Artificial Neural Networks: Behind The scenes....

Hands on Example: Building Artificial Neural Network from A to Z

(without coding: you just need a paper & pencil !)

All the Algorithms (with a lot of **beautiful visuals**)

Forward propagation, Computing the Total Error, Backpropagation and Updating the weights

(Mainly in the Backpropagation: I present a slight overview on the **"math**" behind artificial neural networks)

By Abir ELTAIEF

An **artificial neural network** (or simply neural network) is a predictive model that reproduces the **brain function**! The brain is seen as a collection of neurons connected to each other. Each neuron examines the outputs of the other neurons, which become its inputs, performs a "computation" then is triggered or not (...). Likewise, artificial neural networks (or just neural networks) are made up of artificial neurons that perform the same kind of "calculations" on their inputs(...).

Neural networks can **solve many problems** such as **facial recognition, fraud detection** , **tumor detection**, **performing complex tasks** aimed at **saving ecosystems** (**impacting water, environment, diversity, etc**.), **optimization of a company's processes and resources..**. It is one of **the most promising fields of Data Science**.

Before moving forward, I want to ask an "open" question: **Are algorithms stronger than humans?**

My answer is: **Loud NO** (you will understand why, if you take a paper and a pencil and follow what I am going to do).

Why this paper ? What is its added value?

Admittedly there are too **many scientific papers** on artificial neural networks (you can search on the net, yourself, mainly if you're a data scientist or machine learning engineer: you have **certainly** read tons of papers on this). But imagine ! There are **no papers** which give examples of real "**hand computation**" of these **fascinating algorithms** (behind artificial neural networks).

In fact the advantage of this **practical example**, that I will present, is to show you the **"BEHIND THE SCENES"** of theses algorithms, plunges you into **the brains of those who built them**, and then allows you to appropriate these algorithms (so, they will no longer be "**black boxes**", just ready to use or build on them!):

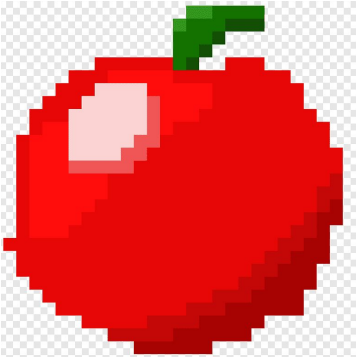
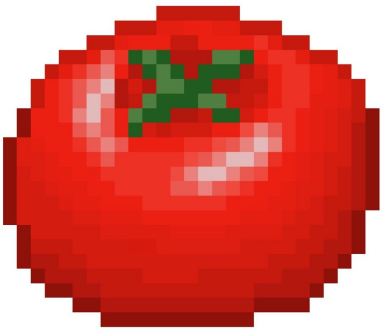
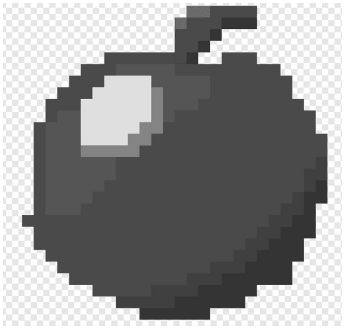
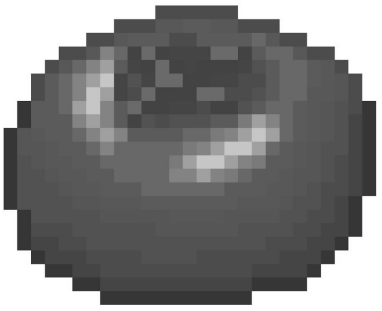
My ultimate goal is to be crystal clear and engaging, but with a touch of fun and uniqueness !

This paper includes **5 steps**:

Step1: The concept of this practical example (building the neural network structure)

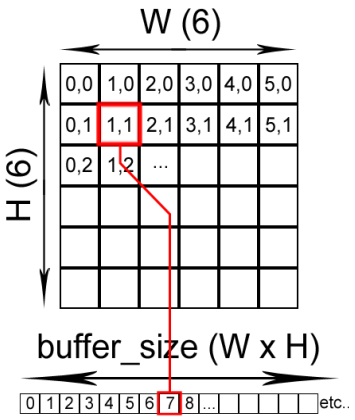
I built a **very simple** artificial neural network (using and applying all the concepts and algorithms "behind the scenes").
I built it to be, **hopefully**, **very intuitive!** (easy to reproduce by any human **using paper and pencil)**, but above all to allow anyone to understand **"the why**" of each step (**behind the algorithms of artificial neural networks**…).

1-1 The Scenario (Classification problem: image recognition): On a high level, we are going to build a neural network that can distinguish between **two 6x6 grayscale** pixel characters: **red apple** or **tomato** (Just for fun: These are my favorite fruit and vegetable :) :))

<table>
<tr>
<td> 
Red Apple (in pixels)</td>
<td> 
Tomato(in pixels)</td>
</tr>
<tr>
<td> 
grayscale pixels</td>
<td> 
grayscale pixels</td>
</tr>
</table>

Highly Important Note:
Convolutional neural networks (CNN's), and **not traditional feedforward networks** (like the one we're going to build), are typically used for **image classification**. CNN's performance is **superior** in this regard, which is why major technology companies such as Google and Facebook use CNN's to classify images. However, images provide a **fantastic visua**l that can help in this practical example, **which is why I will be using them!**

***** Based on our simplified example: each image, <u>will contain data</u> like this :**



<u>1-2 The Structural elements of our simple artificial neural network:</u>

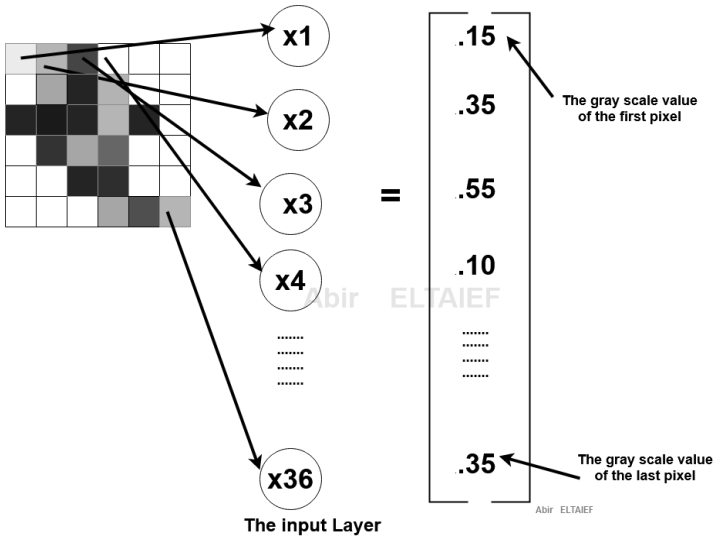
we need to define a number of structural elements.

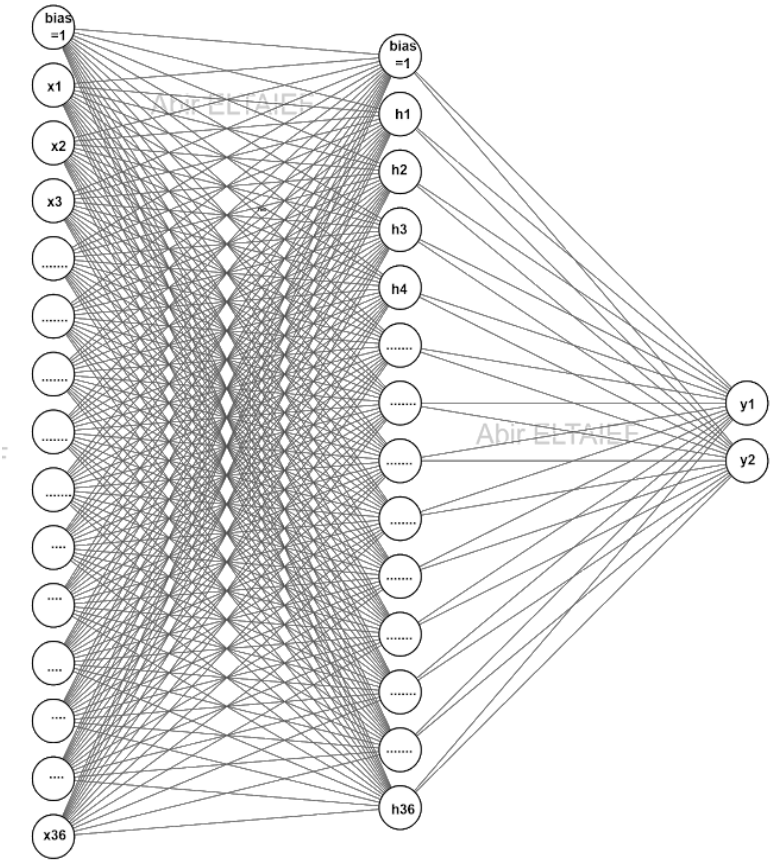
- **<u>Total input nodes</u>:** 36. This number is derived from each image, which **is 6x6 pixels**.
- **<u>Total hidden layers</u>**: 1. We will follow **Stanford's <u>Andrew Ng</u>'s** recommendation , which is to begin with a single hidden layer.
- **<u>Total hidden nodes</u>**: 36. It is a good place to start (...): the same number of hidden layers.
- **<u>Total output nodes</u>**: 2. Due to the fact that we are classifying two elements we will use <u>two output nodes</u>. Each node will represent a unique class<u>: red apple or tomato</u>. If we pass an image of a **<u>red apple</u>** through the network, our goal is for the output node that represents the red apple *<u>class to output a "1</u>*", and the tomato class output a 0 (zero).
- Likewise, if we pass an image of a **<u>tomato</u>** through the network, our goal is for the output node that represents the tomato *<u>class to output a "1"</u>*, and the red apple class output a 0 (zero). it is called "**One-hot encoding method** "
- **<u>Bias value</u>**: 1: (this is common practice…Don't worry too much about it yet.)
- **<u>Weight values</u>:** Random. We <u>will assign random values to begin</u>.
- **<u>Learning rate</u>:** 0.5. We will begin with an initial learning rate of 0.5. This number is somewhat arbitrary, and <u>it can be changed as the network learns</u>...

<u>1-3 The artificial neural network Architecture:</u>

Before transitioning to a simplified version of our network, we will take a closer look at the input, hidden and output layers. As you see, below (second picture), we have 36 inputs (features), so 37x36 (interior weights between input layer and hidden layer, 37 because we include the bias weight) + 37 x2 (output weights between output layer and the last hidden layer, in our example we have only one hidden layer), so we have **1406 weights (in total) ! ,** with an already <u>simple neural network</u> with only **one hidden layer** (and only **2 outputs** because it is a <u>simple problem of binary classification</u>).

Note: for each image of our "**hypothetical training dataset"(that our neural network should be able to classify),** the inputs are a vector like this(in the first picture, below) and it is the case for each image(=each instance of our dataset)), so if our dataset contains **"m"** images, we will compute **"m" artificial neural networks(like you see in the picture below: it is only a neural network for only one instance!)**, with "**n**" inputs, for each instance(image)! (in our case n= 36 (features), by image of "m" images in the dataset, m= 10.000 images for example).

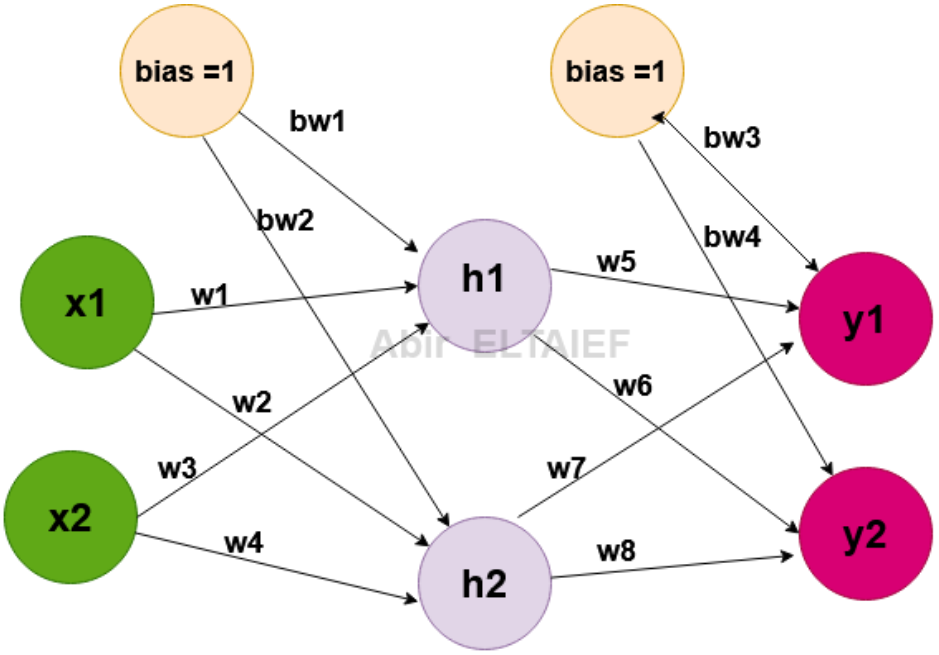




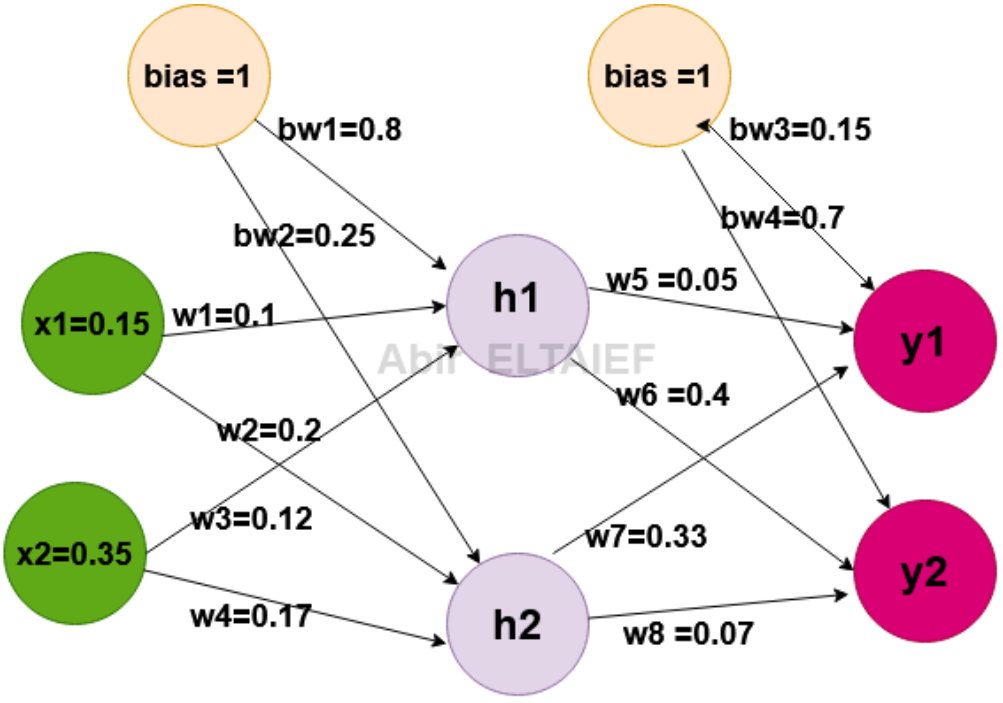
Our Artificial neural network architecture (input layer(36 inputs), hidden layer and output layer(for only one image)

1-4 Simplifying our network (with only two features(inputs)):

We will **imagine** that our image can be classified using **only two input nodes** and **two hidden nodes**. To do this : we will continue using the input values we introduced earlier for x1 and x2. These inputs have values of 0.15 for x1 and 0.35 for x2.



The simplified artificial neural network with two inputs, one hidden layer(2 hidden neurons), and an output layer with two neurons



The iniitial values of weights (without any treatment, Ichoose them randomly(...)

=> In this step1, we created our network structure, which includes the number of **input nodes**, **hidden nodes/layer**, **output nodes**, **bias values**, **weight values**, and the **initial learning rate(=0.5)**, this learning rate is somewhat arbitrary and it can be changed as the network learns.
We also concluded the following:

- If the input is a red apple , we want the output to be a vector of [1, 0].
- If the input is a tomato, we want the output to be [0, 1].
- We also show how an image is broken down, pixel, by pixel, and stored within an input vector.
- Lastly, we **simplified the structure** so that it can be used for purely **practical reasons** to be able to do it **by hand** (but the real network is far more complex)

Step2: Forward propagation

In this step, we will run our input data through the neural network, this step includes two sub-steps:

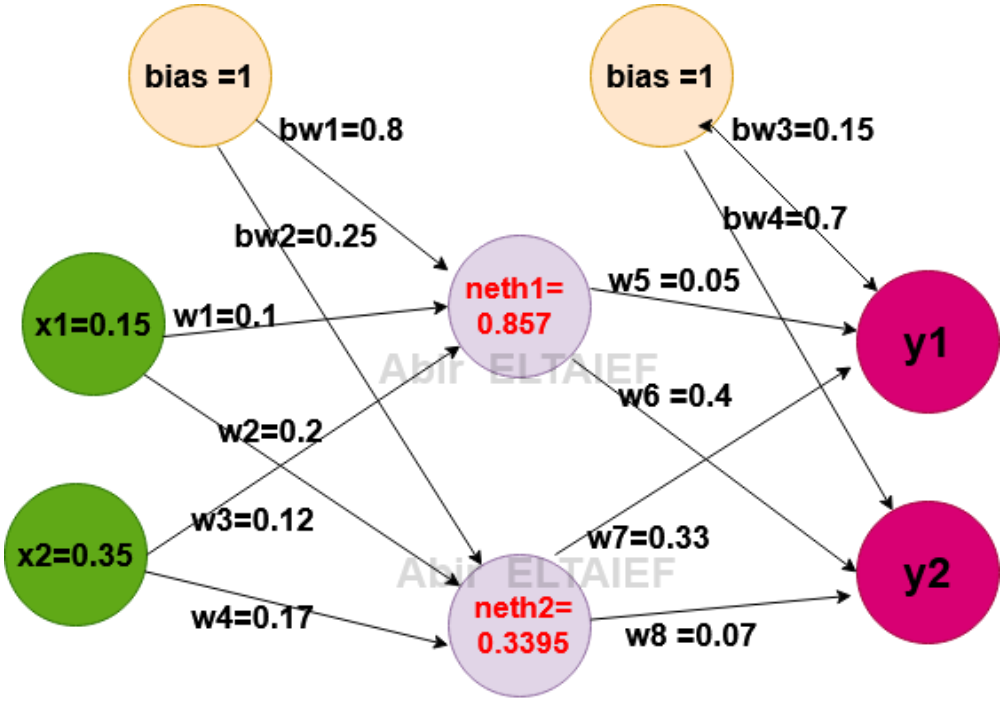
- **Pat1**: Moving From the Input Layer to the Hidden Layer (Computing net hidden nodes + Applying activation function)
- **Part2**: Moving From the Hidden Layer to the Output Layer (Computing net output nodes + Applying activation function)

Part1: Moving From the Input Layer to the Hidden Layer:

Computing net hidden nodes

Let's begin by calculating the net input to hidden nodes h1 and h2, which we will label as **neth1** and **neth2**, respectively. (**Recall**, **neth1**= sum(xi.wi) + bias with i ∈ [1,n], n=2 in our simplifying example).

-- **neth1**= (0.15 x 0.1)+ (0.35 x 0.12) + 0.8 = **0.857**
--n**eth2** = (0.15 x 0.2)+(0.35 x 0.17) + 0.25 = **0.3395**
=> so our neural networks becomes:

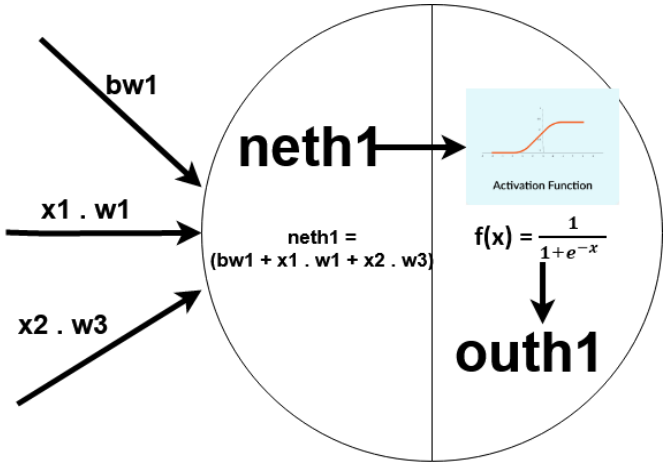


Applying activation function

To move our values from the hidden layer to the *output layer* (part3 below), we must first apply activation function to our **net input values**.

This will provide us with the **output for our hidden nodes**, which we can, then move forward along their respective edges.

Recall: Here is a picture of a node to remind you of how the summation operator and activation function (sigmoid) work together:



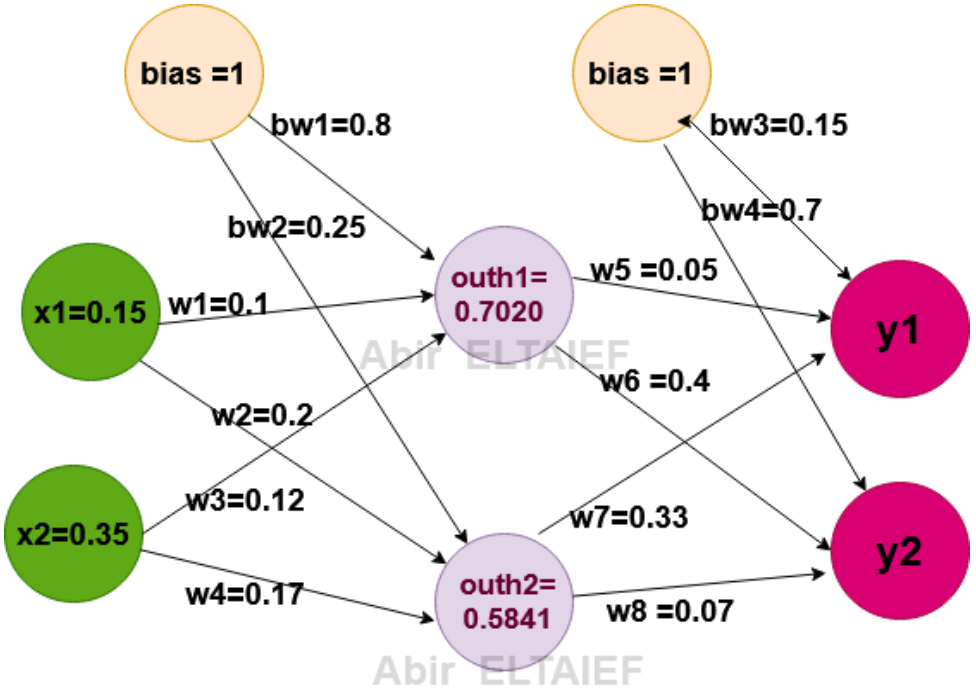
by applying the activation function (sigmoid, called also logistic function):

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

-- **outh1**= **f(neth1)**= 1/(1+ exp(-neth1))= (1/(1+ exp(-0.857))= **0.7020**

-- **outh2**= **f(neth2)**= 1/(1+ exp(-neth2))= (1/(1+ exp (-0.3395))= **0.5841**

=> so our neural networks becomes:



Part2: Moving from the Hidden layer to the Output layer:

All we need to do now is move this output of node h1 and h2 to nodes y1 and y2. To do this, we will follow the same steps as I did previously when I moved from the input layer to the hidden layer. Since I have already explained the steps involved, I will only show the calculations.

******computing net output nodes y1 and y2***

--**nety1**= (outh1 x w5) + (outh2 x w7) + bw3= (0.7020 x 0.05)+(0.5841x 0.33)+0.15= **0.3779**

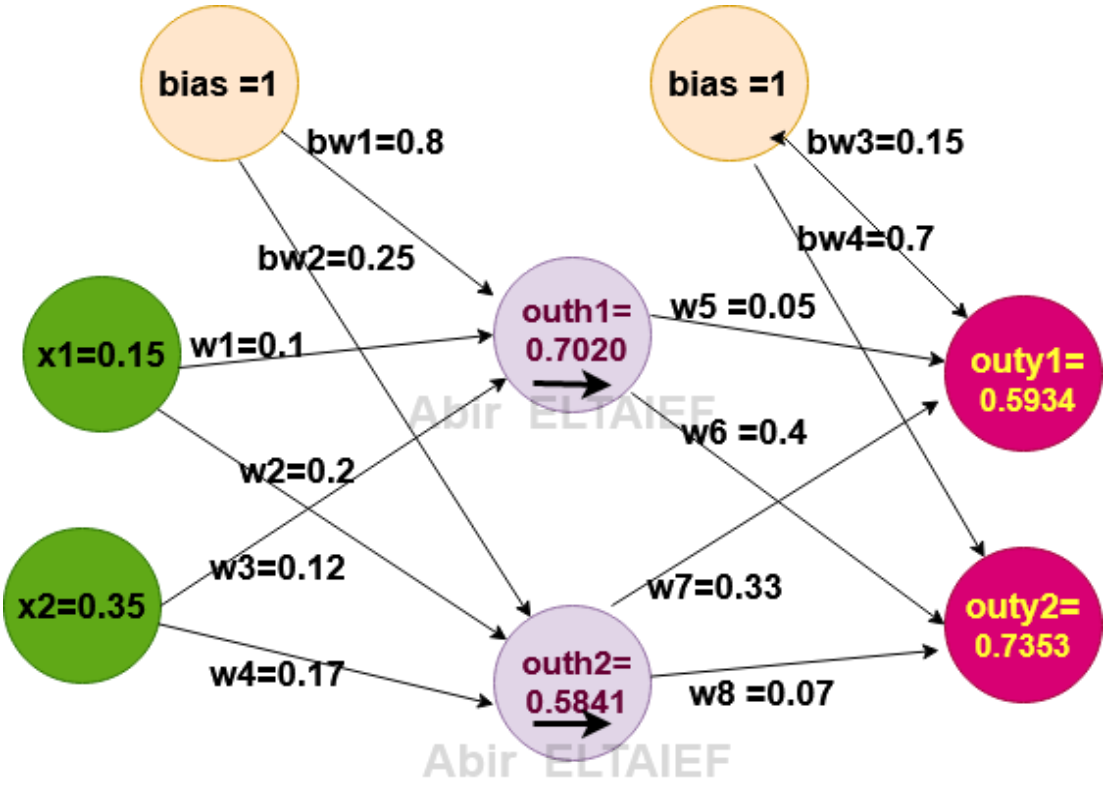
--nety2 =(outh1 x w6) + (outh2 x w8) + bw4= (0.7020 x 0.4)+(0.5841x 0.07)+0.7= **1.0217**

******Applying activation function***

--**outy1**= f(nety1) = 1/(1+ exp(-nety1))= (1/(1+ exp(-0.3779))= **0.5934**

--outy2 = f(nety2)= 1/(1+ exp(-nety1))= (1/(1+ exp(-1.0217))= **0.7353**

=> so our neural networks becomes:



To wrap up: In this **step 2,** we moved our input through the network to create a final output: using The Forward propagation.

Step3: Calculating the total error

This step3 is brief: we will apply the cost function to the actual output of calculating the total error: the mean squared errors:

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$

with **yi**: the actual value of class image (the real value), and **yi (with hat**), is the predicted value of our target by the neural network.
Note: this formula compute the total error for all the images (all the instances of our dataset), and since we are doing the work for only one image (and since also we have two outputs), the formula (for our simplifying example), becomes:

SE= ½ ((y1 - out y1)² + (y2 - outy2)²)

In our case, our training example is an image of **a tomato**, so our target output is a vector with a value of 0 (zero) for node y1 and value of 1 for node y2

=> **SE=** 0.5 x ((0-0.5934) ²+(1-0.7353)²)= **0.21105 (it is the total error, but if it is for all the instances (images), of our dataset,). All the work of optimization of artificial networks is to reduce this total error!**

Step4: Backpropagation (Calculating the gradients):

******Understanding the mathematics behind the backpropagation!***

There are **two** mathematical functions used in step4. These functions are used together to calculate the gradient of every weight in a neural network:

- Partial derivatives.
- The Chain Rule.

Recall:

- **A partial derivative** is the derivative of a function which has two or more variables but **with respect to only one variable**(all the other variables are treated as constant). For example, the partial derivative of the total error with respect to w1 (the first weight), means: how **a change in weight w1**, **affects the total error** (while all the other weights remain constant).
- A slight change in a single weight (example w1), will affect all variables t**hat occur after it** within the network (the nodes net inputs, the other weights, the nodes outputs,etc).
- The larger the network, the greater the impact of a change.

=> Given **the ramification of a slight change**, calculating how a specific weight impacts the total error is **challenging! :** Partial derivatives are the answer to this challenge.

When calculating **partial derivatives**, it is important to realize that there are **four different types of weights**:

- ***Part 1***: **Weights on the outside** of the neural network situated between a hidden node and output node (w5, w6, w7 and w8),
- ***Part 2***: **Weights on the inside** of the neural network situated between input/hidden nodes (w1, w2, w3 and w4),
- ***Part 3*** : **Bias weights on the outside** of the neural network situated between a hidden node and output node (bw3 and bw4),
- ***Part 4*** : **Bias weights on the inside** of the neural network situated between input/hidden nodes (bw1 and bw2).

Part1: Calculating the gradients for the weights on the outside (w5,w6,w7 & w8):

*****we start with w5:*

we apply *the chain rule*, to *unpack* the *partial derivative of the* ***total error Etotal*****,** with respect to the weight (on the outside), w5:

******* First unpacking:***

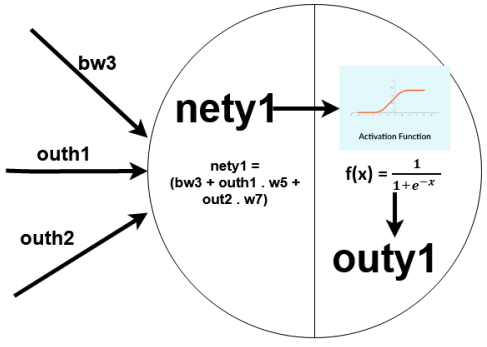
∂Etotal / ∂w5 = (∂Etotal / ∂outy1) x (∂outy1/ ∂w5)

- *with* ***outy1*** *is the* ***output value of the node y1 (in our example)***
- *In fact the total* ***error Etotal****, depends, first on* ***the outputs*** *(the formula of the total error shows it clearly), then the* ***outputs*** *depend on* ***the weights*** *(and the Forward propagation (step 3) shows it clearly): This is what explains this first unpacking!*

******* Second unpacking:***

∂Etotal / ∂w5 =(∂Etotal / ∂outy1) x(∂outy1/ ∂nety1) x(∂nety1/ ∂w5)

- *with* ***nety1*** *is, the* ***net input*** *of the "output node y1" (before applying the activation function that transforms it in final output y)*



Etotal = Ey1 +Ey2

Ey1 = ½ (targety1 - outy1)² with targety1: the desired(real) value that we want our neural network to predict.

=> Etotal = ½ (targety1 - outy1)² + ½ (targety2 - outy2)²

******* Computing the three partial derivative:***

- **First partial derivative:**

∂Etotal / ∂outy1 = 2 x (½) x (targety1 - outy1) x(-1) + 0

∂Etotal / ∂outy1 = (outy1 - targety1) = 0.5934 - 0 = 0.5934

- **Second partial derivative:**

∂outy1/ ∂nety1= ?

Here, the question is how much does a change in **nety1** (the net input to y node), affect the final output y1: outy1?
So, it is simple, we look at the function that transforms the **nety1** to the **outy1**: the **activation function.**

outy1= 1/(1+ exp(-nety1)) = f(nety1)

without going into details (you can do it yourself), the partial derivative of the sigmoid (logistic function) : f is :

∂(f(nety1))/ ∂nety1= f(nety1) . (1 - f(nety1))
= outy1 (1-outy1) = 0.5934 *(1 - 0.5934) = 0.2413

- **Third partial derivative**

It is simple:

∂nety1/ ∂w5 = outh1 = 0.7020

=> ∂Etotal / ∂w5 = (outy1 - targety1) x outy1 (1-outy1) x outh1
= 0.5934 x 0.2413 x 0.7020 = 0.1005

*******we continue the same treatment with w6:***

in this case, y2 depends on w6 (not y1), see our neural network, above: so the formula becomes:

∂Etotal / ∂w6 =(∂Etotal / ∂outy2) x(∂outy2/ ∂nety2) x(∂nety2/ ∂w6)
= (outy2 - targety2) x outy2 (1-outy2) x outh1
= (0.7353 - 1) x 0.7353 x (1- 0.7353) x 0.7020 = - 0.0362

*******we continue the same treatment with w7:***

in this case, y1 depends on w7 (not y2), see our neural network, above: so the formula becomes:

∂Etotal / ∂w7 =(∂Etotal / ∂outy1) x(∂outy1/ ∂nety1) x(∂nety1/ ∂w7)
= (outy1 - targety1) x outy1 (1-outy1) x outh2
= (0.5934-0) x 0.5934(1-0.5934) x 0.5841 = 0.0836

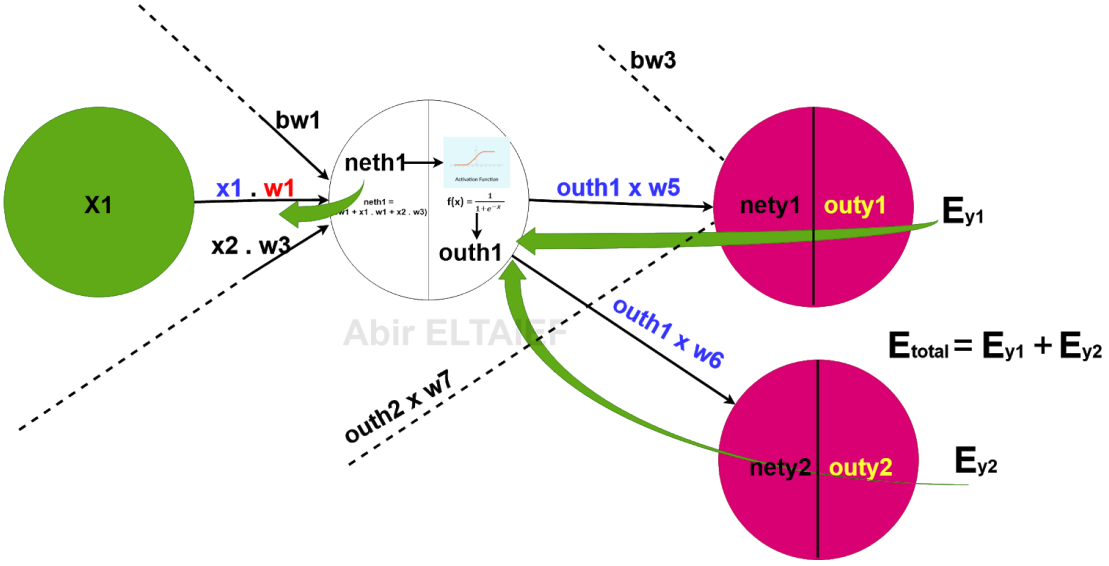
*******we continue the same treatment with w8:***

in this case, y2 depends on w8 (not y1), see our neural network, above: so the formula becomes:

∂Etotal / ∂w8 =(∂Etotal / ∂outy2) x(∂outy2/ ∂nety2) x(∂nety2/ ∂w8)
= (outy2 - targety2) x outy2 (1-outy2) x outh2
= (0.7353 - 1) x 0.7353(1-0.7353) x 0.5841 = -0.0301

Part2: Calculating the gradients for the weights on the inside (w1,w2,w3 & w4):
*****we start with w1:***

For the **weights on the inside** of the neural network: the treatment is slightly different, because every hidden neuron contributes to the **two outputs** (of the two output neurons y1 and y2), not only one output neuron as it was the case for the weights on the outside of the network(lik w5,w6,etc).
below, **a picture highlighting t**his reality:



∂Etotal / ∂w1 =(∂Etotal / ∂outh1)x(∂outh1/ ∂neth1) x(∂neth1/ ∂w1)
we will calculate the three partial derivative, each separately

***** First unpacking (First partial derivative):***

∂Etotal / ∂outh1 = (∂Ey1 / ∂outh1) + (∂Ey2 / ∂outh1)
***First term:
∂Ey1 / ∂outh1 = (∂Ey1 / ∂outy1) x (∂outy1 / ∂outh1)
= (∂Ey1 /∂outy1) x (∂outy1 /∂nety1) x (∂nety1 /∂outh1)
= (outy1 - targety1) x outy1(1-outy1) x w5
= (0.5934 - 0) x 0.5934 x(1- 0.5934) x 0.05 = 0.1432 x 0.05 = 0.0072

***Second term:
∂Ey2 / ∂outh1 = (∂Ey2 / ∂outy2) x (∂outy2 / ∂outh1)
= (∂Ey2 /∂outy2) x (∂outy2 /∂nety2) x (∂nety2 /∂outh1)
= (outy2 - targety2) x outy2(1-outy2) x w6
= (0.7353 - 1) x 0.7353 (1-0.7353) x 0.4
= -0.0206

******* Second unpacking (Second partial derivative):***

∂outh1/ ∂neth1 = outh1 x (1-outh1) = 0.7020 x (1 - 0.7020) = 0.2092

******* Third unpacking (Third partial derivative):***

∂neth1/ ∂w1= x1 = 0.15

=> ∂Etotal / ∂w1= (0.0072 + (-0.0206)) x 0.2092 x 0.15 =
= -0.0134 x 0.2092 x 0.15 = - 0.000420492

According to the calculations made previously for w1, the formula for calculating the gradients for the interior weights (weights on the inside of the neural network) is:

∂Etotal / ∂w1 =
[(outy1 - targety1)x outy1(1-outy1) x w5 +(outy2 - targety2) x outy2(1-outy2) x w6] x outh1 x (1-outh1) x x1

*******we continue the same treatment with w2:***

According to the formula above (for calculating partial derivative error with respect to interior weight w1):

∂Etotal / ∂w2 =(∂Etotal / ∂outh2)x(∂outh2/ ∂neth2) x(∂neth2/ ∂w2)
=[(outy1- targety1) x outy1(1-outy1) x w7 + (outy2- targety2) x outy2(1-outy2) x w8] x outh2 x (1 -outh2) x x1
= [(0.5934-0) x0.5934 x (1- 0.5934) x 0.33 + (0.7353-1) x 0.7353 (1- 0.7353) x 0.07] x 0.5841 x (1- 0.5841) x 0.15
= 0.0437 x 0.2429 x 0.15
= 0.00159221

*******we continue the same treatment with w3:***

According to the formula above (for calculating partial derivative error with respect to interior weight w1):

∂Etotal / ∂w3 =(∂Etotal / ∂outh1)x(∂outh1/ ∂neth1) x(∂neth1/ ∂w3)
=[(outy1- targety1) x outy1(1-outy1) x w5 + (outy2- targety2) x outy2(1-outy2) x w6] x outh1 x (1 -outh1) x x2
= -0.0134 x 0.2092 x 0.35
= - 0.000981148

*******we continue the same treatment with w4:***

According to the formula above (for calculating partial derivative error with respect to interior weight w1):

∂Etotal / ∂w4 =(∂Etotal / ∂outh2)x(∂outh2/ ∂neth2) x(∂neth2/ ∂w4)
= [(outy1- targety1) x outy1(1-outy1) x w7 + (outy2- targety2) x outy2(1-outy2) x w8] x outh2 x (1 -outh2) x x2

= 0.0437 x 0.2429 x 0.35
= 0.003715156

Part3: Calculating the gradients for the bias weights on the outside (bw3 & bw4):

*******we start with bw3:***

∂Etotal / ∂bw3 = (∂Etotal / ∂outy1) x (∂outy1/ ∂nety1) x (∂nety1/ ∂bw3)
= (outy1 - targety1) x outy1 (1- outy1) x 1
= 0.1432

*******we continue the same treatment with bw4:***

∂Etotal / ∂bw4 = (∂Etotal / ∂outy2) x (∂outy2/ ∂nety2) x (∂nety2/ ∂bw4)
= (outy2 - targety2) x outy2 (1- outy2) x 1
= -0.0515

Part4: Calculating the gradients for the bias weights on the inside (bw1 & bw2):

*******we start with bw1:***

∂Etotal / ∂bw1 = (∂Etotal / ∂outh1) x (∂outh1 / ∂neth1) x (∂neth1 / ∂bw1)
=[(outy1- targety1) x outy1(1-outy1) x w5 + (outy2- targety2) x outy2(1-outy2) x w6] x outh1 x (1 -outh1) x 1
= -0.0134 x 0.2092
= - 0.0028

*******we continue the same treatment with bw2:***

∂Etotal / ∂bw2 = (∂Etotal / ∂outh2) x (∂outh2 / ∂neth2) x (∂neth2 / ∂bw2)
= [(outy1- targety1) x outy1(1-outy1) x w7 + (outy2- targety2) x outy2(1-outy2) x w8] x outh2 x (1 -outh2) x 1

= 0.0437 x 0.2429
= 0.0106

Part4: let's wrap it up: all the gradients

∂Etotal / ∂w1= - 0.000420492	∂Etotal / ∂w2= 0.00159221	∂Etotal / ∂w3= -0.000981148	∂Etotal / ∂w4= 0.003715156
∂Etotal / ∂w5= 0.1005	∂Etotal / ∂w6= - 0.0362	∂Etotal / ∂w7= 0.0836	∂Etotal / ∂w8= -0.0301
∂Etotal / ∂bw1= - 0.0028	∂Etotal / ∂bw2= 0.0106	∂Etotal / ∂bw3= 0.1432	∂Etotal / ∂bw4= -0.0515

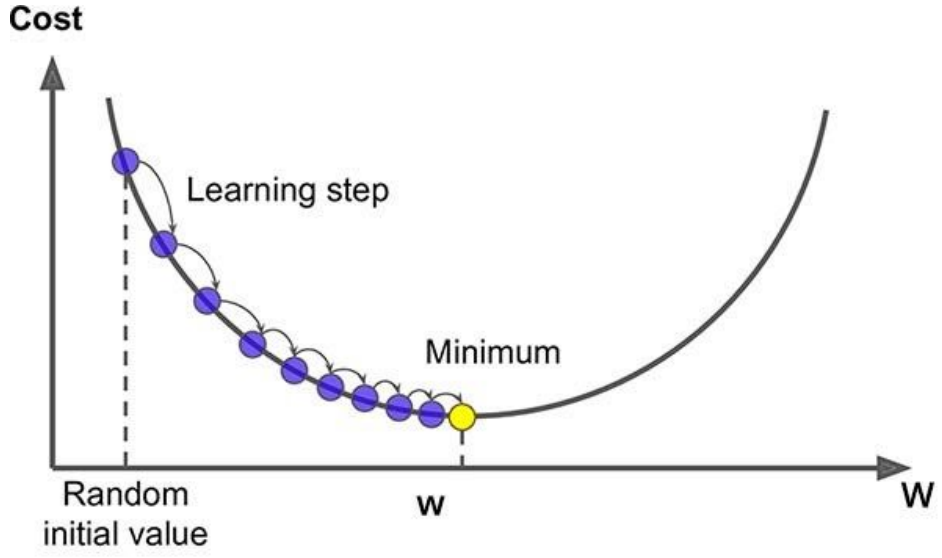
Step5: Updating Weights :

The goal of Step 5, is to update the weights in a neural network. This is accomplished by continuing to apply the backpropagation I began in Step 4 by using the gradients I calculated through an optimization process known as **gradient descent**.

Gradient descent is an optimization method that helps us find **the exact combination** of **weights** for a network that will minimize **the output error.**

With **gradient descent**, we descend down the slope of gradients to find the lowest point, which is where **the error** is **smallest.**

By **descending down the gradients**, we are actively trying to **minimize the cost function** and arrive at **the global minimum**.
Our steps are determined by the **steepness of the slope** (the gradient itself) and the **learning rate**.
The **learning rate** is **a value** that speeds up or slows down **how quickly** an algorithm learns.
below, a figure that gives an intuitive explanation of how gradient descent works:
the cost function = Total Error.



The formula of updating weights is the following:
New weight_wi = Old weight_wi - learning rate x (∂Etotal / ∂wi)

=> Results for first iteration:

- **w1_new = w1 - 0.5 x (∂Etotal / ∂w1) = 0.1 - 0.5 x(- 0.000420492) = 0.1002**
- **w2_new = w2 - 0.5 x (∂Etotal / ∂w2) = 0.2 - 0.5 x (0.00159221) = 0.1992**
- **w3_new = w3 - 0.5 x (∂Etotal / ∂w3) = 0.12 - 0.5 x (-0.000981148) = 0.1205**
- **w4_new = w4 - 0.5 x (∂Etotal / ∂w4) = 0.17 - 0.5 x (0.003715156) = 0.1681**
- **w5_new = w5 - 0.5 x (∂Etotal / ∂w5) = 0.05 - 0.5 x (0.1005) = 0.00025**
- **w6_new = w6 - 0.5 x (∂Etotal / ∂w6) = 0.4 - 0.5 x (- 0.0362) = 0.4181**
- **w7_new = w7 - 0.5 x (∂Etotal / ∂w7) = 0.33 - 0.5 x (0.0836) = 0.2882**
- **w8_new = w8 - 0.5 x (∂Etotal / ∂w8) = 0.07 - 0.5 x (-0.0301) = 0.0851**

- **bw1_new = bw1 - 0.5 x (∂Etotal / ∂bw1) = 0.8 - 0.5x (- 0.0028) = 0.8014**
- **bw2_new = bw2 - 0.5 x (∂Etotal / ∂bw2) = 0.25 - 0.5x (0.0106) = 0.2447**
- **bw3_new = bw3 - 0.5 x (∂Etotal / ∂bw3) = 0.15 - 0.5 x (0.1432) = 0.0784**
- **bw4_new = bw4 - 0.5 x (∂Etotal / ∂bw4) = 0.7 - 0.5 x (-0.0515) = 0.7258**

Note!

So, our hands on example is concluded (we stop here), because we have broken down all the algorithms and we did all the calculations **by hand**, but **in the real world, the algorithm of neural network**, continues **to loop through step 3 to step 5**, until the network **converges** , that means, it reaches **the global minimum**(so the total error reaches an acceptable level)...

By Abir ELTAIEF
Great passion for Maths, Programming & Data science...