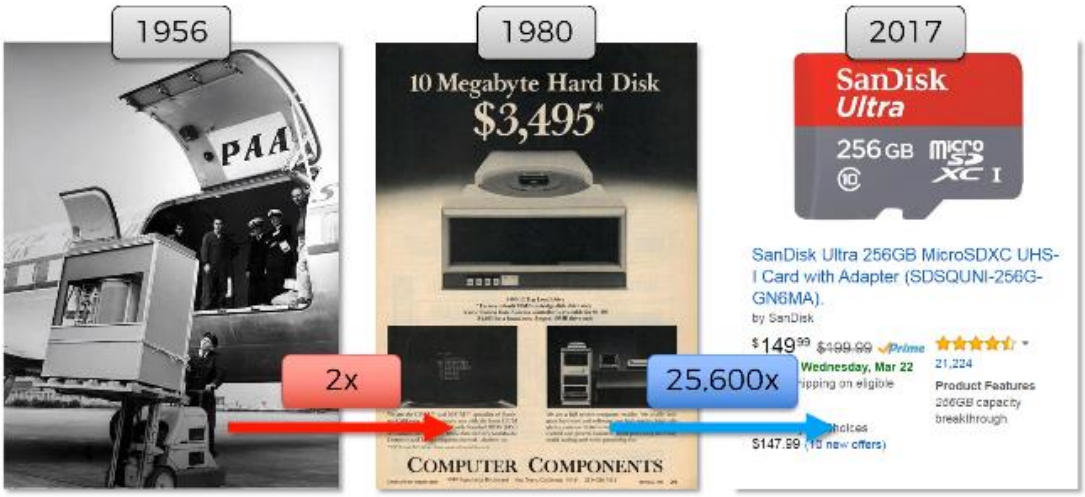
Deep Learning
Artificial Neural Networks

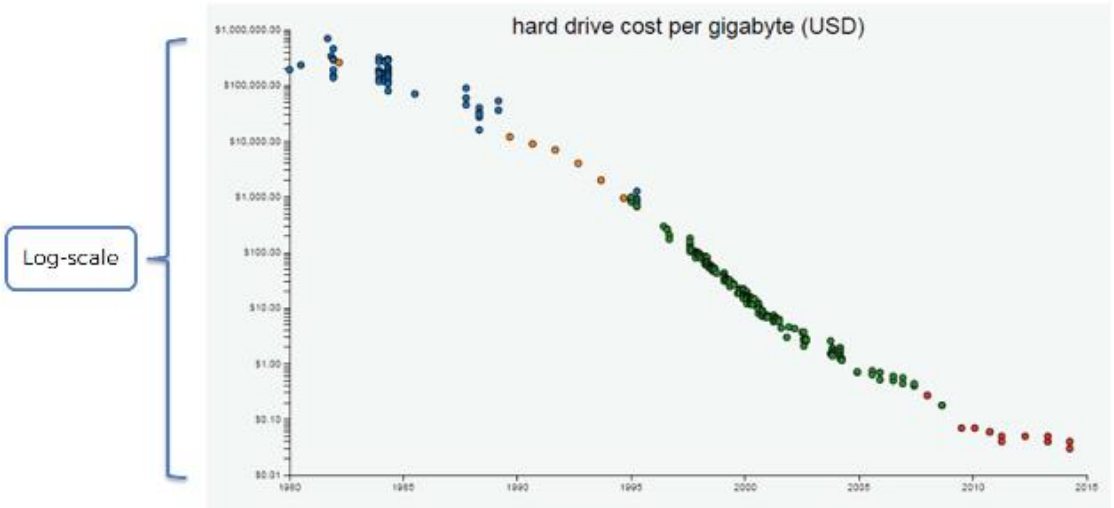
1. What is Deep Learning?

Neuronal networks and deep learning were invented in the 1960s, but did not begin to have a major impact until a few years ago. Why?

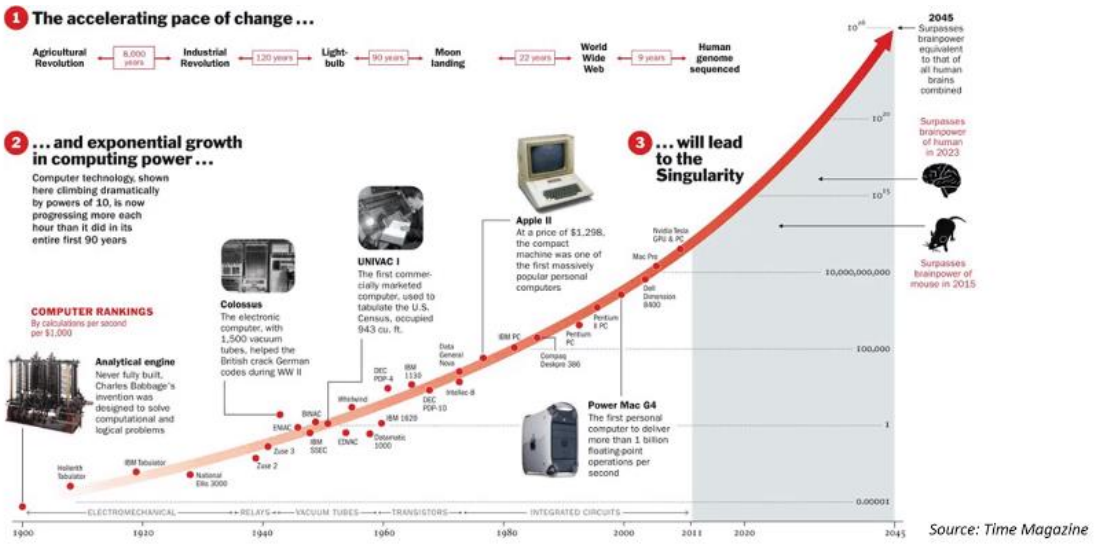
The reason of the lack of success for neuronal networks is the fact that technology at that time was not in the right standard. Basically, neuronal networks need two things => a lot of data and processing power. That is, you need solid computers to process that data and provide it.

How has technology evolved over time?





The cost of the hard drive has been reduced, while the power has increased.



So, **what is Deep Learning?**

Geoffrey Hinton is known as the father of the Deep Learning.

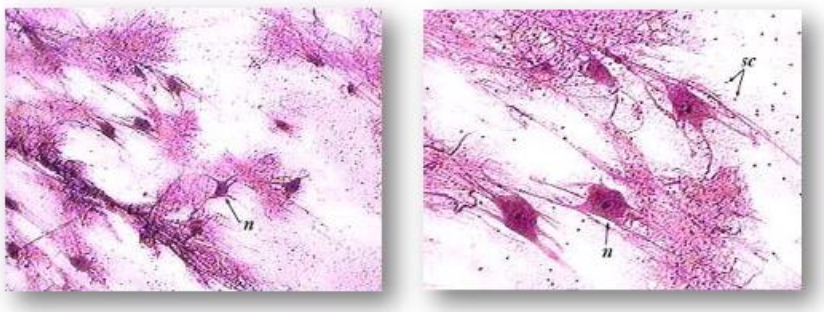
The concept of Deep Learning is to check how the human brain operates in order to mimic it and recreate it.

Human brain seems to be one of the most powerful tools on the planet for learning adapting skills and then applying them and if computers could copy that then we could just leverage what natural selection has already decided for us.

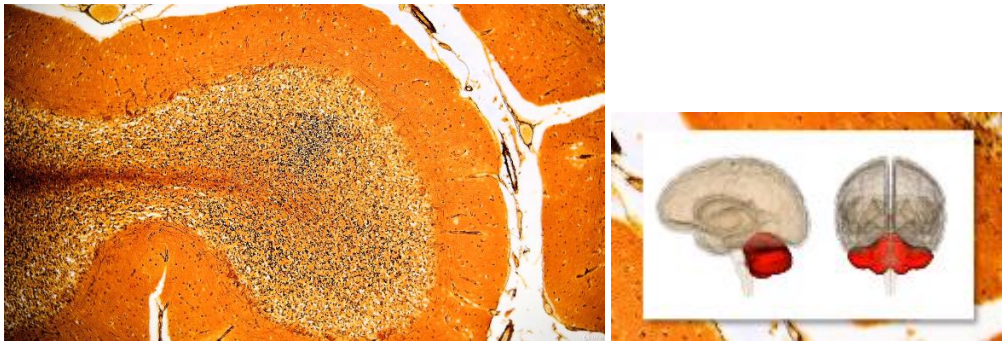
Let's check how it works:

The human brain has 100 billons of neurons. Each neuron is connected to hounds of neurons through neighbours.

In the following picture, you can see how a neuron looks like:



This is the cerebellum, which is this part of your brain at the back:

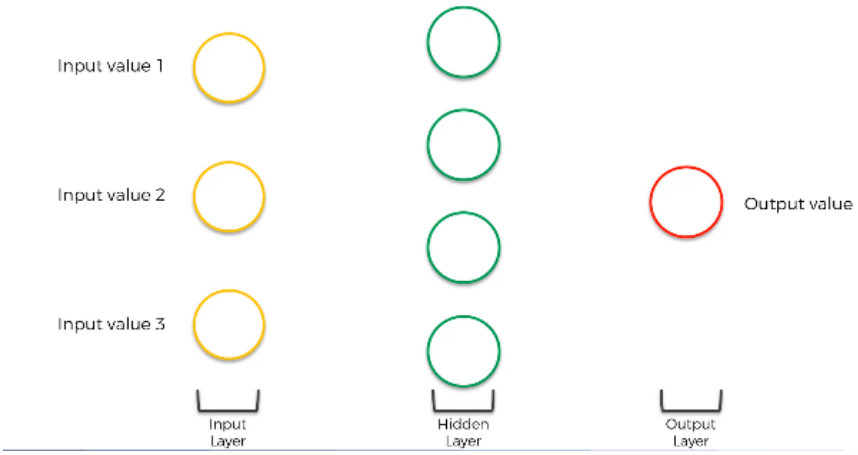


The cerebellum is responsible for your know keeping a balance and some language capabilities. This picture is interesting to show you how many neurons are like billions and billions all connected in the brain. So, we are going to try to recreate this picture in a computer. But, how?

In order to do it, we have to create an artificial structure called an **artificial neural network** where we have nodes or neurons and we are going to have some neurons for input value, so these are values that you that you know about a certain situation.

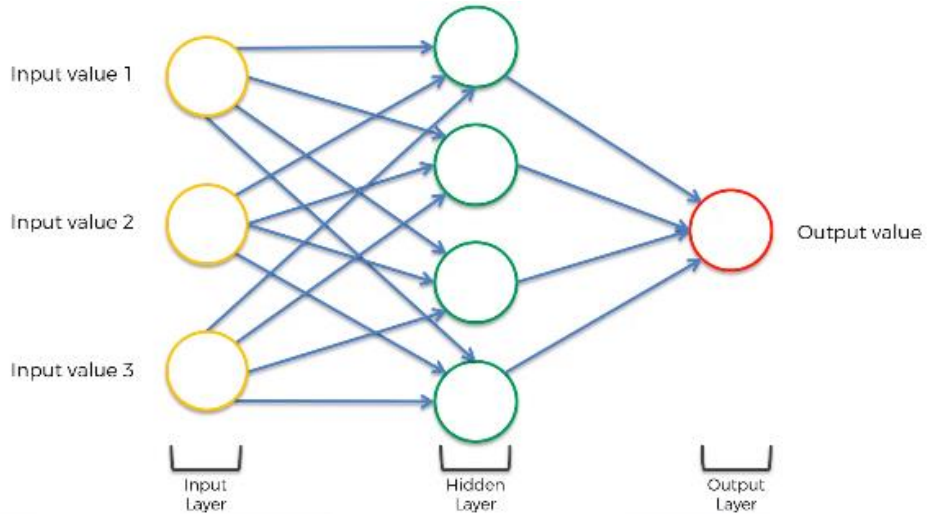
Imagine that you are modelling something that you want to predict. In this case, you have always some input (something) to start. So, your prediction is off then Your prediction is off, then that's called the **input layer**. Then, you get the output (the value that you want to predict).

Finally, we are going to have a hidden layer between the input layer and the output layer:

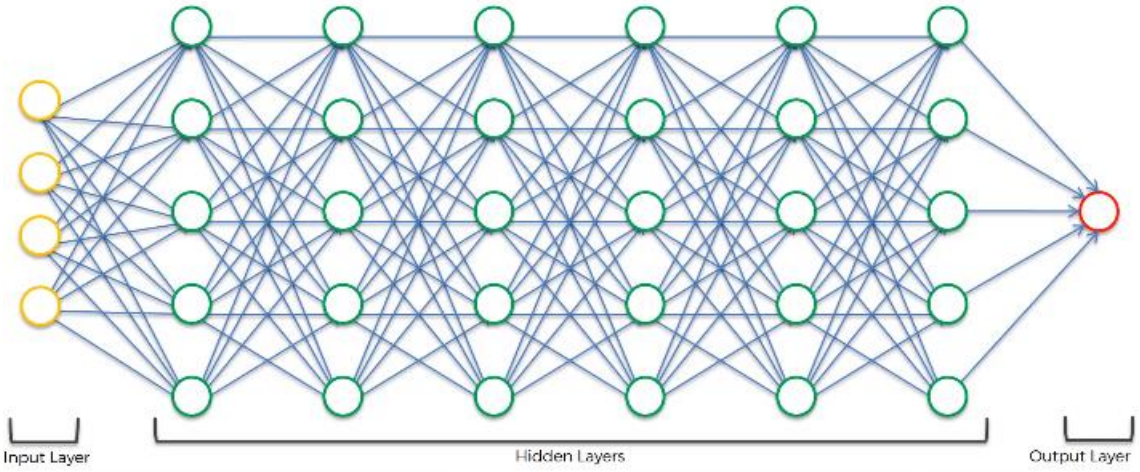


As you could see in your brain, you have so many neurons with so some information that is coming in through your senses (eyes, nose, etc). The information is not just going right away to the output where you have the result, it is going through all of these billions of neurons before guess output and this is the whole concept behind the brain modelling in deep learning.

At the end, we need these hidden layers that are before the output. Therefore, ***the input layers neurons are connected to a hidden layer neurons, which are connected to the output layer***:



Why should we associate this concept to Deep Learning? =>> We have to take this to the next level separating it even further. So, at the end are not going to have just one hidden layer or neurons, we are going to have lots and lots of hidden layers and we have to connect everything just like the human brain connect everything.



That is how the input values are processed through all these hidden layers just like in the human brain. Then we have an output value and now we are talking about deep learning.

Applications of Deep Learning:

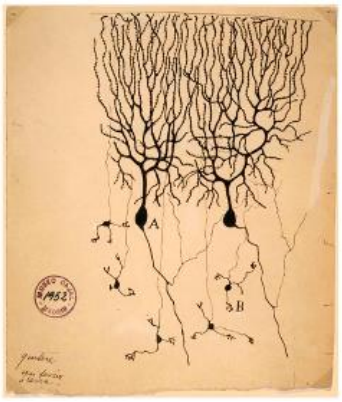
Deep Learning models can be used for a variety of complex tasks:

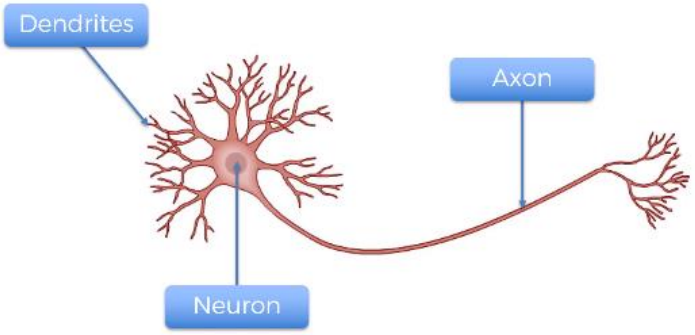
- Artificial Neural Networks for Regression and Classification.
- Convolutional Neural Networks for Computer Vision.
- Recurrent Neural Networks for Time Series Analysis.
- Self Organizing Maps for Feature Extraction.
- Deep Boltzmann Machines for Recommendation Systems.
- Auto Encoders for Recommendation Systems.

2. Artificial Neural Networks

2.1. The Neuron

What is a neuron? Let's see how the neuron looks like:

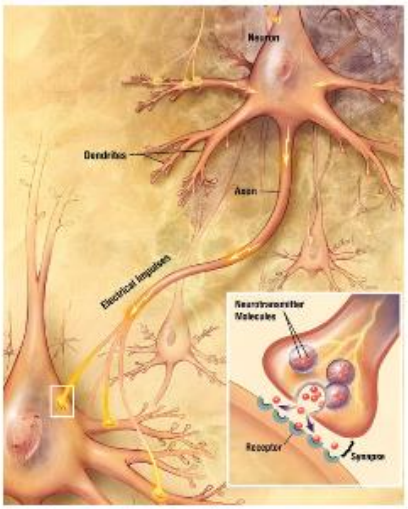




> Dendrites => receptors of the signal from the neuron
>
> Axon => the sender of the signal for the neuron

The important point here is that when a lot of neurons work together they can do magic.

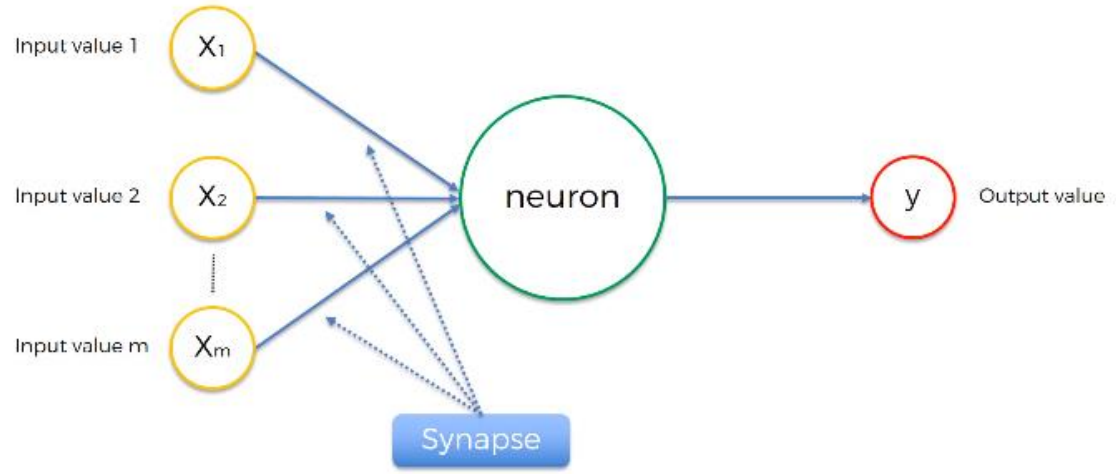
The ***dendrites are like the receivers of the signal*** for the neuron and the ***axon is the transmitter of the signal*** for the neuron. Here is an image of how it all works conceptually:



At the top, you have the neuron and you can see that its dendrites are connected to axons of other neurons that are like even further away above it. Then, the signal from your own travels down its axon and connects or passes on to the dendrites of the next neuron and that is how they are connected. In addition, we can see that the axon does not actually touch the dendrite lot (there is no physical connection there).

How are we going to recreate a neuron into the machine?

The goal is to create an amazing infrastructure for machines to be able to learn.



Here the green neuron is receiving information from the yellow neurons (input layer).

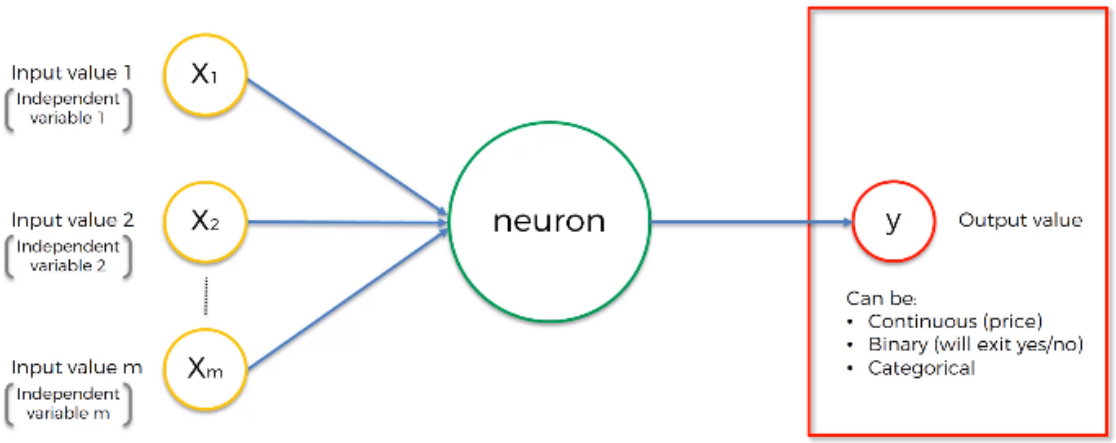
This jump process of the signal passing is called the synapse. For simplicity's sake, this is the term I will also use when referring to the passing of signals in our Neural Networks.

The inputs on the left side represent the incoming signals to the main neuron in the middle. In a human neuron, this these would include smell or touch.

In your Neural Network these inputs are independent variables. They travel down the synapses, go through the big grey circle, then emerge the other side as output values. It is a like-for-like process, for the most part.

The main difference between the biological process and its artificial counterpart is the level of control you exert over the input values; the independent variables on the left-hand side.

How can we the output value?



Signal (your senses) => Input Value

Predicted Value => Output Signal

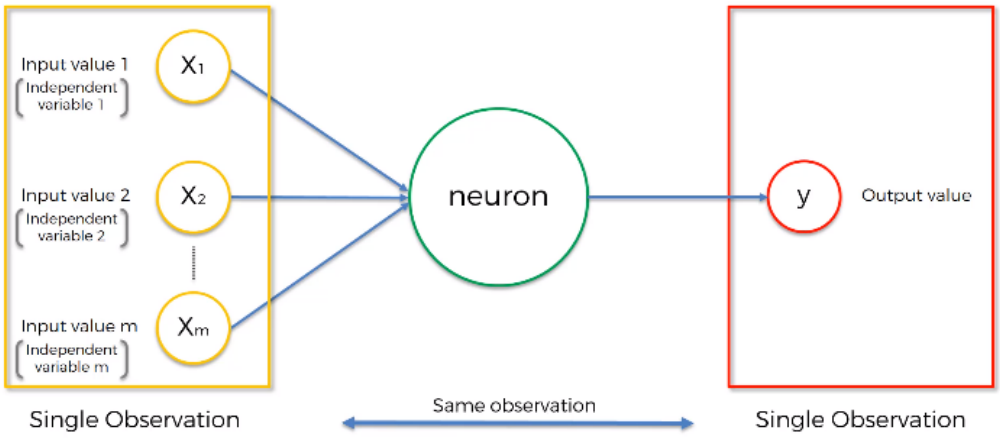
Basically, you have inputs that are coming in terms of human brain.

Important => The independent variables (input layer) are only for one single observation.

Once you have the information, which passes through sing synapse to the neuron, you go the output values.

The green neuron is the “hidden layer”.

The important thing to remember here is that in that case your output value will not be just one because it will be several output values because these will be a dummy variables which will be representing your categories and that just this how it works. In addition, it is important to remember how you are going to be getting your categories out of the artificial neural network.



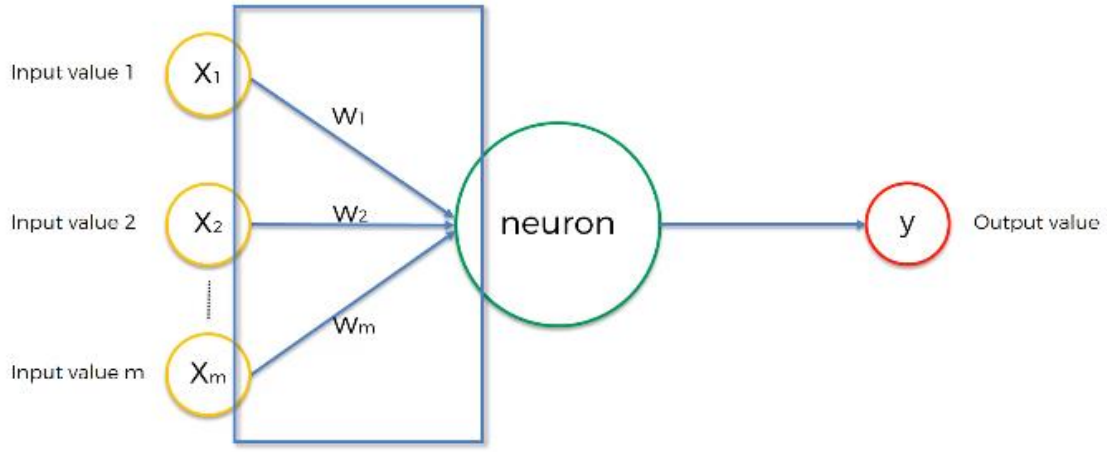
Observations: It is equally important to note that each variable does not stand alone. **All the variables are together as a singular observation**. For example, you may list a person’s height, age, and weight. These are three different descriptors, but they pertain to one individual person.

Now, once these values pass through the main neuron and break on through to the other side, they become output values. Important => **the independent variables are only for one single observation**. We need “standardize” them, it means that we have to make sure that they have a mean of 0 and a variance of 1.

Basically, you want all of these variables to be quite similar in about the same range of values.

Why? => To make easier the processing of the neural network.

Let’s talk about the signus:



Here we need to talk about ***weights***, which are crucial to artificial neural network and it works functioning because weights are how neural networks learn by adjusting the weights. Therefore, the neural network decides in every single case what single signal is poor and what signal is not important to certain neuron.

What single gets passed along and what signal doesn't get passed along or what strength to what extent signals get passed along.

In addition, the weights are the things that get adjusted through the process of learning, like when you are training an artificial neural network you are basically adjusting all of the weights in all of the sign says across this whole neural network. Here are where gradient descent and back propagation come into play.

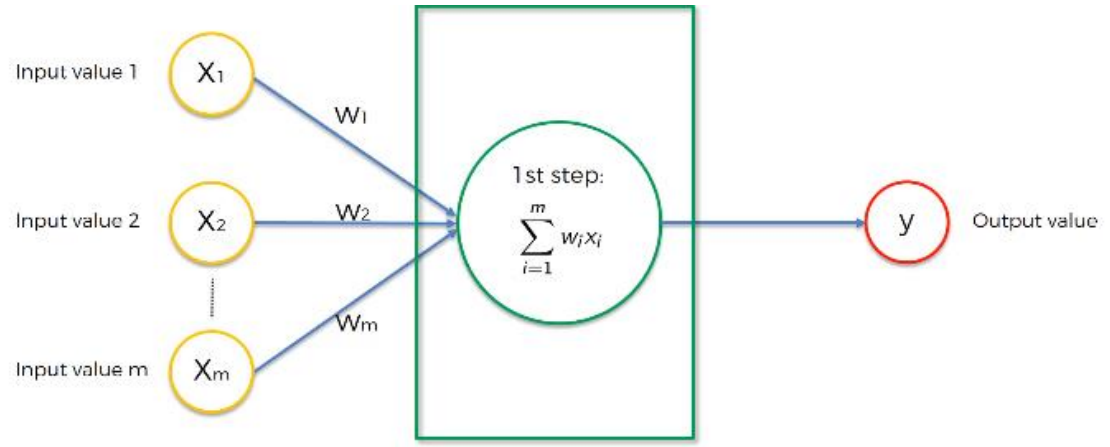
Weights => **Each synapse is assigned a weight**.

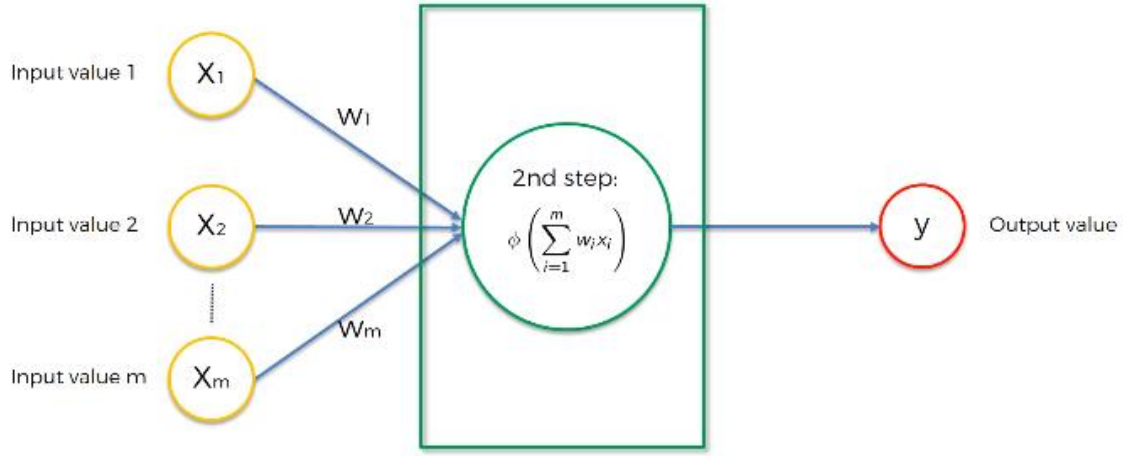
Just as the tautness of the tightrope is integral to the stuntman’s survival, so is the weight assigned to each synapse for the signal that passes along it.

The weight determines which signals get passed along or not, or to what extent a signal gets passed along. **The weights are what you will adjust through the process of learning**. When you are training your Neural Network, not unlike with your body, the work is done with weights.

What happens inside the neuron? => Two main things:

It takes that added so the weighted sum of all of the input values that is getting very simple it's very straightforward just add up multiply by the way add them up and then it applies an activation function.

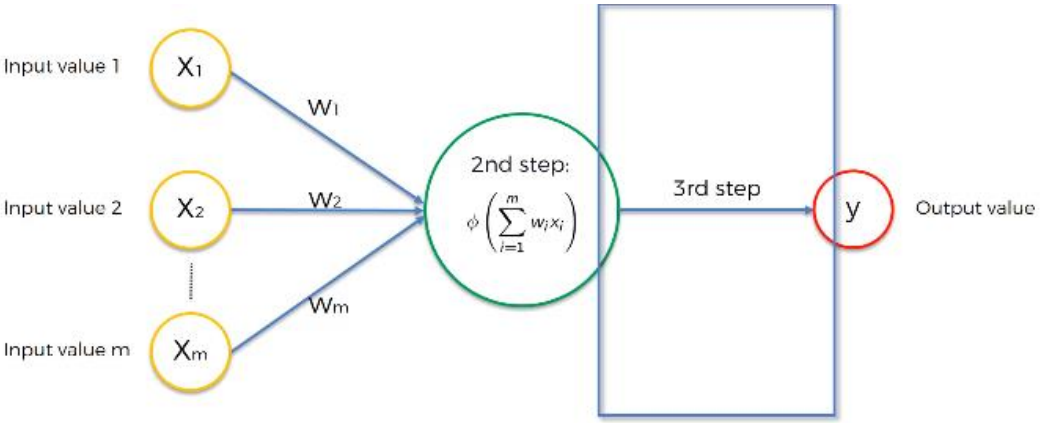




Now we're going to talk more about activation functions further down but it's basically a function that is assigned to this neuron or to this whole layer and it is applied to this weighted some.

The signal passes on that the function applied to the way that some.

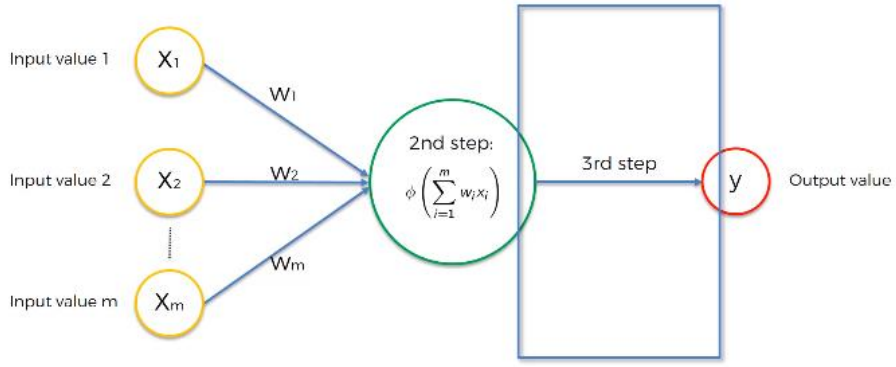
But basically depending on the function the neuron will either pass on a signal it or it won't pass the signal on.



And that's exactly what happened here in step three. The neuron passes on that signal to the next neuron down the line

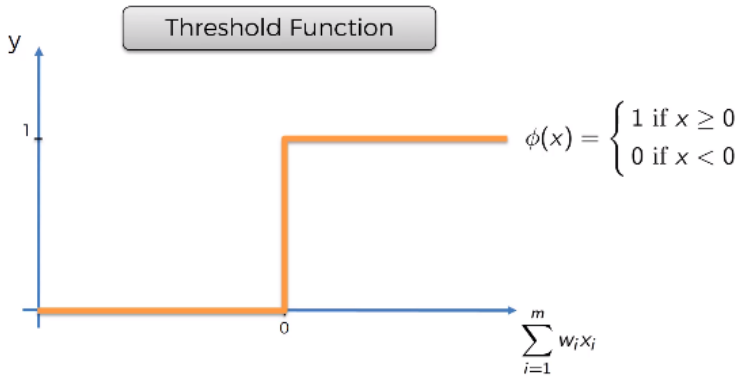
2.2. The Activation Function

As we saw in the last section, the activation function occurs in the Step 3 of the weights:



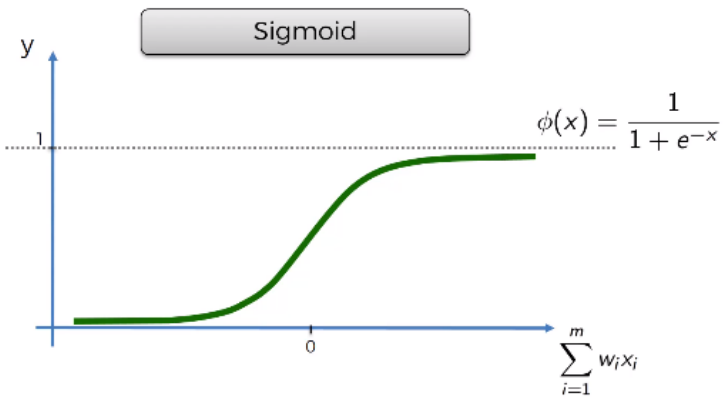
How many options do we have for the “activation functions”? => There are 4 types of activation functions.

a) Threshold Function



The first is the simplest. The x-axis represents the weighted sum of inputs. On the y-axis are the values from 0 to 1. If the weighted sum is valued as less than 0, the TF will pass on the value 0. If the value is equal to or more than 0, the TF passes on 1. It is a yes or no, black or white, binary function.

b) Sigmoid Function



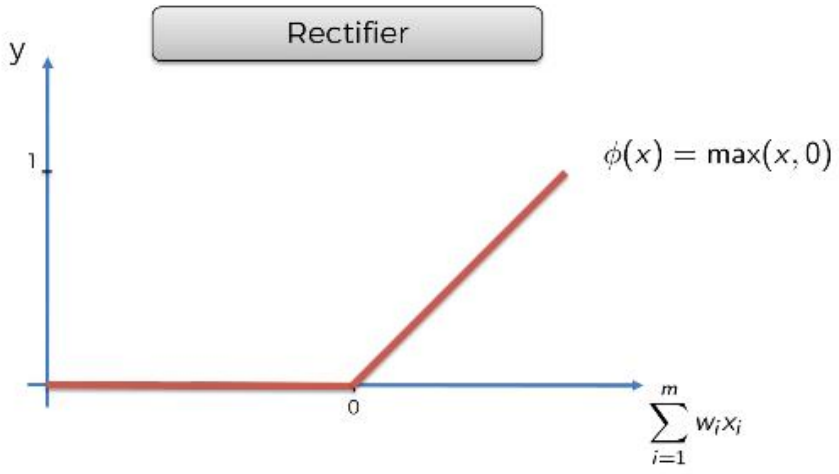
Here is the second method. With the Sigmoid, anything valued below 0 drops off and everything above 0 is valued as 1.

This function is useful when you try to predict probabilities.

Shaplier than its rigid cousin, the Sigmoid's curvature means it is far better suited to probabilities when applied at the output layer of your NN. If the *Threshold tells you the difference between 0 and 1 dollar, the Sigmoid gives you that as well as every cent in between.*

c) Rectifier Function

Here we have one of the most popular functions applied in global Neural Networking today, and with the most medieval name.



The rectifier function is the most popular in neural networks.

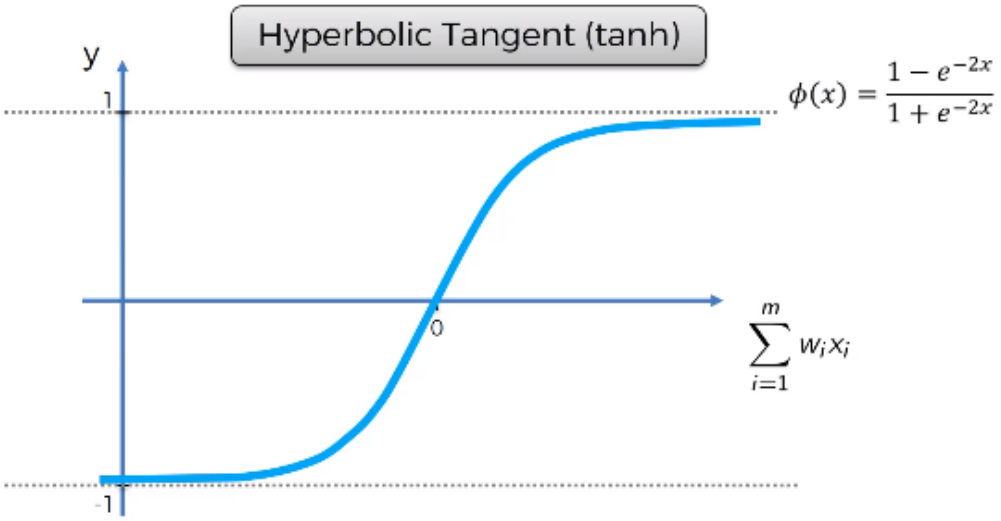
Look at how the red line presses itself against the x-axis before exploding in a fiery arrow at 45 degrees to a palace above the clouds.

Who wouldn't want a piece of that sweet action? Its spectacle is matched only by its generosity. Every value is welcome at the party. If a weighted value is below 0 it doesn't just get abandoned, left to float through time and space between stars and pie signs. It is recruited, indoctrinated. It becomes a 0.

d) Hyperbolic Tangent Function

It is willing to delve deep below the x-axis and its 0 value to the icy pits of the lowest circle, where the value -1 slumbers.

From there it soars on a route similar to that of the Sigmoid, though over a greater span, all the way to blessed 1.

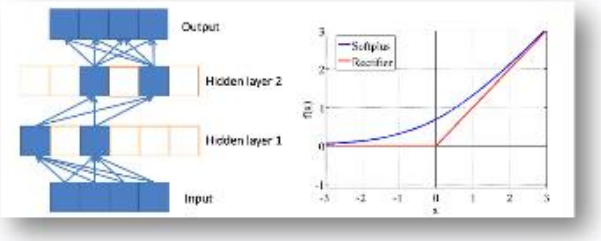


The hyperbolic tangent function is similar to the sigmoid function, but the first one goes below 0. It means, the values go from 0 to 1 or approximately from 0 to minus 1 on the other side.

Additional Reading:

Deep sparse rectifier neural networks

By Xavier Glorot et al. (2011)



Link:

http://jmlr.org/proceedings/papers/v15/glorot11a/glorot11a.pdf

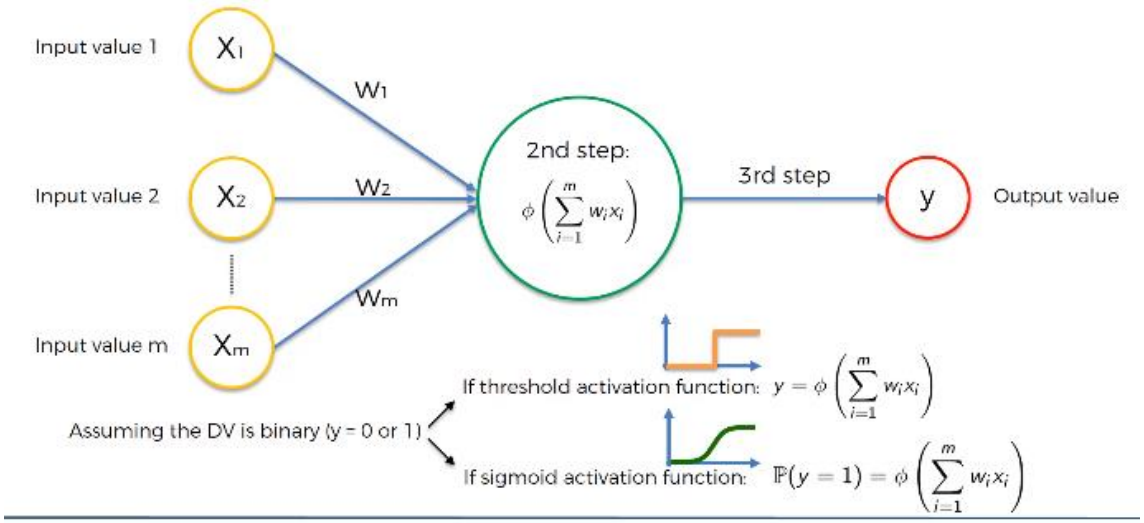
Questions to apply the theory in the practical side:

Question 1: Assuming that the dependent variables is binary (y=0 or y=1). What should be the properly "activation function"?

The "Threshold activation" because we can predict 2 values and the "sigmoid activation function" because it works between 0 and 1, just what we need.

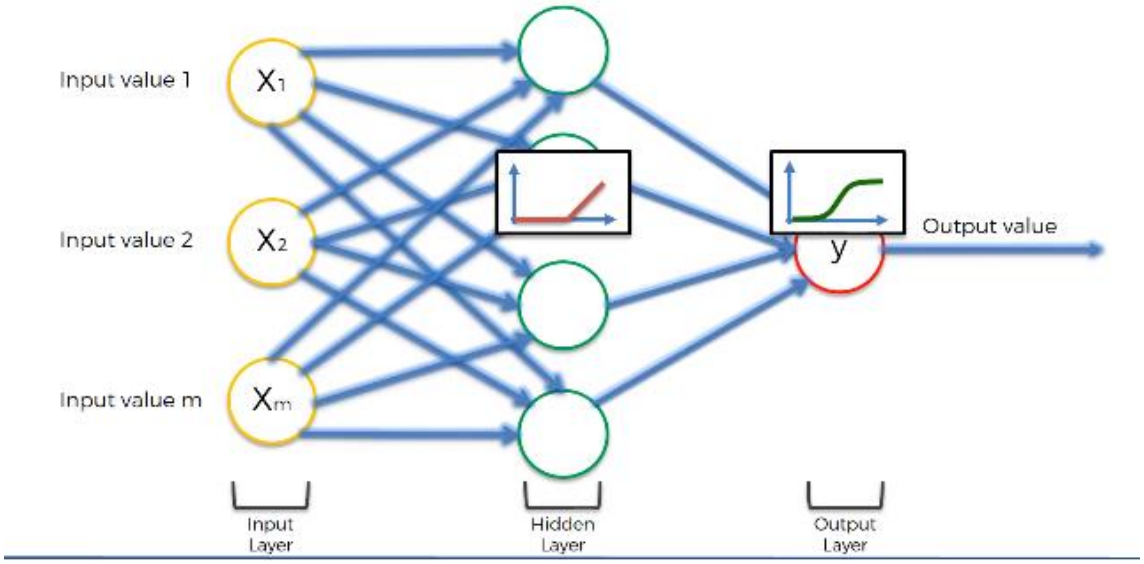
We need exactly 0 or 1, but we can use the probability of y=YES or y=NO.

Basically the closer you get the top, the more likely it is indeed a one or a yes rather than a NO or 0 (very similar to the logistics regression approach).



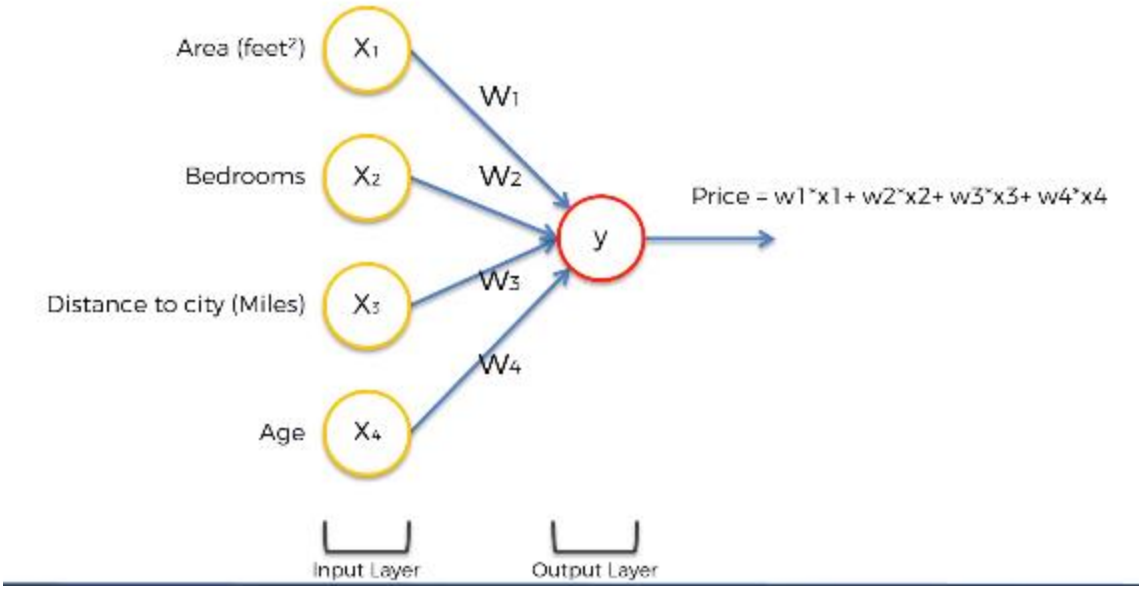
In the cases that we need to apply the “rectifier function”, the information comes from the signals (senses) to the output layer, where the sigmoid function would be applied and that would be our final output, which could predict a probability this specific case. This combination is common when we apply the rectifier functions in the hidden layers and the sigmoid function in the output layer.

In the following case, we are going to apply a combination of the “rectifier function” (in the hidden layer) and the “sigmoid function” (in the ouput layer).



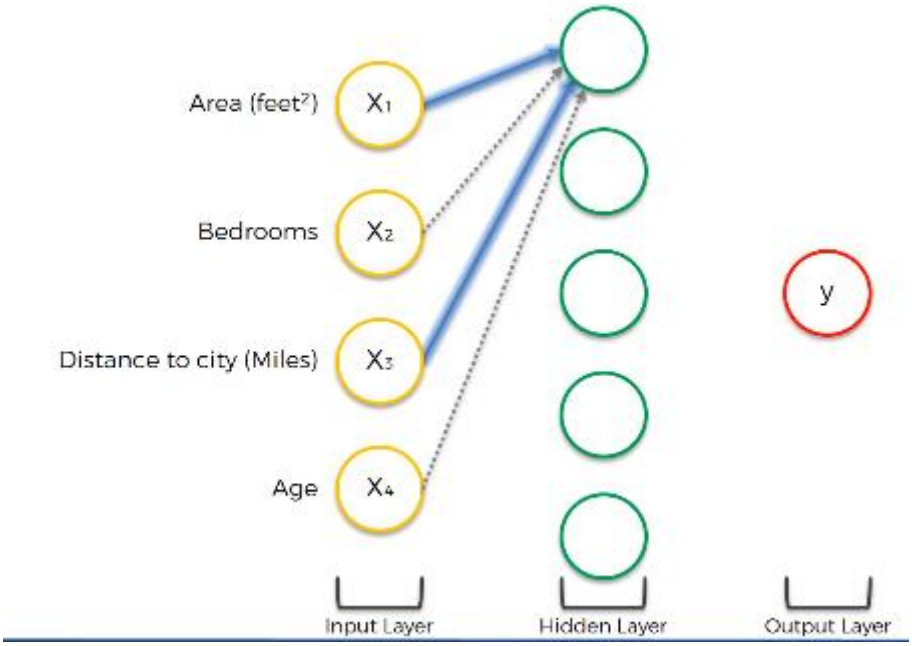
2.3. How do Neural Networks work? (example)

Let's recap the concept of Neural Networks with a simple case:



The power of the neural networks compared to other Machine Learning algorithms is the hidden layer, and we are going to check the details of the hidden layer.

As we saw previously, all the neurons from the input layer have synapses, it means that all the neurons are connected to the top neuron located in the hidden layer. In addition, those synapses have weights:

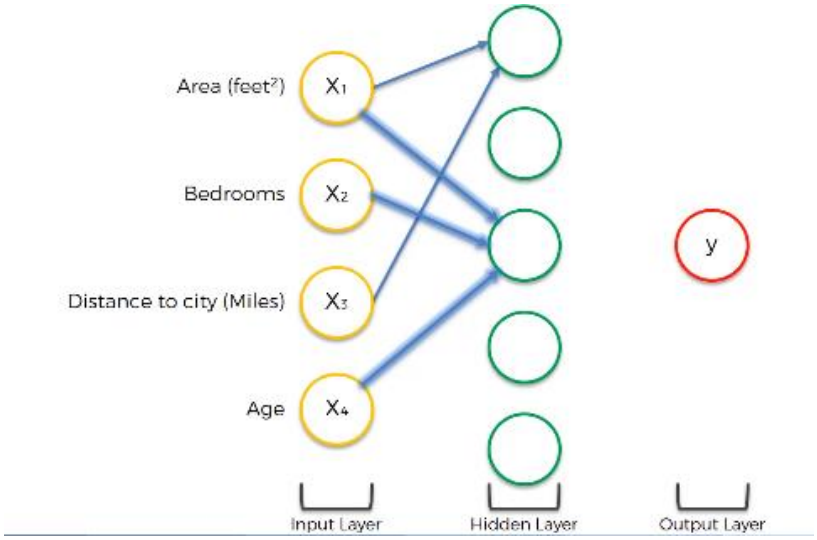


Sometimes some weights will have a non-zero value, while other weights will have zero value because not all inputs will be valid or not all inputs will be important for every single neuron. Therefore, some inputs will not be important.

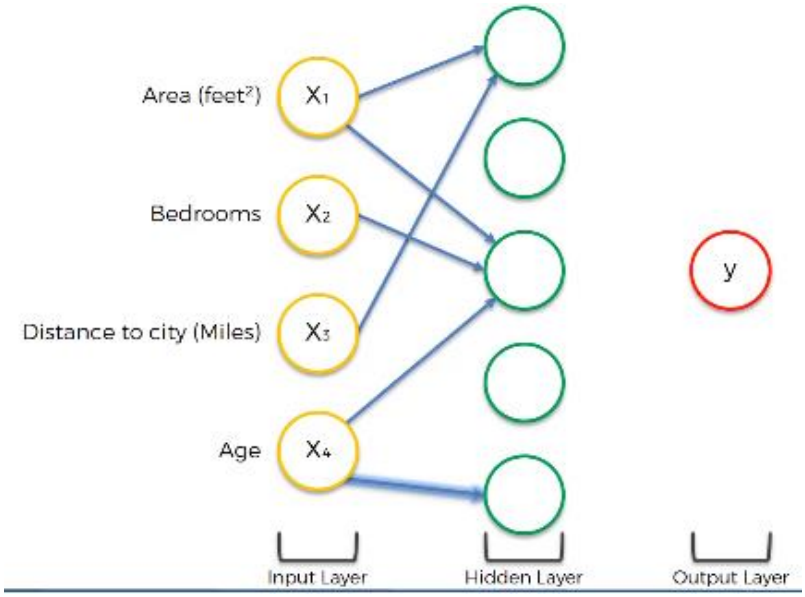
What is the reason? => Maybe the neuron can speculate, but here it is picking those specific properties and it will activate the function and fire up only when the certain criteria is met. Then, it performs on calculation and it combines those two and as soon as certain areas where it fires up and that contributes to the price in output.

For this reason, the previous neuron did not consider the variables “bedrooms” and “age” because it was focused on that specific thing. (“area” and “distance to city”).

Let’s move on to the next:

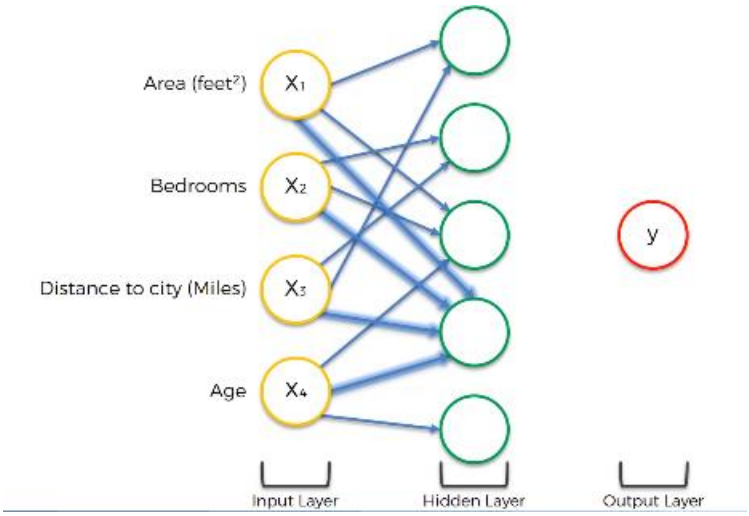


Another criteria:



Therefore, the neuron combines different parameters/variables in a new attribute and it makes the neuron more accurate. In addition, the neuron keeps in mind things that we cannot do on our own.

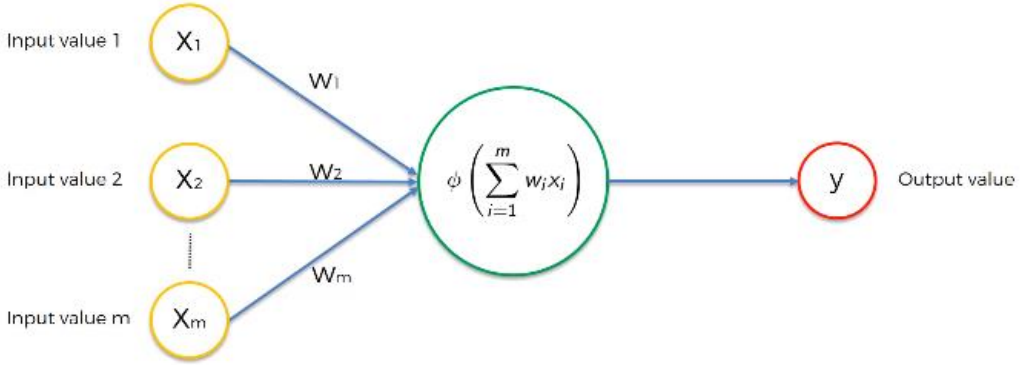
Finally, we have to keep in mind that the hidden layer allows us to increase the flexibility of the neural network and allows the neural network to look for very specific things and then in combination. That's where the power comes from.



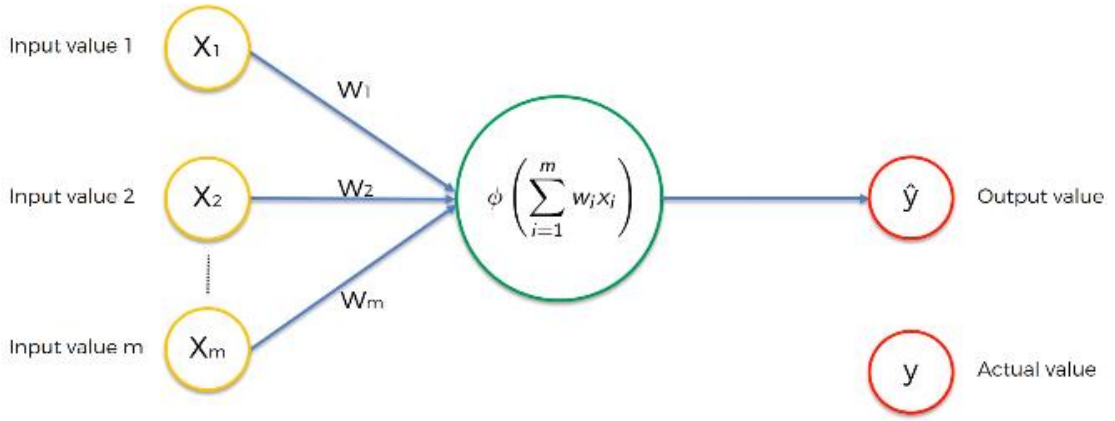
Each one of these neurons by itself cannot predict the price, but together they can predict the price and they can do quite an accurate job if trained properly set up properly.

2.4. How do Neural Networks learn?

Here we have a very basic neural network with a one layer is called a single layer feedforward neural network, which is also called a **perception**.



When you code the Neural Network you are coding the architecture and then you point the Neural Network at a folder with all the images of the things that it has to predict, which are already categorized. Then, the NN will learn, will understand everything and it needs to be trained up. Finally, you will give a new image to the NN and it will be able to understand what it was.



Output value => predicted value by the Neural Network

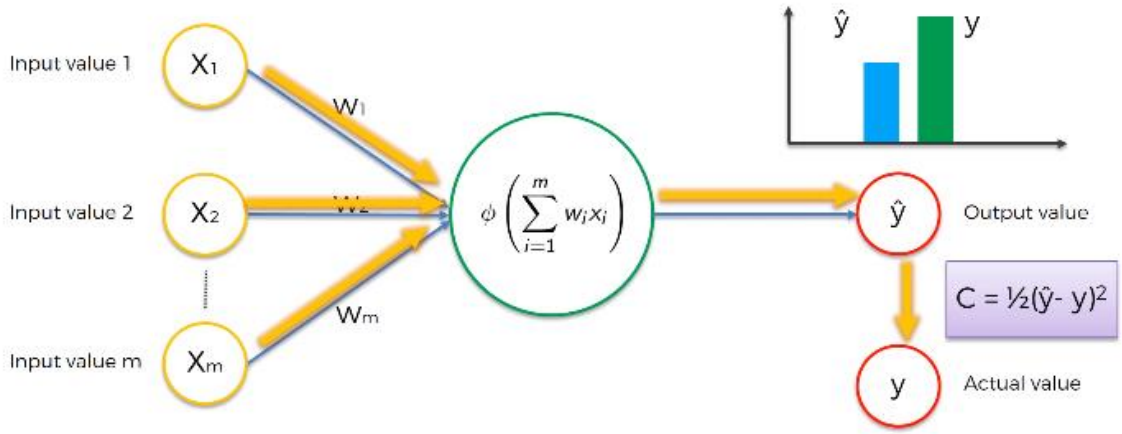
Actual Value => Real value that we see in the reality

How the perception learns?

Imagine that we have some input values, which have been supplied to the perception or to our own network. Then, the activation function is applied and we get an output value.

Now, we are going to plot the output on a chart (our output y hat). We need to compare the output value to the actual value that we want the neural network to get right.

We realize that there is a bit difference between both values. In order to fix it, we need to calculate a function called “cost function”, which tells us what is the error that you have in your prediction.

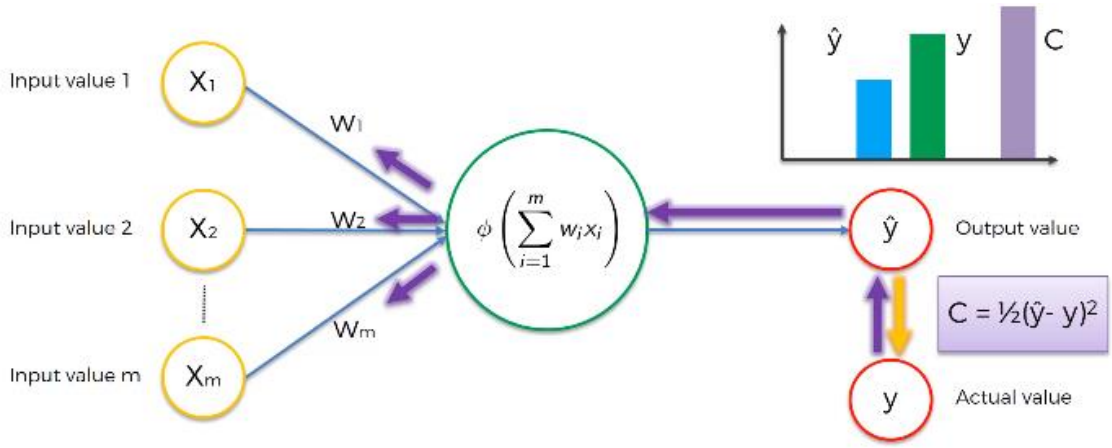


Our goal is to minimize the cost function because the lower the cost function the closer the y hat is to y.

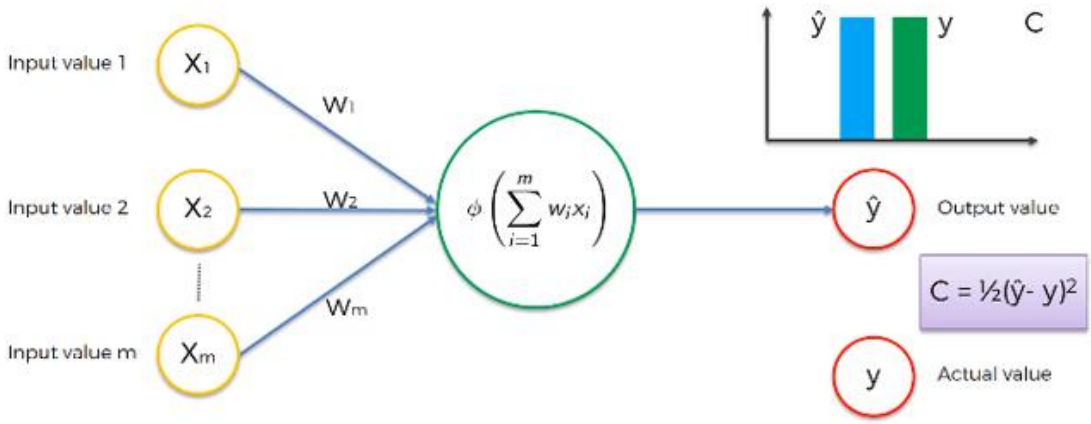
How can we minimize the cost function?

Firstly, we have compared the output with the y value and the cost function, we are going to feed this information back into the neural network. Then, the information goes back into the neural network and it goes to the weights and the weights get updated.

Therefore, the *only thing that we have control of in this very simple neural network are the weights*. So, all we can do is update the weights and tweak them a little bit.



The “output value” is equal to the “y actual value” when the cost function is equal to 0. Usually, you will not get cost function equal to zero in the reality, but we need to try to get this value in our prediction models.

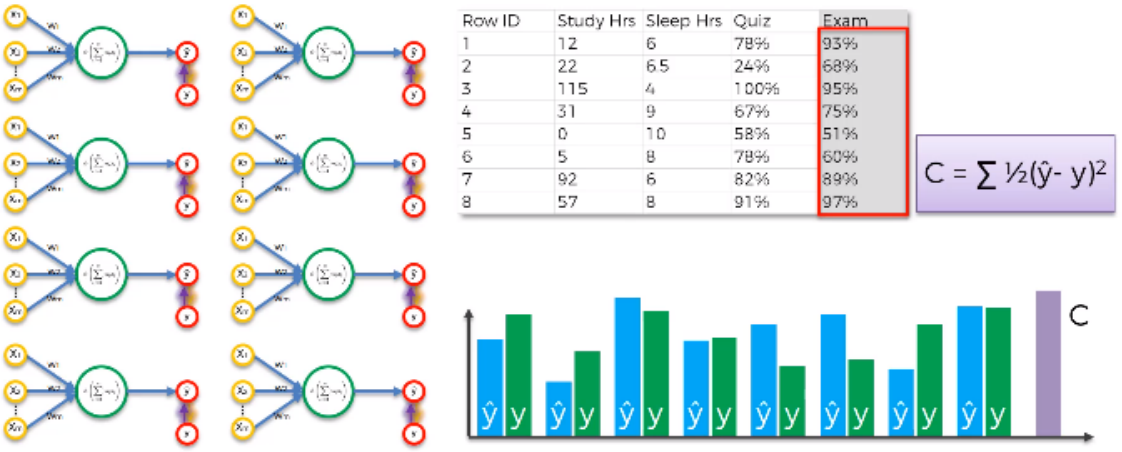


<u>Let's see what happens when you have multiple rows to analyse (complex examples)</u>:

In the following example, there is our first row and there is “output value” for the first row there is a second role there is “output value” for the second round. So again, it is being fed into the same neural network every time.

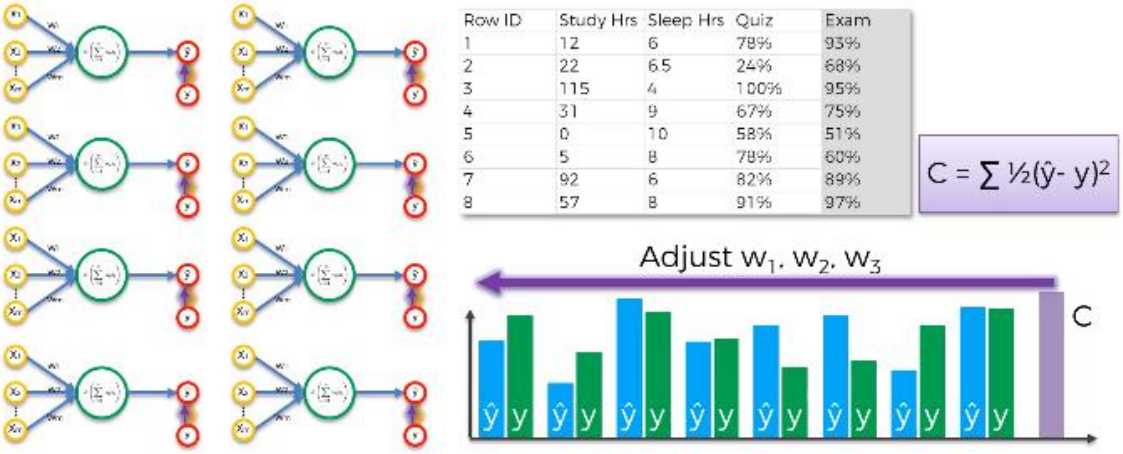
Then, we compare to the “output values” with the “actual values” in the plot.

Now, based on all of these differences between “output values” and “actual values” we can calculate the cost function, which is the sum of all of those squared differences between “output values” and “actual values”.



Now, what we can do is going back and update the weights:

The important thing to remember here is that all of these perceptions or neural networks are actually just one neural network. In addition, when we update the weights we are going to *update the weights in that one neural network.* So, the weights are going to be the same for all of the rows.



Next, we are going to run this whole thing again. We have to feed every single row into the neural network find out our cost function and do this whole process again. This whole process is called "***back propagation***".

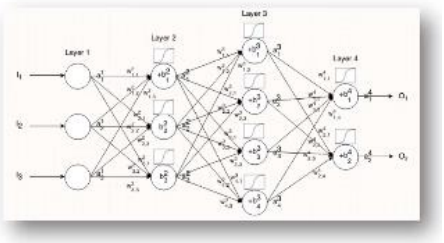
The goal is to minimize the cost function and find the optimal weights.

Additional Reading:

A list of cost functions used in neural networks, alongside applications

CrossValidated (2015)

Link:

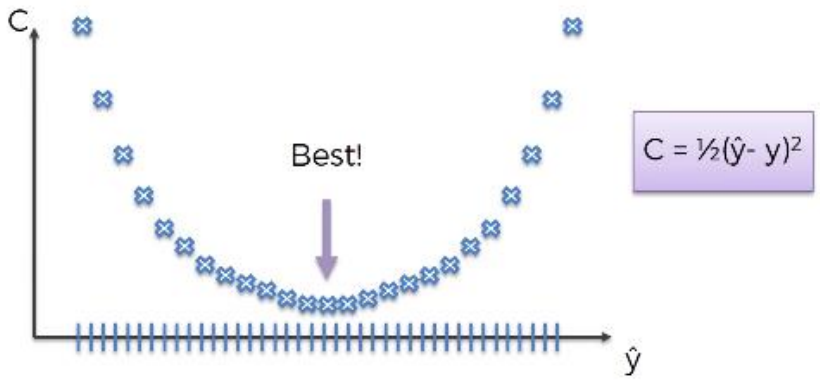


http://stats.stackexchange.com/questions/154879/a-list-of-cost-functions-used-in-neural-networks-alongside-applications

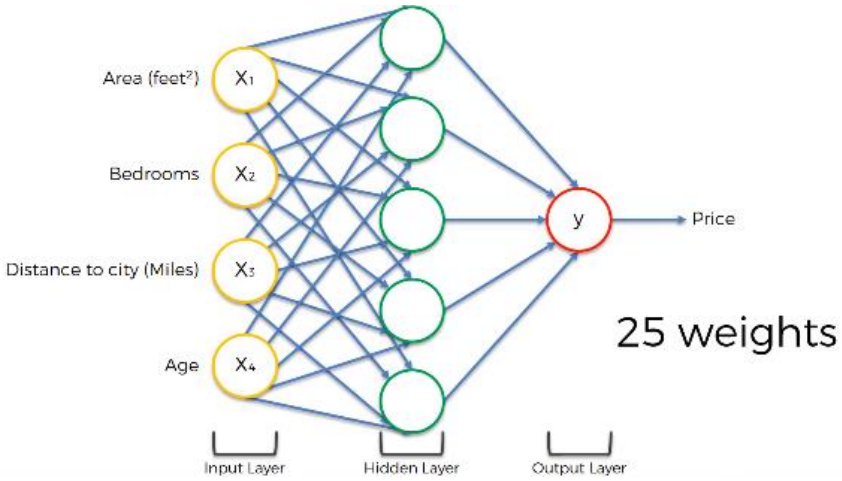
2.5. Gradient Descent

Let's see how the weights are adjusted in order to minimize the cost function.

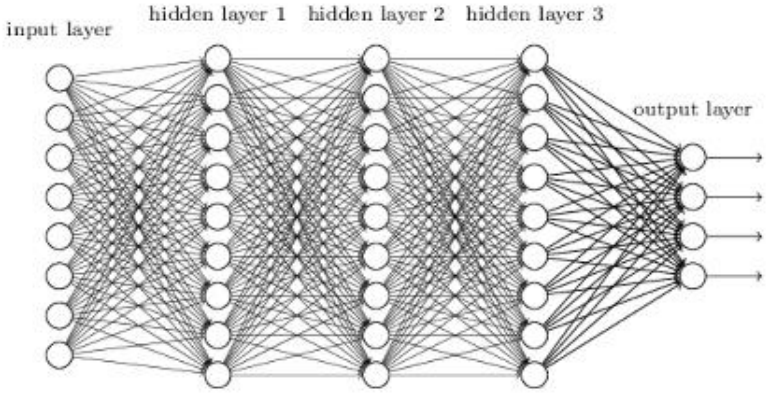
Imagine that you have thousands of weights and we need to look for the best one. So, we are going to try all the possible combinations and choose the best one:



Let's check an example of a neural network *before adjusting the weights*:



And below another example more complex:

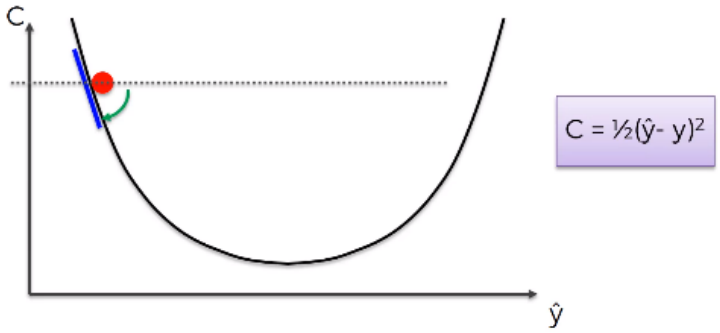


How to adjust the weights?

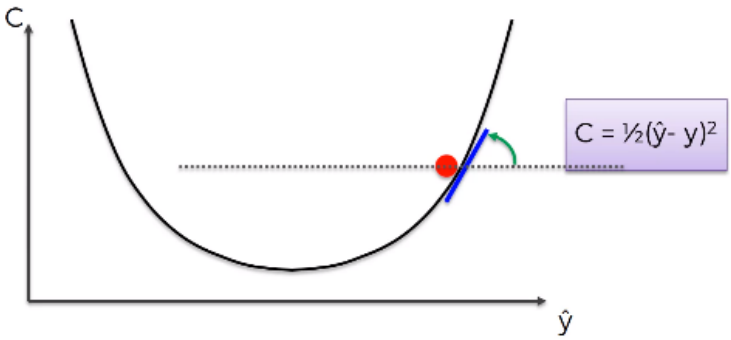
The method used to optimize the weights is called "**Gradient Descent**".

We have to look at the angle of our cost function at the red point, which is called "gradient". Then, we just need to differentiate find out what the slope is in that specific point and find out if the slope is positive or negative.

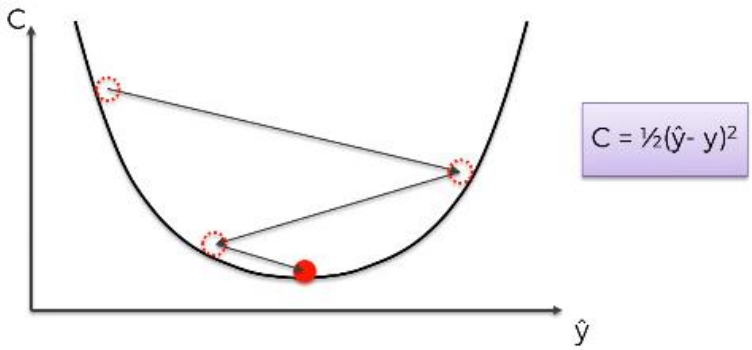
If the slope is negative like in this case means that you are going downhill so to the right is downhill to the right:



Now, the ball rolls down again and you have to do the same thing. You calculate the slope and in case that the slope is positive meaning writer's uphill left is downhill and you need to go left and you're on the ball down.

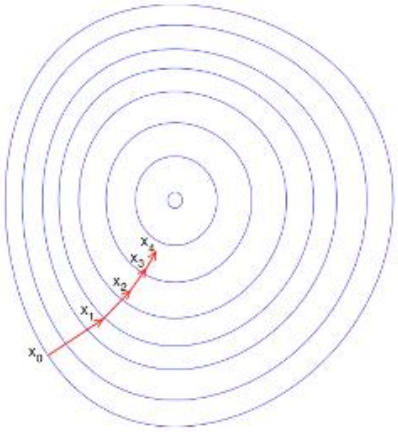


Then, you calculate again the slope and you're all the bull right there you go so that's how you find in simple terms that's how you find the best weights, which are the best situation that minimizes your cost function.

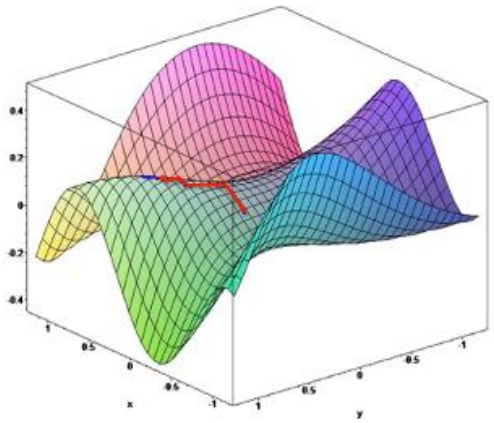


In conclusion, we are getting to the bottom by just understanding which way we need to go.

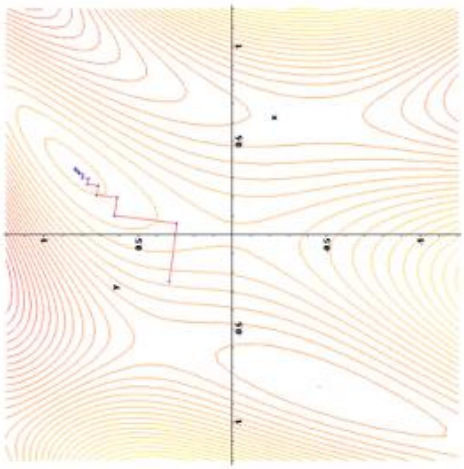
Here we have a two dimensional space for the gradient descent as you can see it's getting closer to the minimum and it's also called gradient descent because you're descending into the minimum of the cost function.



Gradient descent example applied in three dimensions:

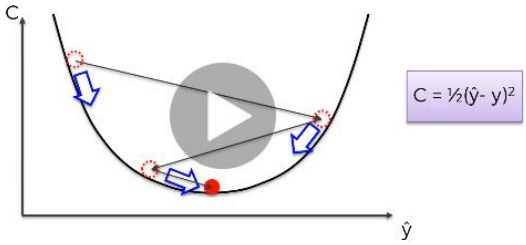


Finally, this is what it looks like if you projected onto two dimensions you can see zigzagging its way into the minimum.



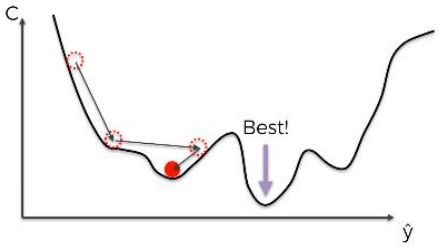
2.6. Stochastic Gradient Descent

The gradient descent method requires that the cost function has to be convex and it has in essence one global minimum.



But, **what happens if the cost function is not convex?**

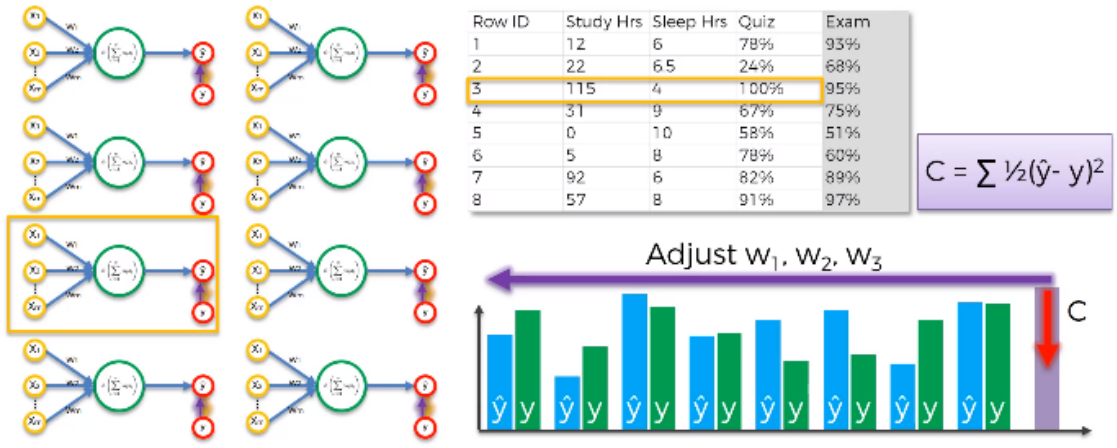
In this case, we could find a local minimum of the cost function rather than the global one. So, this one was the best one and we found the wrong one. Therefore, we do not have an optimized neural network because we have a subpar neural network.



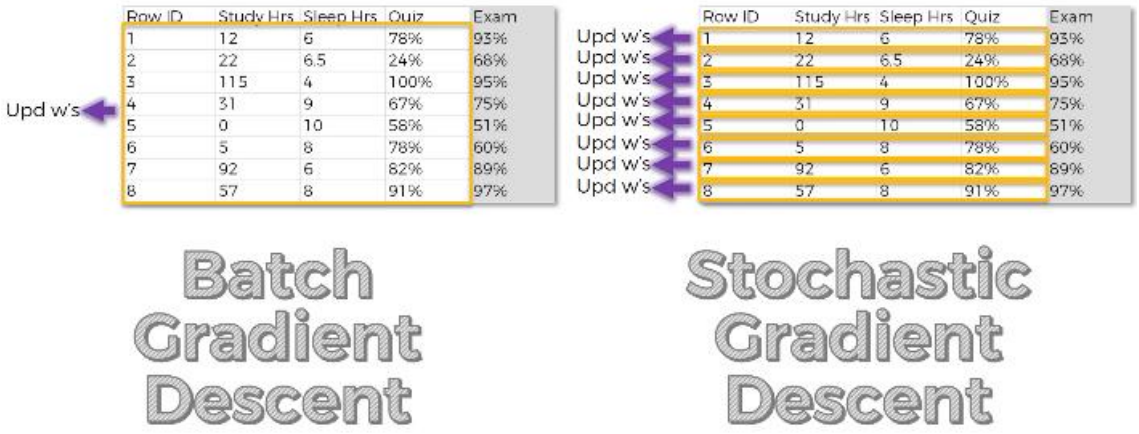
How can we fix this situation? =>> Using the Stochastic Gradient Descent method, which does not have the requirement that the cost function should be convex.

The Stochastic Gradient Descent method is a bit different compared to the Gradient Descent method because we take the rows one by one. Let's see how this new method works:

First, we take the first row, we run the neural network and we adjust the weights. Then, we move into the second row, we take the second row and we run our neural network. We look at the cost function and then, we adjust the weights again and then we take another row.



In summary, we are looking at we are adjusting the weights after every single row rather than doing everything together and then testing weights two different approaches.



The main two differences are that the sarcastic gradient descent method helps you avoid the problem where you find those local extremities or local minimums rather than the overall global minimum.

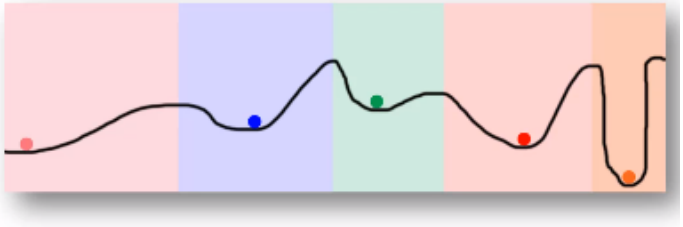
The second benefit of the Stochastic Gradient Descent method it that it is faster compared to the other method.

Additional Reading:

A Neural Network in 13 lines of Python (Part 2 - Gradient Descent)

Andrew Trask (2015)

Link:



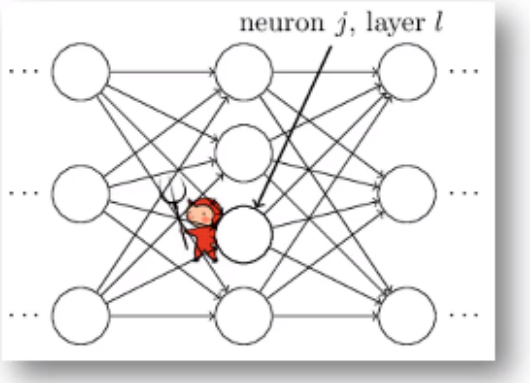
https://iamtrask.github.io/2015/07/27/python-network-part2/

Additional Reading:

Neural Networks and Deep Learning

Michael Nielsen (2015)

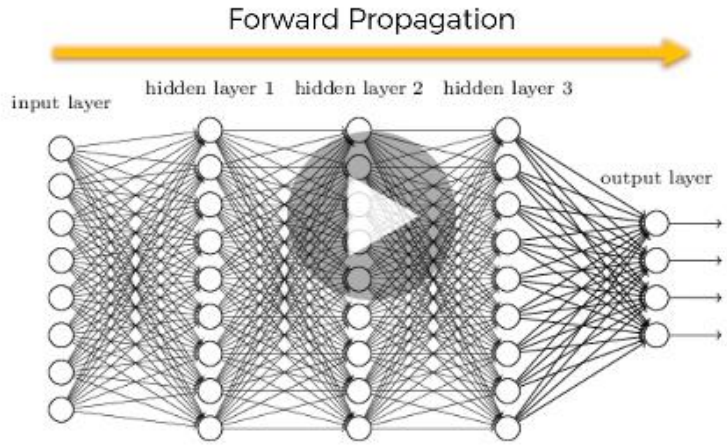
Link:



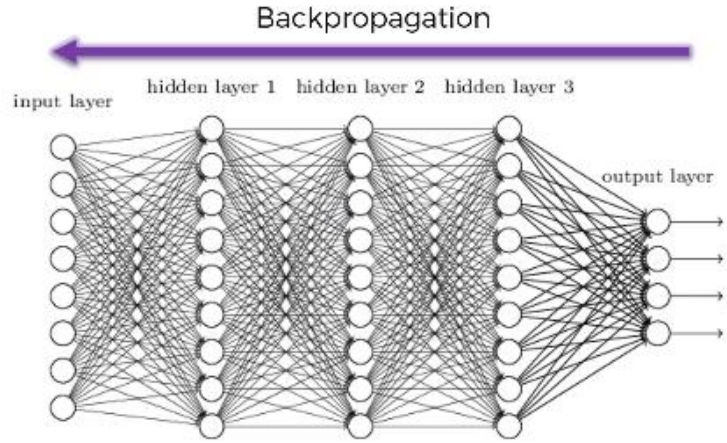
http://neuralnetworksanddeeplearning.com/chap2.html

2.7. Backpropagation

Previously, we learned the concept of ***Forward Propagation***, where information is entered into the input layer and then it is propagated forward to get the output values. Then, we compare those to the actual values that we have in our training set.



Then, we calculate the errors and them are back propagated through the network in the opposite direction and that allows us to train the network by adjusting the weights.



Backpropagation is an advanced algorithm driven by very interesting and sophisticated mathematics which allows us to adjust the weights. **All of the weights are adjusted simultaneously at the same**. Basically, it knows which part of the error each of your weights in the neural network is responsible for.

Just wrap everything up with a step-by-step walkthrough of what happens in the training of a neural network.

Step 1 => Randomly initialise the weights to small numbers close to 0 (but not 0).

We have to start with random values for weights near zero. Then, through the process for propagation these weights are adjusted until the error is minimized. So, the cost function is minimized.

Step 2 => Input the first observation of your dataset in the input layer, each feature in one input node.

Take the combs and put them into the input nodes separately for propagation from left to right.

Step 3 => Forward-propagation: from left to right, the neurons are activated in a way that the impact of each neuron's activation is limited by the weights. Propagate the activations until getting the predicted result "y".

The neurons are activated limited by the weights. So, the weights basically determine how important each neurons activation is then propagate the activation until getting the output value.

Step 4 => Compare the predicted result to the actual result. Measure the generated result.

Compare the output result to the actual result and measure the generated error. Then, you do the backwardation from right to left.

Step 5 => Backward-propagation: from right to left, the error is back-propagated. Update the weight according to how much they are responsible for the error. The learning rate decided by how much we update the weights.

The backward-propagation algorithm is structured in the way that the learning rate decides by how much we update the weights the learning rate as parameter you can control in your neural network.

Step 6 => "Repeats steps 1 to 5 and update the weights after each observation (Reinforcement learning)" or "Repeats steps 1 to 5 but update the weights only after a batch of observations (Batch Learning)".

In the case of Reinforcement Learning, we apply the stochastic gradient descent method.

For the Batch Learning, we only repeat the steps 1 to 5 only after a batch of observations or batch learning it could be a full gradient descent, or badge green Nissan or mini batched gradient descent.

Step 7 => When the whole training set passed through the ANN, that makes an epoch. Redo more epochs.

So, just keep doing that to allow your neural network to train better and constantly adjust itself as you minimize the cost function.

In summary, those are the **steps you need to take to build your artificial neural networks and train it**.

3. Building Artificial Neural Networks in Python

3.1. Artificial Neural Network for Classification

3.1.1. Business Case

In this business case, we are looking at the data set of a bank. If you scroll down to the bottom is going to be 10000 customers. The bank has been seeing unusual churn rates. "*Churn*" is when people leave the company.

The bank has seen customers leaving at unusually high rates and they want to understand what the problem is and they want to assess and address that problem.

That is why they've hired you to look into this data set for them and give them some insights. This fictional bank operates in Europe in three countries: France, Spain and Germany. The column **"Exit"** tells you whether or not the person left the bank within those six months. So, it will be **our independent variable**.

"The goal is to create a job demographic segmentation model to tell the bank which of their customers are at highest risk of leaving."

Please, keep in mind that you would have a binary outcome. So, whenever you have a scenario when you have a binary outcome and you have lots of independent variables you can build a proper robust and that will tell you which factors influence the outcome.

3.1.2. Step 1 – Data Preprocessing

We are going to build a deep neural network with neurons and fully connected layers connecting these neurons. In order to get it, we will use the TensorFlow 2.0. library, which is available in Python.

In this case we will apply ANN for classification.

a) Importing the required packages & Loading the dataset

Firstly, we will import the required libraries and load the dataset.

```
import numpy as np
import pandas as pd
import tensorflow as tf
```

With the following script, we will check what version of the tensorflow library we are using:

```
#check the tensorflow version that we are using
tf.__version__
```

```
'2.1.0'
```

Secondly, let's load the dataset and visualize the information:

```
dataset = pd.read_csv('Churn_Modelling.csv')
dataset.head()
```

RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary
1	15634602	Hargrave	619	France	Female	42	2	0.00	1	1	1	101348.88
2	15647311	Hill	608	Spain	Female	41	1	83807.86	1	0	1	112542.58
3	15619304	Onio	502	France	Female	42	8	159660.80	3	1	0	113931.57
4	15701354	Boni	699	France	Female	39	1	0.00	2	0	0	93826.63
5	15737888	Mitchell	850	Spain	Female	43	2	125510.82	1	1	1	79084.10

After checking the dataset, we realized that we do not need the A, B and C columns because they do not have any impact on the dependent variable. So, we will start our analysis from the column D (CreditScore).

As we explained in the description of the Business Case, the dependent variable is the last column of the dataset, which is called “Exited” and it is composed of the following options:

- 0 => Stay in the bank
- 1 => Left the bank

It is really important to understand the correlations between the features and the dependent variable to build the model.

b) Encoding the categorical variables

Encoding the "Gender" column

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
X[:, 2] = le.fit_transform(X[:, 2])
```

```
print(X)
```

```
[[619 'France' 0 ... 1 1 101348.88]
 [608 'Spain' 0 ... 0 1 112542.58]
 [502 'France' 0 ... 1 0 113931.57]
 ...
 [709 'France' 0 ... 0 1 42085.58]
 [772 'Germany' 1 ... 1 0 92888.52]
 [792 'France' 0 ... 1 0 38190.78]]
```

Encoding the "Geography" column

```
#encoding the Geography column creating dummy variables

from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
ct = ColumnTransformer(transformers=[('encoder', OneHotEncoder(), [1])], remainder='passthrough')
X = np.array(ct.fit_transform(X))
```

```
print(X)
```

```
[[1.0 0.0 0.0 ... 1 1 101348.88]
 [0.0 0.0 1.0 ... 0 1 112542.58]
 [1.0 0.0 0.0 ... 1 0 113931.57]
 ...
 [1.0 0.0 0.0 ... 0 1 42085.58]
 [0.0 1.0 0.0 ... 1 0 92888.52]
 [1.0 0.0 0.0 ... 1 0 38190.78]]
```

c) *Splitting the data into training and test*

```
#getting the training set and test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)
```

d) *Applying Feature Scaling*

Feature Scaling is absolutely compulsory for deep learning and categorization.

```
#applying feature scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

3.1.3. Step 2 – Building the ANN

a) *Initializing the ANN.*

In order to build an ANN, we have to create a new variable that will be the properly ANN itself as an object of a certain class. With this object, we will initialize or activate the ANN.

That certain class is the "sequential class", which allows us to build the ANN. The sequential class comes from the Keras module of TensorFlow 2.

The ANN is actually a sequence of layers, which starts from the input layer and then we have hidden layers fully connected until the output layer.

object =>> ann

class =>> sequential class

Let's check how the code looks like:

```
#create the object for the ANN
ann = tf.keras.models.Sequential()
```

b) *Adding the Input layer and the first hidden layer.*

The way to add a fully connected layer into an ANN at whatever phase you are is using the "*dance class*". So, we have to take our object and call the "add" method of the sequential class.

```
#add a simple connected layer
ann.add(tf.keras.layers.Dense(units=6, activation='relu'))
```

The "***units***" parameter is related to the number of neurons that we have in our first hidden layer.

How do i know how many units i need to enter in the hidden level of my neuron? => There is no rule, it is just based on experimentation. So, we have to experiment with a set of hyper-parameters, which are those parameters that will not be trained during the training process. We can start with a number of 6 hyper-parameters.

Then, we will use the *"rectifier" activation function in the first hidden layer of the ANN*.

c) *Adding the second hidden layer.*

For the second hidden layer, we will use the same code as the first hidden layer:

```
#add the second hidden layer
ann.add(tf.keras.layers.Dense(units=6, activation='relu'))
```

d) *Adding the output layer.*

We want an output layer fully connected to that second hidden layer and we need to use the “dense class” again.

We want to predict a binary variable [0, 1]. So, with only one neuron is enough. However, in case that we are doing classification with a non-binary dependent variable, we will need as neurons as possible classes could have the dependent variable because there is no relation between the different classes of the dependent variable.

On another hand, as we are in the output layer, we have to replace the “rectifier activation function” for the “sigmoid activation function”. What will we get?

- Get the predictions of whether the customers choose to leave or not the bank.
- We have for each customer the probability that the customers leave the bank.

Let’s do a recap:

- If you are doing classification with only two categories to predict in the output (predict a YES/NO or 0/1), you have to choose a sigmoid activation function.
- If you are doing classification with more than 2 categories or classes to predict the output, you should use a soft max activation function.
- If you are doing regression, you have to predict a continuous real number as a final output, then you should choose any activation function.

Let’s check how the code looks like:

```
#add the output layer
ann.add(tf.keras.layers.Dense(units=1, activation='sigmoid'))
```

3.1.4. Step 3 – Training the ANN

a) *Compiling the ANN.*

We will use the “compile” method to compile the ANN. Here we have to include three parameters:

- **“Optimizer”** =>> choose an optimizer to adjust the weights through stochastic gradient descent and reduce the loss function in the next iteraction. The most common is the “adam” optimizer.
- “**Loss function**” =>> It computes the difference between the predictions and the real result. For binary classification, we can use the “binary_crossentropy” loss function. For non-binary classification, we use the “categorical_crossentropy” loss function when we are predicting more than two categories.
- “**Matrix**” =>> “accuray”

Let's check how the code looks like:

```
#compiling the ANN
ann.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])
```

b) Training the ANN on the training set.

We have to use the "fit" method to train out ANN. Here we have to enter two main parameters:

- "**Number of epochs**" => Forward-propagation and Backward-propagation happens over many epochs and over each epoch the loss functions is slightly reduced. Therefore, we want to repeat these epochs in order to reduce more little by little the loss function. By default, we can use 100 epochs.
- "**Batch size**" => Instead of propagating all the features one by one, we propagate them in batches of a certain number of elements of a certain sets of the features. By default, we can use the 32 in the batch size.

When we apply backward-propagation, we can adjust the weights of the connections between the neurons. With this action, the loss function approaches 0 the next time that we will use the ANN for a prediction.

In order to get it, we need to use the "gradient descent". It will change the weights in small increments with the calculation of the derivate (gradient) of the loss function, which allow us to see the descent direction until the global minimum. This calculation is done in batches in the following interactions (epochs) of the data that we are sending along the ANN.

Let's check how the code looks like:

```
#training the ANN
ann.fit(X_train, y_train, batch_size = 32, epochs = 100)
```

```
Train on 8000 sampl
Epoch 1/100
8000/8000 [==                                                    0.4953 - accuracy: 0.7960
Epoch 2/100
8000/8000 [==                                                    .4560 - accuracy: 0.7960
Epoch 3/100
8000/8000 [==============================] - 0s 40us/sample - loss: 0.4438 - accuracy: 0.7960
Epoch 4/100
8000/8000 [==============================] - 0s 48us/sample - loss: 0.4388 - accuracy: 0.7960
```

We train the model in a numpy array form.

- **Batch size** =>> Number of data that we will use in each interaction, when we adjust the weights of the model. Partition of training data into small batches to through it over the ANN. The size depends on several factors, such as the computer memory capacity.
- **Epochs** =>> Number of times that we will use all the data in the learning process. The more epochs, the better we will adjust the "accuracy" metrics with the validation data.

The output of the training code continues until achieving the epoch 100/100.

3.1.5. Step 4 – Making predictions and evaluating the model

a) One single observation

We need to scale one single observation using the "*sc.transform*" method:

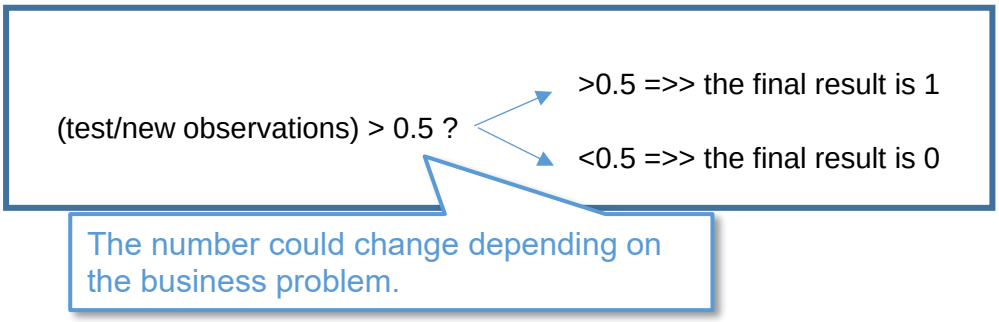
```
#predicting the result of a single observation
#predicted probability
print(ann.predict(sc.transform([[1, 0, 0, 600, 1, 40, 3, 60000, 2, 1, 1, 50000]])))
```

Important: Here we cannot use "fit.transform" in the test set/ new observations on which you deploy your model in production, because it is used in the training sets.

So, for the test set, we only need to call the "transfer" method because if we fit again the scaler, it would case some irregular information.

As we selected the "sigmoid activation function" in the output layer, we can get results in two different ways:

- Outcome in a probability form =>> it is a predicted probability
- Outcome in a non-probability form =>> enter "> 0.5" to convert it into the final prediction.



Important: We have to apply scaling from the ann object because the ANN is trained with scaled values.

b) Check the results of the Test Set

We have to use the "predict" method in the test set. We will get all these predictions of the test in a new vector, which is called the ***y_pred***.

Let's check how the model predict the test results:

```
#predicting the absolute results of the test set
y_pred = ann.predict(X_test)
y_pred = (y_pred > 0.5) #show the results in a non-probability form
print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))
```

```
[[0 0]
 [0 1]
 [0 0]
 ...
 [0 0]
 [0 0]
 [0 0]]
```

Comments =>> In general, the results are good!

3.2. Artificial Neural Network for Regression

3.2.1. Business Case

The goal of this study case is to implement an Artificial Neural Network (ANN) Regression model using tensorflow and python to predict the electrical energy output of a Combined Cycle Power Plant.

3.2.2. Step 1 – Data Pre-processing

We are going to build a deep neural network with neurons and fully connected layers connecting these neurons. In order to get it, we will use the TensorFlow 2.0. library, which is available in Python.

In this case we will apply ANN for regression.

a) Importing the required packages & Loading the dataset

Firstly, we will import the required libraries and load the dataset.

```
import numpy as np
import pandas as pd
import tensorflow as tf
```

With the following script, we will check what version of the tensorflow library we are using:

```
#check the tensorflow version that we are using
tf.__version__
```

```
'2.1.0'
```

Secondly, let's load the dataset and visualize the information:

```
dataset = pd.read_excel('Folds5x2_pp.xlsx')
dataset.head()
```

	AT	V	AP	RH	PE
0	14.96	41.76	1024.07	73.17	463.26
1	25.18	62.96	1020.04	59.08	444.37
2	5.11	39.40	1012.16	92.14	488.56
3	20.86	57.32	1010.24	76.64	446.48
4	10.82	37.50	1009.23	96.62	473.90

Descriptions of the variables of the dataset:

AT: Temperature

V: Pressure

AP: Humidity

RH: Vacuum

PE: Electrical Energy =>> the output value

b) Splitting the data into training and test

Let's check how the code looks like:

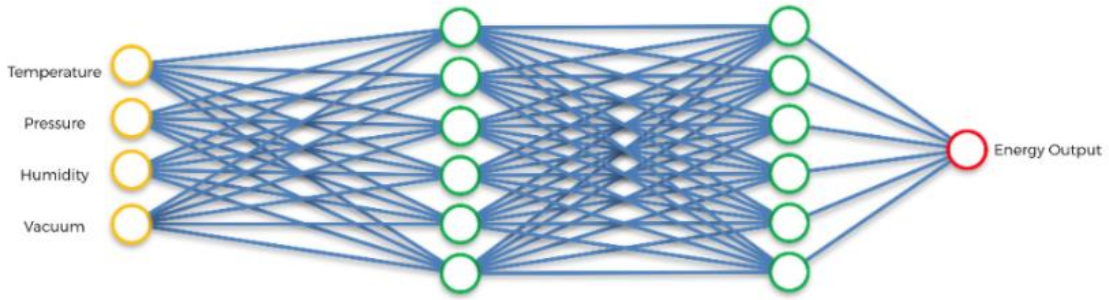
```
#getting the training set and test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)
```

Important Note =>> It is not necessary to apply Feature Scaling for regression models!

3.2.3. Step 2 – Building the ANN

a) Initializing the ANN.

Before building the ANN, let's check its architecture:



We can identify the input layer (the line composed of yellow neurons), which contains the independent variables (temperature, pressure, humidity and vacuum). Then, we have the first hidden layer with 6 hidden neurons and the second hidden layer with 6 hidden neurons as well. Finally, we have the output layer, which contains only one output neuron because we have just to predict one real value (the energy output).

Let's initialize or activate the ANN as a sequence of layers. To do it, we have to create an object of the sequential class, which allows us to build the ANN. The sequential class comes from the Keras module of TensorFlow 2.

The ANN is actually a sequence of layers, which starts from the input layer and then we have hidden layers fully connected until the output layer.

Let's check how the code looks like:

```
#create the object for the ANN
ann = tf.keras.models.Sequential()
```

b) Adding the Input layer and the first hidden layer.

The way to add a fully connected layer into an ANN at whatever phase you are is using the "*dense class*". So, we have to take our object and call the "add" method of the sequential class.

The layers will be created as objects of a new class, which is the "Dence class" (it belongs to the Keras sub-library of tensorflow).

How many hidden neurons do we need to enter? By default, we can choose the option of 6 hidden neurons.

Do we need to enter the number of neurons in the input layer? => NO, because the features will be recognized automatically by tensorflow. So, once we included the matrix

of features in the training, the ANN will automatically collect these four features. Therefore, we do not need to specify that we have 4 features.

Regarding the activation function, we will use the “rectifier” function, which will break the linearity of the operations happening between this input layer and the first hidden layer.

Let’s check how the code looks like:

```
#add a simple connected layer
ann.add(tf.keras.layers.Dense(units=6, activation='relu'))
```

c) Adding the second hidden layer.

For the second hidden layer, we will use the same code as the first hidden layer:

```
#add the second hidden layer
ann.add(tf.keras.layers.Dense(units=6, activation='relu'))
```

d) Adding the output layer.

We want an output layer fully connected to that second hidden layer and we need to use the “dense class” again, but we need to change some parameters to build the output layer.

We will include only one output neuron here because we want to predict one real value.

Regarding the activation function, it is recommended to include the sigmoid function for classification cases. However, for regression we keep the output layer without any activation function.

Let’s do a recap:

- If you are doing classification with only two categories to predict in the output (predict a YES/NO or 0/1), you have to choose a sigmoid activation function.
- If you are doing classification with more than 2 categories or classes to predict the output, you should use a soft max activation function.
- If you are doing regression, you have to predict a continuous real number as a final output, then you should choose any activation function.

Let’s check how the code looks like:

```
#add the output layer
ann.add(tf.keras.layers.Dense(units=1))
```

3.2.4. Step 3 – Training the ANN

a) Compiling the ANN.

We are going to connect our ANN with an optimizer and loss function.

The optimizer is the tool with which we will perform stochastic gradient descent, a technique that consists on updated the weights of each of the neurons in the hidden layers to reduce the loss function over the epochs.

Let’s explain the process step by step:

First, we perform forward propagation (the signal will be forward propagated up to the energy output). So, the energy output/prediction will be compared to the real energy

output of the training set. This incur into a loss function, which will be exactly the squared difference between the predicted energy output and the real energy output.

The predictions come into batches (we actually compute the sum of the squared differences between the predicted energy output and the real energy outputs in the batch).

Once we get a loss, backward propagation is applied. The loss is back propagated into the neural network and stochastic gradient is applied with the optimizer to reduce the loss. The optimizer will update all the weights inside this new networks in order to reduce the loss function.

Now, let's connect the ANN with the optimizer and the loos function. We will use the "compile" method.

- ***Optimizer*** =>> "adam", which is the most used in stochastic gradient descent. The adam optimizer will reduce the difference between the predicted values and the real values. We can use this both for regression and classification.
- ***Loss function*** =>> "mean squared error" or "root mean squared error", as we are facing a regression issue.

The "mean squared error" calculates the sum of the squared differences between the predictions and the real energy output in the batch.

Let's check how the code looks like:

```
#compiling the ANN
ann.compile(optimizer = 'adam', loss = 'mean_squared_error')
```

b) Training the ANN on the training set.

Let's use the fit method in the training sets to train our ANN.

Here we need to specify the options for two main parameters:

- ***Batch size*** =>> Instead of propagating all the features one by one, we propagate them in batches of a certain number of elements of a certain sets of the features. By default, we can use the 32 in the batch size.
- ***Epochs*** =>> Forward-propagation and Backward-propagation happens over many epochs and over each epoch the loss functions is slightly reduced. Therefore, we want to repeat these epochs in order to reduce more little by little the loss function. By default, we can use 100 epochs.

Now, we are goint to execute each of the cells one by one. At the end of this operation, we will see that the loss function getting reduced epoch by epoch.

Let's check how the code looks like:

```
#training the ANN
ann.fit(X_train, y_train, batch_size = 32, epochs = 100)
```

```
Train on 7654 samples
Epoch 1/100
7654/7654 [==============================] - 1s 118us/sample - loss: 5752.9360
Epoch 2/100
7654/7654 [==============================] - 0s 36us/sample - loss: 261.9719
Epoch 3/100
7654/7654 [==============================] - 0s 44us/sample - loss: 247.6824
Epoch 4/100
7654/7654 [==============================] - 0s 57us/sample - loss: 230.4209
Epoch 5/100
7654/7654 [==============================] - 0s 42us/sample - loss: 210.9364
Epoch 6/100
7654/7654 [==============================] - 0s 38us/sample - loss: 189.9985
Epoch 7/100
7654/7654 [==============================] - 0s 51us/sample - loss: 168.3451
Epoch 8/100
7654/7654 [==============================] - 0s 42us/sample - loss: 147.1466
Epoch 9/100
7654/7654 [==============================] - 0s 40us/sample - loss: 126.7308
Epoch 10/100
```

The output of the training code continues until achieving the epoch 100/100.

- **Batch size** =>> Number of data that we will use in each interaction, when we adjust the weights of the model. Partition of training data into small batches to through it over the ANN. The size depends on several factors, such as the computer memory capacity.
- **Epochs** =>> Number of times that we will use all the data in the learning process. The more epochs, the better we will adjust the "accuracy" metrics with the validation data.

3.2.5. Step 4 – Predicting the results of the Test set

We have to use the "predict" method in the test set. Then, we will get all these predictions of the test in a new vector, which is called the ***y_pred***. Finally, we can compare the ***y_pred*** with the real results (y_test).

Let's check how the code looks like:

We use the numpy function to stablish the number of decimals after the comma in the predictions and in the real results, 2 in this case. Otherwise, we will get a huge number of decimals.

```
y_pred = ann.predict(X_test)
np.set_printoptions(precision=2)
print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))
```

```
[[429.3  431.23]
 [460.33 460.01]
 [463.84 461.14]
 ...
 [471.06 473.26]
 [437.89 438.  ]
 [457.09 463.28]]
```

We use the "reshape" function to display the output results vertically.

We use the "concatenate" function to display at the same time the ***y_test*** and ***y_pred*** results and make comparisons.

Comments: **We observe that the predictions are quite close to the real results!**