**Neural Networks – part I**

A neural network (NN) model is very similar to a non-linear regression model, with the exception that the former can handle an incredibly large amount of model parameters. For this reason, neural network models are said to have the ability to approximate any continuous function. Neural network is very good at learning non-linear function and also multiple outputs can be learnt at the same time. However, the training time is relatively long and it is also susceptible to local minimum traps. This can be mitigated by doing multiple rounds and pick the best learned model.

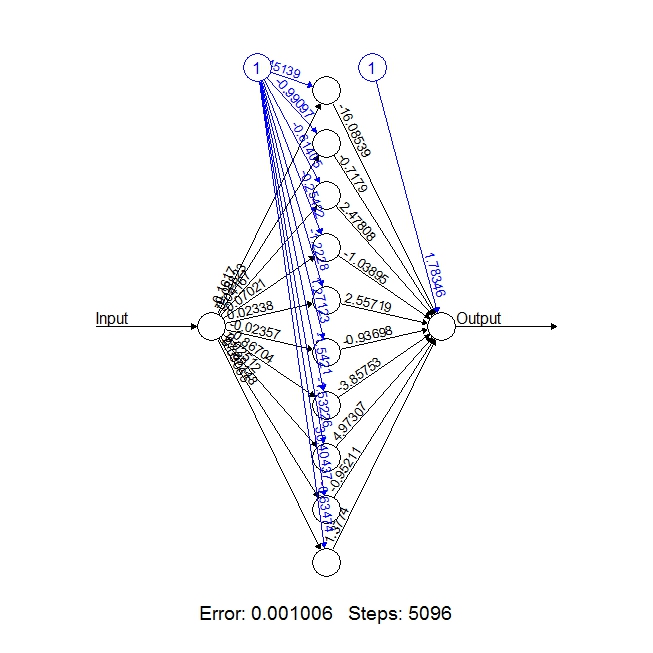
R has several packages for dealing with NNs, like neuralnet, nnet, and caret

**neuralnet** depends on two other packages: **grid** and **MASS**. It is used is primarily with functions dealing with regression analyses like linear models (**lm**) and general linear models (**glm**). As essential arguments, we must specify a formula in terms of *response variables* ~ *sum of covariates* and a data set containing covariates and response variables. Default values are defined for all other parameters. The function **neuralnet,** used for training a neural network, provides the opportunity to define the required number of hidden layers and hidden neurons according to the needed complexity.  The complexity of the calculated function increases with the addition of hidden layers or hidden neurons. The default value is one hidden layer with one hidden neuron. The most important arguments of the function are the following:

* **formula**, a symbolic description of the model to be fitted. No default.
* **data**, a data frame containing the variables specified in formula. No default.
* **hidden**, a vector specifying the number of hidden layers and hidden neurons in each layer. For example the vector (3,2,1) induces a neural network with  three hidden layers, the first
* **startweights**, a vector containing prespecified starting values for the weights.  Default:  random numbers drawn from the standard normal distribution
* **algorithm**,   a string containing the algorithm type. Possible values are “**backprop**“, “**rprop**+”, “**rprop**-“, “**sag**“, or “**slr**“. “backprop” refers to traditional  backpropagation, “**rprop**+” and “**rprop**-” refer to resilient backpropagation  with and  without weight backtracking and “**sag**” and “**slr**” refer to the modified  globally convergent algorithm  (gr-prop). “**sag**” and “**slr**” define the learning rate that is changed according to all others. “**sag**” refers to the smallest absolute derivative, “**slr**” to the smallest learning rate. Default: “**rprop**+”
* **err.fct**, a differentiable error function. The strings “**sse**” and “**ce**” can be used, which refer to ‘sum of squared errors’ and ‘cross entropy’. Default: “**sse**“
* **act.fct**, a differentiable activation function. The strings “**logistic**” and “**tanh**” are possible for the logistic function and tangens hyperbolicus (that is the Latin name for hyperbolic tangent). Default: “**logistic**“
* **linear.output**, logical.   If act.fct should not be applied to the output   neurons, **linear.output** has to be **TRUE**. Default: **TRUE**
* **likelihood**, logical. If the error function is equal to the negative log-likelihood function, **likelihood** has to be **TRUE**. Akaike’s Information Criterion and Bayes Information Criterion will then be calculated. Default: **FALSE**
* **exclude**, a vector or matrix specifying weights that should be excluded from training.  A matrix with *n* rows and three columns will exclude *n* weights, where the first column indicates the layer, the second column the input neuron of the weight, and the third neuron the output neuron of the weight. If given as vector, the exact numbering has to be known.  The numbering can be checked using the provided plot or the saved starting weights. Default: **NULL**
* **constant.weights**, a vector specifying the values of weights that are excluded from training and treated as fixed. Default: **NULL**

**Our first NN**

In this first exercise, a neural network (or Multilayer perceptron) will be build that is able to take a number and calculate the square root (or as close to as possible). The exercise will produce the neural network shown in the image below. It is going to take a single input (the number that you want square rooting) and produce a single output (the square root of the input). The middle of the image contains 10 hidden neurons which will be trained.



**install.packages**('neuralnet')

[**library**](http://astrostatistics.psu.edu/su07/R/html/graphics/html/library.html)("neuralnet")

#Going to create a neural network to perform square rooting

#Type ?neuralnet for more information on the neuralnet library

#Generate 50 random numbers uniformly distributed between 0 and 100, and #store them as a dataframe

traininginput <- [**as.data.frame**](http://astrostatistics.psu.edu/su07/R/html/graphics/html/as.data.frame.html)(**runif**(50, [**min**](http://astrostatistics.psu.edu/su07/R/html/graphics/html/min.html)=0, [**max**](http://astrostatistics.psu.edu/su07/R/html/graphics/html/max.html)=100))

trainingoutput <- [**sqrt**](http://astrostatistics.psu.edu/su07/R/html/graphics/html/sqrt.html)(traininginput)

#Column bind the data into one variable

trainingdata <- [**cbind**](http://astrostatistics.psu.edu/su07/R/html/graphics/html/cbind.html)(traininginput,trainingoutput)

[**colnames**](http://astrostatistics.psu.edu/su07/R/html/graphics/html/colnames.html)(trainingdata) <- [**c**](http://astrostatistics.psu.edu/su07/R/html/graphics/html/c.html)("Input","Output")

#Train the neural network

#Going to have 10 hidden layers

#Threshold is a numeric value specifying the threshold for the partial

#derivatives of the error function as stopping criteria.

net.sqrt <- neuralnet(Output~Input,trainingdata, hidden=10, threshold=0.01)

[**print**](http://astrostatistics.psu.edu/su07/R/html/graphics/html/print.html)(net.sqrt)

## Call: neuralnet(formula = Output ~ Input, data = trainingdata, hidden = 10, threshold = 0.01)

##

## 1 repetition was calculated.

##

## Error Reached Threshold Steps

## 1 0.0006282191137 0.009245995019 6812

#Plot the neural network

[**plot**](http://astrostatistics.psu.edu/su07/R/html/stats/html/plot.html)(net.sqrt)

#Test the neural network on some training data

testdata <- [**as.data.frame**](http://astrostatistics.psu.edu/su07/R/html/graphics/html/as.data.frame.html)((1:10)^2) #Generate some squared numbers

net.results <- compute(net.sqrt, testdata) #Run them through the neural network

#Lets see what properties net.sqrt has

[**ls**](http://astrostatistics.psu.edu/su07/R/html/graphics/html/ls.html)(net.results)

## [1] "net.result" "neurons"

#Lets see the results

[**print**](http://astrostatistics.psu.edu/su07/R/html/graphics/html/print.html)(net.results$net.result)

## [,1]

## [1,] 0.9143358513

## [2,] 1.9942238404

## [3,] 3.0005677587

## [4,] 3.9981018905

## [5,] 5.0008748616

## [6,] 6.0022207302

## [7,] 6.9938605784

## [8,] 7.9987651833

## [9,] 9.0110881921

## [10,] 9.9793247077

#Lets display a better version of the results

cleanoutput <- [**cbind**](http://astrostatistics.psu.edu/su07/R/html/graphics/html/cbind.html)(testdata,[**sqrt**](http://astrostatistics.psu.edu/su07/R/html/graphics/html/sqrt.html)(testdata),

[**as.data.frame**](http://astrostatistics.psu.edu/su07/R/html/graphics/html/as.data.frame.html)(net.results$net.result))

[**colnames**](http://astrostatistics.psu.edu/su07/R/html/graphics/html/colnames.html)(cleanoutput) <- [**c**](http://astrostatistics.psu.edu/su07/R/html/graphics/html/c.html)("Input","Expected Output","Neural Net Output")

[**print**](http://astrostatistics.psu.edu/su07/R/html/graphics/html/print.html)(cleanoutput)

## Input Expected Output Neural Net Output

## 1 1 1 0.9143358513

## 2 4 2 1.9942238404

## 3 9 3 3.0005677587

## 4 16 4 3.9981018905

## 5 25 5 5.0008748616

## 6 36 6 6.0022207302

## 7 49 7 6.9938605784

## 8 64 8 7.9987651833

## 9 81 9 9.0110881921

## 10 100 10 9.9793247077

You can also use 2 hidden layers: replace hidden=c(10,8), just to show the application of multiple hidden layers.

**Modelling of nonlinear systems using NN**

In this example we will be providing estimates of the performance of building materials. Estimate requirements are important to develop safety guidelines governing the materials used in the construction of buildings, bridges, and roadways. An NN model could predict concrete strength given a listing of composition of the input materials could results in safer construction practices.

**Step 1 - Data Collection**

For this NN analysis, we will utilize the concrete data donated to the UCI Machine Learning Data Repository (<http://archive.ics.uci.edu/ml>). We will attempt to replicate his work using a simple neural network model in R. (concrete.csv)

**Step 2 - Exploring and Preparing the Data**

The concrete dataset contains 1,030 examples of concrete with eight features describing the components used in the mixture. These features are thought to be related to the final compressive strength and they include the amount (in kilograms per cubic meter) of cement, slag, ash, water, superplastic, coarse aggregate, and fine aggregate used in the product in addition to the aging time (measured in days).

concrete <- read.csv("concrete.csv")

str(concrete)

## 'data.frame': 1030 obs. of 9 variables:

## $ cement : num 141 169 250 266 155 ...

## $ slag : num 212 42.2 0 114 183.4 ...

## $ ash : num 0 124.3 95.7 0 0 ...

## $ water : num 204 158 187 228 193 ...

## $ superplastic: num 0 10.8 5.5 0 9.1 0 0 6.4 0 9 ...

## $ coarseagg : num 972 1081 957 932 1047 ...

## $ fineagg : num 748 796 861 670 697 ...

## $ age : int 28 14 28 28 28 90 7 56 28 28 ...

## $ strength : num 29.9 23.5 29.2 45.9 18.3 ...

The nine variables in the data frame correspond to the eight features and one outcome. Neural networks work best when the input data are scaled to a narrow range around zero, and here, we see values ranging anywhere from zero up to over a thousand. Typically, the solution to this problem is to rescale the data with a normalizing or standardization function. If the data follow a bell-shaped curve (a normal distribution), then it may make sense to use standardization via R’s built-in scale() function. On the other hand, if the data follow a uniform distribution or are severely non-normal, then normalization to a 0-1 range may be more appropriate. In this case, we’ll use the latter.

So, we’ll go ahead defined our own normalize() function as:

normalize <- function(x) {

return((x - min(x)) / (max(x) - min(x)))

}

Let’s apply to every column in the concrete data frame using the lapply() function and verify.

concrete\_norm <- as.data.frame(**lapply**(concrete, normalize))

summary(concrete\_norm$strength)

## Min. 1st Qu. Median Mean 3rd Qu. Max.

## 0.0000 0.2664 0.4001 0.4172 0.5457 1.0000

As you can see below, the original minimum and maximum values were 2.33 and 82.60:

summary(concrete$strength)

## Min. 1st Qu. Median Mean 3rd Qu. Max.

## 2.33 23.71 34.44 35.82 46.14 82.60

Check the summary for all variables

summary(concrete\_norm)

Check the difference between sapply and lapply commands. It depends on what you want to receive back. If you want a list returned, use lapply. If you want a vector, use sapply.

**Step 3 - Training a Model on the Data**

We will partition the data into a training set with 75% of the examples and a testing set with 25%.

concrete\_train <- concrete\_norm[1:773, ]

concrete\_test <- concrete\_norm[774:1030, ]

Alternative method:

In order to split our data into a training set and a test set, we can also use the caTools package to randomly split the data into a training set and test set.

Let’s begin by installing the neuralnet library package with the training set as the simplest multilayer feedforward network with only a single hidden node:

#install.packages("neuralnet")

library(neuralnet)

## Loading required package: grid

## Loading required package: MASS

library(grid)

library(MASS)

concrete\_model <- neuralnet(strength ~ cement + slag + ash + water + superplastic + coarseagg + fineagg + age, data = concrete\_train)

The following will produce a plot visually the network topology using the plot() function on the resulting model object.

plot(concrete\_model)

# alternative plot

#install.packages("NeuralNetTools")

library(NeuralNetTools)

# plotnet

par(mar = numeric(4), family = 'serif')

plotnet(concrete\_model, alpha = 0.6)

As you can see above in this simple mode, there is one input node for each of the eight features followed by a single hidden node and a single output node that predicts the concrete strength. The weights for each connection are displayed. Some bias terms are indicated by nodes labelled with the number 1. These bias terms are numeric constants that allow the value at the indicated nodes to be shifted upward or downward, much like the intercept in a linear equation. The bottom of the plot displays the number of training steps and Sum of Squared Errors (SSE) (sum of the squared predicted minus actual values). In our mode, we have an SSE of 5.077771 and 4293 training steps. A lower SSE implies better predictive performance. This is helpful for estimating the model’s performance on the training data, but tells us little about how it will perform on unseen data.

**Step 4 - Evaluating Model Performance**

The network topology diagram gives us a peek into the black box of the NN, but it doesn’t provide much information about how well the model fits future data. To generate predictions on the test dataset, we can use the compute() as follows:

model\_results <- compute(concrete\_model, concrete\_test[1:8])

Note: The compute() function works returns a list with two components: $neurons, which stores the neurons for each layer in the network, and $net.result, which stores the predicted values. Let’s use \*\* $net.result \*\*.

predicted\_strength <- model\_results$net.result

Since this is a numeric prediction problem rather than a classification problem, we cannot use a confusion matrix to examine model accuracy. Instead, we must measure the correlation between our predicted concrete strength and the true value. This provides insight into the strength of the linear association between the two variables. Thus, we will use the cor() function to obtain a correlation between the two numeric vectors.

cor(predicted\_strength, concrete\_test$strength)

## [,1]

## [1,] 0.8063884572

A correlation close to 1 indicates strong linear relationships between two variables. Therefore, the correlation here of about 0.8063904949 indicates a fairly strong relationship. This implies that our model is doing a fairly good job, even with only a single hidden node. Given that we only used one hidden node, it is likely that we can improve the performance of our model. Let’s improve on this in Step 5.

**Step 5 - Improving Model Performance**

As networks with more complex topologies are capable of learning more difficult concepts, let’s see what happens when we increase the number of hidden nodes to five. We use the neuralnet() function as before, but add the hidden = 5 parameter:

concrete\_model2 <- neuralnet(strength ~ cement + slag + ash + water + superplastic + coarseagg + fineagg + age, data = concrete\_train, hidden = 5)

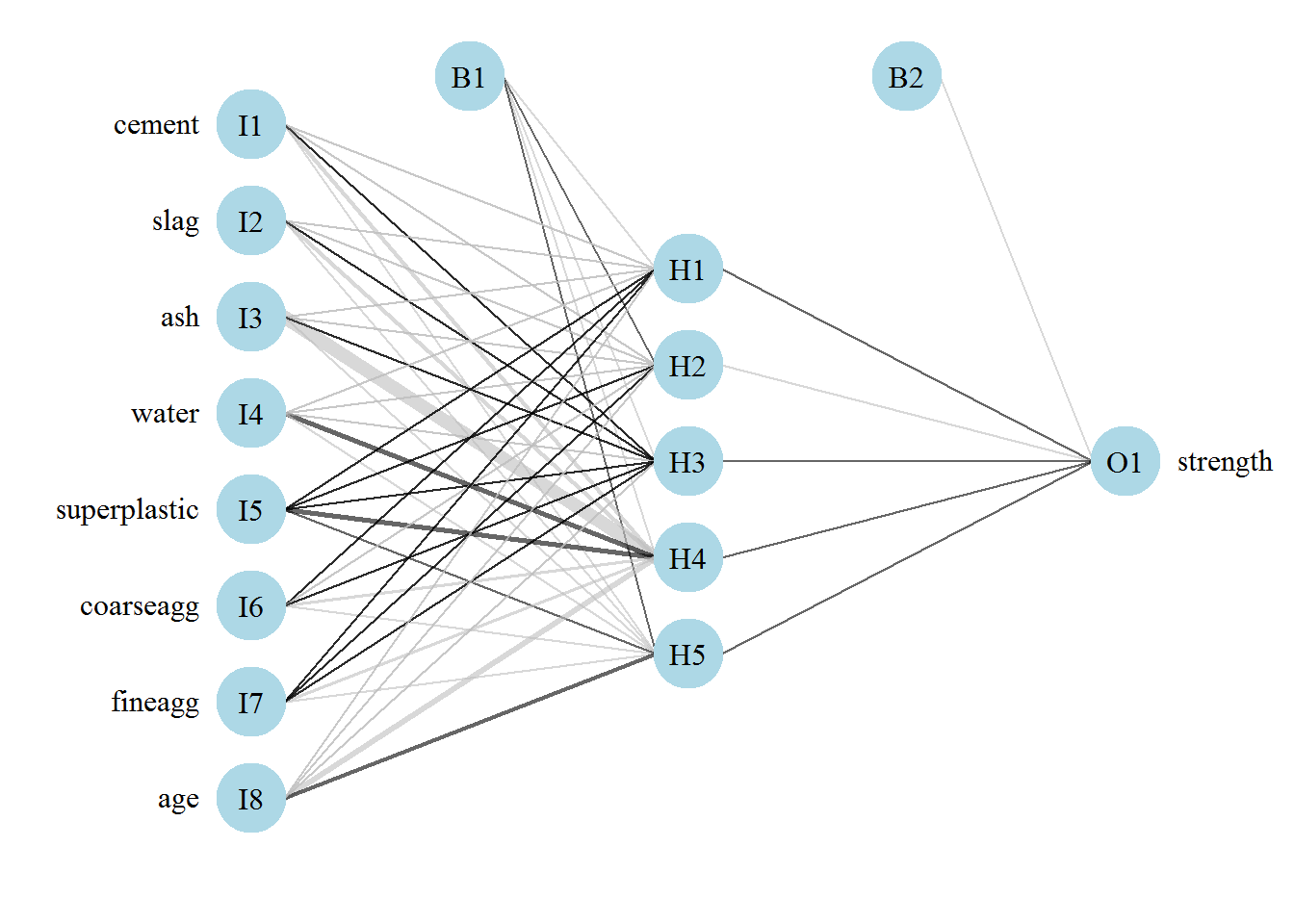
Let’s plot the network again if we see any drastic increase in the number of connections.

plot(concrete\_model2)

#alternative plot using plotnet

par(mar = numeric(4), family = 'serif')

plotnet(concrete\_model2, alpha = 0.6)



As you can see from our newly plotted model, we have a much reduced SSE of 1.692125 compared to from the previous model 5.077771. We also have an increased training steps of 12,932 compared to 4,293 from the previous model along with a few bias terms - which are indicated by nodes labelled with the number 1. This should come as no surprise given how complex the model has become. A more complex network takes more iterations to find the optimal weights. Let’s continue with our predictions and correlations.

model\_results2 <- compute(concrete\_model2, concrete\_test[1:8])

predicted\_strength2 <- model\_results2$net.result

cor(predicted\_strength2, concrete\_test$strength)

## [,1]

## [1,] 0.929033617

As you can see from above, the correlation is close to 1 - 0.932956629 - which is a drastic improvement from our earlier correlation of 0.8063904949. Check the same problem, using two hidden layers.

**Classification using NN**

We use the data set **infert** that is provided by the package **datasets** to illustrate its application. This data set contains data of a case-control study that investigated infertility after spontaneous and induced abortion. The data set consists of 248 observations, 83 women, who were infertile (cases), and 165 women, who were not infertile (controls).  It includes amongst others the variables **age**, **parity**, **induced**, and **spontaneous**. The variables **induced** and **spontaneous** denote the number of prior induced and spontaneous abortions, respectively.  Both variables take possible values 0, 1, and 2 relating to 0, 1, and 2 or more prior abortions. The age in years is given by the variable **age** and the number of births by **parity**.

The usage of **neuralnet** is described by modelling the relationship between the case-control status (case) as response variable and the four covariates **age**, parity, induced and **spontaneous**. Since the response variable is binary, we could choose the logistic function (default) as the activation function and (**err.fct="ce"**) as the error function as cross-entropy. Additionally, we should state the item **linear.output** as **FALSE** to ensure that the output is mapped by the activation function to the interval [0, 1]. The number of hidden neurons should be determined in relation to the needed complexity.

[**library**](http://inside-r.org/r-doc/base/library)([**datasets**](http://inside-r.org/r-doc/datasets))  
   
[**library**](http://inside-r.org/r-doc/base/library)([neuralnet](http://inside-r.org/packages/cran/neuralnet" \t "_blank))

Install packages: grid & MASS

[**names**](http://inside-r.org/r-doc/base/names)(**[infert](http://inside-r.org/r-doc/datasets/infert" \t "_blank)**)

   
nn <- [neuralnet](http://inside-r.org/packages/cran/neuralnet)(case~age+parity+induced+spontaneous, [**data**](http://inside-r.org/r-doc/utils/data)=**[infert](http://inside-r.org/r-doc/datasets/infert" \t "_blank)**, hidden=2, err.fct="ce", linear.output=**FALSE**)

Alternative:

nn = neuralnet ( formula = case~age+parity+induced+spontaneous, data = infert, hidden = 2, err.fct = "ce", linear.output = FALSE)

The default activation function is the sigmoid or logistic function that exists in neuralnet. In case you need to use another one that does not exist in the built-in package you can create it, like the following example:

sigmoid = function(x) {

1 / (1 + exp(-x))

}

nn2 <- neuralnet(case~parity+induced+spontaneous, infert, err.fct="ce", linear.output=FALSE, likelihood=TRUE, act.fct = sigmoid)

Output training results

# basic

nn  
   
# results options  
[**names**](http://inside-r.org/r-doc/base/names)(nn)  
   
We save basic information about the training process and the trained neural network in **nn**.  This includes all information that has to be known to reproduce the results as for instance the starting weights. Important values are the following:

* **net.result**, a list containing the overall result, i.e. the output, of the neural network for each replication.
* **weights**, a list containing the fitted weights of the neural network for each replication.
* **generalized.weights**, a list containing the generalized weights of the neural network for each replication.
* **result.matrix**, a matrix containing the error, reached threshold, needed steps, AIC and BIC (computed if likelihood=TRUE) and estimated weights for each replication. Each column represents one replication.
* **startweights**, a list containing the starting weights for each replication.

# result matrix  
nn$result.matrix

The given data is saved in nn$covariate and nn$response as well as in nn$data for the whole data set inclusive non-used variables. The output of the neural network, i.e. the fitted values o(x), is provided by nn$net.result:

out <- cbind(nn$covariate,nn$net.result[[1]])

[**dimnames**](http://inside-r.org/r-doc/base/dimnames)(out) <- [**list**](http://inside-r.org/r-doc/base/list)(**NULL**, [**c**](http://inside-r.org/r-doc/base/c)("age", "parity","induced","spontaneous","nn-output"))  
 [**head**](http://inside-r.org/r-doc/utils/head)(out)  
 The generalized weight expresses the effect of each ovariate xi and thus has an analogous interpretation as the ith regression parameter in regression models. However, the generalized weight depends on all other covariates. Its distribution indicates whether the effect of the covariate is linear since a small variance suggests a linear effect The columns refer to the four covariates age (j = 1), parity (j = 2), induced (j = 3), and spontaneous (j=4)

[**head**](http://inside-r.org/r-doc/utils/head)(nn$generalized.weights[[1]])  
 # visualization

[**plot**](http://inside-r.org/r-doc/graphics/plot)(nn)

**Homework**

Check also

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