

THE SIMPLE PERCEPTRON

Neural Representation of AND and OR Logic Gates

A project made by :

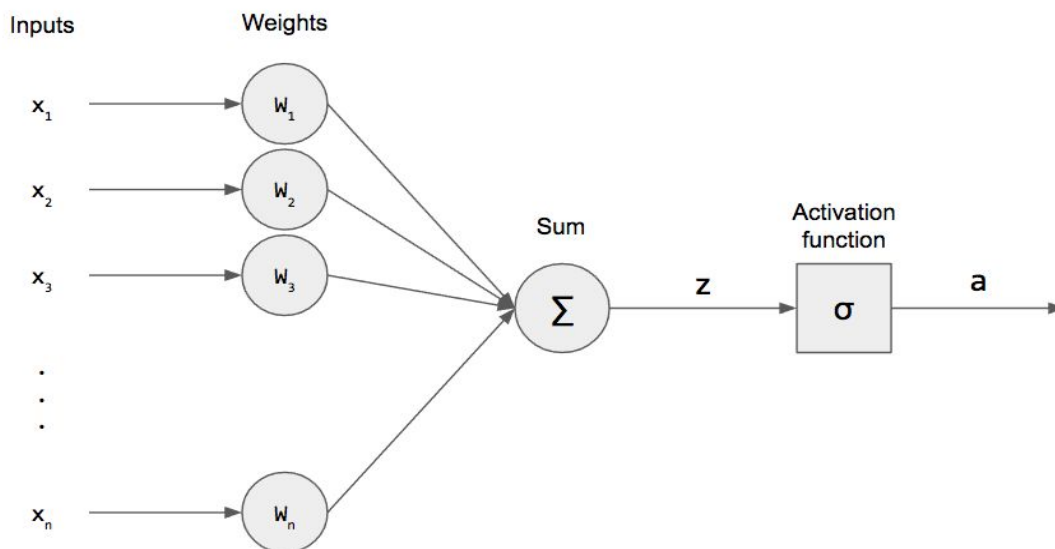
Noah Scharrenberg 1 and Lorenzo Pompigna 2

1 Cognitive and Computational Neuroscience Department of Data Science AI at Maastricht University

n.scharrenberg@student.maastrichtuniversity.nl - id : i6254952

2 Cognitive and Computational Neuroscience Department of Data Science AI at Maastricht University

l.pompigna@student.maastrichtuniversity.nl -id : i6233748



Index:

Abstract3

Introduction.....4

Methods.....6

Pseudocode.....8

Results.....9

Conclusions.....15

Code.....17

Sources.....20

Abstract

An abstract representation of the neuronal process of the action potential can be expressed in machine learning with a simple algorithm that performs a simple output evaluation if a general threshold has been reached.

A key task for connectionist research is the development and analysis of learning algorithms. An examination is made of several supervised learning algorithms for single-cell and network models. The heart of these algorithms is the pocket algorithm, a modification of perceptron learning that makes perceptron learning well-behaved with nonseparable training data, even if the data are noisy and contradictory.

We are intended to provide an algorithm capable of classifying a known and an unknown set of inputs by autocorrecting the weight vectors it is provided and analyse the result obtained by a meaning of a linear function.

We have experienced some graphic issues as we will discuss later, and we noticed that this algorithm can be expensive from a computational point of view, but it can help us simplify our results in an intuitive way.

The algorithm is intended to be used to evaluate on a small or big set of points the outcome of a logic gate table and the decisions made by the machine.

Introduction

We want to implement an algorithm with weighted vectors, able to classify inputs incoming in a linear space, by providing a supervised learning approach with a small constant rate, following a variation of the Hebbian learning rule of a single layer, feed forward network.

1. Given an activation function, we will be able to classify a given dataset with two possible output values (1,-1), depending on the result of the dot product of the inputs vectors with the weight vectors. However, if we look at this with an arithmetical approach we can easily recognise the pattern hidden behind this problem with an easier approach.

This method can be implemented in machine learning to classify a known dataset with a given number of inputs, and giving an output based on the outcome of a truth table



With this approach we can classify our outputs based on some basic approaches like

AND, OR, NOT, !OR. However our weights will represent the strength of each branch that is fed with one set of our data elements, and therefore the result can be interpreted as well as a linear representation of two correlated inputs that give an output after evaluating a linear formula of the type $y = mx + q$.

Furthermore with an easy implementation we desire to be able to visualise those data elements being classified in the runtime environment by the hebbian learning rule change

algorithm and the activation function. There are two types of Perceptrons: Single layer and Multilayer. **Single layer Perceptrons** can learn only **linearly separable** patterns. Multilayer Perceptrons or feedforward neural networks with two or more layers have greater processing power.

2. We decided to implement an algorithm based on a linear classification of a set of two inputs with an activation function of the type $(\text{Input } a = 2 * \text{Input } b + 1)$, meaning that if the input b is bigger than the input a multiplied by $2b + 1$, we will be able to classify it with a 1, else -1 if otherwise.

A Boolean output like the last one is based on inputs such as salaried, married, age, past credit profile, etc. It has only two values: Yes and No or True and False. The summation function " Σ " multiplies all inputs of " x " by weights " w " and then adds them up as follows:

In the equation given below:

" w " = vector of real-valued weights

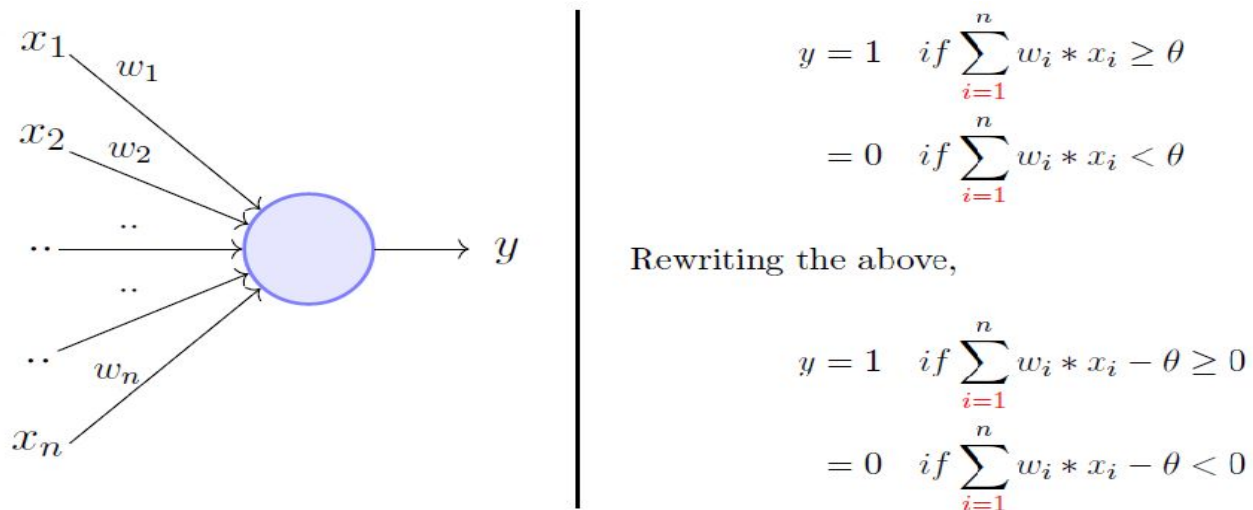
" b " = bias (an element that adjusts the boundary away from origin without any dependence on the input value)

" x " = vector of input x values

"1 and 0" = outputs that may vary due to a free interpretation (need to be different between each other)

Weight correction

3. A decision function $\phi(z)$ of Perceptron is defined to take a linear combination of x and w vectors. The value z in the decision function is given by: The activation function is +1 if z is greater than a threshold θ , and it is -1 otherwise. The Perceptron learning rule converges if the two classes can be separated by the linear hyperplane. However, if the classes cannot be separated perfectly by a linear classifier, it could give rise to errors.



Methods

Our perceptron can work both with a random set of inputs and weights, and initialised inputs and weights as well, in order to have the best possible overview of the process that is going to take place in this sequence of operations.

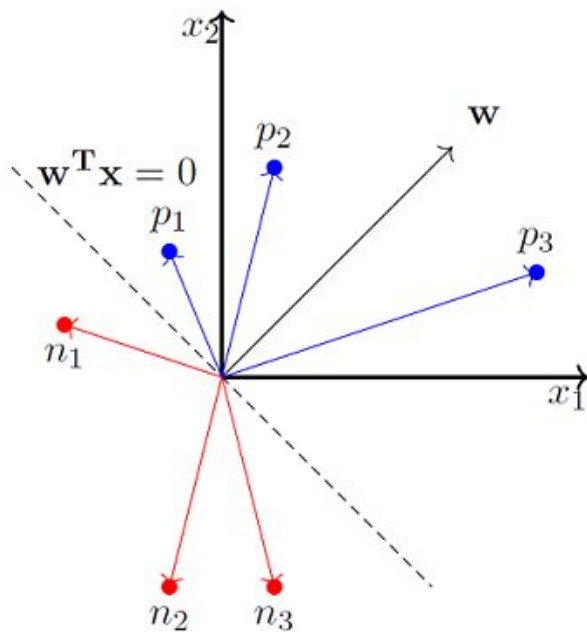
We will first try to analyse the outcome of a known dataset of values and weights to see what are the operations involved in the calculation of the activation function, then we will move a bit further trying to convert our results in an intuitive representation of the learning process and its speed and behaviour.

We decided to implement the algorithm in a java based environment and to make our results possible to be visualised by a dynamic GUI that gives us in the runtime the result of the weighted calculation on the inputs in an arithmetic representation of the slope and the rise of a line, that get evaluated after each points it's been first evaluated by checking on the x and y coordinates and feeded to the perceptron in order to give it a target where to calculates its application.

4. As we can see from our function, if a given input corresponds to $(x: 0, y: 0)$ and we want to calculate the activation function given by $w_1 * x + w_2 * y + b$ we will never come up with a right solution if we don't add the bias node. Therefore our solution will look like : $w_1 * x + w_2 * y + w_3 * b = 0$ and to calculate the y parameter of the line we want to display giving an x value and assuming the perceptron has corrected its weight learning from our targets, we could just ask the perceptron to give us it's calculation on x based on:

$$y = -(w_3/w_2) * b - (w_1/w_2) * x$$

Here we can already notice that the bias node is necessary to give us the right answer to the point $(0,0)$ given b as the rise of the given line, otherwise we could only hope for a coincidence.



$$W = [w_0, w_1, w_2, \dots, w_n]$$

$$X = [1, x_1, x_2, \dots, x_n]$$

$$W \cdot X = W^T X = \sum_{i=0}^n w_i * x_i$$

Pseudocode

-Initialise random weights

-Initialise a random dataset of points

(x and y coordinates) -> inputs

-Initialise sum of errors to 0

while true ->

for each Point in the dataset

-Calculate target of inputs

-Calculate Activation function (inputs, weights)

error = target – activation

sum of errors += abs(error)

foreach weight in weights

*deltaWeight = learning constant*error*inputNode*

weight += deltaWeight

-Classify the dataset based on new values given by the weighted activation

-display the inputs as they are classified by mapping them on the frame

-Calculate the rise and the slope of the weights and display the outcome

-Evaluate if sum of errors is equal to a desired value

-> close

Results:

Training a known set of datas

*If we decide to analyse a known set of datas, more specifically 4 points with a given x and y coordinate, then we may consider to increase our learning constant to some higher value due to the not needed precision in dividing points with such a big distance. As we can notice, due to randomness in the initialisation of the weighted vectors, we can encounter different type of results, and if we take a closer look to the weights they should be able to give us the values of the **activation function (3*)** by evaluating if*

weight [x] * x + weight[y]* y + weight [bias] *1 >= 0 -> { 1 }

weight [x] * x + weight[y]* y + weight [bias] *1 < 0 -> { 0 }

therefore our weights, once trained ,should be able to express the factors of the linear function we discussed before. (4*)

Training with assigned weights:

Let's try to analyse what is the outcome of a given learning process started with assigned weights:

As we can see, the maximum amount of the error for each set of inputs can either be 1 or -1,(2 or -2 later in a variation) therefore we take the sum of the absolute value for each point and we run the simulation until we don't get it to 0, in order to have a complete view at the precision of the classification given by the perceptron. Having the same weights initialized for the perceptron leads us to the conclusion that indeed the steering factor is being evaluated from the weights and therefore we can only interact with the learning constant in order to access the key for the faster evaluation, or make some considerations about how to implement the initialization of the weights.

Now let's take an even closer look at what happens in our weight changing method.

Here, the weights are first outputted in square brackets and the activation function is defined as we discussed before(3*), So :

Weights initialised as:

{ 0.1 ; 0.3 ; (bias) -0.3 }

Points given :

[0,0] [0,1] [1,0] [1,1]

Activation function(3*): where i is the output of the evaluation of the inputs combined with the weights defined as the dot multiplication we discussed in the beginning.

```
if (i >= 0)
```

```
{ //if the input is positive (or 0) return 1; }
```

```
else
```

```
{ //if the input is negative return 0; }
```

```
point : 1 [0, 0, 1] > { target : 0 }
0 x 0.1 + 0 x 0.3 + 1 x -0.3 + = -0.3 --output ->0 correct!
point : 2 [0, 1, 1] > { target : 0 }
0 x 0.1 + 1 x 0.3 + 1 x -0.3 + = 0.0 --output ->1 error ! -1
dW0 [ -1 ] x 1 * k .... >>-0.1 || updating weight0--> [ -0.3 + -0.1 ] = -0.4
dW2 [ -1 ] x 1 * k .... >>-0.1 || updating weight2--> [ 0.3 + -0.1 ] = 0.2
point : 1 [0, 0, 1] > { target : 0 }
0 x 0.1 + 0 x 0.2 + 1 x -0.4 + = -0.4 --output ->0 correct!
point : 2 [0, 1, 1] > { target : 0 }
0 x 0.1 + 1 x 0.2 + 1 x -0.4 + = -0.2 --output ->0 correct!
point : 3 [1, 0, 1] > { target : 0 }
1 x 0.1 + 0 x 0.2 + 1 x -0.4 + = -0.3 --output ->0 correct!
point : 4 [1, 1, 1] > { target : 1 }
1 x 0.1 + 1 x 0.2 + 1 x -0.4 + = -0.1 --output ->0 error ! 1
dW0 [ 1 ] x 1 * k .... >>0.1 || updating weight0--> [ -0.4 + 0.1 ] = -0.3
dW1 [ 1 ] x 1 * k .... >>0.1 || updating weight1--> [ 0.1 + 0.1 ] = 0.2
dW2 [ 1 ] x 1 * k .... >>0.1 || updating weight2--> [ 0.2 + 0.1 ] = 0.3
point : 1 [0, 0, 1] > { target : 0 }
0 x 0.2 + 0 x 0.3 + 1 x -0.3 + = -0.3 --output ->0 correct!
point : 2 [0, 1, 1] > { target : 0 }
0 x 0.2 + 1 x 0.3 + 1 x -0.3 + = 0.0 --output ->1 error ! -1
dW0 [ -1 ] x 1 * k .... >>-0.1 || updating weight0--> [ -0.3 + -0.1 ] = -0.4
dW2 [ -1 ] x 1 * k .... >>-0.1 || updating weight2--> [ 0.3 + -0.1 ] = 0.2
point : 1 [0, 0, 1] > { target : 0 }
0 x 0.2 + 0 x 0.2 + 1 x -0.4 + = -0.4 --output ->0 correct!
point : 2 [0, 1, 1] > { target : 0 }
0 x 0.2 + 1 x 0.2 + 1 x -0.4 + = -0.2 --output ->0 correct!
point : 3 [1, 0, 1] > { target : 0 }
1 x 0.2 + 0 x 0.2 + 1 x -0.4 + = -0.2 --output ->0 correct!
point : 4 [1, 1, 1] > { target : 1 }
1 x 0.2 + 1 x 0.2 + 1 x -0.4 + = 0.0 --output ->1 correct!
process finished with weights : [-0.4, 0.2, 0.2]
```

After the results evaluated in the first 4 iterations we then get to the last attempt, that is nothing else than a double check on the evaluation of the inputs, and as we can notice, for every data element passed to the evaluation and calculation function, we get 0 errors from

each point, that takes our global sum of errors to be zero, and this stops the execution of our algorithm, giving us the last variation of the weights that should be able to classify all the points correctly. * **The time needed to perform this analysis was around 37 ms.** *

Training with randomized weights:

Training a known set of 4 points [0,0] [0,0.5] [-0.5,0] [-0.5,0.5] with a learning constant of 1% , and in the first try it took more than 1000 iteration to appropriately adjust the weights and make the learning successful. For this particular example we might assume that having an higher learning constant will lead to a faster solution but as we will see in the next case, not always having an high learning rate will give us a faster solution, but we will state indeed that the lower the last one is, the higher is the probability for the perceptron to steer in the desired direction instead of moving too far from one side to the other.

Now instead, weights are initialised as:

{ random ; random ; (bias) 1 }

Points given :

[0,0] [0,.5] [-0.5,0] [-0.5,0.5] "ideally [0,0][0,1][1,0][1,1]"

Activation function : where i is the output of the evaluation of the inputs combined with the weights defined as the dot multiplication we discussed in the beginning.

```
if (i >= 0)
```

```
{ //if the input is positive (or 0)
```

```
  return 1; }
```

```
else
```

```
{ //if the input is negative
```

```
  return -1; }
```

The output of this function will then be stored in the classifier instance field of the Point , so that when the point will be displayed we can set a different color based on this classifier, and furthermore we can trace a black outline on the point if the perceptron guessed the point or not.

x_1	x_2	OR	
0	0	0	$w_0 + \sum_{i=1}^2 w_i x_i < 0$
1	0	1	$w_0 + \sum_{i=1}^2 w_i x_i \geq 0$
0	1	1	$w_0 + \sum_{i=1}^2 w_i x_i \geq 0$
1	1	1	$w_0 + \sum_{i=1}^2 w_i x_i \geq 0$

$$w_0 + w_1 \cdot 0 + w_2 \cdot 0 < 0 \implies w_0 < 0$$

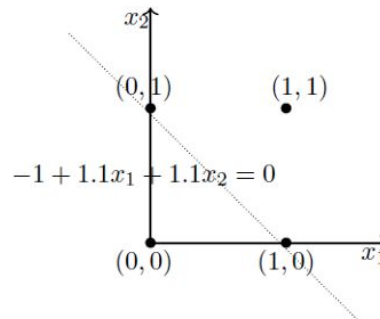
$$w_0 + w_1 \cdot 0 + w_2 \cdot 1 \geq 0 \implies w_2 > -w_0$$

$$w_0 + w_1 \cdot 1 + w_2 \cdot 0 \geq 0 \implies w_1 > -w_0$$

$$w_0 + w_1 \cdot 1 + w_2 \cdot 1 \geq 0 \implies w_1 + w_2 > -w_0$$

One possible solution is

$$w_0 = -1, w_1 = 1.1, w_2 = 1.1$$



Here, we can easily imagine what's happening inside the perceptron:

```

real classifier :1.0
errors : 0.0
coordinates : 0.0 0.5
real classifier :1.0
errors : 0.0
coordinates : -0.5 0.0
real classifier :1.0
errors : 0.0
coordinates : -0.5 0.5
real classifier :-1.0
errors : 0.0
Tentative n : 1073
errors : 0.0
weights : [0.004834114957562241, -0.0012594706244624023, 0.0026534989738613395]

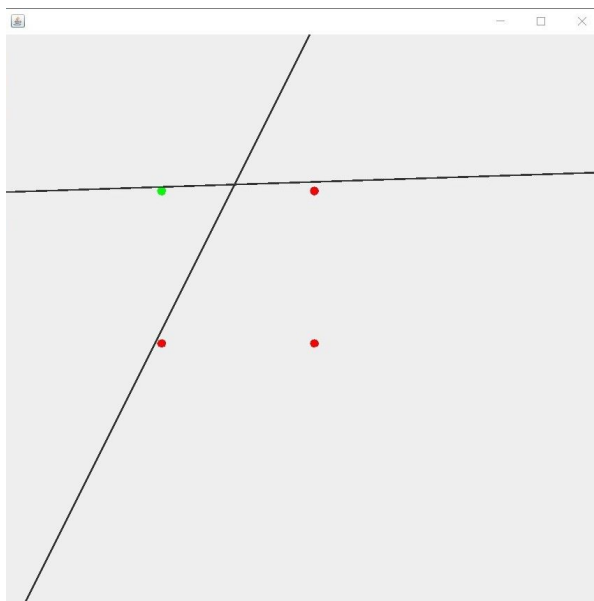
```

```

Terminal: Local (2) x +
real classifier :1.0
errors : 0.0
coordinates : 0.0 0.5
real classifier :1.0
errors : 0.0
coordinates : -0.5 0.0
real classifier :1.0
errors : 0.0
coordinates : -0.5 0.5
real classifier :-1.0
errors : 0.0
Tentative n : 1
errors : 0.0
weights : [0.7390882022350117, -0.8836424366119956, 0.7308507084434795]

```

Giving randomized weights can lead us to either having the solution at the first iteration, or having the completely opposite from the ideal initialization, therefore these uncertainties can potentially make the search hard to be solved, especially with a small set of points and a slow learning constant.



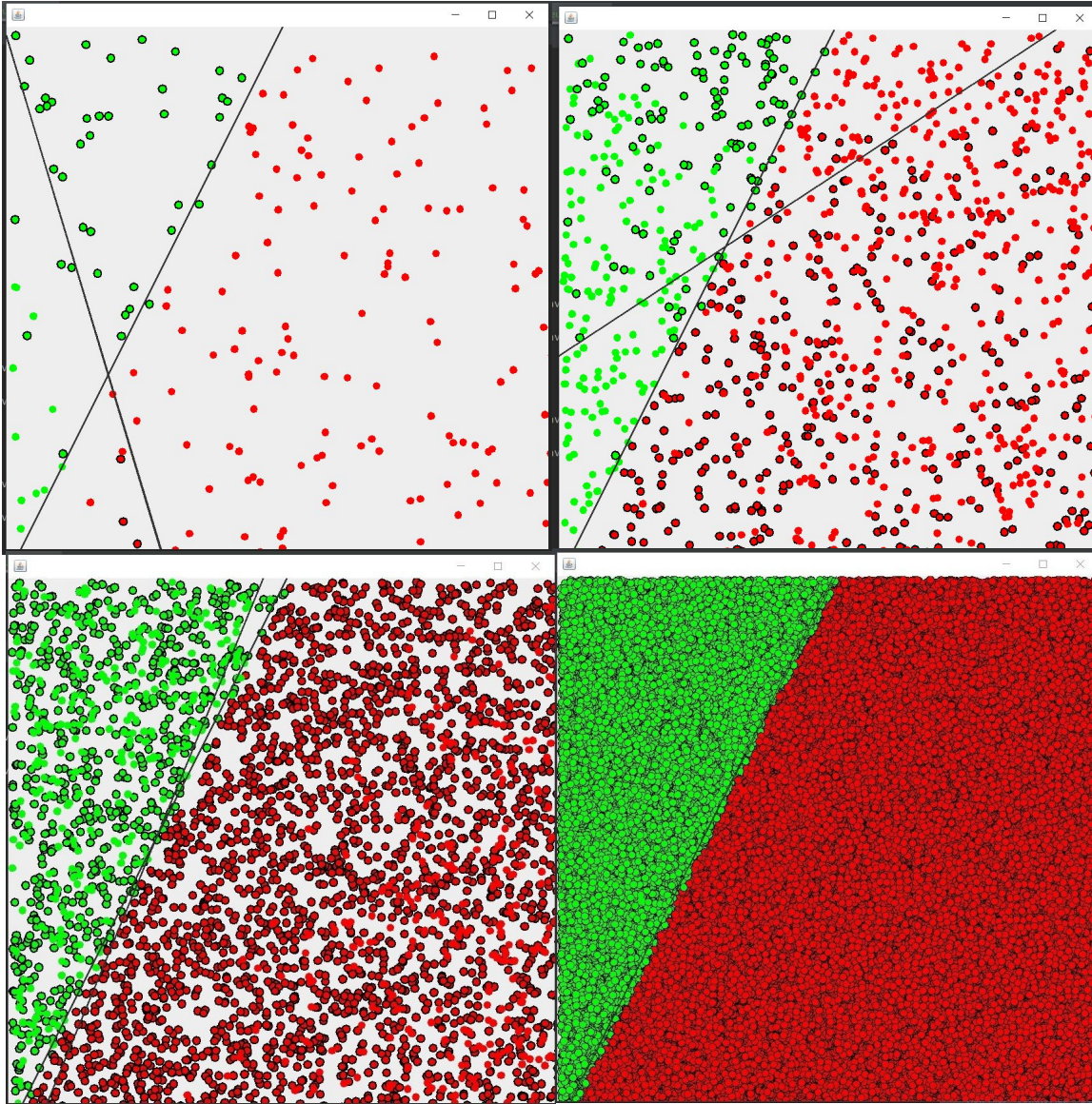
Here instead we faced a small issue with the graphics, it looks like the y space of the frame is considered to be extended to the top of the title bar, but the line or the points get mapped with some pixels of difference between each other: with an interactive visualisation of the perceptron acting on those points we can see the line slowly rotating from one side or from the other (due to the completely randomness in the weights initialisation), and this will stop exactly when the line hits / "ideally should go over" / the green point from the exact side, being able to place itself in the middle of the points, as we expected, and as the target line it's defined. Of course, for a specified target and with required high

precision we can definitely forget to achieve the specific weights desired, because the perceptron won't be able to classify them correctly after such a small set of data . (learning = practice!!)

Training a big set of data elements

Thanks to the brilliant idea found on the book '[The nature of Code](https://natureofcode.com/book/chapter-10-neural-networks/)' Chapter 10 - The perceptron¹ written by Daniel Shiffman, as we already saw in the previous chapter, we are now able to display the instant result of the weight power and precision by displaying a line that acts as an interpretation of the weights as they are calculated on two given values corresponding to the outer values of our frame, and by that we can also make some very intuitively consideration about the importance of the learning rate in a learning algorithm. Indeed we can easily observe how changing the learning rate can influence the learning steer of the line the perceptron is guessing. As we know the steering factor is the most important value to determine the efficiency of a turning point. After training our perceptron with an huge dataset of points defined by an x and y coordinate, we are now able to give the perceptron the right tools to guess, after being fed with a reinforced supervised learning, what are the factor coefficients of our desired function, giving us the weight vectors that have been corrected. Eventually after a sequence of iteration over new or already given data values we will be able to see the result achieved from the perceptron with an easy check on the line it's providing us and indeed we can consider the precision achieved from the perceptron resulting fairly similar to the line we are testing it on.

¹ <https://natureofcode.com/book/chapter-10-neural-networks/>



How we can notice here, we can only be able to classify a dataset of a maximum of **two inputs**, with this technique, because of course we can divide with a line the space if and only if the space is represented by **two dimensions** as well as our datas are. If we might think of dividing into two different pieces a space represented by **3 dimensions**, then we might need a plane in order to do that, and this can be

achieved in machine learning by combining the power of more than one perceptron and combination of adapted weights, and backpropagating the learning rule from the bottom to the top of the 'neurons layer' and classify our datas as a product of a combination of multiples boolean truth tables..

***The time needed for this analysis can take up to 5-10 minutes ***

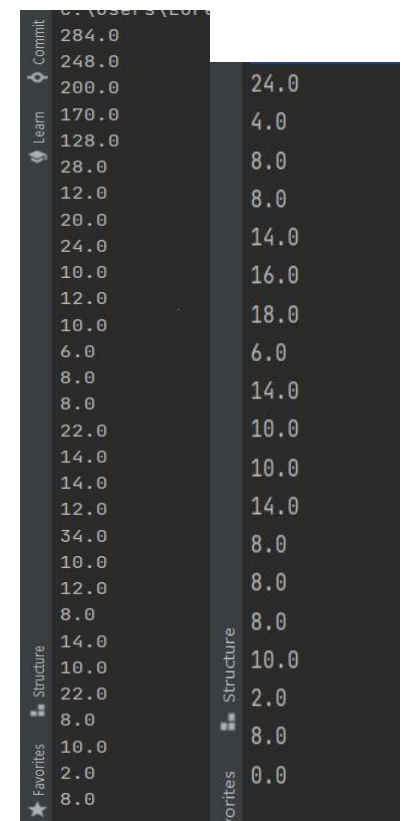
Conclusions:

AT THE RIGHT IN THE PICTURES -> RESULTS OF THE ITERATION OF THE SAME PERCEPTRON INSTANCE LEARNING FROM A DATASET OF 200 POINTS
***DIFFERENT AFTER EACH ITERATION* : DISPLAYED IS THE SUM OF THE ERRORS RETRIEVED AT EACH STEP.**

We decided now to run the perceptron algorithm based on a bigger dataset, therefore we iterate through a bigger collection of our randomly initialized dataset. For each iteration the data elements are always randomly changed from the beginning to the end of the test, but the weights are just randomly established and then adjusted based on the hebbian rule. As we can notice, we can have different types of solutions for each learning constant rate we

want to assign, and eventually after a series of iteration the weights will always be adjusted to get the sum of errors being 0, or at least we couldn't find an iteration that took more than 2-3 minutes (without graphics) . But this is well known as the 'monkeys and typewriters' topic we discussed in the genetic algorithm report, therefore what we are interested in now is the efficiency of this algorithm and the behaviour of its learning performance. We decided to initialise our weights and inputs to be double type variables and we noticed that it gets slowed by the big increment in computational times and space.

Without any complaint from the graphic unit of our machines we then decided to run a simulation of the learning process by initialising a dataset with random inputs and random weights, and an activation function that gives us a different classifier if the second input is bigger of a function of the first input.



284.0	24.0
248.0	
200.0	
170.0	4.0
128.0	8.0
28.0	8.0
12.0	14.0
20.0	16.0
24.0	18.0
10.0	6.0
12.0	14.0
10.0	10.0
6.0	10.0
8.0	14.0
8.0	10.0
22.0	14.0
14.0	8.0
14.0	8.0
12.0	10.0
34.0	2.0
10.0	8.0
12.0	0.0
8.0	
14.0	
10.0	
22.0	
8.0	
10.0	
2.0	
8.0	

Therefore we created a Map object intended to appropriately display our results on a frame with the aim of understanding our data by the meaning of points below or under a given linear function (line across the screen), getting the second input mapped to be rightly displayed in the Java plane, rather than getting it swapped because of the Java graphic default settings. We also noticed that there could be a possible lack of precision due to the high level of decimals given and the possible lossy conversion from all the mathematical processes done.

Code

```
package Perceptron.src;
import java.util.Arrays;
import static java.util.concurrent.TimeUnit.*;
/**
 * Class to implement the perceptron based on given weights and points
 * output is the value of the learning at each weight correction
 *
 */
public class pract2
{
    private static Neuron perceptron;
    private final int [] input;
    private final int id;
    private final int target;
    private static final double lk = 0.1; // the learning constant for the assignment
    private static double START;
    private static pract2 [] test;
    public pract2(int []x, int t, int n)
    {
        this.input = x;
        this.target = t;
        this.id = n;
    }
}
```

```

public static void main (String [] args)
{
    START = System.nanoTime();
    setPerceptron();
    int [] coord1 = new int[]{0,0,1}; //data values
    pract2 a = new pract2(coord1, t: 0, n: 1); //data element (values, classifier, id)
    int [] coord2 = new int[]{0,1,1};
    pract2 b = new pract2(coord2, t: 0, n: 2);
    int [] coord3 = new int[]{1,0,1};
    pract2 c = new pract2(coord3, t: 0, n: 3);
    int [] coord4 = new int[]{1,1,1};
    pract2 d = new pract2(coord4, t: 1, n: 4);
    test = new pract2[] { a,b,c,d};
    do {
        for (pract2 in : test) {
            System.out.print(" point : " + in.id + " "); //train the perceptron with the inputs
            perceptron.train(in.input, in.target);
            if (perceptron.getSum() != 0) // if the value hasn't been guessed correctly
            { //restart the loop and correct the weights
                break;
            }
        }
    } while (perceptron.getSum() != 0);
    System.out.println("process finished with weights : " + Arrays.toString(perceptron.getWeights()));
    //display results obtained
    System.out.println("time needed :");
    double end =System.nanoTime()-START;
    long convert = MILLISECONDS.convert((long) end, NANOSECONDS);
    System.out.println(String.format("Time needed > %s ms", convert));
}

```

```

public static void setPerceptron() //implementation of the abstract class Neuron
{
    perceptron = new Neuron() //initialize perceptron with the constructor providing given weights
    {
        /**
         * Training function for the assignment:
         * All the abstract methods are implemented here and for the purpose
         * of the analysis for the given points and weights.
         * Small differences from the Core constructor :
         * Without Graphical Interface
         * The analysis is restarted after each error
         * For this performance integers have been used for classifier and target
         * Order and application of the evaluation - correction process
         */
        @Override
        public void train(int [] values , int target)
        {
            perceptron.sum_Errors = 0;
            System.out.println(Arrays.toString(values)+ " > { target : "+ target+ " }");
            double sumActivation = 0;

            for (int i = 1; i< w.length; i++) //calculate the activation function of the value
            {
                sumActivation += values[i-1]*w[i];
                System.out.print(values[i-1]+ " x "+ w[i] );
                System.out.print(" + ");
            }
            sumActivation += values[2]*w[0]; //calculate the activation function for the bias
            System.out.print(values[2]+ " x "+ w[0] );
            System.out.print(" + ");
            sumActivation = (double) Math.round(sumActivation*10d)/10d;
        }
    }
}

```

```

int guess = activation(sumActivation);
System.out.print(" = " + sumActivation + " --output ->"+guess);
int error;
if (guess == target)
{
    System.out.println(" correct!");
}
else
{
    error = target - guess;
    System.out.println(" Incorrect ! "+error);
}
error = target - guess;
//the error is the correct answer - guess
if (guess != target )
{
    double deltaWeight; // deltaWeight is the change for the weight vectors
    deltaWeight = error * values[2] * lk; //wb is now wb + error * c
    deltaWeight = (double) Math.round(deltaWeight*10d)/10d;
    {
        System.out.print(" dW"+ 0 + " [ "+ error + " ] x "+ values[2] + " * k .... >>"+ deltaWeight);
        System.out.print(" || updating weight" +0+ "--> [ "+w[0]+ " + " + deltaWeight);
        w[0] = w[0] + deltaWeight;
        w[0] = (double) Math.round(w[0]*10d)/10d;
        System.out.println(" ] = "+ w[0]);
    }
    for (int i = 1;i<w.length; i++)
    {
        deltaWeight = error * values[i-1] * lk; //wx is now wy + error * y * c
        if (deltaWeight != 0) //wy is now wx + error * x * c
        {

```

```

        System.out.print(" dW"+ i + " [ "+ error + " ] x "+ values[i-1] + " * k .... >>"+ deltaWeight);
        System.out.print(" || updating weight" +i+ "--> [ "+w[i]+ " + " + deltaWeight);
        w[i] = w[i] + deltaWeight;
        w[i] = (double) Math.round(w[i]*10d)/10d;
        System.out.println(" ] = "+ w[i]);
    }
}
}
perceptron.sum_Errors += Math.abs(error); //instance field of the perceptron
}
public int activation(double i) //activation function with t = 0
{
    if(i>= 0) //return integer 1 or 0
        return 1;
    return 0;
}
};
}
}

```

References :

[1] THE NATURE OF CODE -The Perceptron (Chapter 10) - by Daniel Shiffman-

<https://natureofcode.com/book/chapter-10-neural-networks/> - (2012)

[2] The simple Perceptron - -The perceptron learning algorithm

**http://130.243.105.49/~lilien/ml/seminars/2007_02_01b-Janecek-Perceptron.pdf-
2007_02_01b-Janecek-Perceptron.pdf**

[3] The perceptron learning algorithm-by Towards Data Science- 22/08/2018 -

<https://towardsdatascience.com/perceptron-learning-algorithm-d5db0deab975> WRITTEN BY

Akshay L Chandra - DL Research