The Catholic University of America

CSC 527 Fundamentals of Neural Networks

Project 1: Double moon classification

Student: Lan Nguyen

Instructor: Dr. Hieu Bui

Date: 9/30/2020

Contents

I. Introduction

Rosenblatt's perceptron is a binary single neuron model. The input integration is implemented through the addition of the weighted inputs that have fixed weights obtained during the training step[1]. The neuron produces an output equal to +1 if the result of this addition is positive, and -1 if it is negative. The objective of this project is apply Rosenblatt's perceptron to classify a double moon problem and draw a decision boundary.

II. Working environment

I am using python 3.7, matplotlib and numpy libraries for plotting and storing data. A python notebook also been created for a better overview of all the results. This file can be viewed directly on my Github or can be opened via Anaconda/Google Colab.

III. Double-moon implementation

I was provided with the function "moon" for generating a double-moon shape. The input of this function are: total pairs of data points, radius of each moon, width of each moon and the distance between two moons. The output of this function are 4 lists contain position of every point of two moons. In order to draw these two moons we just use the matplotlib library and call the scatter() function for each moon.

Code:

```
x1, x2, y1, y2 = moon(2000, -4, 10, 6)
plt.scatter(x1, y1, marker='x', color='r')
plt.scatter(x2, y2, marker='o', color='b')
plt.show()
```

Result:

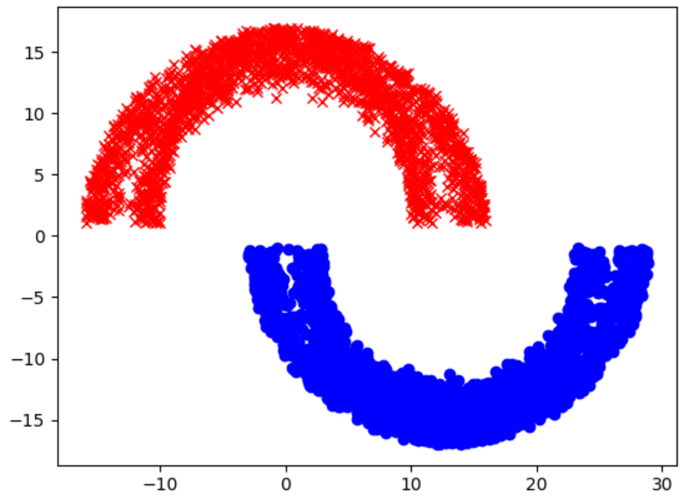
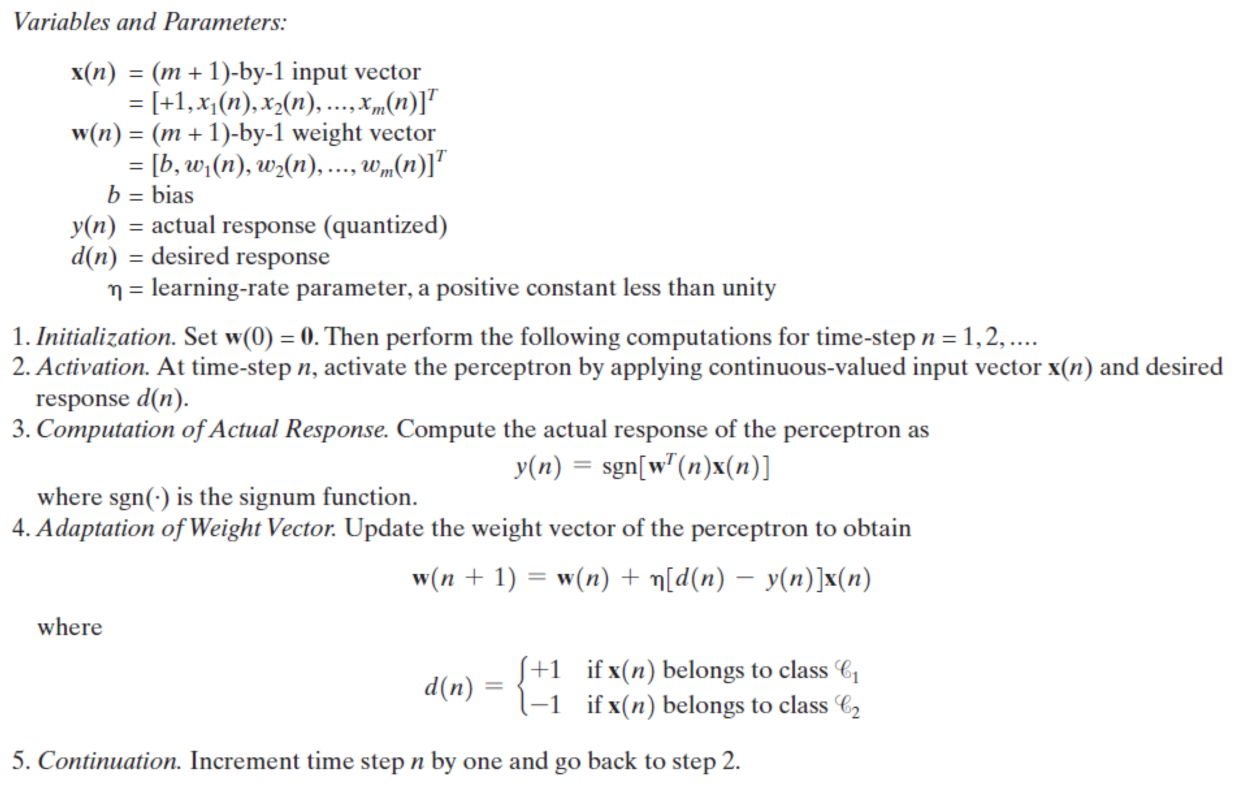


Figure 1

IV. Perceptron Convergence Implementation

The pseudocode for this algorithm are given via the textbook (page 54):



Variables and Parameters:

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

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
$$y(n) = \operatorname{sgn}[\mathbf{w}^T(n)\mathbf{x}(n)]$$
where $\operatorname{sgn}(\cdot)$ 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 2

I separated this implementation into 6 steps:

Step 1 – Preprocess data

Since the size of each input is 2 (position for x and y axis). I will put a third element for each input as the desired output. If that data point is belong to the upper moon then the output is 1, if it belong to the lower moon then the output is -1. I named this function as getDataSet().

Code:

```
def getDataSet(num_points, distance, radius, width):
    '''
    Add third element as desired output in each data point
    Return the dataset.
    '''

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

Step 2 – Initialization

Set the original weight vector = 0, bias = 0.1 and learning rate = 0.1. The bias can be any value and the learning rate varied linearly from 10^{-1} down to 10^{-5}.

Code:

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

Step 3 – Create activation function

The purpose of this function is to compute the response of the perceptron. It will calculate the dot product between weight vector and input vector. The response is +1 if the summation >= 0 and -1 if summation < 0.

Code:

```
def sgnFunc(x, w):
    '''
    Signum function, return 1 if sum(w*x)+b >= 0 else return -1
    '''

    activation = w[0]
    for i in range(2):
        activation += sum([i * j for i, j in zip(w[1:],x[0:2])])
    if activation >= 0:
        return 1
    else:
        return -1
```

Step 4 – Create train function

The objective of this function is to loop and update the weight vector using the formula:

w(n+1) = w(n) + learningRate*error*x(n)

Where error = expected output – predicted output.

I also create a MSE array to store the mean square error[2] value for each iteration and sumError to keep track the error. With the maximum of epochs or iterations is 50, the looping will terminate when epochs > 50 or the sumError = 0. The result of the train function will be the final weight vector.

Code:

```
def train(data, learningRate, w):
    """
    Trains all the vector in data.
    """

    epochs = 0
    MSE = []
    while True:
        sumError = 0.0
        for x in data:

            predicted = sgnFunc(x, w)
            expected = x[2]

            if expected != predicted:

                error = expected - predicted
                w[1:] = updateWeight(x, learningRate, error, w)
                w[0] = w[0] + learningRate*error
                sumError += error**2
        epochs += 1
        MSE.append(sumError/50)
        if epochs >= 50 or sumError == 0.0:
            break
    #plt.close()
    return w, MSE

def updateWeight(x, learningRate, error, w):
    '''
    Update function for weights
    w(n+1) = w(n) + learningRate * (d(n) - y(n)) * x(n)
    iterError is (d(n) - y(n))
    '''

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

Step 6 – Draw the boundary

After the training step we will get the final weight vector. For a perceptron with a 2-dimension input vector[3], the equation of the decision boundary should be:

w1x + w2y + b = 0

Solve this equation to get the formula for the line. I will draw the line as a function of x:

y(x) = - (b + w1x)/ w2

Code:

```
def draw_line(w, data):
    '''
    Draw decision boundary
    w0 + w1x + w2y = 0 => y = -(w0 + w1x)/w2
    '''

    x = np.linspace(np.amin(data),np.amax(data),100)
    y = -(w[0] + x*w[1])/w[2]
    plt.plot(x, y, '--k',label="DB")
```

V. Task1 implementation

The requirements of task 1 is to implement the Rosenbatt's perceptron using the same parameters as discussed in Chapter 1 (pages 61-62).

Step 1: I created a training dataset with 1000 points and a testing dataset with 2000 points. The distance between two moons is 1.

Code:

```
#Create dataset for training
dataTrain = getDataSet(1000, 1, 10, 6)

#Create dataset for testing
dataTest = getDataSet(2000, 1, 10, 6)
```

Step 2: Call the training function using the dataTrain. The results will be stored as result (weight vector) and MSE variables.

Code:

```
#Train the model
result, MSE = train(dataTrain, learningRate, w)
```

Step 3: Using dataTest to draw the double-moon. All the points will be predicted using the weight vector from step 2. If the predicted value is 1 then I colored the data point as red (upper moon) and if the predicted value is -1 then I colored it as blue (lower moon). Finally, call the draw boundary function.

Code:

```
for x in dataTest:
    plt.figure(1)
    predict = sgnFunc(x, result)
    if predict == 1:
        plt.plot(x[0], x[1], marker='x', color='r',label="bl")
    else:
        plt.plot(x[0], x[1], marker='o', color='b',label='rd')
```

Results:

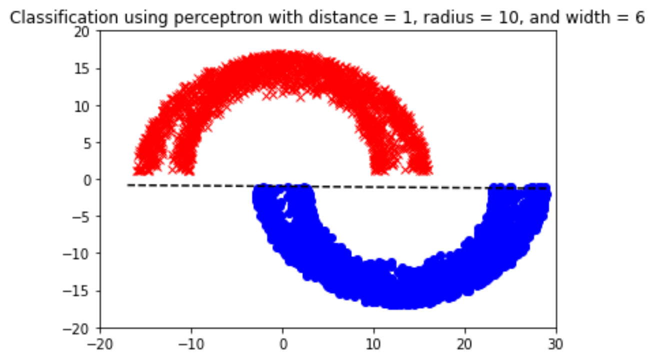


Figure 4

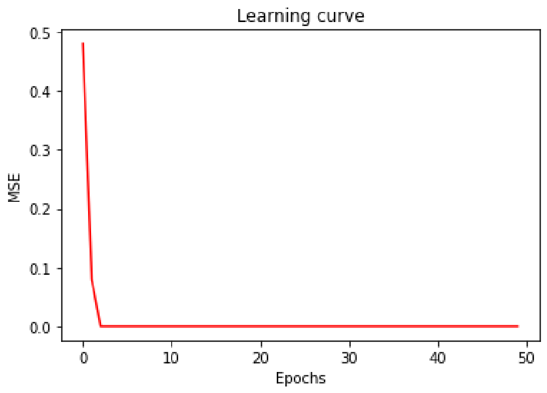


Figure 3

VI. Task 2 implementation

Task 2 is similar to task 1, just change the distance d = 1. I already set a variable to keep track the classification error and draw learning curve base on that value.

Results:

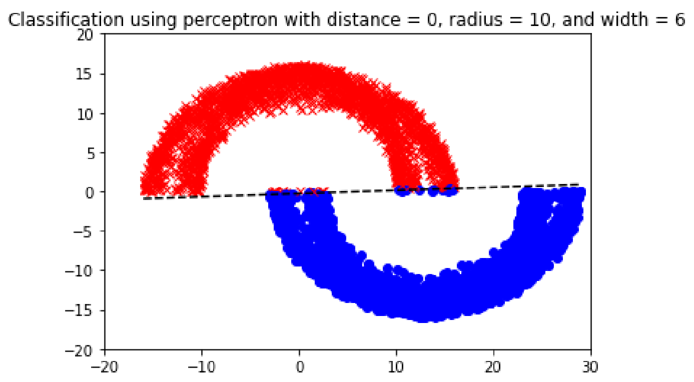


Figure 5

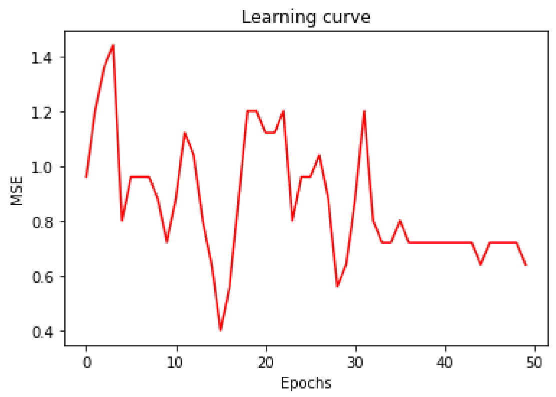


Figure 6

VII. Conclusion

This project is a great opportunity to learn about perceptron and fundamental of neural network. As we can see from task 1, it is possible to find a boundary because there is a significant gap between the two moons and we can say that the perceptron has convergence to a weight vector. The decision boundary is the line that perpendicular to the weight vector. In task 2, the perceptron cannot convergence and there are some misclassify data points in both of the moon.

VIII. Github repository

https://github.com/khoalan/CSC527/tree/master/Project1

References

[1] David, I. (2016, August 2). Artificial Neural Networks-The Rosenblatt Perceptron. *Neuroelectrics*. Retrieved from: https://www.neuroelectrics.com/blog/2016/08/02/artificial-neural-networks-the-rosenblatt-perceptron/

[2] Stephanie, G. (2013, November 2). Mean Squared Error: Definition and Example. *StatisticsHowTo*. Retrieved from: https://www.statisticshowto.com/mean-squared-error/

[3] Thomas, C. (2018, April 13). Calculate the Decision Boundary of a Single Perceptron. *Medium*. Retrieved from: https://medium.com/@thomascountz/calculate-the-decision-boundary-of-a-single-perceptron-visualizing-linear-separability-c4d77099ef38#3778