The Catholic University of America

School of Engineering

Department of Electrical Engineering and Computer Science

CSC/ECE/DA 427/527 Fundamentals of Neural Networks

Project 1

Double-moon Classification using the Rosenblatt's perceptron

Han Nguyen

Instructor: Dr. Hieu Bui

October 1st, 2020

Table of Contents

1 Introduction

1.1 History of Deep Learning

The first mathematical model of an artificial neuron was the Threshold Logic Unit developed by the Warren S. McCulloch and Walter H. Pitts Jr in 1943 [1], they got inspired by how a biological neuron work and mimic it by create an artificial neuron network with mathematical and statistic behinds it. The history of deep learning is shown in figure 01.

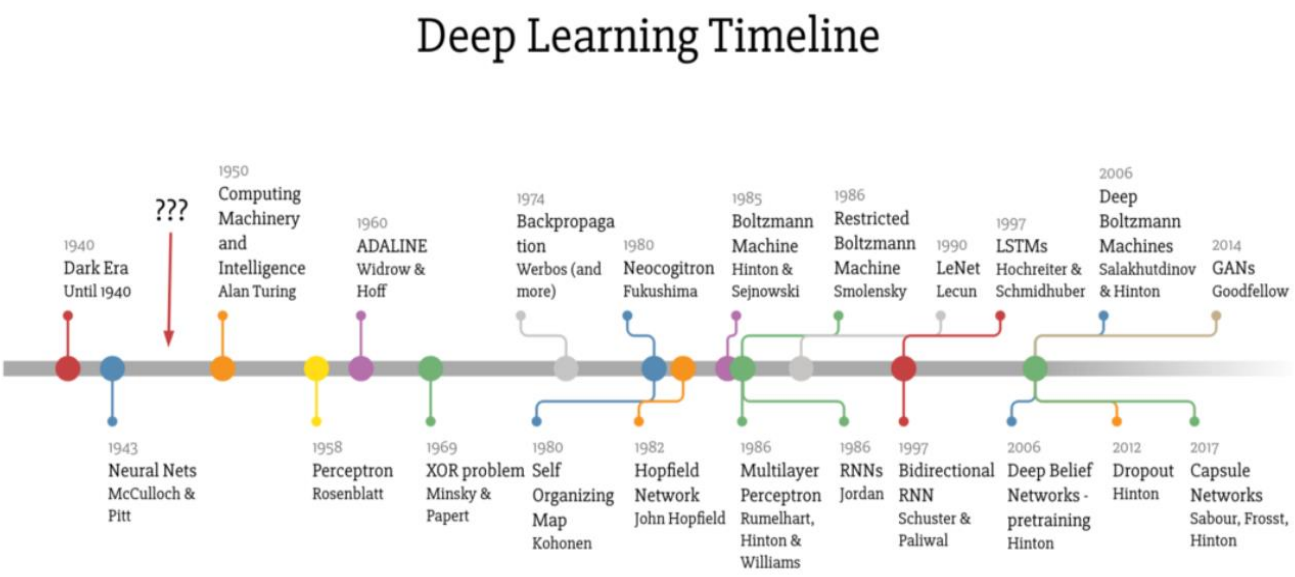


Figure 01 – Deep Learning Timeline.

1.2 Rosenblatt's Perceptron

The perceptron was invented by Rosenblatt in 1958, it was the first algorithmically described neural network and prove that the artificial neurons could learn from data [2]. The perceptron model enables the artificial neuron was fed with supervised learning algorithm, and it could learn the correct weights directly from training data by itself.

The perceptron used for the classification of patterns which are linearly separable as shown in figure 02. It is consisting of a single neuron with adjustable bias and weights. Additionally, when the perceptron being train with the patterns (vectors) which are came from two hyperplanes (classes), then the perceptron algorithm converges and positions the points of the vectors in its classes. [2]

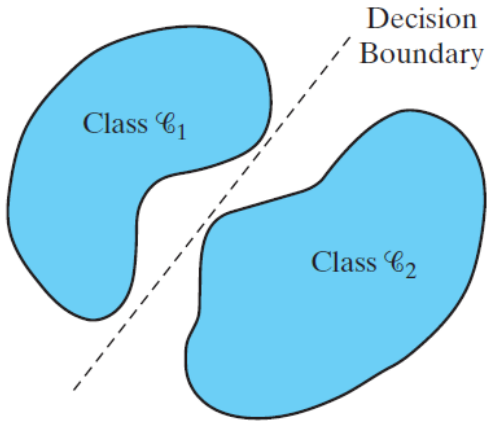
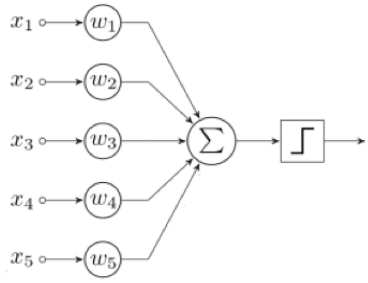


Figure 02 – A pair of linearly separable patterns.

The Rosenblatt's perceptron is the linear combiner that performing the signum function as shown in figure 03. The summing node of the model calculates the linear combination of the input vectors, also corporates with the bias to provide classification output. [2]



$$h(x) = \begin{cases} 1 \ if & w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_d \cdot x_d \geq \theta \\ 0 \ if & w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_d \cdot x_d < \theta \end{cases}$$

The equation can be re-written as follows including what it's known as the bias term: $x_0 = 1, w_0 = \theta$.

$$h(x) = \begin{cases} 1 \ if & w_0 \cdot x_0 + w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_d \cdot x_d \geq 0 \\ 0 \ if & w_0 \cdot x_0 + w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_d \cdot x_d < 0 \end{cases}$$

$$h(x) = \begin{cases} 1 \ if \ w^t \cdot x \geq 0 \\ 0 \ if \ w^t \cdot x < 0 \end{cases}$$

Figure 03 – Signal-flow graph of the perceptron.

In summary, the Rosenblatt's perceptron is a binary single model that takes the vectors as inputs, calculates the weights of the inputs. Then the weight values are pass to the activation (threshold), if the neuron is activating, give output 1, otherwise it is return -1.

2 Integrated development environments (IDE)

This project is implemented on Python 3 with two different virtual environments: Jupyter notebook and Pycharm.

Libraries: matplotlib, math, numpy, random

3 Project objective

3.1 Project description

Figure 04 shows a pair of "moons" facing each other in an asymmetrically arranged manner. The moon classified "Region A" is positioned symmetrically with respect to the y-axis, and the moon "Region B" is positioned on the right of the y-axis by an amount equal to the radius r, and below the x-axis by distance d with the parameters radius r = 10, and width w = 6. [2]

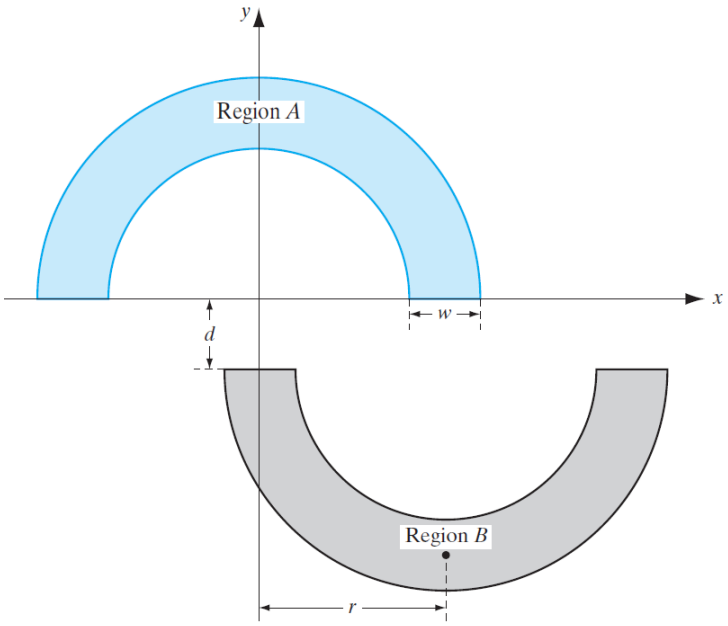


Figure 04 – The double-moon classification problem.

There are two main goals for the project. First, demonstrate the capability of Rosenblatt's perceptron algorithm to correctly classify linearly separable patterns. Second, show its breakdown when the condition of linear separability is violated.

3.2 Approach

The "moon" function was given for generating a double-moon classification problem as showed above (figure 04) with the same parameters as described:

- Num_points: the number of points in the data that will be generate randomly in each region.
- Distance: the distance of the "region B" with respect to the x-axis, it is adjustable.
- Radius: the radius of each regions.
- Width: the width of each regions.

After implemented the "moon" function, then we will do the following:

- Create the function to generate data points for the double-moon, initialize the bias and learning rate.
- Create the computation of actual response of the perceptron.
- Create the function to update the weights vector of the perceptron.
- Train the data
- Create the decision boundary
- Test

This implementation approach is based on the summary of the Perceptron Convergence Algorithm as illustrated in figure 05. [2]

Variables and Parameters:

$\mathbf{x}(n)$ = $(m+1)$-by-1 input vector
$= [+1, x_1(n), x_2(n), \ldots, x_m(n)]^T$
$\mathbf{w}(n)$ = $(m+1)$-by-1 weight vector
$= [b, w_1(n), w_2(n), \ldots, w_m(n)]^T$
b = bias
$y(n)$ = actual response (quantized)
$d(n)$ = desired response
η = learning-rate parameter, a positive constant less than unity

1. *Initialization.* Set $\mathbf{w}(0) = \mathbf{0}$. Then perform the following computations for time-step $n = 1, 2, \ldots$.
2. *Activation.* At time-step n, activate the perceptron by applying continuous-valued input vector $\mathbf{x}(n)$ and desired response $d(n)$.
3. *Computation of Actual Response.* Compute the actual response of the perceptron as
$$y(n) = \text{sgn}[\mathbf{w}^T(n)\mathbf{x}(n)]$$
where sgn(·) is the signum function.
4. *Adaptation of Weight Vector.* Update the weight vector of the perceptron to obtain
$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta[d(n) - y(n)]\mathbf{x}(n)$$
where
$$d(n) = \begin{cases} +1 & \text{if } \mathbf{x}(n) \text{ belongs to class } \mathscr{C}_1 \\ -1 & \text{if } \mathbf{x}(n) \text{ belongs to class } \mathscr{C}_2 \end{cases}$$
5. *Continuation.* Increment time step n by one and go back to step 2.

Figure 05 – Summary of the Perceptron Convergence Algorithm

There two main tasks for this project:

1. Implement the Rosenblatt's perceptron using the same parameters as in the Chapter 1 (page 61-62) [2] that consists the distance = 1, and distance = -4 with 2000 test points, radius = 10, and width = 6. Determine the classification error rate with 50 epochs.
2. Repeat task 1, but the distance = 0.

4 Implementation processes

4.1 Pre-processing

As the results of function "moon", there are 4 lists are return as x1, x2, y1, y2 – these are the position of 2 regions of the moon (x-coordinate, y-coordinate). Therefore, we will need to add the 3rd vector as the desired output (labeling the positions of the data for the 2 regions). 1 will be labeled for the 'region A' and -1 is the 'region B' as shown in figure 04.

```
def generateData(num_points, distance, radius, width):
    x1, x2, y1, y2 = moon(num_points, distance, radius, width)
    dt = []
    dt.extend([x1[i], y1[i],  1] for i in range(num_points))
    dt.extend([x2[i], y2[i], -1] for i in range(num_points))
    
    return dt
```

4.2 Initialization

We define the (m + 1)-by-1 weight vector as $w(n) = [b, w_1(n), w_2(n), ..., w_m(n)]^T$

The $w_0(n)$ corresponding to i = 0, represents the bias b. Therefore, we will set the bias = 0.1, the learning rate = 0.01; these values are adjustable from range 10^{-1} to 10^{-5}. The weight vector = 0 with addition with the bias. [2]

```
learningRate = 0.01
b = [0.1]
w = b + [0 for _ in range(2)]
```

4.3 Computation of actual response

In this step, we compute the actual response of the perceptron as y(n) = sgn[$\mathbf{w}^T(n)\mathbf{x}(n)$] where sgn() is the signum function [2]. The output will have to be follow this condition: sgn(v) = +1 if v > 0 or -1 if v < 0

```
def perceptronOutput(x, w):

    y = w[0]
    for i in range(2):
        y += sum([i*j for i, j in zip(w[1:],x[0:2])])  # dot product between w and x
    return 1 if y >= 0 else -1
```

4.4 Adaption of Weight Vector

This function is to update the weight vector of the perceptron to obtain:

- w(n + 1) = w(n) + learning rate * [d(n) - y(n)] * (n)
- error signal = d(n) - y(n)
- d(n) = +1 if x(n) belongs to class c1
- d(n) = -1 if x(n) belongs to class c2

```
def weightAdaptation(x, learningRate, errorSignal, w):

    return [i + learningRate*errorSignal*j for i, j in zip(w[1:], x[0:2])]
```

4.5 Training

In this function, the dataset will be implemented repeatedly to calculate the error signal, then use it to update the weight vector (as mentioned in step 4.4) by each epoch. The function will predict the data point and compare to the 3rd data point (the label). If the data point is wrongly classified, then the model will compute the error signal and update it to the weight, and the learning rate to increase the prediction accuracy of the model.

```
def train(dt, learningRate, w):

    epochs_num = 0 # number of epoch
    MSE = [] # mean squared error

    while True:
        totalError = 0.00
        for x in dt: # for each sample in the data set
            desired = perceptronOutput(x, w) # predicted response y(n)
            actual = x[2] # actual response d(n)
            if actual != desired:
                errorSignal = actual - desired  # e(n) = d(n) - y(n)
                w[1:] = weightAdaptation(x, learningRate, errorSignal, w)
                w[0] = w[0] + learningRate * errorSignal
```

Additionally, this function will also compute the Mean Squared Error (MSE) for each epoch as the algorithm shows in figure 06. The process will stop whenever the MSE = 0 or reached number of epochs = 50 (this condition is following the experiment in the textbook page. 62) [2].

$$MSE = \frac{1}{n}\Sigma\underbrace{\left(y - \hat{y}\right)}^{2}$$

The square of the difference between actual and predicted

Figure 06 – MSE algorithm

```
        totalError += errorSignal**2
epochs_num += 1
MSE.append(totalError/50)

if totalError == 0.0 or epochs_num >= 50: # stop condition
    break
```

4.6 Decision Boundary

The decision boundary is plotting in a 2-D space; therefore, it only takes input vector as 2-dimensional that each input in the vector representing a point on the graph. The summation of

perceptron uses to determine its output is the dot product of the inputs and weights vectors, addition with the bias (w[x]*w[y] + b), and use this equation to calculate the y value. [3]

```
def boundary(w, dt):
    x = np.linspace(np.amin(dt), np.amax(dt), 50)
    y = -(w[0] + x*w[1])/w[2]

    plt.plot(x, y, 'black')
```

5 Task 1 implementation

5.1 Distance = 1

The parameters of the double-moon in will be train with 1000 points, distance = 1, radius = 10, width = 6, learning rate = 0.01, bias = 0.1 and test with 2000 points.

```
trainSet = generateData(1000, 1, 10, 6)
testSet = generateData(2000, 1, 10, 6)

result, MSE = train(trainSet, learningRate, w)

# perceptron test
for x in testSet:
    plt.figure(1)
    predicted = perceptronOutput(x, result)
    if predicted == 1:
        plt.plot(x[0], x[1], 'x', color = 'red')
    else:
        plt.plot(x[0], x[1], 'x', color = 'blue')
```

Then we will plot the classified double-moon with decision boundary computed through training of the perceptron algorithm demonstrating separability of 2,000 test points, and the result is illustrated in figure 07.

```
# decision boundary
plt.figure(1)
plt.xlabel("x")
plt.ylabel("y")
plt.title("Classification using perceptron with d=1, r=10, w=6")
boundary(result, testSet)
plt.axis([-20, 30, -20, 20])
```

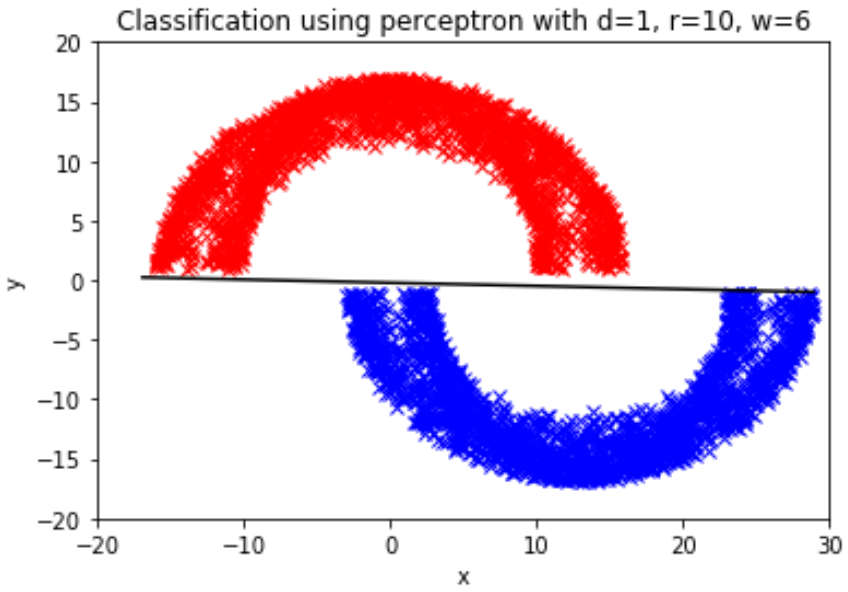


Figure 07 – Classification using Perceptron with d=1, r=10, w=6.

Additionally, we plot the learning curve where the mean-square error (MSE) is plotted versus the number of epochs with the results as show in figure 08.

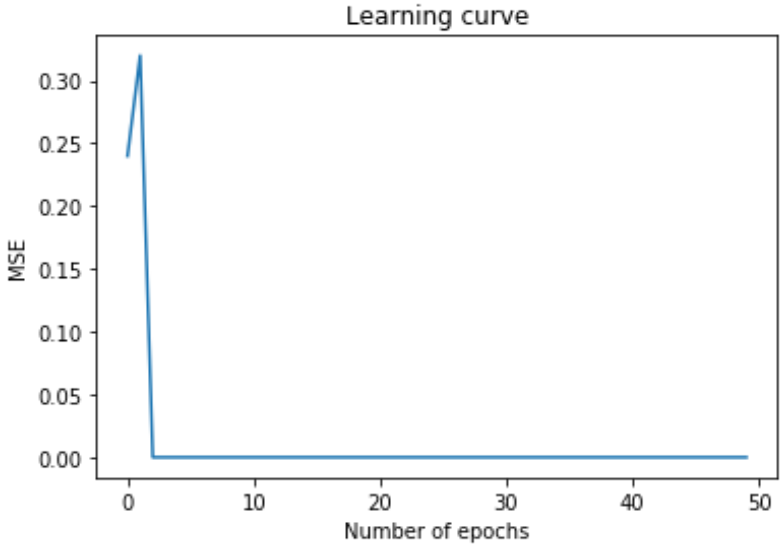


Figure 08 – Learning Curve of distance = 1

5.2 Distance = -4

The parameters of the double-moon in will be train with 1000 points, distance = -4, radius = 10, width = 6, learning rate = 0.01, bias = 0.1 and test with 2000 points.

```
trainSet = generateData(1000, -4, 10, 6)
testSet = generateData(2000, -4, 10, 6)
```

Similar to task 1 (5.1), the results of the distance = - 4 as illustrated in figure 09 and 10.

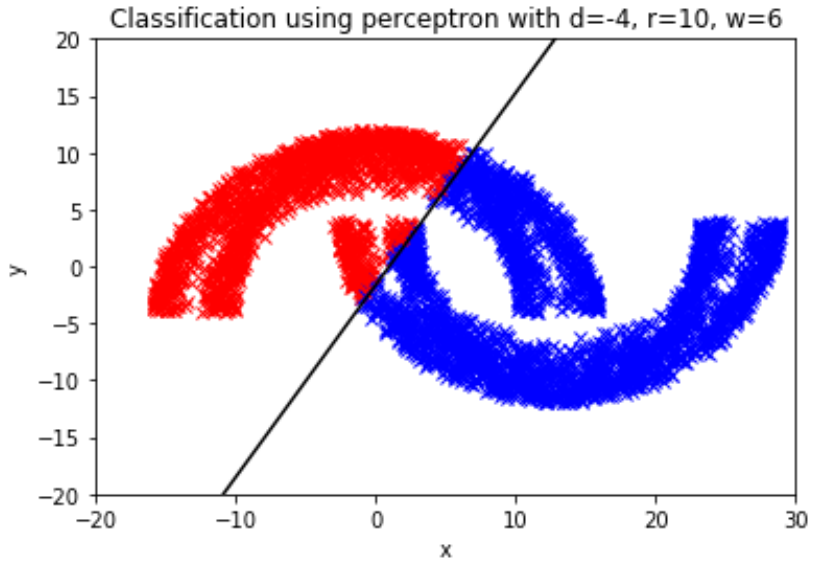


Figure 09 – Classification using perceptron with d= -4, r=10, w=6.

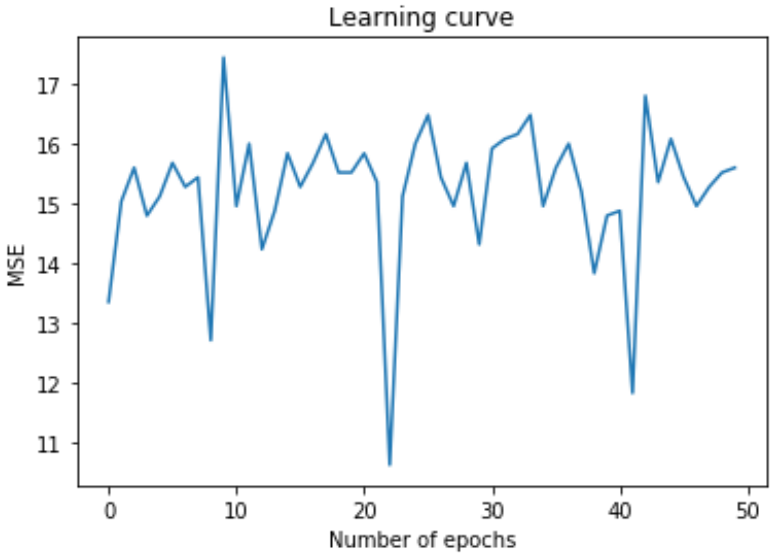


Figure 10 – Learning Curve of distance = -4

6 Task 2 implementation

The parameters of the double-moon in will be train with 1000 points, distance = 0, radius = 10, width = 6, learning rate = 0.01, bias = 0.1 and test with 2000 points.

```
trainSet = generateData(1000, 0, 10, 6)
testSet = generateData(2000, 0, 10, 6)
```

The results of the distance = 0 as illustrated in figure 11 and 12.

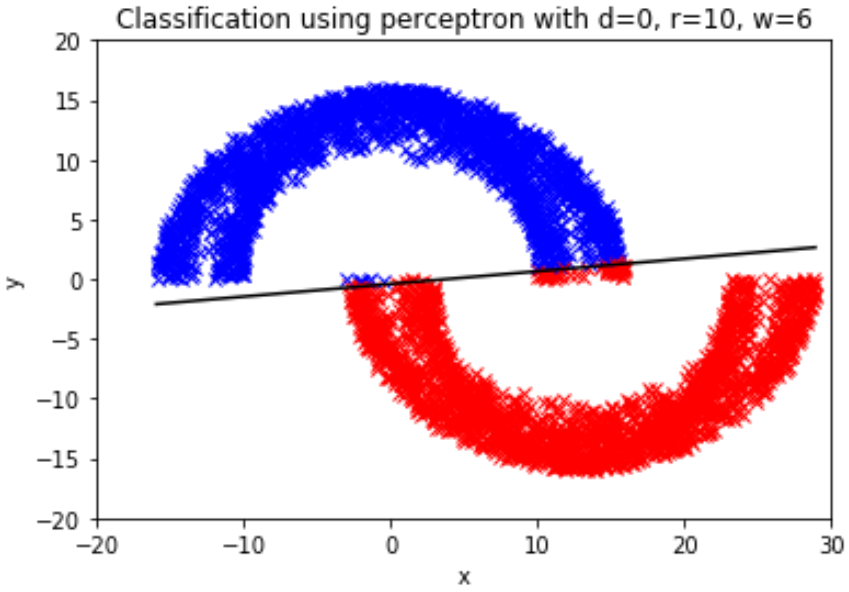


Figure 11 – Classification using perceptron with d=0, r=10, w=6.

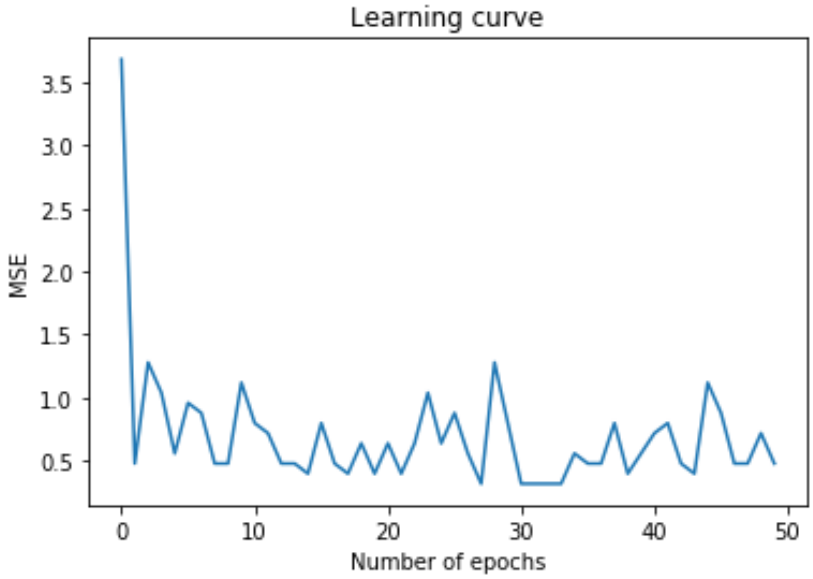


Figure 12 – Learning Curve of distance = 0

7 Conclusion

We have learned many behind-the-scenes facts of the very first single neuron. The Rosenblatt's perceptron is only able to classify 2 classes, but it could be any type of data (input vectors). Moreover, the perceptron could only perform linearly classification as we observe in the figure 07 & 08. The model was able to classify 2 regions perfectly within 3 epochs with the distance = 1.

However, when it comes to the scenario that 2 regions starting to merge with distance = -4 (overlap the x-coordinate and y-coordinate), the model was unable to classify 2 regions correctly as shown in figure 9 and 10. Therefore, Rosenblatt's perceptron is not suitable for non-linearly classification scenario.

Overall, the project is the fascinating opportunity to learn the mathematical algorithm behind the perceptron along with visualizing the ability to classify classes of the perceptron.

For Github repository of this project, named "Project 1", please visit the provided link below: **https://github.com/nguyenhq15/CSC527**

8 References

[1] B.Loiseau, J., 2019. *Rosenblatt'S Perceptron, The Very First Neural Network*. [online] towardscience. Available at: <https://towardsdatascience.com/rosenblatts-perceptron-the-very-first-neural-network-37a3ec09038a> [Accessed 2 October 2020].

[2] Haykin, S., 2017. *Neural Networks and Learning Machines*. 3rd ed.

[3] Countz, T., 2018. *Calculate The Decision Boundary Of A Single Perceptron - Visualizing Linear Separability*. [online] Medium. Available at: <https://medium.com/@thomascountz/calculate-the-decision-boundary-of-a-single-perceptron-visualizing-linear-separability-c4d77099ef38> [Accessed 2 October 2020].

[4] Github repository of Dingjie He. https://github.com/hedingjie/DoubleMoonClassfy