## Introduction

Deep Learning isn't a recent discovery. The seeds were sown back in the 1950s when the first artificial neural network was created. Since then, progress has been rapid, with the structure of the neuron being "re-invented" artificially.

Computers and mobiles have now become powerful enough to identify objects from images.

Not just images, they can chat with you as well! Haven't you tried Google's Allo app ? That's not all—they can drive, make supersonic calculations, and help businesses solve the most complicated problems (more users, revenue, etc).

But, what is driving all these inventions? It's Deep Learning!

With increasing open source contributions, R language now provides a fantastic interface for building predictive models based on neural networks and deep learning. However, learning to build models isn't enough. You ought to understand the interesting story behind them.

In this tutorial, I'll start with the basics of neural networks and deep learning (from scratch). Along with theory, we'll also learn to build deep learning models in R using MXNet and H2O package. Also, we'll learn to tune parameters of a deep learning model for better model performance.

**Note: This article is meant for beginners and expects no prior understanding of deep learning (or neural networks).**

## Table of Contents



## What is Deep Learning ? How is it different from a Neural Network?

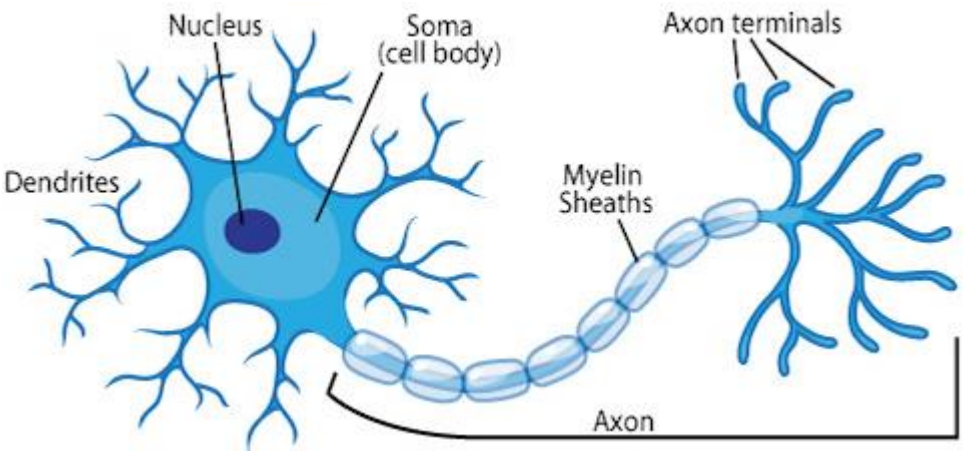
Deep Learning is the new name for multilayered neural networks. You can say, deep learning is an enhanced and powerful form of a neural network. The difference between the two is subtle.

The difference lies in the fact that, deep learning models are build on several hidden layers (say, more than 2) as compared to a neural network (built on up to 2 layers).

Since data comes in many forms (tables, images, sound, web etc), it becomes extremely difficult for linear methods to learn and detect the non - linearity in the data. In fact, many a times even non-linear algorithms such as tree based (GBM, decision tree) fails to learn from data. In such cases, a multi layered neural network which creates non - linear interactions among the features (i.e. goes deep into features) gives a better solution. You might ask this question, 'Neural networks emerged in 1950s. But, deep learning emerged just few years back. What happened all of a sudden in last few years?' In the last few years, there has been tremendous advancement in computational devices (specially GPUs). High performance of deep learning models come with a cost i.e. computation. They require large memory for computation. The world is continually progressing from CPU to GPU (Graphics Processing Unit). Why ? Because, a CPU can be enabled with max. 22 cores, but a GPU can contain thousands of cores, thereby making it exponentially powerful than a CPU.

## How does Deep Learning work ?

To understand deep learning, let's start with basic form of neural network architecture i.e. perceptron. A Neural Network draws its structure from a human neuron. A human neuron looks like this:

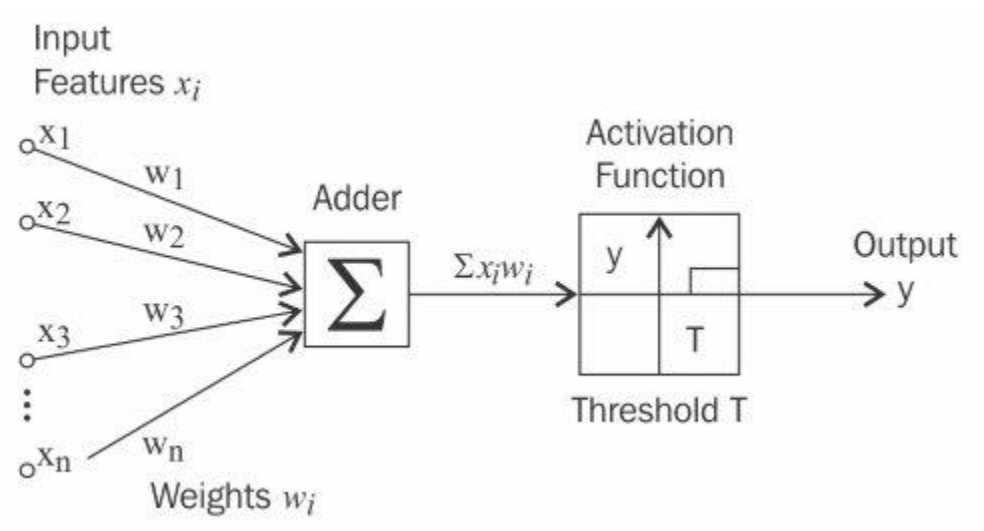



Yes, you have it too. And, not just one, but billions. We have billions of neurons and trillions of synapses (electric signals) which pass through them. Watch this short video (~2mins) to understand your brain better.

It works like this:

1. The dendrites receive the input signal (message).
2. These dendrites apply a weight to the input signal. Think of weight as "importance factor" i.e. higher the weight, higher the importance of signal.
3. The soma (cell body) acts on the input signal and does the necessary computation (decision making).
4. Then, the signal passes through the axon via a threshold function. This function decides whether the signal needs to be passed further.
5. If the input signal exceeds the threshold, the signal gets fired though the axon to terminals to other neuron.

This is a simplistic explanation of human neurons. The idea is to make you understand the analogy between human and artificial neurons. Now, let's understand the working of an artificial neuron. The process is quite similar to the explanation above. Make sure you understand it well because it's the fundamental concept of neural network. A simplistic artificial neuron looks like this:




Here x1, x2, ... xn are the input variables (or independent variables). As the input variables are fed into the network, they get assigned some random weights (w1,w2...wn). Alongside, a bias (wo) is added to the network (explained below). The adder adds all the weighted input variable. The output (y) is passed through the activation function and calculated using the equation:

$$y = g\left( w_0 + \sum_{i=1}^{p} w_i x_i \right)$$

where wo = bias, wi = weights, xi = input variables. The function g() is the activation function. In this case, the activation function works like this: if the weighted sum of input variables exceeds a certain threshold, it will output 1, else 0.

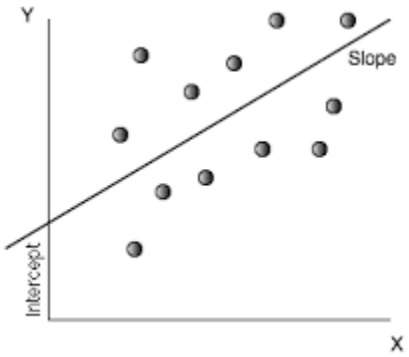
$$g(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$$

This simple neuron model is also known as **McCulloch-Pitts model or Perceptron**. In simple words, a perceptron takes several input variables and returns a binary output. Why binary output ? Because, it uses a sigmoid function as the activation function (explained below). If you remove the activation function, what you get is a simple regression model. After adding the sigmoid activation function, it performs the same task as logistic regression.

However, perceptron isn't powerful enough to work on linearly inseparable data. Due to its limitations, Multilayer Perceptron (MLP) came into existence. If the perceptron is one neuron, think of MLP as a complete brain which comprises several neurons.

### Why is bias added in the neural network ?

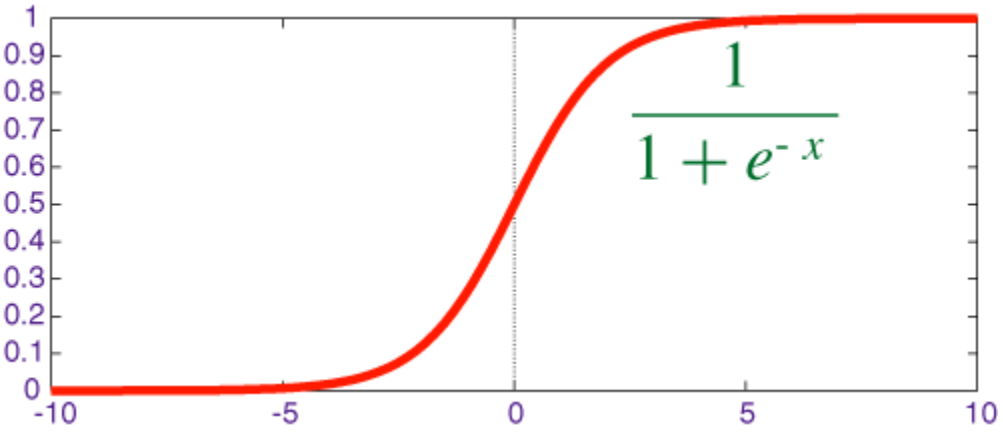
Bias (wo) is similar to the intercept term in linear regression. It helps improve the accuracy of prediction by shifting the decision boundary along Y axis. For example, in the image shown below, had the slope emerged from the origin, the error would have been higher than the error after adding the intercept to the slope.




Similarly, in a neural network, the bias helps in shifting the decision boundary to achieve better predictions.

### What are activation functions and their types ?

The perceptron classifies instances by processing a linear combination of input variables through the activation function. We also learned above that the perceptron algorithm returns binary output by using a sigmoid function (shown below).

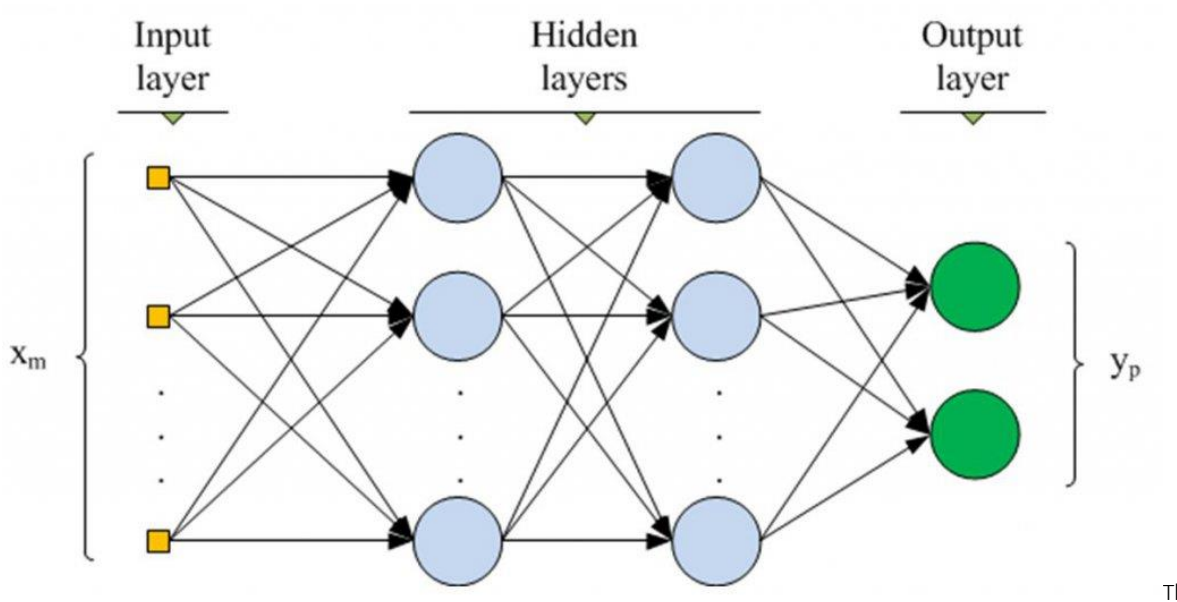



A sigmoid function (or logistic neuron ) is used in logistic regression. This function caps the max and min values at 1 and 0 such that any large positive number becomes 1 and large negative number becomes 0. It is used in neural networks because it has nice mathematical properties (derivative is easier to compute), which help calculate gradient in the backpropagation method (explained below).

In general, activation functions govern the type of decision boundary to produce given a non-linear combination of input variables. Also, due to their mathematical properties, activation functions play a significant role in optimizing prediction accuracy. Here is a complete list of activation functions you can find.

### Multi Layer Neural Network (or Deep Learning Model)

A multilayered neural network comprises a chain of interconnected neurons which creates the neural architecture. As shown below, along with input and output layers, it consists of multiple hidden layers also. Don't worry about the word "hidden;" it's how middle layers are named.




The input layer consists of neurons equal to the number of input variables in the data. The number of neurons in the hidden layer depends on the user. In R, we can find the optimum number of neurons in the hidden layer using a cross-validation strategy. Multilayered neural networks are preferred when the given data set has a large number of features. That's why this model is being widely used to work on images, text data, etc. There are several types of neural networks; two of which are most commonly used:

1. **Feedforward Neural Network:** In this network, the information flows in one direction, i.e., from the input node to the output node.
2. **Recurrent (or Feedback) Neural Network:** In this network, the information flows from the output neuron back to the previous layer as well. It uses the backpropagation algorithm.

What is the Backpropagation Algorithm? How does it work ?

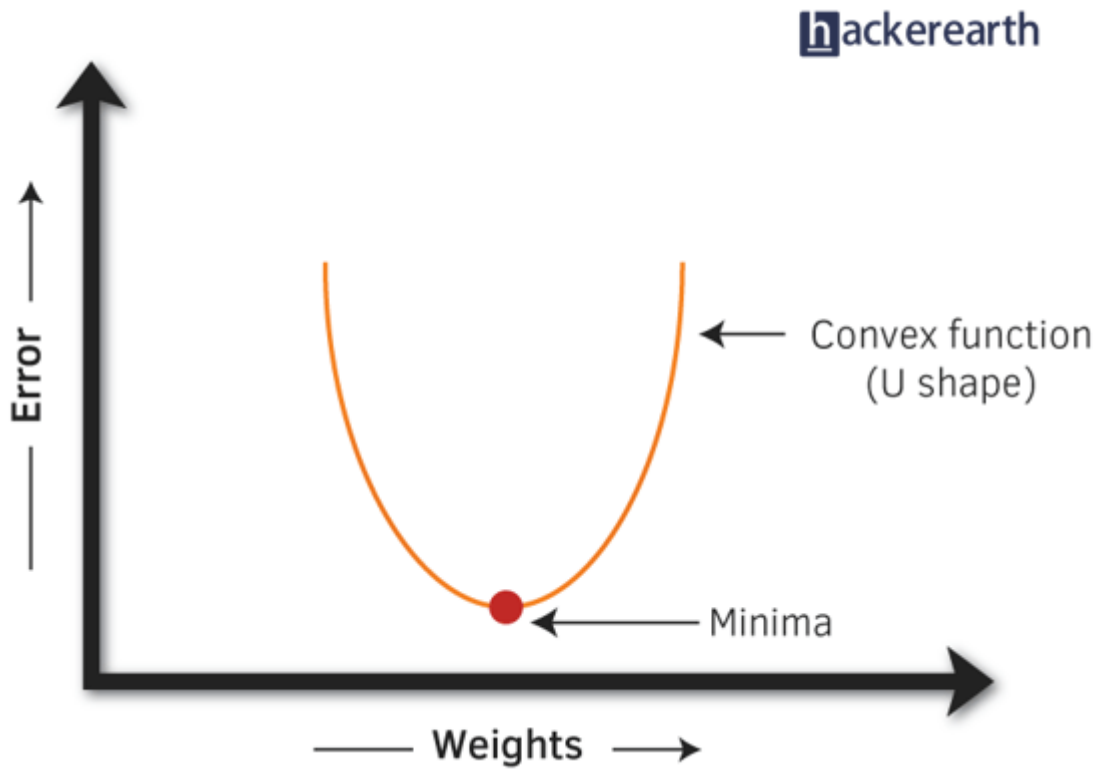
The goal of the backpropagation algorithm is to optimise the weights associated with neurons so that the network can learn to predict the output more accurately. Once the predicted value is computed, it propagates back layer by layer and re-calculates weights associated with each neuron. In simple words, it tries to bring the predicted value as close to the actual value. It's quite interesting!

The backpropagation algorithm optimises the network performance using a cost function. This cost function is minimized using an iterative sequence of steps called the **gradient descent** algorithm. Let's first understand it mathematically. Then, we'll look at an example. We'll take the cost function as squared error. It can be written as:

$$\frac{1}{2n}\sum_{i=1}^{n}\left(\hat{y}_i - y_i\right)^2$$

where n = the number of observations in the data, yi = the actual value, yi cap = the predicted value. Let's call it equation 1.

The constant value 1/2 is added in front for ease of computational purposes. You'll understand it in a while. This cost function is convex in nature. A convex function can be identified by a U-shaped curve (shown below). A great property of the convex function is that it is guaranteed to provide the lowest value when differentiated at zero. Think of a ball rolling down the curve. It will take a few rounds of rolling (up and down) to slow down and it settles at the bottom. That bottom point is the minimum. And, that's where we want togo!




If we assume that the data is fixed and the resultant cost function is a function of weights, we can re-write equation 1 as:

$$J\left(\vec{w}\right)=\frac{1}{2n}\sum_{i=1}^{n}\left(\left(\sum_{j=1}^{p}w_j x_j\right)-y_i\right)^2$$

where J(w) is the vector of weights. Note that we have only substituted ycap with its functional form (wjxj); the rest of the equation is same. Now, this equation is ready for differentiation.

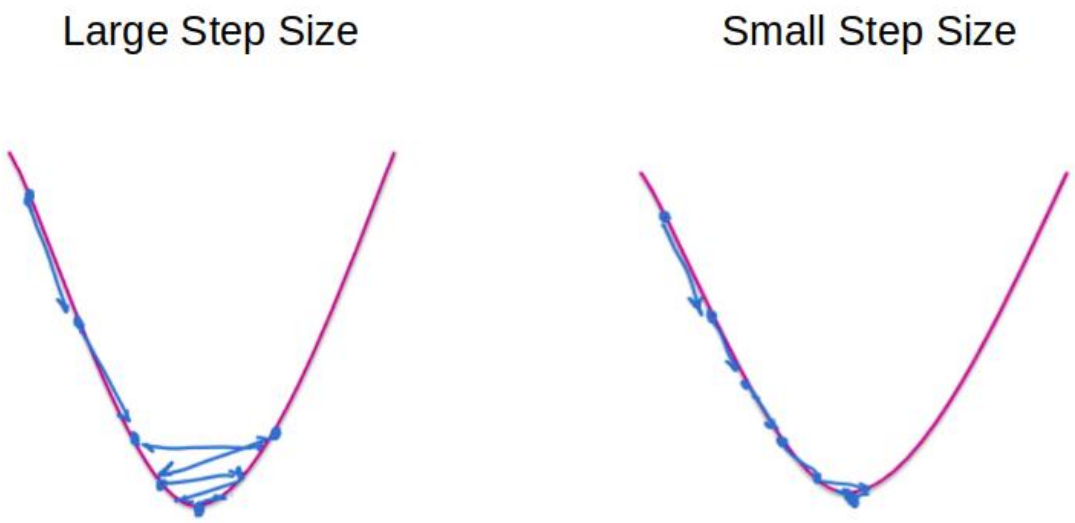
After partially differentiating the equation with respect to weights, we get a general equation as

$$\frac{\partial J\left(\vec{w}\right)}{\partial w_k}=\frac{1}{n}\sum_{i=1}^{n}\left(\left(\sum_{j=1}^{p}w_j x_j\right)-y_i\right)x_{ik}$$

$$\frac{\partial J\left(\vec{w}\right)}{\partial w_k}=\frac{1}{n}\sum_{i=1}^{n}\left(\hat{y}_i-y_i\right)x_{ik}$$

As you can see, the constant 1/2 got cancelled. In partial differentiation, we differentiate the entire equation with respect to one variable, keeping other variables constant. We also learn that the partial derivative of this cost function is just the difference between actual and predicted values multiplied by the respective weights averaged over all observations (n). The weight vector [J(w)] comprises weights corresponding to every row in the data. To compute these weights more effectively, gradient descent comes into picture. For a particular value of weight, gradient descent works like this:

1. First, it calculates the partial derivative of the weight.
2. If the derivative is positive, it decreases the weight value.
3. If the derivative is negative, it increases the weight value.
4. The motive is to reach to the lowest point (zero) in the convex curve where the derivative is minimum.
5. It progresses iteratively using a step size (η), which is defined by the user. But make sure that the step size isn't too large or too small. Too small a step size will take longer to converge, too large a step size will never reach an optimum.



Remember that the motive of gradient descent is to get to the bottom of the curve. The gradient descent equation can be written as:

$$w_k \leftarrow w_k - \eta \frac{1}{n} \sum_{i=1}^{n} \left( \hat{y}_i - y_i \right) x_{ik}$$

Let's understand it using an example. Suppose, we have a data set with 2 variables (inputs are scaled between 0 and 1):

| Age | CGPA | Target |
|---|---|---|
| 0.2 | 0.1 | 1 |
| 0.4 | 0.6 | 0 |
| 0.5 | 0.2 | 1 |

Let's run a recurrent neural network model on this data with 2 input neurons and an output neuron. The activation function is a sigmoid function. If you understand this, calculations with hidden neurons are similar. Output from one layer becomes input for the hidden layers.

**Iteration 1:** Initial Weight (randomly chosen(wo,w1,w2)): 0.1,0.1,0.1
Bias: 1
Input value: 0.2, 0.1 [1st Row]
y = 1*0.1 + 0.1*(0.2) + 0.1*(0.1) = 0.13*
y = 1 / (1 + e ^ (0.13)) = 0.467
ycap = 0 **[prediction is incorrect]**

Now, we'll re-calculate the weights using the equation above:

w1 = 0.1 - 1*1/3[(0 -1)*1] = 0.43
w2 = 0.1 - 1*1/3[(0 -1)*0.2] = 0.16
w3 = 0.1 - 1*1/3[(0 -1)*0.1] = 0.13

New weights: 0.43,0.16,0.13
Input value: 0.4, 0.6 [2nd Row]

y = 1*0.43 + 0.16*(0.4) + 0.13*(0.6) = 0.572*
y = 1 / (1 + e ^ (0.572)) = 0.36
ycap = 0 **[prediction is correct]**

Since the prediction is correct, we'll continue with the same weights:

Weights: 0.6,0.2,0.15
Input value: 0.5, 0.2

y =1*0.6 + 0.2*(0.5) + 0.15*(0.2) = 0.73*
y = 1 / (1 + e ^ (0.73)) = 0.323
ycap = 0 **[prediction is incorrect]**

Again, the algorithm will recompute the weights for **Iteration 2** and so on.

Practically, this iteration goes on until the user defined stopping criteria is reached or the algorithm converges. Since the algorithm finds weights for every row in the data, what if your data set has 10 million rows ? You are lucky if you have a powerful computational machine. But for the unlucky ones ? Don't get upset, you can use the **stochastic gradient descent** algorithm.

The only difference between gradient descent and stochastic gradient descent (SGD) is that SGD takes one observation (or a batch) at a time instead of all the observations. It assumes that the gradient for a cost function computed for a particular row of observations will be approximately equal to the gradient computed across all rows. It updates parameters (bias and weights) for each training example. Also, SGD is being widely using in online learning algorithms.

### Practical Deep Learning (+ Tuning) with H2O and MXNet

Until here, we focused on the conceptual part of deep learning. Now, we'll get some hands-on experience in building deep learning models. R offers a fantastic bouquet of packages for deep learning. Here, we'll look at two of the most powerful packages built for this purpose. For this tutorial, I've used the adult data set from the UC Irivine ML repository. Let's start with H2O. This data set isn't the most ideal one to work with in neural networks. However, the motive of this hands-on section is to make you familiar with model-building processes.

### H2O Package

H2O package provides h2o.deeplearning function for model building. It is built on Java. Primarily, this function is useful to build multilayer feedforward neural networks. It is enabled with several features such as the following:

- Multi-threaded distributed parallel computation
- Adaptive learning rate (or step size) for faster convergence
- Regularization options such as L1 and L2 which help prevent overfitting
- Automatic missing value imputation
- Hyperparameter optimization using grid/random search

There are many more! For optimization, this package uses the hogwild method instead of stochastic gradient descent. Hogwild is just parallelized version of SGD. Let's understand the parameters involved in model building with h2o. Both the packages have different nomenclatures, so make sure you don't get confused. Since most of the parameters are easy to understand by their names, I'll mention the important ones:

1. hidden - It specifies the number of hidden layers and number of neurons in each layer in the architechture.
2. epochs - It specifies the number of iterations to be done on the data set.
3. rate - It specifies the learning rate.
4. activation - It specifies the type of activation function to use. In h2o, the major activation functions are Tanh, Rectifier, and Maxout.

Let's quickly load the data and get over with sanitary data pre-processing steps:

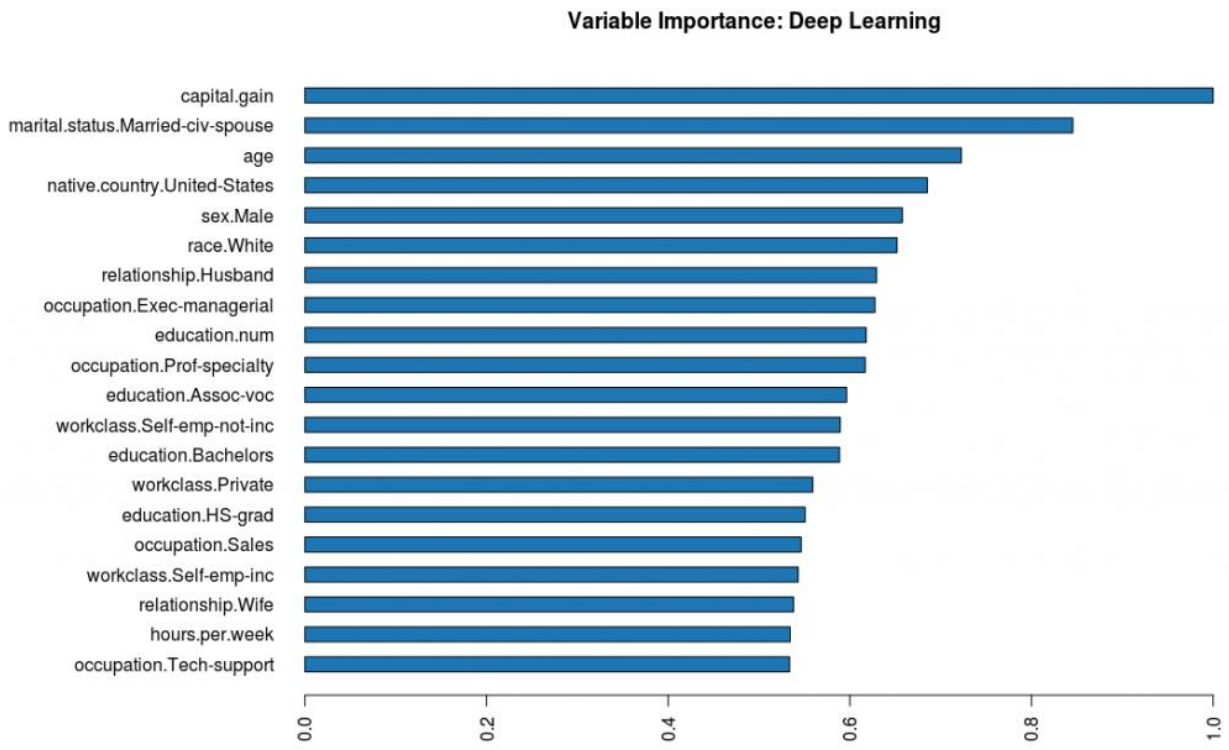
```
path = "~/mydata/deeplearning"
setwd(path)

#load libraries
library(data.table)
library(mlr)

#set variable names
setcol <- c("age",
        "workclass",
        "fnlwgt",
        "education",
        "education-num",
        "marital-status",
        "occupation",
        "relationship",
        "race",
        "sex",
        "capital-gain",
        "capital-loss",
        "hours-per-week",
        "native-country",
        "target")
```

```
#load data
train <- read.table("adultdata.txt",header = F,sep = ",",col.names = setcol,na.strings = c(" ?"),stringsAsFactors = F)
test <- read.table("adulttest.txt",header = F,sep = ",",col.names = setcol,skip = 1, na.strings = c(" ?"),stringsAsFactors = F)

setDT(train)
setDT(test)

#Data Sanity
dim(train)
[1] 32561 X 15

dim(test)
[1] 16281 X 15

str(train)
str(test)

#check missing values
table(is.na(train))
sapply(train, function(x) sum(is.na(x))/length(x))*100
table(is.na(test))
sapply(test, function(x) sum(is.na(x))/length(x))*100

#check target variable
#binary in nature check if data is imbalanced
train[,.N/nrow(train),target]
test[,.N/nrow(test),target]

#remove extra characters
test[,target := substr(target,start = 1,stop = nchar(target)-1)]

#remove leading whitespace
library(stringr)
char_col <- colnames(train)[sapply(test,is.character)]

for(i in char_col) set(train,j=i,value = str_trim(train[[i]],side = "left"))

#set all character variables as factor
fact_col <- colnames(train)[sapply(train,is.character)]

for(i in fact_col) set(train,j=i,value = factor(train[[i]]))
for(i in fact_col) set(test,j=i,value = factor(test[[i]]))

#impute missing values
imp1 <- impute(data = train,target = "target",classes = list(integer = imputeMedian(), factor = imputeMode()))
imp2 <- impute(data = test,target = "target",classes = list(integer = imputeMedian(), factor = imputeMode()))

train <- setDT(imp1$data)
test <- setDT(imp2$data)
```

Now, let's build a simple deep learning model. Generally, computing variable importance from a trained deep learning model is quite pain staking. But, h2o package provides an effortless function to compute variable importance from a deep learning model.

```
#load the package
require(h2o)

#start h2o
localH2o <- h2o.init(nthreads = -1, max_mem_size = "20G")

#load data on H2o
```

```
trainh2o <- as.h2o(train)
testh2o <- as.h2o(test)

#set variables
y <- "target"
x <- setdiff(colnames(trainh2o),y)

#train the model - without hidden layer
deepmodel <- h2o.deeplearning(x = x
                              ,y = y
                              ,training_frame = trainh2o
                              ,standardize = T
                              ,model_id = "deep_model"
                              ,activation = "Rectifier"
                              ,epochs = 100
                              ,seed = 1
                              ,nfolds = 5
                              ,variable_importances = T)

#compute variable importance and performance
h2o.varimp_plot(deepmodel,num_of_features = 20)
h2o.performance(deepmodel,xval = T)
#84.5 % CV accuracy
```




Now, let's train a deep learning model with one hidden layer comprising five neurons. This time instead of checking the cross-validation accuracy, we'll validate the model on test data.

```
deepmodel <- h2o.deeplearning(x = x
                              ,y = y
                              ,training_frame = trainh2o
                              ,validation_frame = testh2o
                              ,standardize = T
                              ,model_id = "deep_model"
                              ,activation = "Rectifier"
                              ,epochs = 100
```

```
                    ,seed = 1
                    ,hidden = 5
                    ,variable_importances = T)

h2o.performance(deepmodel,valid = T)
#85.6%
```

For hyperparameter tuning, we'll perform a random grid search over all parameters and choose the model which returns highest accuracy.

```
#set parameter space
activation_opt <- c("Rectifier","RectifierWithDropout", "Maxout","MaxoutWithDropout")
hidden_opt <- list(c(10,10),c(20,15),c(50,50,50))
l1_opt <- c(0,1e-3,1e-5)
l2_opt <- c(0,1e-3,1e-5)

hyper_params <- list( activation=activation_opt,
                hidden=hidden_opt,
                l1=l1_opt,
                l2=l2_opt )

#set search criteria
search_criteria <- list(strategy = "RandomDiscrete", max_models=10)

#train model
dl_grid <- h2o.grid("deeplearning"
                ,grid_id = "deep_learn"
                ,hyper_params = hyper_params
                ,search_criteria = search_criteria
                ,training_frame = trainh2o
                ,x=x
                ,y=y
                ,nfolds = 5
                ,epochs = 100)

#get best model
d_grid <- h2o.getGrid("deep_learn",sort_by = "accuracy")
best_dl_model <- h2o.getModel(d_grid@model_ids[[1]])
h2o.performance (best_dl_model,xval = T)
#CV Accuracy - 84.7%
```

### MXNetR Package

The mxnet package provides an incredible interface to build feedforward NN, recurrent NN and convolutional neural networks (CNNs). CNNs are being widely used in detecting objects from images. The team that created xgboost also created this package. Currently, mxnet is being popularly used in kaggle competitions for image classification problems.

This package can be easily connected with GPUs as well. The process of building model architecture is quite intuitive. It gives greater control to configure the neural network manually.

Let's get some hands-on experience using this package.

Follow the commands below to install this package in your respective OS. For Windows and Linux users, installation commands are given below. For Mac users, here's the installation procedure.

```
# Installation - Windows
install.packages("drat", repos="https://cran.rstudio.com")
drat:::addRepo("dmlc")
install.packages("mxnet")
library(mxnet)

#Installation - Linux
#Press Ctrl + Alt + T and run the following command
```

```
sudo apt-get update
sudo apt-get -y install git
git clone https://github.com/dmlc/mxnet.git ~/mxnet --recursive
cd ~/mxnet/setup-utils
bash install-mxnet-ubuntu-r.sh
```

In R, mxnet accepts target variables as numeric classes and not factors. Also, it accepts data frame as a matrix. Now, we'll make the required changes:

```
#load package
require(mxnet)

#convert target variables into numeric
train[,target := as.numeric(target)-1]
test[,target := as.numeric(target)-1]

#convert train data to matrix
train.x <- data.matrix(train[,-c("target"),with=F])
train.y <- train$target

#convert test data to matrix
test.x <- data.matrix(test[,-c("target"),with=F])
test.y <- test$target
```

Now, we'll train the multilayered perceptron model using the mx.mlp function.

```
#set seed to reproduce results
mx.set.seed(1)

mlpmodel <- mx.mlp(data = train.x
                   ,label = train.y
                   ,hidden_node = 3 #one layer with 10 nodes
                   ,out_node = 2
                   ,out_activation = "softmax" #softmax return probability
                   ,num.round = 100 #number of iterations over training data
                   ,array.batch.size = 20 #after every batch weights will get updated
                   ,learning.rate = 0.03 #same as step size
                   ,eval.metric= mx.metric.accuracy
                   ,eval.data = list(data = test.x, label = test.y))
```

Softmax function is used for binary and multi-classification problems. Alternatively, you can also manually craft the model structure.

```
#create NN structure
data <- mx.symbol.Variable("data")
fc1 <- mx.symbol.FullyConnected(data, num_hidden=3) #3 neuron in one layer
lrm <- mx.symbol.SoftmaxOutput(fc1)
```

We have configured the network above with one hidden layer carrying three neurons. We have chosen softmax as the output function. The network optimizes for squared loss for regression, and the network optimizes for classification accuracy for classification. Now, we'll train the network:

```
nnmodel <- mx.model.FeedForward.create(symbol = lrm
                                       ,X = train.x
                                       ,y = train.y
                                       ,ctx = mx.cpu()
                                       ,num.round = 100
                                       ,eval.metric = mx.metric.accuracy
                                       ,array.batch.size = 50
                                       ,learning.rate = 0.01)
```

Similarly, we can configure a more complex network fed with hidden layers.

```
#configure another network
data <- mx.symbol.Variable("data")
fc1 <- mx.symbol.FullyConnected(data, name = "fc1", num_hidden=10) #1st hidden layer
act1 <- mx.symbol.Activation(fc1, name = "sig", act_type="relu")
fc2 <- mx.symbol.FullyConnected(act1, name = "fc2", num_hidden=2) #2nd hidden layer
out <- mx.symbol.SoftmaxOutput(fc2, name = "soft")
```

Understand it carefully: After feeding the input through data, the first hidden layer consists of 10 neurons. The output of each neuron passes through a relu (rectified linear) activation function. We have used it in place of sigmoid. relu converges faster than a sigmoid function. You can read more about relu here.

Then, the output is fed into the second layer which is the output layer. Since our target variable has two classes, we've chosen num_hidden as 2 in the second layer. Finally, the output from second layer is made to pass though softmax output function.

```
#train the network
dp_model <- mx.model.FeedForward.create(symbol = out
                                        ,X = train.x
                                        ,y = train.y
                                        ,ctx = mx.cpu()
                                        ,num.round = 100
                                        ,eval.metric = mx.metric.accuracy
                                        ,array.batch.size = 50
                                        ,learning.rate = 0.005)
```

As mentioned above, this trained model predicts output probability, which can be easily transformed into a label using a threshold value (say, 0.5). To make predictions on the test set, we do this:

```
#predict on test
pred_dp <- predict(dp_model,test.x)
str(pred_dp)
#contains 2 rows and 16281 columns

#transpose the pred matrix
pred.val <- max.col(t(pred_dp))-1
```

The predicted matrix returns two rows and 16281 columns, each column carrying probability. Using the max.col function, we can extract the maximum value from each row. If you check the model's accuracy, you'll find that this network performs terribly on this data. In fact, it gives no better result than the train accuracy! On this data set, xgboost tuning gave 87% accuracy.

If you are familiar with the model building process, I'd suggest you to try working on the popular MNIST data set. You can find tons of tutorials on this data to get you going!

## Summary

Deep Learning is getting increasingly popular in solving most complex problems such as image recognition, natural language processing, etc. If you are aspiring for a career in machine learning, this is the best time for you to get into this subject. The motive of this article was to introduce you to the fundamental concepts of deep learning.

In this article, we learned about the basics of deep learning (perceptrons, neural networks, and multilayered neural networks). We learned deep learning as a technique is composed of several algorithms such as backpropagration and gradient descent to optimize the networks. In the end, we gained some hands-on experience in developing deep learning models.