Artificial Neural Network

# Exercise 1

**Show the output from your program reproducing the AND learning in the final lecture slides, where the weights are outputted at every application of the weight-change rule.**

When we first created our perceptron program, we had decided upon 6 fixed points which could be separated by the line . These 6 points were as follows:

* ( 6, 9), ( 1, 8), ( -4, 2), ( 1, -1), ( -3, -4), ( 7, 2)

The original graph with points can be found in Appendix A.

Here we can see our perceptron updating the weights in order to create a line which will be similar enough to x = y to still separate our points. The original weights are randomly generated values between 0 and 1:

Text

Description automatically generated

A visual representation of our graph along with 5 lines drawn by our perceptron shows that randomly generated values for the weights will result in the perceptron outputting various different lines which separate our points. Pictured in red is the line from the example above:

Chart

Description automatically generated

# Exercise 2

**Adjust your program so that it calculates and show that it works.**

To explain this question we dove deeper into what the significance of the weights in our perceptron are, leading us to the conclusion that they can be used to create a line.

If we have a weight vector [1,2,3], it will represent the following line:

This can be rearranged to:

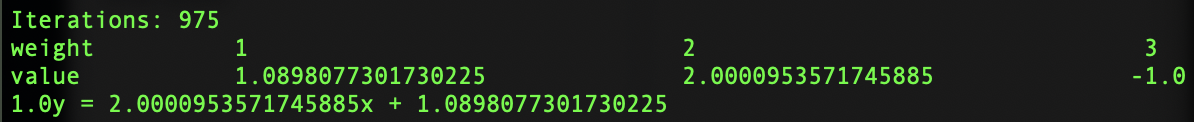
In order to obtain a line in the for , where we have exactly one y, we decided to prevent our perceptron from editing the weight that represents y. In other words, we will only let our perceptron update the weights for x and the constant (b).

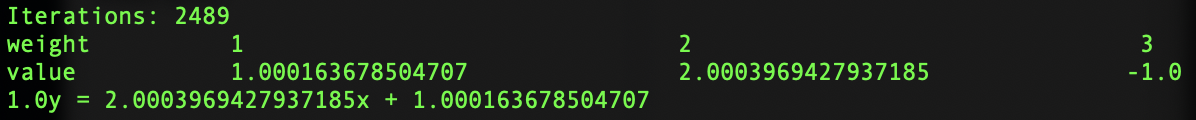
Now in order to let our perceptron find the line , we would have to generate two groups of inputs which are linearly separable by the line . To create our points, we used a random point generator which assigned half of the points above the line, and half of the points under the line, and put our perceptron to work.

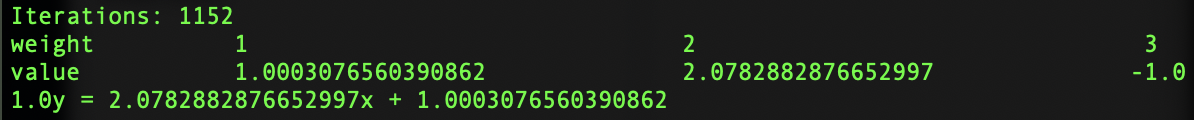
As is shown in exercise 3, we would technically need only two points to create the line, however, the more points we use, the more accurate our line is likely to become.

For the purpose of this exercise we have decided to use 100 points; 50 above the line, and 50 below. Furthermore, to verify that our points are close enough to the line to make a relevant impact, we are only generating the points with max/min values of 10/-10.

The resulting weights generated by our perceptron were both expected and unexpected. Although we anticipated that our perceptron would generate the line , it never quite reached the target. The use of double types in our weight vector made it nearly impossible for our perceptron to reach exactly, however, most of our results amounted to lines which were incredibly close to the desired line.







These examples demonstrate the infinite amount of lines which could separate our points accurately if you just add enough decimal places to the coefficient of x and the constant (b).

# Exercise 3

**What is the effect when you add more target values (if there is any effect...)? Try to show data/results of running your program that support your answer.**

Similarly to the prior exercise, for this experiment we used our random point generator, but for the sake of simplicity, our target line this time was. In attempt to find out the effect of adding more target values, we simply increased the amount of points that our point generator constructed. The next step consisted of running the perceptron 1000 times, documenting the average weights and the line they draw.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| # of points | W1 (constant) | W2 (x-value) | W3 (y-value) | Line |
| 2 | 0.45914 | 0.79083 | -0.85018 | 0.85018y = 0.79082x + 0.45914 |
| 10 | 0.34967 | 4.23365 | -5.17335 | 5.17335y = 4.23365x + 0.34967 |
| 100 | -0.55502 | 49.10631 | -49.4799 | 49.4799y = 49.10631x - 0.55502 |
| 1000 | 0.00728 | 24.17015x | -24.17355y | 24.17355y=24.17015x+-0.00728 |

At first glance, it is not very clear what all these numbers mean, so let us break it down into the two components of a line in 2D: the slope and the y-intercept [1]. For reference, the line has a slope of 1 and a y-intercept of 0.

|  |  |  |
| --- | --- | --- |
| # of points | Slope | y-intercept |
| 2 | 0.93018 | 0.54005 |
| 10 | 0.81836 | 0.06758 |
| 100 | 0.99245 | 0.01122 |
| 1000 | 0.99986 | 0.00030 |

As you can see from the processed data, the average line given by the weights of the perceptron over 1000 runs clearly shows that the line gets gradually closer to the target line when we increase the amount of target values.

# Appendix

### Appendix A

A picture containing diagram

Description automatically generated

### Appendix B

import java.util.Arrays;

import java.util.ArrayList;

import java.util.Random;

public class ANN4CCN{

public static final double LEARNING\_RATE = 0.01;

public static double[] omega = new double[3];

public static boolean convergence = false;

public static ArrayList<Point> allPoints = new ArrayList<>();

public static int TARGET\_VALUES=1000; //Amount of random points we are creating above/below a chosen line

public static int runs = 1000; //Amount of times the perceptron will run before outputting an average line

public static void main(String[] args){

//The original points used in question 1

/\*

int[] p1A = {1,6,9};

int[] p2A = {1,1,8};

int[] p3A = {1,-4,2};

int[] p4A = {1,0,-2};

int[] p5A = {1,-3,-4};

int[] p6A = {1,7,2};

Point p1 = new Point(1,p1A);

Point p2 = new Point(1,p2A);

Point p3 = new Point(1,p3A);

Point p4 = new Point(0,p4A);

Point p5 = new Point(0,p5A);

Point p6 = new Point(0,p6A);

ArrayList<Point> allPoints = new ArrayList<Point>();

allPoints.add(p1);

allPoints.add(p2);

allPoints.add(p3);

allPoints.add(p4);

allPoints.add(p5);

allPoints.add(p6);

\*/

int iterations = 0;

double w1Average = 0;

double w2Average = 0;

double w3Average = 0;

for(int b = 0; b <= runs -1; b++) { //This loop runs the perceptron "runs" times

convergence = false;

allPoints.clear();

iterations = 0;

createValues(); //This creates our points, method found below

//Initializing omega with random values

for (int i = 0; i < omega.length; i++) {

omega[i] = Math.random();

}

while (!convergence) { //Here we calculate the outputs for the current weights, and update the weights

for (int i = 0; i < allPoints.size(); i++) {

Point p = allPoints.get(i);

p.setOutput(dotProduct(omega, p.getVector()));

}

for (int i = 0; i < allPoints.size(); i++) {

Point p = allPoints.get(i);

if (p.getTarget() != p.getOutput()) {

p.setClassified(false);

update(omega, p);

} else {

p.setClassified(true);

}

}

iterations++;

for (int i = 0; i < allPoints.size(); i++) { //If points are correctly classified we converge, otherwise we start over

Point p = allPoints.get(i);

if (p.getClassified() == false) {

break;

}

if (i == allPoints.size() - 1) {

convergence = true;

System.out.println();

System.out.println("Iterations: " + iterations);

System.out.println("weight 1 2 3");

System.out.println("value " + omega[0] + " " + omega[1] + " " + omega[2]);

System.out.println(-omega[2] + "y = " + omega[1] + "x + " + omega[0]);

System.out.println();

}

}

}

System.out.println();

w1Average += omega[0];

w2Average += omega[1];

w3Average += omega[2];

}

//Print statement outputs the average weights and the average line the weights result in

System.out.println("Average W1: " + w1Average/runs);

System.out.println("Average W2: " + w2Average/runs);

System.out.println("Average W3: " + w3Average/runs);

System.out.println(-w3Average/runs + "y = " + w2Average/runs + "x + " + w1Average/runs);

}

/\*\*

\* This is the method to update the weights.

\* @param omega

\* @param point

\*/

public static void update(double[] omega, Point point){

for(int i = 0; i<omega.length; i++){

omega[i] = omega[i] + LEARNING\_RATE \* (point.getTarget() - point.getOutput()) \* point.getVectorValue(i);

}

}

/\*\*

\* First we calculate the dot product between the weight vector and the input, then use the stepfunction to

\* return either 0 or 1.

\* @param omega

\* @param pointVector

\* @return

\*/

public static int dotProduct(double[] omega, double[] pointVector){

double total = 0;

for(int i = 0; i<omega.length; i++){

total += omega[i]\*pointVector[i];

}

if (total <= 0){

return 0;

}

return 1;

}

/\*\*

\* Here we create "TARGET\_VALUES" randomly generated points and assign them a target value of 1 or 0 depending on if

\* they are above or below a predefined line. Right now it's set up for the line y=x.

\*/

public static void createValues(){

Random rnd = new Random();

ArrayList<double[]> allVectors= new ArrayList<>();

int aboveCount=0;

int underCount=0;

for (int i=0; i<TARGET\_VALUES ; i++){

while(allVectors.size()<TARGET\_VALUES){

double [] vector = {1, (Math.random() \* (10 + 10 + 1) -10), (Math.random() \* (10 + 10 + 1) -10)};

if (allVectors.contains(vector)) { continue; }

if(vector[2]< vector[1]){ //if the point is over the line

if(aboveCount==TARGET\_VALUES/2){continue;}

allPoints.add(new Point(1,vector));

allVectors.add(vector);

aboveCount++;

break;

}

else if (vector[2] > vector[1]){ //if the point is under the line

if(underCount==TARGET\_VALUES/2){continue;}

allPoints.add(new Point(0,vector));

allVectors.add(vector);

underCount++;

break;

}

else if(vector[2] == vector[1]){ //if the point is on the line

continue;

}

}

}

}

}

public class Point{

int target;

double[] vector;

int output = 0;

boolean classified = false;

public Point(int target, double[] vector){

this.target = target;

this.vector = vector;

}

public int getTarget(){

return target;

}

public void setOutput(int output){

this.output = output;

}

public int getOutput(){

return output;

}

public double[] getVector(){

return vector;

}

public double getVectorValue(int x){

return vector[x];

}

public boolean getClassified(){

return classified;

}

public void setClassified(boolean x){

classified = x;

}

}

# References:

# [1] Purple Math. “The Meaning of Slope and y-Intercept in the Context of Word Problems”. Available: <https://www.purplemath.com/modules/slopyint.htm#:~:text=In%20the%20equation%20of%20a,%22slope%2Dintercept%20form%22>.

[2] Ritvikmath. Explanation of Basic Perceptron usage, 2019. Available: https://www.youtube.com/watch?v=4Gac5I64LM4