#> iter  30 value 2517965.483941
#> iter  40 value 2464882.178217
#> iter  50 value 2444238.700834
#> iter  60 value 2424302.290643
#> iter  70 value 2395226.949866
#> iter  80 value 2375558.751266
#> iter  90 value 2343011.050867
#> iter 100 value 2298860.593948
#> final  value 2298860.593948
#> stopped after 100 iterations
plot.nnet(mod5,nid=T)
```

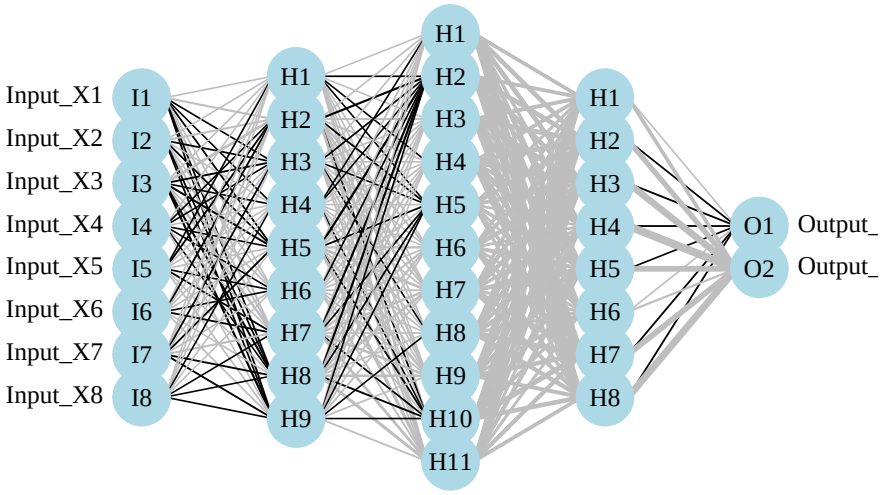


3.2 Multiple hidden layers

More updates… I've now modified the function to plot multiple hidden layers for networks created using the mlp function in the `RSNNS` package and neuralnet in the neuralnet package. As far as I know, these are the only neural network functions in R that can create multiple hidden layers. All others use a single hidden layer. I have not tested the plotting function using manual input for the weight vectors with multiple hidden layers.

My guess is it won't work but I can't be bothered to change the function unless it's specifically requested. The updated function can be grabbed here (all above links to the function have also been changed).

```
library(RSNNS)

# neural net with three hidden layers, 9, 11, and 8 nodes in each
tic()
mod <-mlp(rand.vars, resp,
          size = c(9,11,8),
          linOut = T)
toc()
#> 0.425 sec elapsed
par(mar=numeric(4),family='serif')
plot.nnet(mod)
#> Warning in plot.nnet(mod): Bias layer not applicable for rsnns object
```

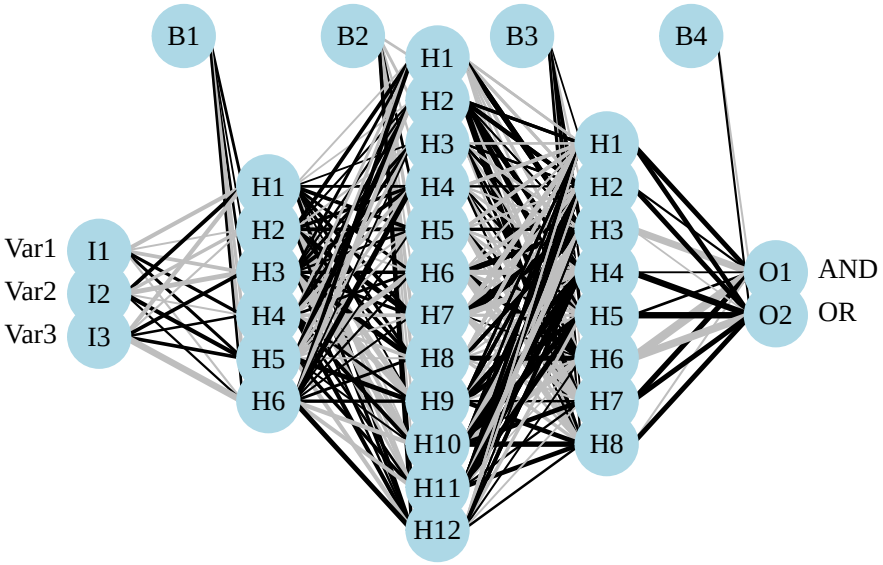


3.3 Binary predictors

Here's an example using the `neuralnet` function with binary predictors and categorical outputs (credit to Tao Ma for the model code).

```
library(neuralnet)

#response
AND<-c(rep(0,7),1)
OR<-c(0,rep(1,7))

# response with predictors
binary.data <- data.frame(expand.grid(c(0,1), c(0,1), c(0,1)), AND, OR)

#model
tic()
net <- neuralnet(AND+OR ~ Var1+Var2+Var3,
                 binary.data, hidden =c(6,12,8),
                 rep = 10,
                 err.fct="ce",
                 linear.output=FALSE)
```

```
toc()
#> 0.164 sec elapsed
#plot ouput
par(mar=numeric(4),family='serif')
plot.nnet(net)
```



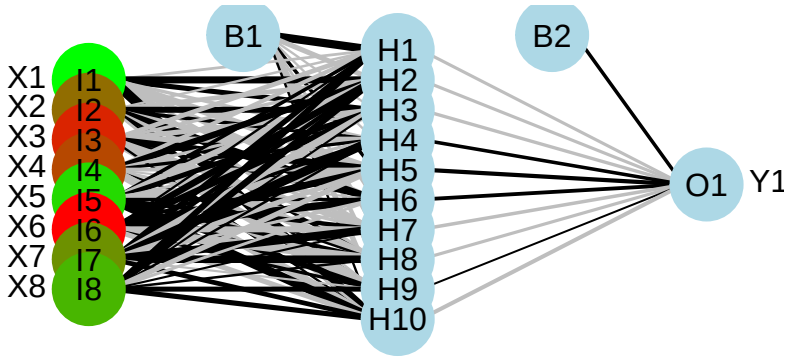
3.4 color coding the input layer

The color vector argument (circle.col) for the nodes was changed to allow a separate color vector for the input layer.

The following example shows how this can be done using relative importance of the input variables to color-code the first layer.

```
# example showing use of separate colors for input layer
# color based on relative importance using 'gar.fun'

##
#create input data
seed.val<-3
set.seed(seed.val)

num.vars<-8
num.obs<-1000

#input variables
library(clusterGeneration)
cov.mat<-genPositiveDefMat(num.vars,covMethod=c("unifcorrmat"))$Sigma
rand.vars<-mvrnorm(num.obs,rep(0,num.vars),Sigma=cov.mat)

# output variables
parms<-runif(num.vars,-10,10)
y1<-rand.vars %*% matrix(parms) + rnorm(num.obs,sd=20)
```

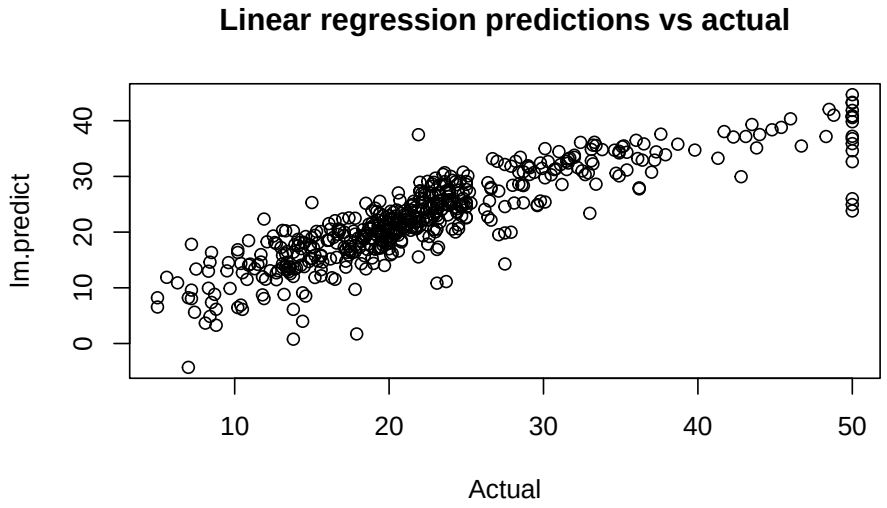
```
# final datasets
rand.vars<-data.frame(rand.vars)
resp<-data.frame(y1)
names(resp)<-'Y1'
dat.in <- data.frame(resp,rand.vars)

##
# create model
library(nnet)
mod1 <- nnet(rand.vars,resp,data=dat.in,size=10,linout=T)
#> # weights:  101
#> initial  value 844959.580478
#> iter  10 value 543616.101824
#> iter  20 value 479986.887846
#> iter  30 value 465607.784054
#> iter  40 value 454237.073298
#> iter  50 value 445032.412421
#> iter  60 value 433191.158624
#> iter  70 value 426321.161292
#> iter  80 value 424900.966883
#> iter  90 value 423816.437605
#> iter 100 value 422064.114812
#> final  value 422064.114812
#> stopped after 100 iterations

##
# relative importance function
library(devtools)
source_url('https://gist.github.com/fawda123/6206737/raw/2e1bc9cbc48d1a56d2a79dd1d33f414213f5f1b1/gar_fu
#> SHA-1 hash of file is 9faa58824c46956c3ff78081696290d9b32d845f

# relative importance of input variables for Y1
rel.imp <- gar.fun('Y1',mod1,bar.plot=F)$rel.imp

#color vector based on relative importance of input values
cols<-colorRampPalette(c('green','red'))(num.vars)[rank(rel.imp)]

##
#plotting function
source_url('https://gist.githubusercontent.com/fawda123/7471137/raw/466c1474d0a505ff044412703516c34f1a4
#> SHA-1 hash of file is 74c80bd5ddbc17ab3ae5ece9c0ed9beb612e87ef

#plot model with new color vector
#separate colors for input vectors using a list for 'circle.col'
plot(mod1,circle.col=list(cols,'lightblue'))
```


B1
B2
X1
X2
X3
X4
X5
X6
X7
X8
I1
I2
I3
I4
I5
I6
I7
I8
H1
H2
H3
H4
H5
H6
H7
H8
H9
H10
O1
Y1

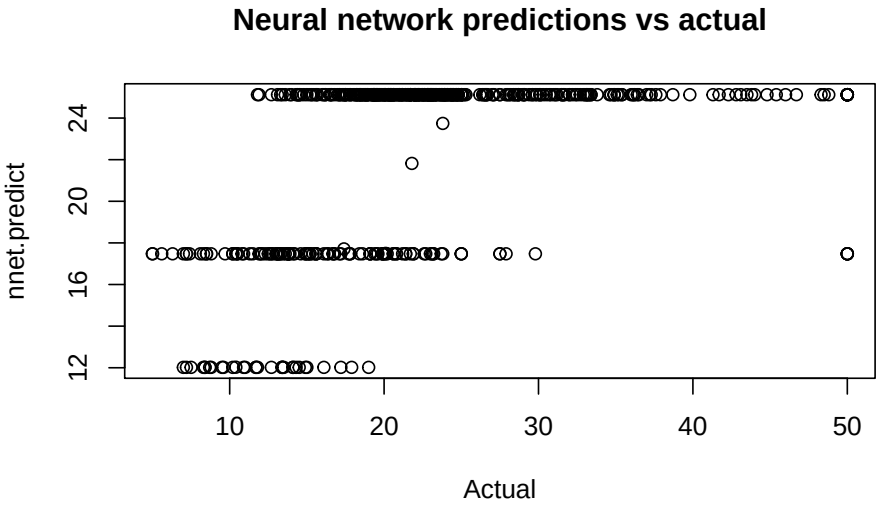
Chapter 4

Regression Boston nnet

```
###
### prepare data
###
library(mlbench)
data(BostonHousing)

# inspect the range which is 1-50
summary(BostonHousing$medv)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>     5.0    17.0    21.2    22.5    25.0    50.0


##
## model linear regression
##

lm.fit <- lm(medv ~ ., data=BostonHousing)

lm.predict <- predict(lm.fit)

# mean squared error: 21.89483
mean((lm.predict - BostonHousing$medv)^2)
#> [1] 21.9

plot(BostonHousing$medv, lm.predict,
    main="Linear regression predictions vs actual",
    xlab="Actual")
```




```
##
## model neural network
##
require(nnet)
#> Loading required package: nnet

# scale inputs: divide by 50 to get 0-1 range
nnet.fit <- nnet(medv/50 ~ ., data=BostonHousing, size=2)
#> # weights:  31
#> initial  value 17.039194
#> iter  10 value 13.754559
#> iter  20 value 13.537235
#> iter  30 value 13.537183
#> iter  40 value 13.530522
#> final  value 13.529736
#> converged

# multiply 50 to restore original scale
nnet.predict <- predict(nnet.fit)*50

# mean squared error: 16.40581
mean((nnet.predict - BostonHousing$medv)^2)
#> [1] 66.8

plot(BostonHousing$medv, nnet.predict,
    main="Neural network predictions vs actual",
    xlab="Actual")
```



4.1 Neural Network

Now, let's use the function train() from the package caret to optimize the neural network hyperparameters decay and size, Also, caret performs resampling to give a better estimate of the error. In this case we scale linear regression by the same value, so the error statistics are directly comparable.

```
library(mlbench)
data(BostonHousing)

require(caret)
#> Loading required package: caret
#> Loading required package: lattice
#> Loading required package: ggplot2
#> Registered S3 methods overwritten by 'ggplot2':
#>   method         from
#>   [.quosures     rlang
#>   c.quosures     rlang
#>   print.quosures rlang

mygrid <- expand.grid(.decay=c(0.5, 0.1), .size=c(4,5,6))
nnetfit <- train(medv/50 ~ ., data=BostonHousing, method="nnet", maxit=1000, tuneGrid=mygrid, trace=F)
print(nnetfit)
#> Neural Network
#>
#> 506 samples
#>  13 predictor
#>
#> No pre-processing
#> Resampling: Bootstrapped (25 reps)
#> Summary of sample sizes: 506, 506, 506, 506, 506, 506, ...
#> Resampling results across tuning parameters:
#>
#>   decay  size  RMSE    Rsquared  MAE
#>   0.1    4     0.0835  0.787     0.0571
#>   0.1    5     0.0822  0.794     0.0565
#>   0.1    6     0.0799  0.806     0.0544
```

```
#>   0.5    4     0.0908  0.757     0.0626
#>   0.5    5     0.0900  0.761     0.0624
#>   0.5    6     0.0895  0.763     0.0622
#>
#> RMSE was used to select the optimal model using the smallest value.
#> The final values used for the model were size = 6 and decay = 0.1.
```

```
506 samples
 13 predictors

No pre-processing
Resampling: Bootstrap (25 reps)

Summary of sample sizes: 506, 506, 506, 506, 506, 506, ...

Resampling results across tuning parameters:

  size  decay  RMSE    Rsquared  RMSE SD  Rsquared SD
  4     0.1    0.0852  0.785     0.00863  0.0406
  4     0.5    0.0923  0.753     0.00891  0.0436
  5     0.1    0.0836  0.792     0.00829  0.0396
  5     0.5    0.0899  0.765     0.00858  0.0399
  6     0.1    0.0835  0.793     0.00804  0.0318
  6     0.5    0.0895  0.768     0.00789  0.0344
```

4.2 Linear Regression

```
lmfit <- train(medv/50 ~ ., data=BostonHousing, method="lm")
print(lmfit)
#> Linear Regression
#>
#> 506 samples
#>  13 predictor
#>
#> No pre-processing
#> Resampling: Bootstrapped (25 reps)
#> Summary of sample sizes: 506, 506, 506, 506, 506, 506, ...
#> Resampling results:
#>
#>   RMSE    Rsquared  MAE
#>   0.0988  0.726     0.0692
#>
#> Tuning parameter 'intercept' was held constant at a value of TRUE
```

```
506 samples
 13 predictors

No pre-processing
Resampling: Bootstrap (25 reps)

Summary of sample sizes: 506, 506, 506, 506, 506, 506, ...
```

```
Resampling results

  RMSE    Rsquared  RMSE SD  Rsquared SD
  0.0994  0.703     0.00741  0.0389
```

A tuned neural network has a RMSE of 0.0835 compared to linear regression's RMSE of 0.0994.