	Version 3.2		
1.	**readme.txt**	**User guide and Anaconda Installation guide**	
2.	**DataEmit.py**	**Data generator**	
3.	**PLA.py**	**PLA algorithm and plot**	
4.	**reportEE5434.docx**	**This report**	
XI.	Python PLAII-Scipy.py	An Python for comparsion purpose and reference only.	
XII.	Colab or Anaconda	The Pythons work in Colab notebook but much better in Anaconda's Spyder IDE.	

By: Hoson LAM, Copyright 2024

Executive Summary

This project implements a PLA[1,2], Perceptron Learning Algorithm, learning algorithm for two-dimensional input by a basic Python class called PerceptronX(object) which can be extended to become a neural network, dense, deep network or CNN in future.

Most of the tests on PLA achieve 100% accuracy within 100 epochs of training. The longest one with error is when the ratio of two labels in the training data is highly unbalanced. Some more works were done on adaptive learning rate and random initialization of weights to improve it. In the future, more works can be done on the random shuffle of input samples and see how much improvement get.

Introduction

The PLA, Perceptron Learning Algorithm, is a binary classifier algorithm for learning function that maps its inputs to an output a single binary value.

The perceptron is a linear classifier and only work well for a linearly separable dataset. The output is a binary value with "+1" or "-1". If the training set is linearly separable, the perceptron is guaranteed to converge. Furthermore, there is an upper bound on the number of times the perceptron can converge to have the expected result. Figure 1.1 is an example of a typical perceptron[3].

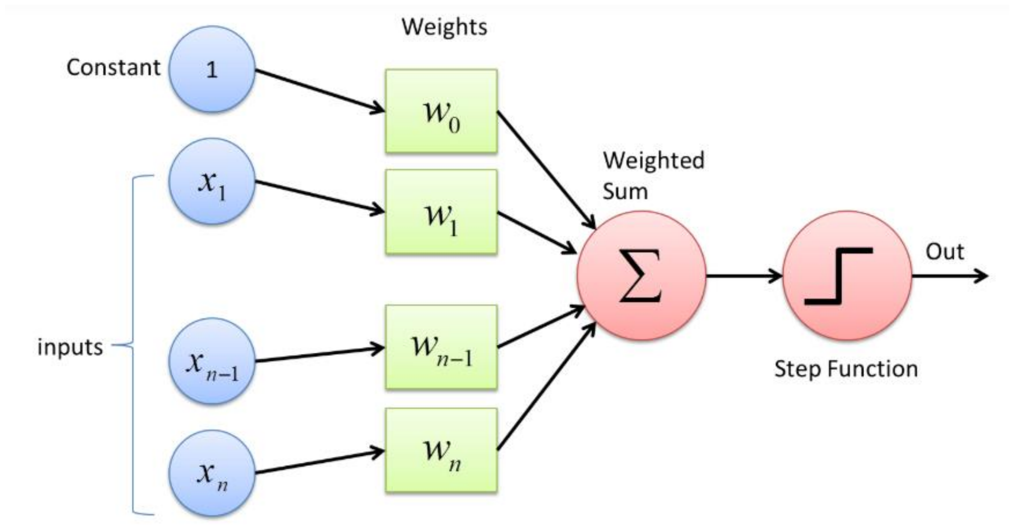


Figure 1.1

The perceptron is a modern but simplified version of biological inspired "neocognitron" [4] created by Kunihiko Fukushima back to early '80s.

System Details

The system is implemented in two-stage. Stage one is training data generation to generate the training data. It takes a three-dimensional vector (w0,w1,w2) as inputs and generate data points (x1,x2) with sign (w1x1+w2x2+w0)>0 or (w1x1+w2x2+w0)<0. If the sign is negative, the data point has labeled "-." Otherwise, it has a label "+". The first stage program is called DataEmit.py

In stage two, we take stage one's output as it's input and then output the learned weight using PLA algorithm. Furthermore, training data points and the line represented by learned weights are plotted. The system is shown in figure 2.1

By: Hoson LAM, Copyright 2024

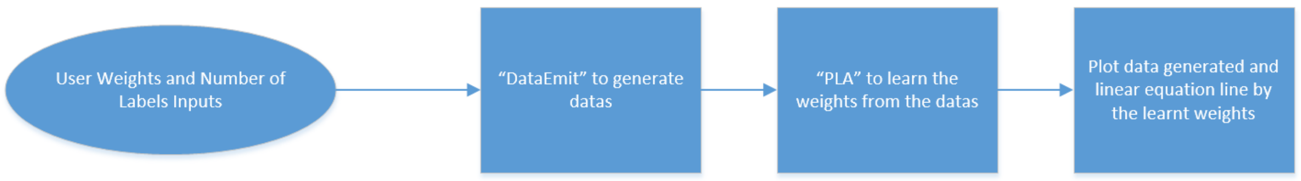
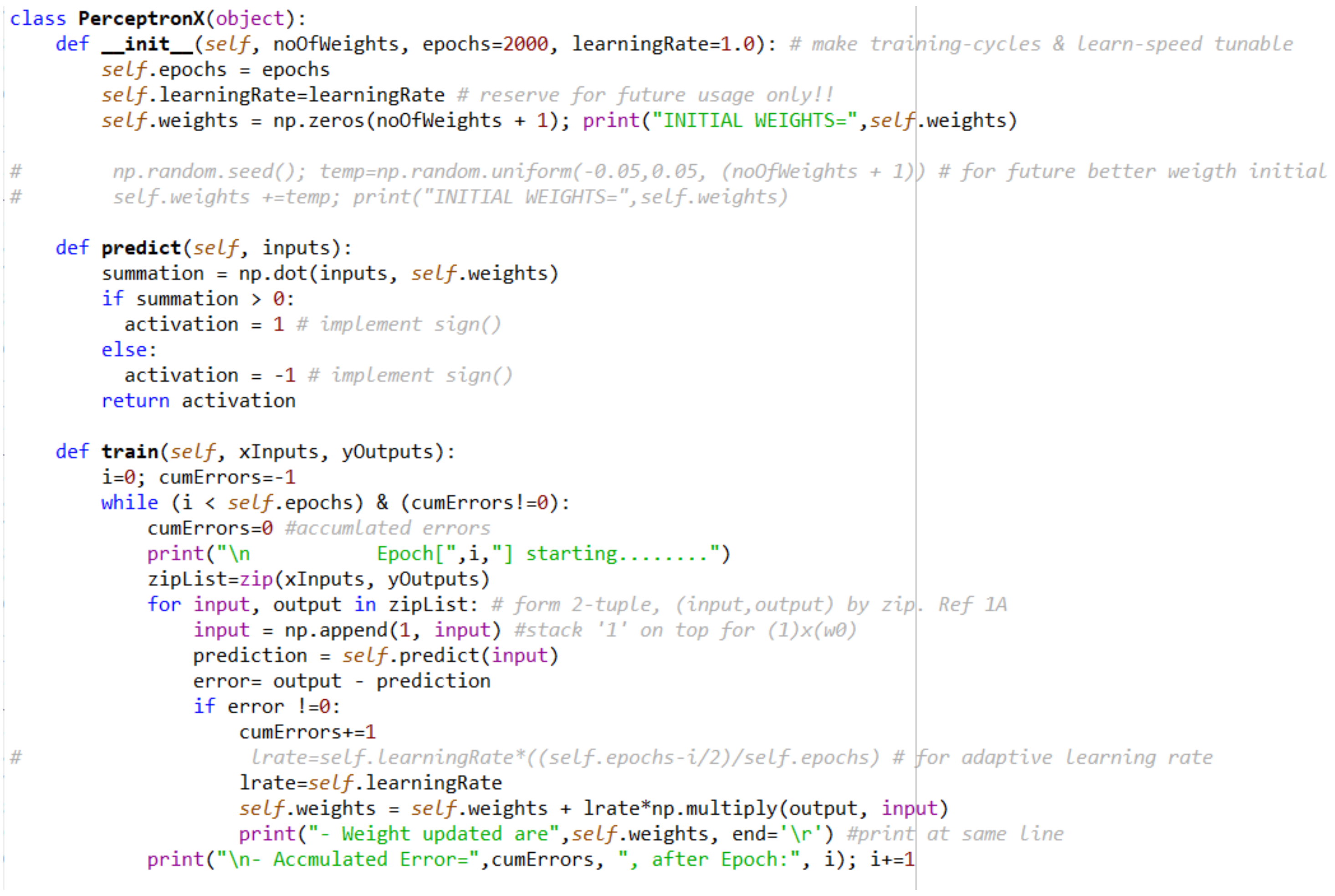

User Weights and Number of Labels Inputs
“DataEmit” to generate datas
“PLA” to learn the weights from the datas
Plot data generated and linear equation line by the learnt weights

Figure 2.1

The perceptron is implemented as a Python class and named as PerceptronX(object). Figure 2.2 shows the code to implement the object class.



```
class PerceptronX(object):
    def __init__(self, noOfWeights, epochs=2000, learningRate=1.0): # make training-cycles & learn-speed tunable
        self.epochs = epochs
        self.learningRate=learningRate # reserve for future usage only!!
        self.weights = np.zeros(noOfWeights + 1); print("INITIAL WEIGHTS=",self.weights)

#        np.random.seed(); temp=np.random.uniform(-0.05,0.05, (noOfWeights + 1)) # for future better weigth initial
#        self.weights +=temp; print("INITIAL WEIGHTS=",self.weights)

    def predict(self, inputs):
        summation = np.dot(inputs, self.weights)
        if summation > 0:
          activation = 1 # implement sign()
        else:
          activation = -1 # implement sign()
        return activation

    def train(self, xInputs, yOutputs):
        i=0; cumErrors=-1
        while (i < self.epochs) & (cumErrors!=0):
            cumErrors=0 #accumlated errors
            print("\n           Epoch[",i,"] starting........")
            zipList=zip(xInputs, yOutputs)
            for input, output in zipList: # form 2-tuple, (input,output) by zip. Ref 1A
                input = np.append(1, input) #stack '1' on top for (1)x(w0)
                prediction = self.predict(input)
                error= output - prediction
                if error !=0:
                    cumErrors+=1
#                    lrate=self.learningRate*((self.epochs-i/2)/self.epochs) # for adaptive learning rate
                    lrate=self.learningRate
                    self.weights = self.weights + lrate*np.multiply(output, input)
                    print("- Weight updated are",self.weights, end='\r') #print at same line
            print("\n- Accmulated Error=",cumErrors, ", after Epoch:", i); i+=1
```

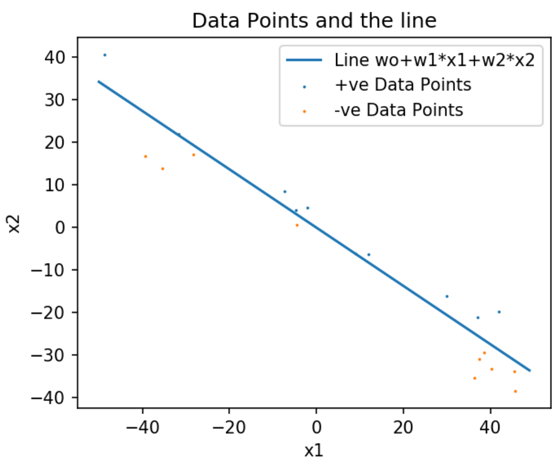
Figure 2.2

Results

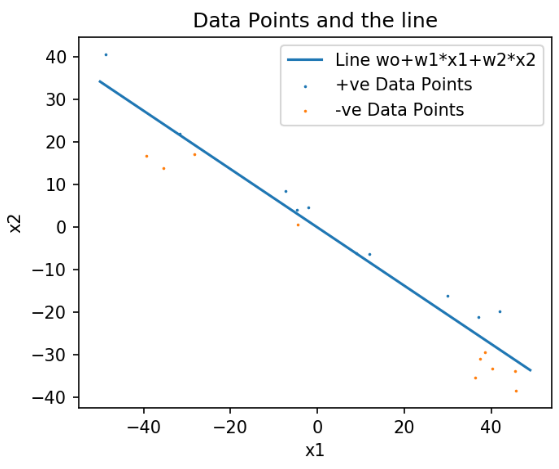
The result is encouraging, except when the ratio of two labels in the training data is highly unbalanced. For the unbalanced case, the maximum number of epochs tried was 1999 with four errors reminded. In each epoch, it will cycle through all the data points and stop until no error or when the maximum of epochs is reached. For this case number of positive labels are 10 and negative labels are 390. Therefore, the number of weights updates are 799,600(1999 x 400) without settling to no error. Table 3.1 shows the numerical results and Graphs 3.1 show the corresponding graphs in Table 3.1.

	M	N	Initial Weights	Learning Rate	No of Epochs	Learnt Weights
w0,w1,w2=5,2,3						
	10	10	(0.0,0.0,0.0)	1.0	8	[6. ,94.61779022, 138.34599209]
	50	50	(0.0,0.0,0.0)	1.0	86	[399. ,282.29516172, 407.11682769]
	100	100	(0.0,0.0,0.0)	1.0	63	[529., 255.87455606, 390.04774934]
	150	150	(0.0,0.0,0.0)	1.0	65	[533., 231.10598844, 345.98248118]
	200	200	(0.0,0.0,0.0)	1.0	331	[1260., 574.55155164, 860.59517533]
	10	390	(0.0,0.0,0.0)	1.0	1999 with 4 errors	[1474. , 1047.27135563, 1517.31052637]

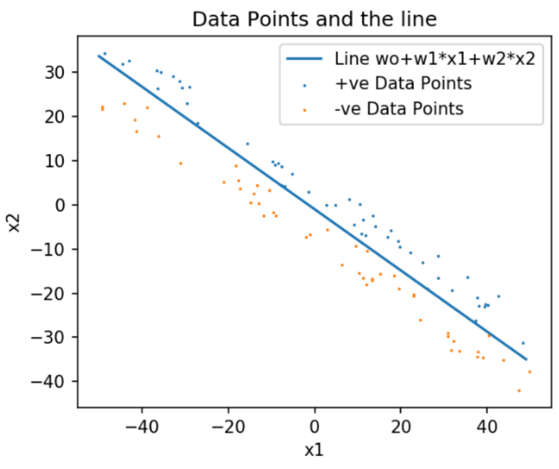
Table 3.1



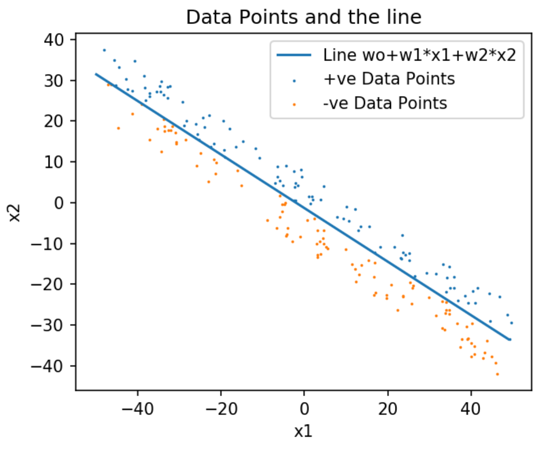
M,N=(10,10)



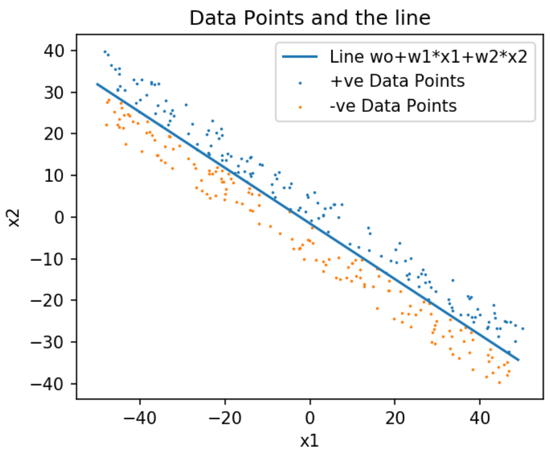
M,N=(50,50)



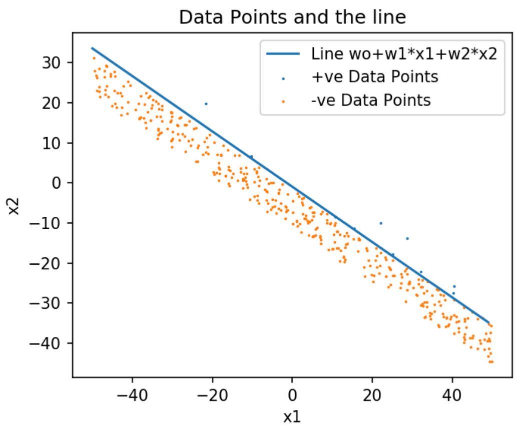
M,N= (100,100)



M,N=(150,150)



M,N= (200,200)



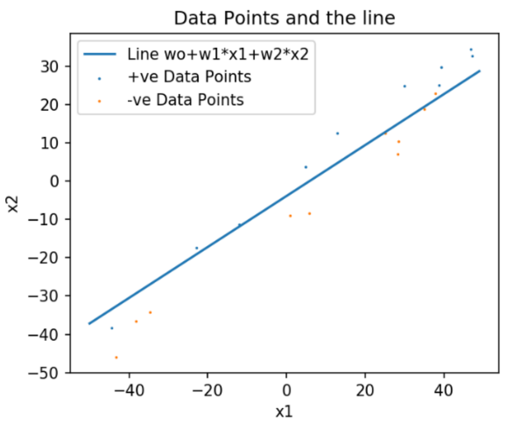
M,N=(10,390)

Graphs 3.1

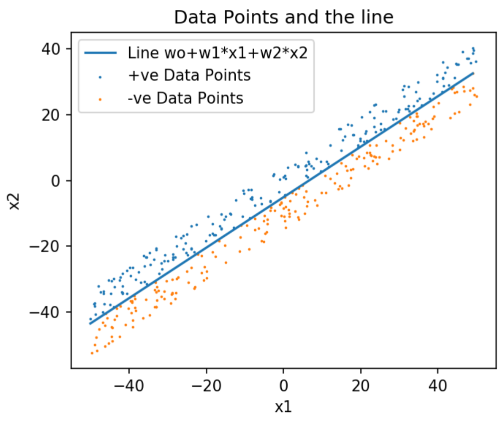
Another set of weights w0,w1,w2=15,-2.3,3 with positive slope are tested, and the results are summarized in Table 3.2 and Graphs 3.2.

	M	N	Initial Weights	Learning Rate	No of Epochs		Learnt Weights
w0,w1,w2=15,-2.3,3							
	10	10			298		[468. , -92.60688305, 123.24255228]
	200	200			44		[1009. , -151.49428576, 197.70114112]
	10	390			1999 with 5 errors		[1606., -358.40240669, 447.63442993]

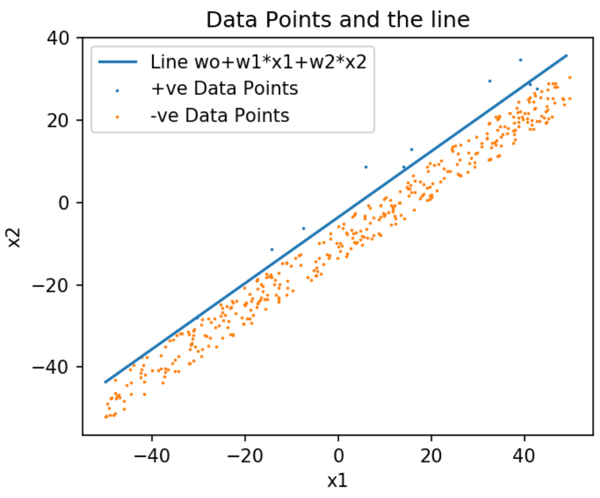
Table 3.2



M,N= (10,10)



M,N= (200,200)



M,N= (10,390)

Graphs 3.2

All the graphs are included in the data folders (refer to the readme file). A file naming convention is created for ease of reference, it is PLAplotW0_W1_W2 M_N (e.g., PLAplot16_-3_4 10_390.png) and the corresponding training data file's naming convention is just similar (e.g., train 15.0_-2.3_3.0 10_10.png).

Furthermore an additiona Python PLAII-Scipy.py used a perceptron from "sklearn.linear_model.Perceptron" for comparsion, it do not help too much for the unbalanced cases and it depends on dataset more than the perceptrons..

Conclusion

In general, the PLA algorithm is effective for the purpose. However, when the ratio of two labels in the training data is highly unbalanced, convergence speed is much slower. For this case, the maximum number of epochs tried was 1999 with four errors in the first set of weights (5,2,3) and five errors in the second set of weights (15,-2.3,3). The weights updates are oscillating without coverage after 799,600 (1999 x 400) times. Furthermore, the total number of epochs is related to the total number of training data size, but there is not obviously a linear relationship. I suspect if I shuffle the input randomly an in batches each time before feeding into the training loop, number of epochs should be more consistent.

For future improvement, adaptive learning rate, random initialization of weights, and random shuffling of input samples can be added to help to improve the algorithm's performance. A new Perceptron is created with adjustable and self-adaptive learning rate but disabled by "#" in figure 2.2. I don't have enough time to do detail evaluations although preliminary result shows that it helps to reduce error from five to one for the case in the second set of weights (15,-2.3,3) and should have an improvement for the first set's case similarly. These should be one of the next steps for more research after this project. In regarding data pre-processing, further normalization of the dataset before feed into training should help. However, current fixed zero initialization of weight make every run precisely the same without heuristics.

References

[1] https://www.csie.ntu.edu.tw/~htlin/mooc/doc/02_handout.pdf
[2] https://en.wikipedia.org/wiki/Perceptron
[3] https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53
[4] Kunihiko Fukushima, Neocognitron: A Self-organizing Neural Network Model for a Mechanism of

Pattern Recognition, 1980.