Introduction to Deep Neural Networks

Esteban Vegas Ferran Reverter Alex Sanchez

2023-05-03

Table of contents

```r
options(width=100)
if(!require("knitr")) install.packages("knitr")
library("knitr")
#getOption("width")
knitr::opts_chunk$set(comment=NA,echo = TRUE, cache=TRUE)
```

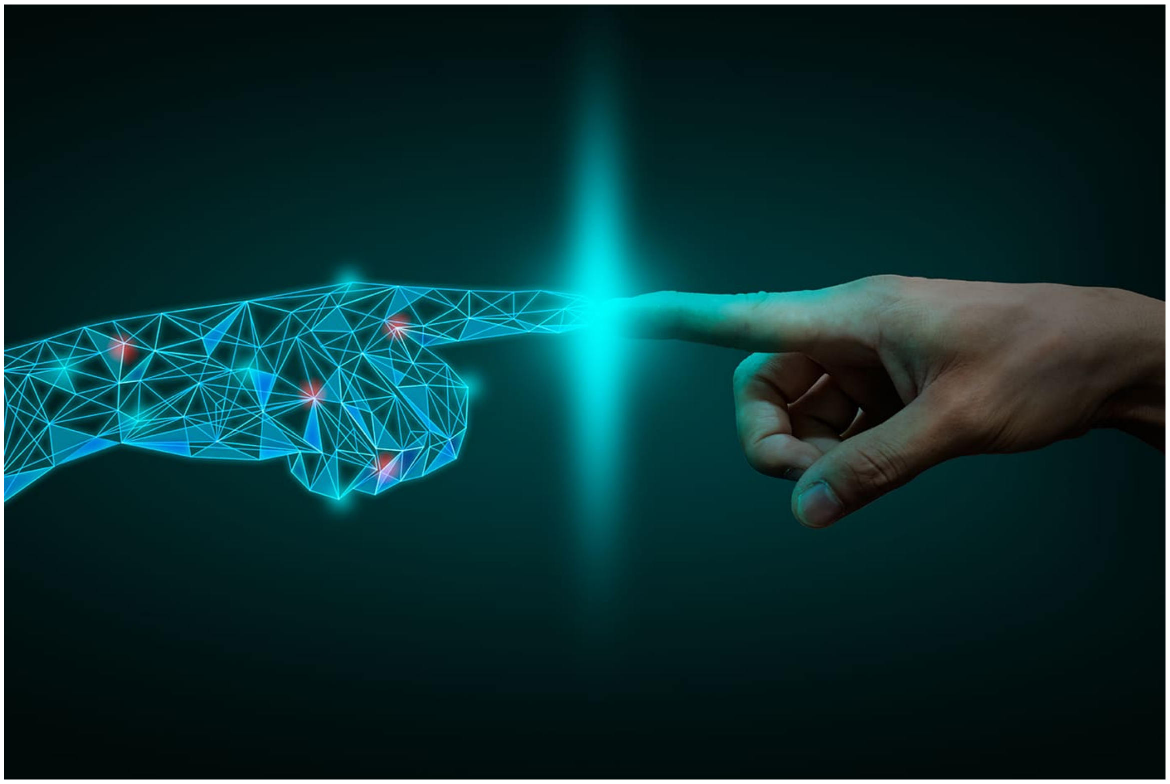
Introduction to Deep Neural Networks

Overview of Deep Learning

Historical Background and Key Milestones

Today, in April 2023, our world is convulsed by the explosion of Artificial Intelligence.

Although it has been growing steadily, it has probably been in the last months (weeks), since ChatGPT has arrived, that everybody has an opinion, or a fear on the topic.



While we are not going to discuss ethical, technological or sociological aspects, what seems clear to the data scientist's eye is that "*most of this is about prediction*".

Most AI engines rely on powerful prediction systems that use statistical learning methods.

Deep learning is a highly successful model in the field of AI, which has powered numerous applications in various domains. It has shown remarkable performance in tasks such as image recognition, natural language processing, and speech recognition.

Deep learning extends the basic principles of artificial neural networks by introducing more complex architectures and algorithms and, at the same time, by enabling machines to learn from large datasets by automatically identifying relevant patterns and features without explicit programming.

One key advantage of deep learning over traditional machine learning algorithms is its ability to handle high-dimensional and unstructured data such as images, videos, and audio.

The early history of artificial [neural networks]/intelligence

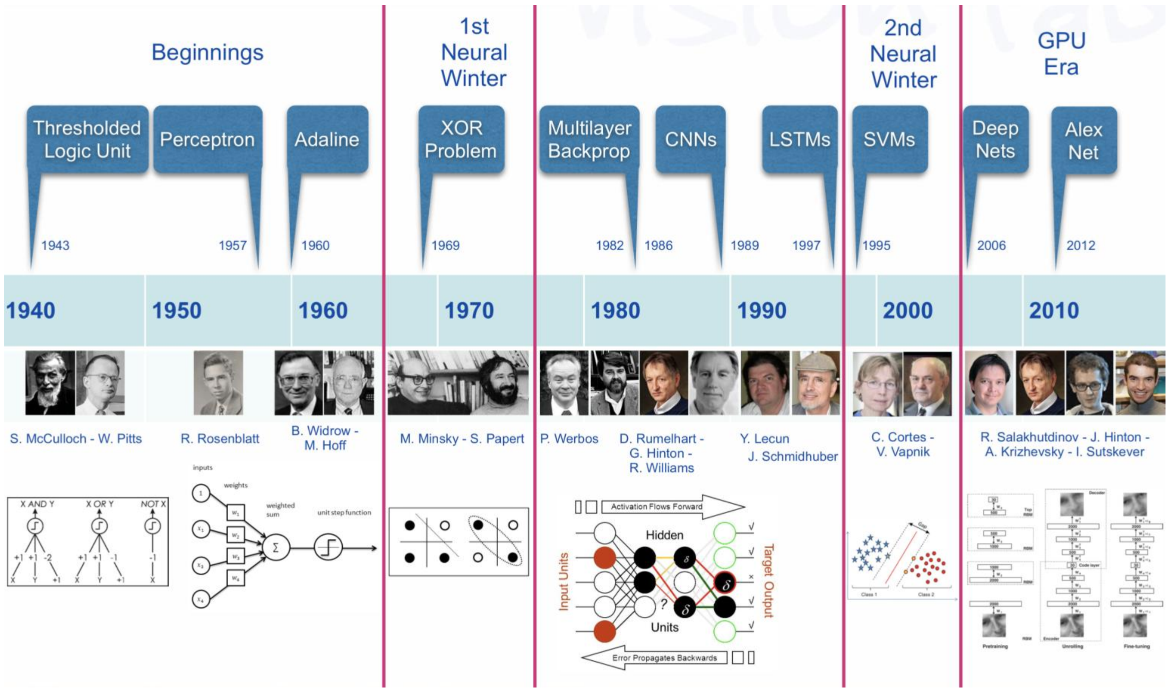


Figure 1: A Brief History of AI from 1940s till Today

The origins of AI, and as such of DL can be traced almost one century backwards. While it is an interesting, or even fascinating, history wee don't go into it (see a summary in A Quick History of AI, ML and DL

We can see there however, several hints worth to account for because we will go through them to understand how a deep neural network works. These are:

- The **Perceptron** and the first **Artificial Neural Network** where the basic building block was introduced.
- The **Multilayered perceptron** and **backpropagation** where complex architechtures were suggested to improve the capabilities.
- **Deep Neural Networks**, with many hidden layers, and auto-tunability capabilities.

In short, there has been an mathematical and a technological evolution that at some point has allowed to meet with

- The required theoretical background (DNN)

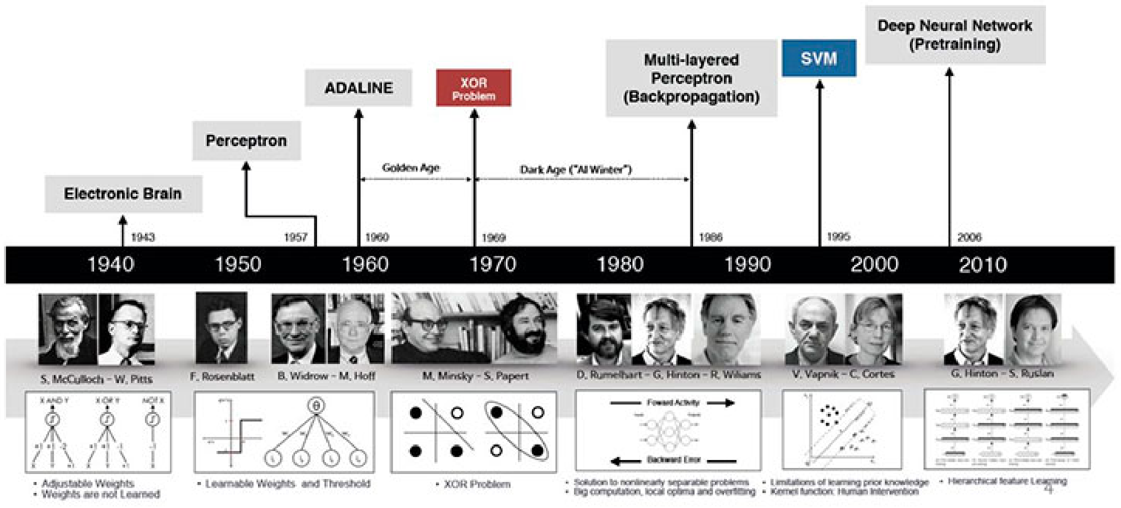
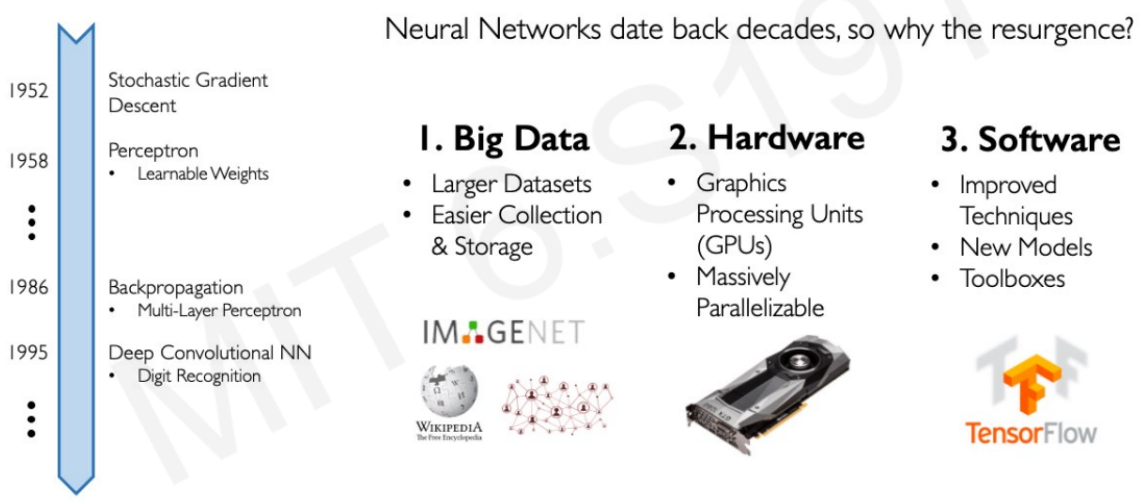

Electronic Brain
Perceptron
ADALINE
XOR Problem
Multi-layered Perceptron (Backpropagation)
SVM
Deep Neural Network (Pretraining)
Golden Age
Dark Age ("AI Winter")
1943
1957
1960
1969
1986
1995
2006
1940
1950
1960
1970
1980
1990
2000
2010
S. McCulloch - W. Pitts
F. Rosenblatt
B. Widrow - M. Hoff
M. Minsky - S. Papert
D. Rumelhart - G. Hinton - R. Wiliams
V. Vapnik - C. Cortes
G. Hinton - S. Ruslan
• Adjustable Weights
• Weights are not Learned
• Learnable Weights and Threshold
• XOR Problem
• Solution to nonlinearly separable problems
• Big computation, local optima and overfitting
• Limitations of learning prior knowledge
• Kernel function: Human Intervention
• Hierarchical feature Learning

Figure 2: The origins of Deep learning and modern Artificial Intelligence can be traced back to the per4ceptron. Source: "A Quick History of AI, ML and DL"

- The required computational capabilities (GPU, HPC)
- The required quantity of data (Big Data, Images, Social Networks)

This has resulted in making artificial intelligence widely accessible to businesses, researchers, and the general public.



Neural Networks date back decades, so why the resurgence?

1952	Stochastic Gradient Descent
1958	Perceptron • Learnable Weights
⋮	
1986	Backpropagation • Multi-Layer Perceptron
1995	Deep Convolutional NN • Digit Recognition
⋮	

1. Big Data
- Larger Datasets
- Easier Collection & Storage

2. Hardware
- Graphics Processing Units (GPUs)
- Massively Parallelizable

3. Software
- Improved Techniques
- New Models
- Toolboxes

Source: Alex Amini's 'MIT Introduction to Deep Learning' course (introtodeeplearning.com)

Success stories such as

- the development of self-driving cars,
- the use of AI in medical diagnosis, and
- the creation of personalized recommendations in online shopping

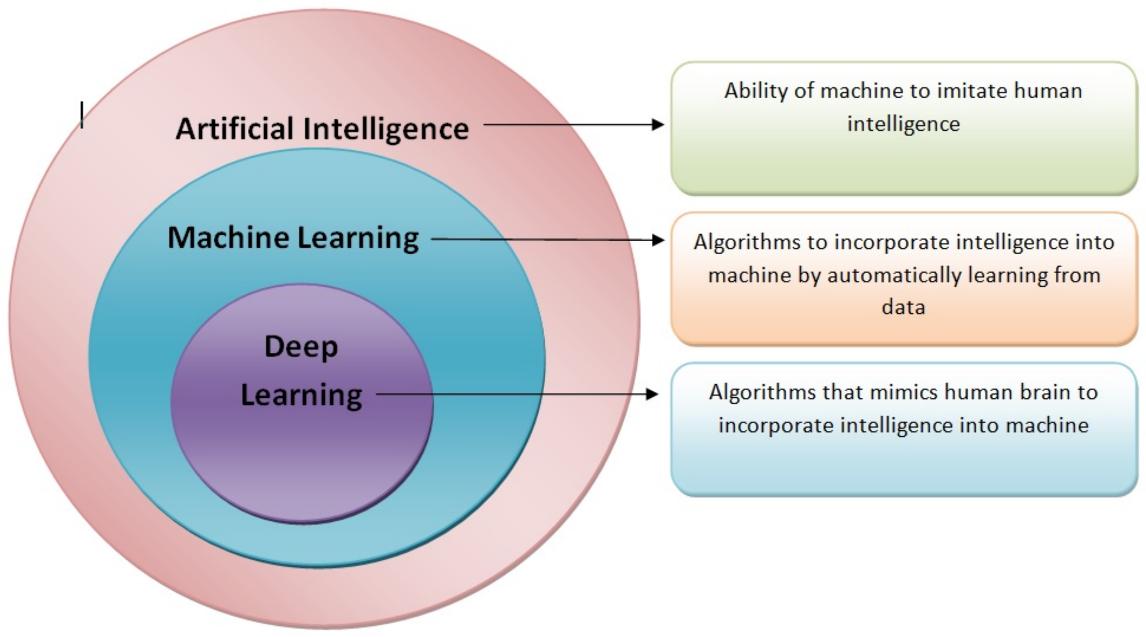
have also contributed to the widespread adoption of AI.

Comparison with Traditional Machine Learning

A reasonable question is "*how are ArtificiaI Intelligence, Machine Learning and Deep learning related*"?

A standard answer can be found in the image below that has a myriad variations:

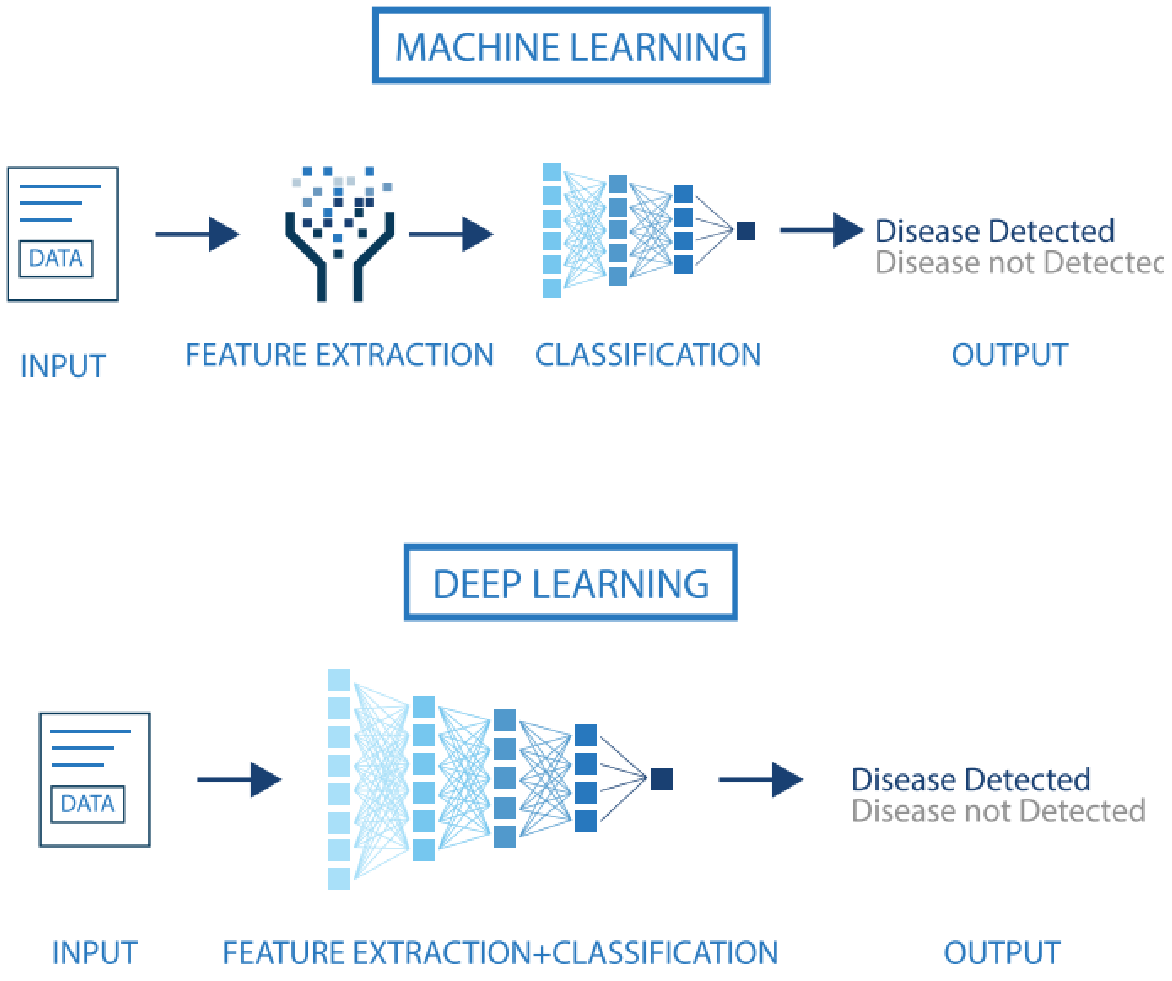
```r
knitr::include_graphics("images/AI-ML-DL-1.jpg")
```


Artificial Intelligence
Machine Learning
Deep Learning
Ability of machine to imitate human intelligence
Algorithms to incorporate intelligence into machine by automatically learning from data
Algorithms that mimics human brain to incorporate intelligence into machine

We can keep three definitions:

- Artificial intelligence is the ability of a computer to perform tasks commonly associated with intelligent beings.
- Machine learning is the study of algorithms that learn from examples and experience instead of relying on hard-coded rules and make predictions on new data
- Deep learning is a subfield of machine learning focusing on learning data representations as successive successive layers of increasingly meaningful representations.

```r
knitr::include_graphics("images/ML_vs_DL-2.png")
```



We will be coming back to the difference between ML and DL, but two strengths of DL that differentiate it from ML should be clear from the beginning:

- DNN combine feature extraction and classification in a way that does not require (or dramatically decreases) human intervention.
- The power of DNN requires in its ability to improve with data availability, without seemingly reaching plateaus as ML.

Deep learning is having a strong impact

- Near-human-level image classification
- Near-human-level speech transcription
- Near-human-level handwriting transcription

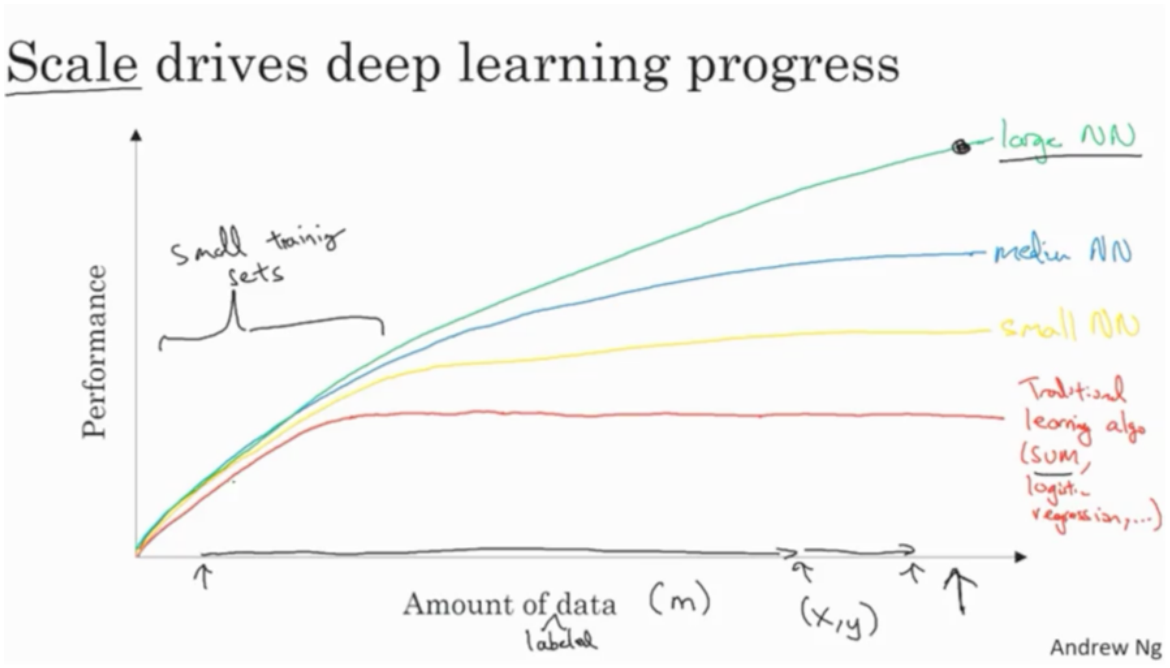


Figure 3: An illustration of the performance comparison between deep learning (DL) and other machine learning (ML) algorithms, where DL modeling from large amounts of data can increase the performance

- Dramatically improved machine translation
- Dramatically improved text-to-speech conversion
- Digital assistants such as Google Assistant and Amazon Alexa
- Near-human-level autonomous driving
- Improved ad targeting, as used by Google, Baidu, or Bing
- Improved search results on the web
- Ability to answer natural language questions
- Superhuman Go playing

According to [@chollet2022] … "*we shouldn't believe the short-term hype, but should believe in the long-term vision. It may take a while for AI to be deployed to its true potential—a potential the full extent of which no one has yet dared to dream—but AI is coming, and it will transform our world in a fantastic way*".

Once the introduction is ready we con move onto the building blocks of neural networks, perceptrons.

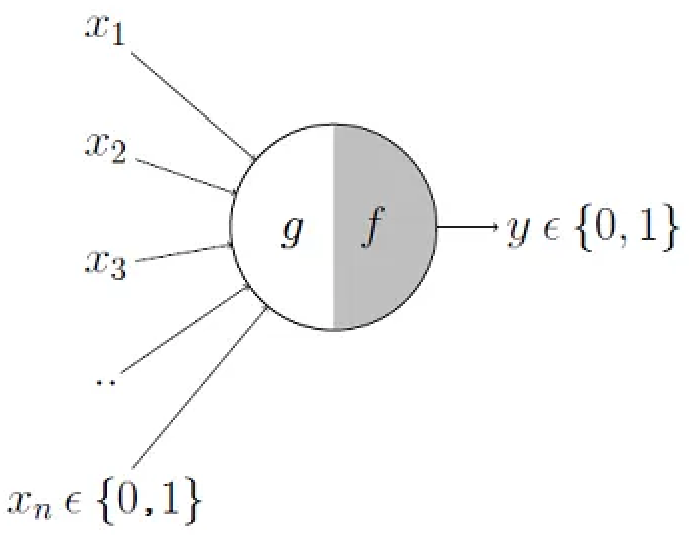
Artificial Neural Networks

The perceptron, the building block

The perceptron, was introduced by Rosenblatt (one version of the perceptron at least), as a mathematical model that might emulate a neuron.

The idea is: *how can we produce a model that given some inputs, and an appropriate set of examples, learn to produce the desired output*?

The first computational model of a neuron was proposed by Warren MuCulloch (neuroscientist) and Walter Pitts (logician) in 1943.



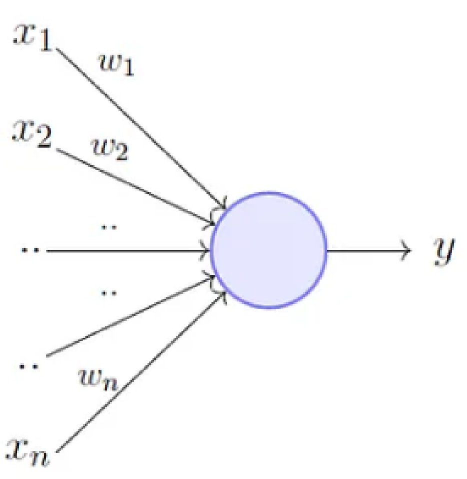
It may be divided into 2 parts. The first part, g,takes an input (ahem dendrite ahem), performs an aggregation and based on the aggregated value the second part, f, makes a decision. See the source of this picture for an illustration on how this can be used to emulate logical operations such as AND, OR or NOT, but not XOR.

This first attempt to emulate neurons succeeded but with limitations:

- What about non-boolean (say, real) inputs?
- What if all inputs are not equal?
- What if we want to assign more importance to some inputs?
- What about functions which are not linearly separable? Say XOR function

To overcome these limitations Frank Rosenblatt, an American psychologist, proposed the classical perception model, the *artificial neuron*, in 1958. It is more generalized computational model than the McCulloch-Pitts neuron where weights and thresholds can be learnt over time.

The perceptron proposed by Rosenblatt this is very similar to an M-P neuron but we take a weighted sum of the inputs and set the output as one only when the sum is more than an arbitrary threshold (***theta***).

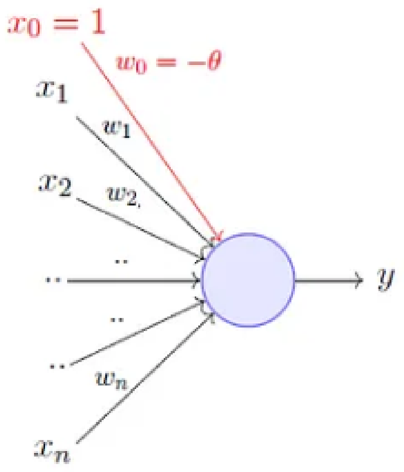


$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i \geq \theta$$
$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i < \theta$$

Rewriting the above,

$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i - \theta \geq 0$$
$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i - \theta < 0$$

Additionally, instead of hand coding the thresholding parameter θ, we add it as one of the inputs, with the weight $w_0 = -\theta$ like shown below, which makes it learnable.



A more accepted convention,

$$y = 1 \quad if \sum_{i=0}^{n} w_i * x_i \geq 0$$
$$= 0 \quad if \sum_{i=0}^{n} w_i * x_i < 0$$
$$where, \quad x_0 = 1 \quad and \quad w_0 = -\theta$$

Now, while this is an improvement (because both the weights and the threshold can be learned and the inputs can be real values) there is still a drawback in that a single perceptron can only be used to implement linearly separable functions.

Artificial Neural Networks improve on this by introducing *Activation Functions* which, eventually, can be non-linear.

McCulloch Pitts Neuron
(assuming no inhibitory inputs)

$$y = 1 \quad if \sum_{i=0}^{n} x_i \geq 0$$
$$= 0 \quad if \sum_{i=0}^{n} x_i < 0$$

Perceptron

$$y = 1 \quad if \sum_{i=0}^{n} w_i * x_i \geq 0$$
$$= 0 \quad if \sum_{i=0}^{n} w_i * x_i < 0$$

Neurons and Activation Functions

An activation function is a function that is added into an artificial neurone in order to help it learn complex patterns in the data.

How come biological and artificial neurons come to compare?

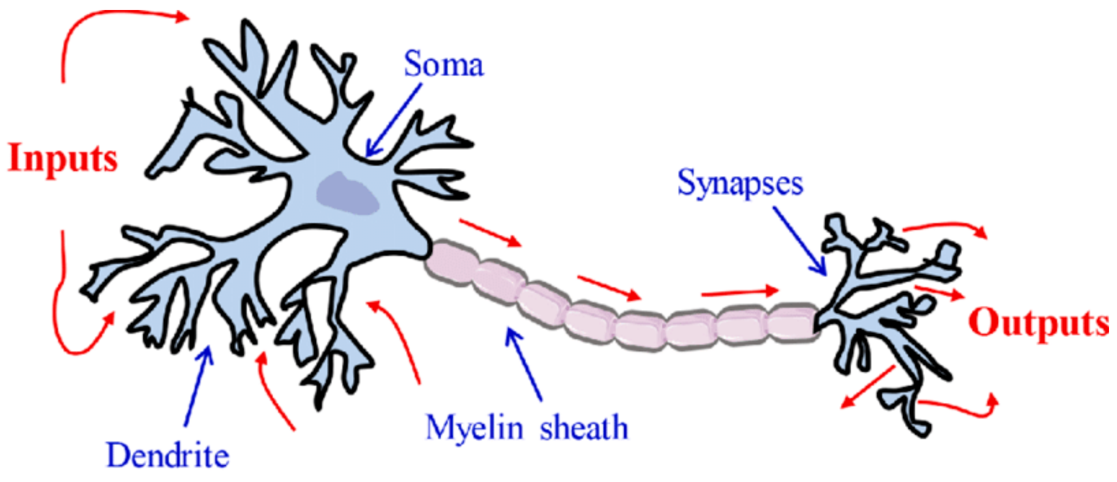
Biological neurons are specialized cells in the central nervous system that transmit electrical and chemical signals to communicate with each other and the rest of the body.

On the other hand, artificial neurons are mathematical models used in neural networks to process information.

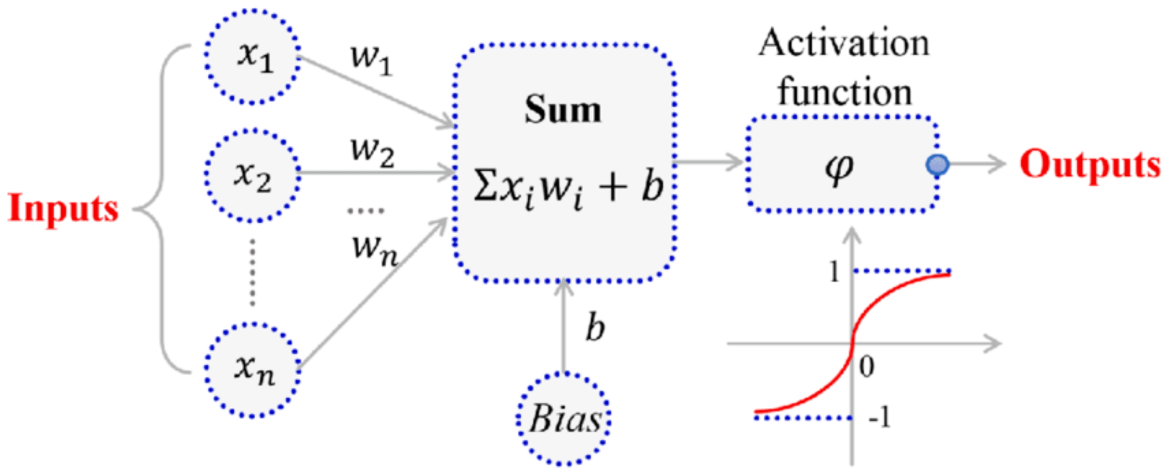
In both biological and artificial neurons, the **activation function** is what is responsible for *deciding whether the neuron activates or not based on the input it receives.*

- In the case of a biological neuron, the activation function is based on the release of neurotransmitters, which are chemical substances that transmit signals between nerve cells. When the electrical signal reaching the neuron exceeds a certain threshold, the neuron releases neurotransmitters, which are received by other neurons or cells in the body to continue the communication process.
- On the other hand, in an artificial neuron, the activation function is a mathematical function applied to the neuron's input to produce an output. Like in the biological neuron, this activation function decides whether the neuron activates or not based on the input it receives.

```r
knitr::include_graphics("images/ActivationFunction0.png")
```



(a) Biological neuron



(b) Artificial neuron

Read more here about activation functions.

With all these inputs in mind we can now define an Artificial Neuron as a *computational unit* that - takes as input $x = (x_0, x_1, x_2, x_3)$ ($x_0 = +1$, called bias), and - outputs $h_\theta(x) = f(\theta^\top x) = f(\sum_i \theta_i x_i)$, - where $f : \mathbb{R} \mapsto \mathbb{R}$ is called the **activation function**.

The goal of the activation function is to provide the Neuron with *the capability of producing the required outputs*.

For instance, if the output has to be a probability, the activation function will only produce values between 0 and 1.

With this idea in mind activation functions are chosen from a set of pre-defined functions:

- the sigmoid function:

$$f(z) = \frac{1}{1+e^{-z}}$$

- the hyperbolic tangent, or `tanh`, function:

$$f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

The `tanh(z)` function is a rescaled version of the sigmoid, and its output range is $[-1, 1]$ instead of $[0, 1]$.

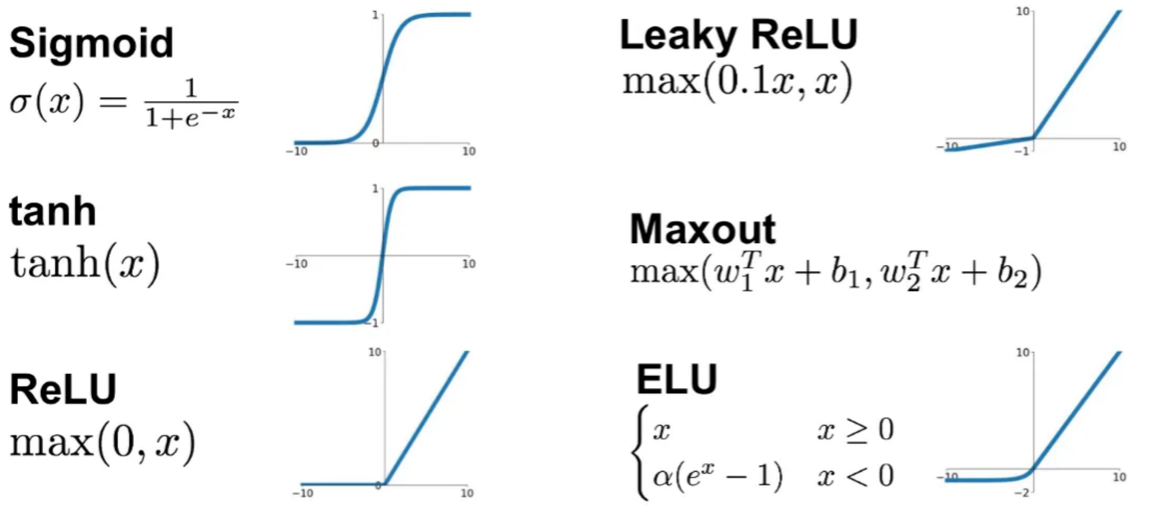
Two useful properties to recall are that: - *If* $f(z) = 1/(1+e^z)$ is the sigmoid function, then its derivative is given by $f'(z) = f(z)(1 - f(z))$.

- *Similarly, if* f is the `tanh` function, then its derivative is given by $f'(z) = 1 - (f(z))^2$.
- In modern neural networks, the default recommendation is to use the *rectified linear unit* or ReLU defined by the activation function $f(z) = \max\{0, z\}$.

This function remains very close to a linear one, in the sense that is a piecewise linear function with two linear pieces.

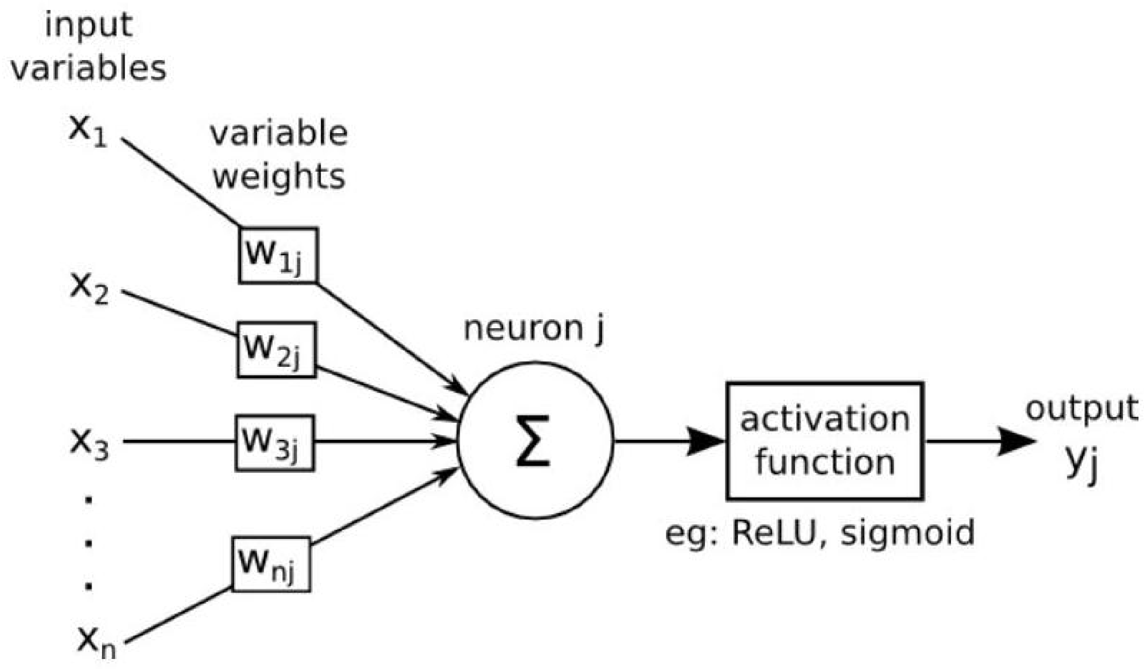
Because rectified linear units are nearly linear, they preserve many of the properties that make linear models easy to optimize with gradient based methods.

They also preserve many of the properties that make linear models generalize well.



.

Putting alltogether we have the following schematic representation of an artificial neuron where $\Sigma = \langle w_j, x \rangle + b_j$ and $\langle w_j, x \rangle$ represents the dot product between vectors w and x.



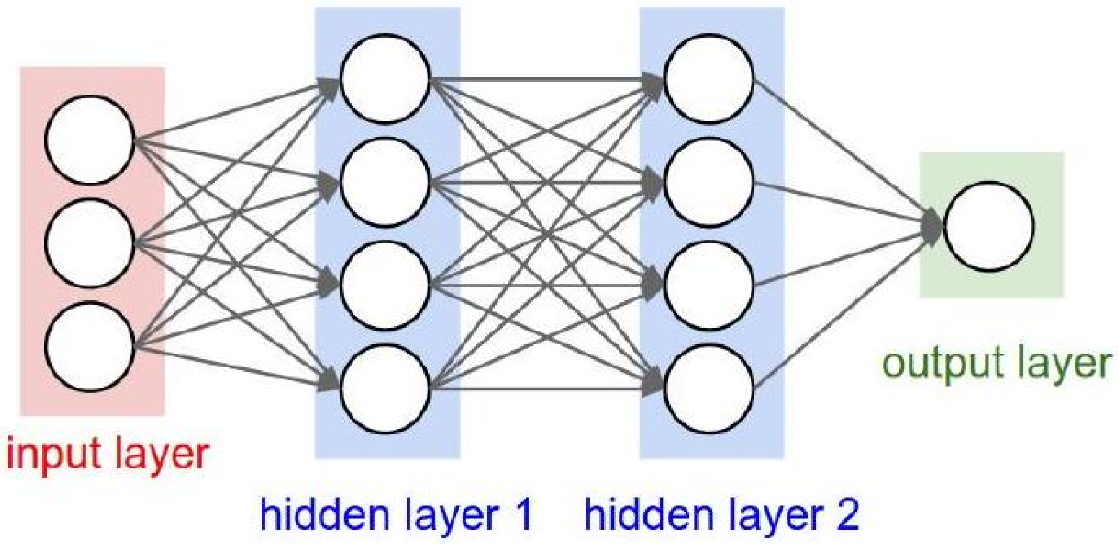
Multilayer perceptrons

A multilayer perceptron (or Artificial neural network) is a structure composed by *several hidden layers of neurons* where the output of a neuron of a layer becomes the input of a neuron of the next layer.

Moreover, the output of a neuron can also be the input of a neuron of the same layer or of neuron of previous layers (this is the case for recurrent neural networks). On last layer, called output layer, we may apply a different activation function as for the hidden layers depending on the type of problems we have at hand : regression or classification.

The Figure below represents a neural network with three input variables, one output variable, and two hidden layers.

```r
knitr::include_graphics("images/MultiLayer1.png")
```



Multilayers perceptrons have a basic architecture since each unit (or neuron) of a layer is linked to all the units of the next layer but has no link with the neurons of the same layer.

The parameters of the architecture are:

- the number of hidden layers and
- the number of neurons in each layer.

The activation functions are also to choose by the user. For the output layer, as mentioned previously, the activation function is generally different from the one used on the hidden layers. For example:.

- For regression, we apply no activation function on the output layer.
- For binary classification, the output gives a prediction of $\mathbb{P}(Y = 1/X)$ since this value is in $[0, 1]$ and the sigmoid activation function is generally considered.
- For multi-class classification, the output layer contains one neuron per class (i), giving a prediction of $\mathbb{P}(Y = i/X)$. The sum of all these values has to be equal to 1. The sum of all these values has to be equal to 1.
    - A common choice for multi-class ANN is the *softmax* activation function:

$$\mathrm{softmax}(z)_i = \frac{\exp\left(z_i\right)}{\sum_j \exp\left(z_j\right)}$$

An example

In this example we train and use a “shallow neural network”, called this way in contrast with “deep neural networks”.

We will use the neuralnet R package, which is not intended to work with deep neural networks, to build a simple neural network to predict if a type of stock pays dividends or not.

```
if (!require(neuralnet))
  install.packages("neuralnet", dep=TRUE)
if (!require(caret))
  install.packages("caret", dep=TRUE)
```

The data for the example are the dividendinfo.csv dataset, available from: https://github.com/MGCodesandStats/datasets

```
mydata <- read.csv("https://raw.githubusercontent.com/MGCodesandStats/datasets/master/divi
str(mydata)
```

```
'data.frame':   200 obs. of  6 variables:
 $ dividend       : int  0 1 1 0 1 1 1 0 1 1 ...
 $ fcfps          : num  2.75 4.96 2.78 0.43 2.94 3.9 1.09 2.32 2.5 4.46 ...
 $ earnings_growth: num  -19.25 0.83 1.09 12.97 2.44 ...
 $ de             : num  1.11 1.09 0.19 1.7 1.83 0.46 2.32 3.34 3.15 3.33 ...
 $ mcap           : int  545 630 562 388 684 621 656 351 658 330 ...
 $ current_ratio  : num  0.924 1.469 1.976 1.942 2.487 ...
```

Data preprocessing

One of the most important procedures when forming a neural network is data normalization. This involves adjusting the data to a common scale so as to accurately compare predicted and actual values. Failure to normalize the data will typically result in the prediction value remaining the same across all observations, regardless of the input values.

We can do this in two ways in R:

- Scale the data frame automatically using the scale function in R
- Transform the data using a max-min normalization technique

In this example We implement the max-min normalization technique.

See this link for further details on how to use the normalization function.

```
normalize <- function(x) {
  return ((x - min(x)) / (max(x) - min(x)))
}
normData <- as.data.frame(lapply(mydata, normalize))
```

As usually, the dataset is separated in a training and a test set. The training set contains a random selection with and (arbitrary) 66% of the observations.

```
perc2Train <- 2/3
ssize <- nrow(normData)
set.seed(12345)
data_rows <- floor(perc2Train *ssize)
train_indices <- sample(c(1:ssize), data_rows)
trainset <- normData[train_indices,]
testset <- normData[-train_indices,]
```

The `trainset` set will be used to train the network and the `testset` set one will be used to evaluate it.

Training a neural network

Setting the parameters of a neural network requires experience and understanding of their meaning, and even so, canges in the parameters can lead to similar results.

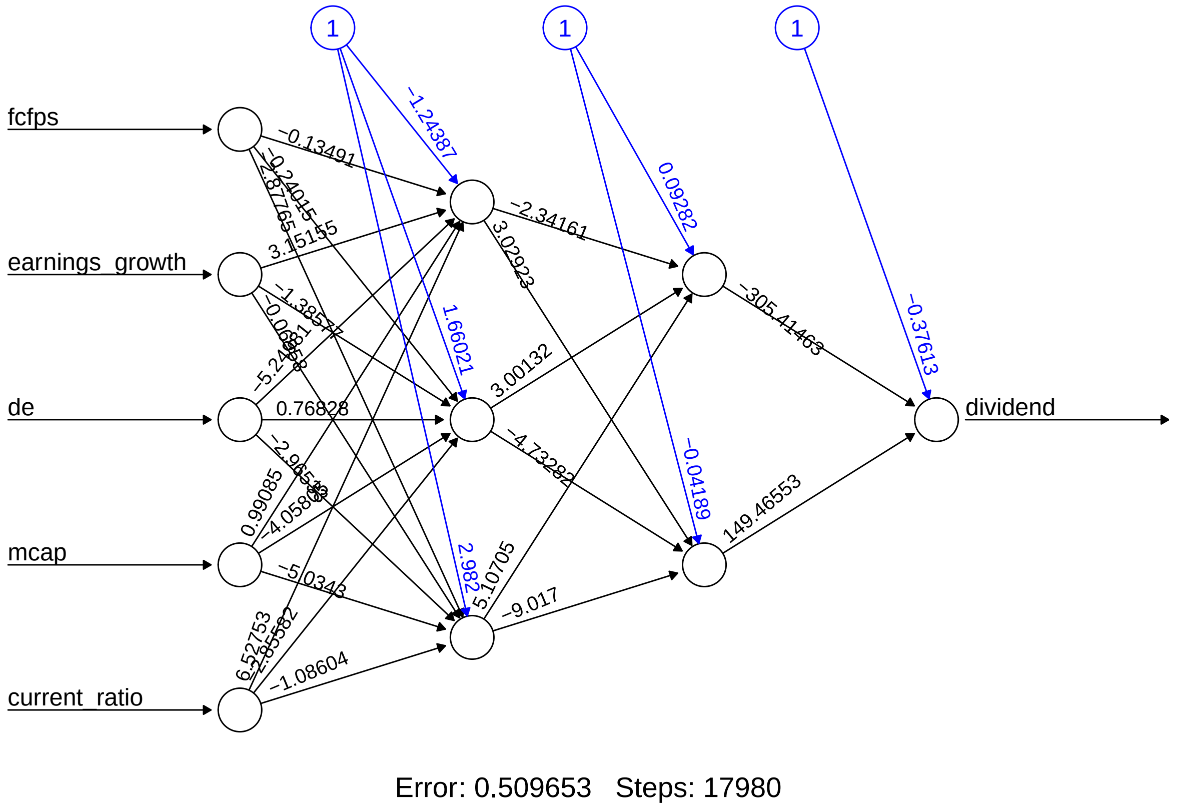
We create a simple NN with two hidden layers, with 3 and 2 neurons respectively. This is specified in the `hidden` parameter. For other parameters see the package help.

```
# Neural Network
library(neuralnet)
nn <- neuralnet(dividend ~ fcfps + earnings_growth + de + mcap + current_ratio,
                data=trainset,
                hidden=c(3,2),
                linear.output=FALSE,
                threshold=0.01)
```

The output of the procedure is a neural network with estimated weights.

This can be seen with a `plot` function (including the `rep` argument).

```
plot(nn, rep = "best")
```



The object `nn`contains information the weights and the results although it is not particularly clear or useful.

```r
summary(nn)
```

```
                      Length Class      Mode    
call                    6    -none-     call    
response              133    -none-     numeric 
covariate             665    -none-     numeric 
model.list              2    -none-     list    
err.fct                 1    -none-     function
act.fct                 1    -none-     function
linear.output           1    -none-     logical 
data                    6    data.frame list    
exclude                 0    -none-     NULL    
net.result              1    -none-     list    
weights                 1    -none-     list    
generalized.weights     1    -none-     list    
startweights            1    -none-     list    
```

```
result.matrix        32    -none-     numeric
```

```r
nn$result.matrix
```

```
                                     [,1]
error                        5.096531e-01
reached.threshold            9.874263e-03
steps                        1.798000e+04
Intercept.to.1layhid1       -1.243872e+00
fcfps.to.1layhid1           -1.349137e-01
earnings_growth.to.1layhid1  3.151554e+00
de.to.1layhid1              -5.249806e+00
mcap.to.1layhid1             9.908495e-01
current_ratio.to.1layhid1    6.527535e+00
Intercept.to.1layhid2        1.660208e+00
fcfps.to.1layhid2           -2.401517e-01
earnings_growth.to.1layhid2 -1.385771e+00
de.to.1layhid2               7.682849e-01
mcap.to.1layhid2            -4.058053e+00
current_ratio.to.1layhid2   -2.855816e+00
Intercept.to.1layhid3        2.982002e+00
fcfps.to.1layhid3           -2.877651e+00
earnings_growth.to.1layhid3 -6.957763e-02
de.to.1layhid3              -2.965334e+00
mcap.to.1layhid3            -5.034300e+00
current_ratio.to.1layhid3   -1.086037e+00
Intercept.to.2layhid1        9.282087e-02
1layhid1.to.2layhid1        -2.341614e+00
1layhid2.to.2layhid1         3.001315e+00
1layhid3.to.2layhid1         5.107051e+00
Intercept.to.2layhid2       -4.188729e-02
1layhid1.to.2layhid2         3.029232e+00
1layhid2.to.2layhid2        -4.732821e+00
1layhid3.to.2layhid2        -9.017001e+00
Intercept.to.dividend       -3.761263e-01
2layhid1.to.dividend        -3.054146e+02
2layhid2.to.dividend         1.494655e+02
```

Model evaluation

A prediction for each value in the `testset` dataset can be built with the `compute` function.

```r
#Test the resulting output
temp_test <- subset(testset, select =
                      c("fcfps","earnings_growth",
                        "de", "mcap", "current_ratio"))
head(temp_test)
```

```
       fcfps earnings_growth        de       mcap current_ratio
9  0.4929006      0.52417860 0.7862595 0.79741379   0.662994637
19 0.8722110      0.89705139 0.5190840 0.31465517   0.631284474
22 0.0811359      0.68272957 0.4554707 0.05747126   0.000785556
26 0.4077079      0.07649537 0.6310433 0.70977011   0.379642293
27 0.4279919      0.70362258 0.1882952 0.30603448   0.628283435
29 0.3509128      0.74203875 0.6030534 0.53017241   0.543404499
```

```r
nn.results <- compute(nn, temp_test)
results <- data.frame(actual =
                  testset$dividend,
                  prediction = nn.results$net.result)
head(results)
```

```
   actual    prediction
9       1  1.000000e+00
19      1  1.000000e+00
22      0 5.442517e-133
26      0  6.801894e-35
27      1  4.548179e-10
29      1  1.000000e+00
```

A confusion matrix can be built to evaluate the predictive ability of the network:

```r
roundedresults<-sapply(results,round,digits=0)
roundedresultsdf=data.frame(roundedresults)
attach(roundedresultsdf)
confMat<- caret::confusionMatrix(table(actual, prediction))
confMat
```

```
Confusion Matrix and Statistics

      prediction
actual  0  1
     0 34  2
     1  6 25
```

```
               Accuracy : 0.8806
                 95% CI : (0.7782, 0.947)
    No Information Rate : 0.597
    P-Value [Acc > NIR] : 3.405e-07

                  Kappa : 0.7577

 Mcnemar's Test P-Value : 0.2888

            Sensitivity : 0.8500
            Specificity : 0.9259
         Pos Pred Value : 0.9444
         Neg Pred Value : 0.8065
             Prevalence : 0.5970
         Detection Rate : 0.5075
   Detection Prevalence : 0.5373
      Balanced Accuracy : 0.8880

       'Positive' Class : 0
```

Some mathematics behind ANN

Mathematical formulation

- An ANN is a predictive model whose functioning and properties can be mathematically characterized.
- In practice this means describing
  - The ANN as a (non linear) function.
  - The (optimization) process for estimating the weights.
- Estimation is done by minimizing an appropriate loss function which requires
  - An appropriate procedure: *gradient based optimization*
  - A method for computing the required derivatives: *back-propagation*.

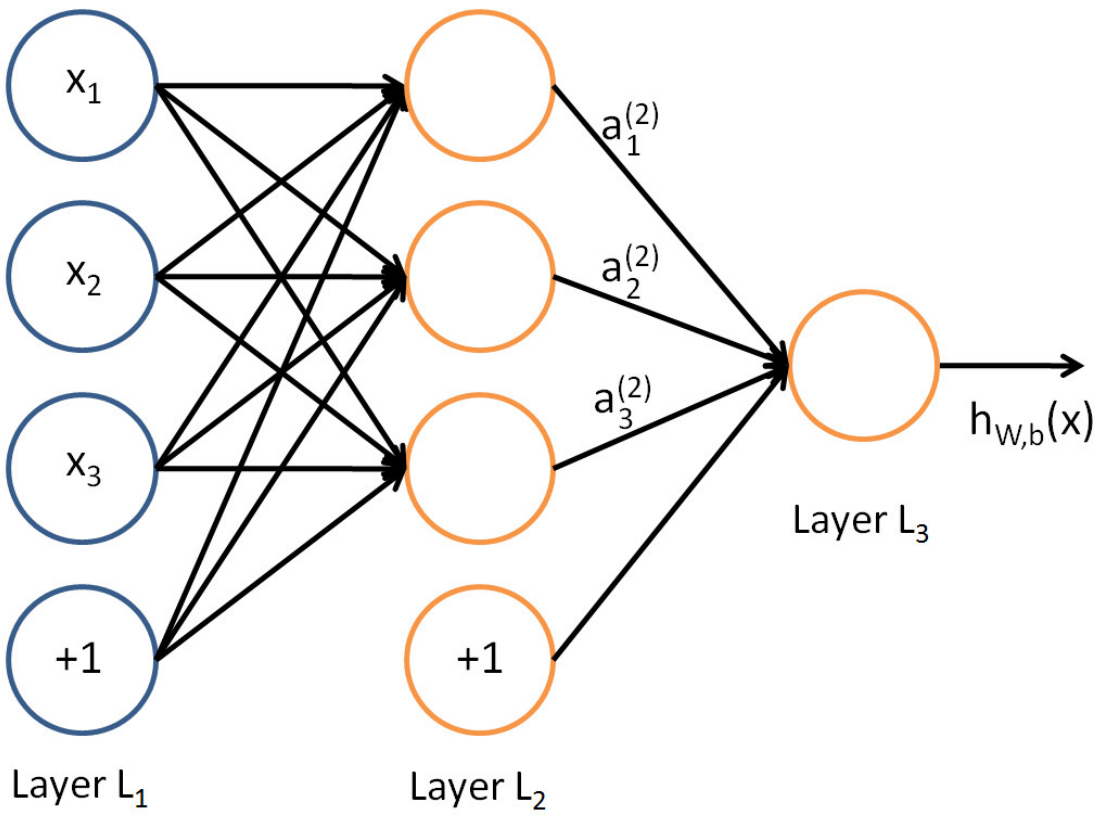
Mathematical formulation of the ANN

We will use a concrete model to explain the concepts, which can be easily generalized to more neurons and layers.

Consider the following simple ANN:

Consider the following ANN:

```r
knitr::include_graphics("images/nn.jpg")
```



- The circles labeled +1 are called bias units, and correspond to the intercept, here named *bias* term.
- The leftmost layer of the network is called the *input layer*.
- The rightmost layer is the *output* layer (which, in this example, has only one node).
- The middle layer(s) is(are) called the *hidden layer(s)*, because its values are not observed in the training set.

So our example network has:

- The input layer with 3 input units (not counting the bias unit),
- 1 hidden layer with 3 hidden units, and
- The output layer with 1 output unit.

A logistic regression ANN

This ANN can be use to build a logistic regression model:

- From input layer to layer 2: non-linear transformation –> new set of complex features.
- From layer 2 to output layer use a sigmoid activation function to produce the following output from the set of *complex features*.

$$\text{The output is: } h_\theta(x) = \frac{1}{1+e^{-\theta^\top x}}$$

Recall that, the logistic regression model is:

$$\log \frac{p(Y=1|x)}{1-p(Y=1|x)} = \theta^\top x$$

Isolating $p(Y=1|x)$ and taking logs in both sides, we have:

$$\frac{p(Y=1|x)}{1-p(Y=1|x)} = e^{\theta^\top x}$$

Thus

$$p(Y=1|x) = \frac{e^{\theta^\top x}}{1+e^{\theta^\top x}} = \frac{1}{1+e^{-\theta^\top x}}$$

That is, *when the activation function of the output node is the sigmoid activation function, the output coincides with a logistic regression on complex features*

- And, with $h_\theta(x)$, the output of the NN, we are estimating $p(Y=1|x)$.

Parametrizing an ANN

- Let n_l denote the number of layers in our network, thus $n_l = 3$ in our example.
- Label layer l as L_l, so layer L_1 is the input layer, and layer $L_{n_l} = L_3$ the output layer.
- Our neural network has parameters: $\Theta = (\Theta^{(1)}, \Theta^{(2)})$, where we will write $\theta_{ij}^{(l)}$ to denote the parameter (or weight) associated with the connection between unit j in layer l, and unit i in layer $l+1$.
- Thus, in our example, we have:
  - $\Theta^{(1)} \in \mathbb{R}^{3\times 4}$, and
  - $\Theta^{(2)} \in \mathbb{R}^{1\times 4}$.

Note that bias units don't have inputs or connections going into them, since they always output the value +1.

We also let s_l denote the number of nodes in layer l (not counting the bias unit).

Now, write $a_i^{(l)}$ to denote the activation (meaning output value) of unit i in layer l.

For $l = 1$, we also use $a_i^{(1)} = x_i$ to denote the i-th input.

Given a fixed setting of the parameters Θ, our neural network defines a model $h_\Theta(x)$ that outputs a real number.

We can now see *how these weights are used to produce the output*:

given by:

$$\begin{aligned}
a_1^{(2)} &= f(\theta_{10}^{(1)} + \theta_{11}^{(1)} x_1 + \theta_{12}^{(1)} x_2 + \theta_{13}^{(1)} x_3) & (1)\\
a_2^{(2)} &= f(\theta_{20}^{(1)} + \theta_{21}^{(1)} x_1 + \theta_{22}^{(1)} x_2 + \theta_{23}^{(1)} x_3) & (2)\\
a_3^{(2)} &= f(\theta_{30}^{(1)} + \theta_{31}^{(1)} x_1 + \theta_{32}^{(1)} x_2 + \theta_{33}^{(1)} x_3) & (3)\\
h_\Theta(x) &= a_1^{(3)} = f(\theta_{10}^{(2)} + \theta_{11}^{(2)} a_1^{(2)} + \theta_{12}^{(2)} a_2^{(2)} + \theta_{13}^{(2)} a_3^{(2)}) & (4)
\end{aligned}$$

Now, letting $z_i^{(l)}$ denote the total weighted sum of inputs to unit i in layer l, including the bias term

$$z_i^{(2)} = \theta_{i0}^{(1)} + \theta_{i1}^{(1)} x_1 + \theta_{i2}^{(1)} x_2 + \theta_{i3}^{(1)} x_3,$$

the output becomes: $a_i^{(l)} = f(z_i^{(l)})$.

Compacting notation

- Note that this easily lends itself to a more compact notation.
- Extending the activation function $f(\cdot)$ to apply to vectors in an elementwise fashion:

$$f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)],$$

then we can write the previous equations more compactly as:

$$\begin{aligned}
z^{(2)} &= \Theta^{(1)} x\\
a^{(2)} &= f(z^{(2)})\\
z^{(3)} &= \Theta^{(2)} a^{(2)}\\
h_\Theta(x) &= a^{(3)} = f(z^{(3)})
\end{aligned}$$

- More generally, recalling that we also use $a^{(1)} = x$ to also denote the values from the input layer,

- then given layer l's activations $a^{(l)}$, we can compute layer $l+1$'s activations $a^{(l+1)}$ as:

$$\begin{aligned} z^{(l+1)} &= \Theta^{(l)} a^{(l)} \qquad (5) \\ a^{(l+1)} &= f(z^{(l+1)}) \qquad (6) \end{aligned}$$

This can be used to provide a matrix representation for the weighted sum of inputs of all neurons:

$$z^{(l+1)} = \begin{bmatrix} z_1^{(l+1)} \\ z_2^{(l+1)} \\ \vdots \\ z_{s_{l+1}}^{(l)} \end{bmatrix} = \begin{bmatrix} \theta_{10}^{(l)} & \theta_{11}^{(l)} & \theta_{12}^{(l)} & \dots & \theta_{1s_l}^{(l)} \\ \theta_{20}^{(l)} & \theta_{21}^{(l)} & \theta_{22}^{(l)} & \dots & \theta_{2s_l}^{(l)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \theta_{s_{l+1}0}^{(l)} & \theta_{s_{l+1}1}^{(l)} & \theta_{s_{l+1}2}^{(l)} & \dots & \theta_{s_{l+1}s_l}^{(l)} \end{bmatrix} \cdot \begin{bmatrix} 1 \\ a_1^{(l)} \\ a_2^{(l)} \\ \vdots \\ a_{s_l}^{(l)} \end{bmatrix}$$

So that, the activation is then:

$$a^{(l+1)} = \begin{bmatrix} a_1^{(l+1)} \\ a_2^{(l+1)} \\ \vdots \\ a_{s_{l+1}}^{(l)} \end{bmatrix} = f(z^{(l+1)}) = \begin{bmatrix} f(z_1^{(l+1)}) \\ f(z_2^{(l+1)}) \\ \vdots \\ f(z_{s_{l+1}}^{(l)}) \end{bmatrix}$$

References and resources