Implementing the Perceptron algorithm for finding the weights of a Linear Discriminant function

Anas Sikder
Department of CSE
Ahsanullah University of Science and Technology
Dhaka, Bangladesh
160204021@aust.edu

***Abstract*—The experiment is about to implement a perceptron algorithm for both single and batch updates and find an analysis of which one performs better. For doing so I took three cases of initial weight vectors i.e, all zeros, ones, and random with a fixed seed. I found all three cases with varying learning rates single update has an overall lower iteration to converge and performs better than a batch update.**

***Index Terms*—Single update, Batch update, Learning rate, initial Weight vector, One at a time, Many at a time, converge, Classified, Miss classified, etc.**

I. Introduction

The perceptron algorithm was invented in 1957 at the Cornell Aeronautical Laboratory by Frank Rosenblatt. It is an algorithm for supervised learning of binary classifiers. It is a linear classifier that makes all of its predictions based on a linear predictor function combining a set of weights with the feature vector.

A. Motivation

My main goal is to implement the Perceptron algorithm on given training sets for finding the weights of a linear discriminant function. Also, make an analysis of how accurate the algorithm performs. Data points for my experiments are:

$$\omega 1 = (1,1),(1,-1),(4,5)$$

$$\omega 2 = (2,2.5),(0,2),(2,3)$$

B. Theory

1) ϕ Function : ϕ function is used to make the data points into higher dimensions.For our experiment a ϕ function was given:

$$\phi = [x_1^2 \;\; x_2^2 \;\; x_1 * x_2 \;\; x_1 \;\; x_2 \;\; 1]$$

2) Normalization: For linear classifier let assume two classes of ω_1 and ω_2 .In normalization concepts one of these two classes needs negation.

3) Basic Components: There are some basic components for a perceptron.

- Input: All the features become the input of a perceptron.
- Weights: Initially starts with an initial weight vector and updated each training errors.
- Bias: Bias neuron enables a classifier to shift the decision boundary.
- Weighted Summations: The some of values after the multiplication of each weight associated with each feature value.
- Step/Activation Function:Activation function helps the neural networks non-linear.For a linear classifier, it is important to make the perceptron as linear as possible.
- Output: The weighted summation passes through the step/activation function and the values get after computation is the predicted output.

4) Single Update: Let assume a data vector 'y' i,e.y contains a set of data points $[x_1, y_1, z_1]$,$[x_2, y_2, z_2]$ and so on.And with an initial weight vector.The task is to update the weight vector until the weights are tuned and scale product of $w^T \cdot y > 0$ for each data points.The condition refers no miss classified data points.The weight updating is done based of below equation:

$$w(t+1) = w(t) + \alpha y \quad \text{if } w^T \cdot y <= 0$$

Here α is the learning rate and $0 < \alpha <= 1$. The weight vector will be updated each data points of y.It is the reason for calling it a single update or one at a time.

5) Batch Update: Let assume a data vector 'y' i,e.y contains a set of data points $[x_1, y_1, z_1]$,$[x_2, y_2, z_2]$ and so on.And with an initial weight vector.The task is to update the weight vector until the weights are tuned and scale product of $w^T \cdot y > 0$ for each data points.The condition refers no miss classified data points.The weight updating is done based of below equation:

$$w(t+1) = w(t) + \alpha \sum y \quad \text{if } w^T \cdot y <= 0$$

Here α is the learning rate and $0 < \alpha <= 1$. The weight vector will be updated after all the data points of y calculated in each iteration.It is the reason for calling it a batch update or many at a time.

II. Experimental Design / Methodology

A. Task1

All the sample points of both classes are plotted with the condition of the same class consists of the same color and

marker. From my observation, these classes are not separable with a linear boundary.

B. Task2

In the second task, I used a given ϕ function to make the data points into higher(6D) dimensions. The main reason for it is **if two classed are not separable with a linear plane data needs to be drawn in higher dimension**. And the task is done through ϕ function.

C. Task3 and Task4

In task 3 implemented the perceptron algorithm both single and batch update for finding the weight coefficient of the discriminate function for the linear classifier. Task 4 is also an extended version of task 3 where the initial weight vector had to be used as all zeros, all ones, and random value with a fixed seed. Also, the learning rate varies between .1 to 1 with a step size of .1. So I combinedly implemented both task3 and task4 i.e, for single update and batch update.

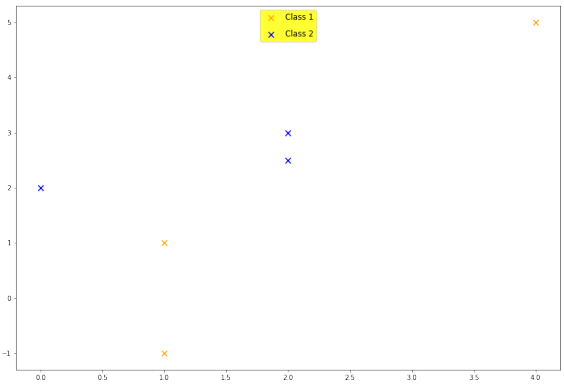

Fig. 1. Plotting of class1 and class2

III. Result Analysis

For analysis of the performance of single and batch update I took 3 cases: Initial Weight Vector All Zeros, Initial Weight Vector All Ones, Initial Weight Vector Random Values with Fixed Seed.

A. Initial Weight Vector All Zeros

For this case, the single update with various learning rates the algorithm takes 94 iterations before converging the weight vector. For a batch update, the results have variations for different learning rates. The algorithm takes 92 or 105 iterations before converging in a batch update. Let's take average in both scenarios.

$$Avg(one\ at\ a\ time) = \frac{[\sum_{n=1}^{10} 94]}{10} = \mathbf{94}$$

$$Avg(many\ at\ a\ time)\frac{[\sum_{n=1}^{7} 105] + [\sum_{n=1}^{3} 92]}{10} = 101.1 = \mathbf{101}$$

So many at a time for initial weight vector zeros needs more iteration for convergence than one at time.So for all initial weight vector zeros the case, one at a time performs better than many at a time.

B. Initial Weight Vector All Ones

For this case, the single update with varies learning rate algorithm has a variety of iterative results between 6 to 115 before converging the weight vector. For batch updates, the results have also variations for different learning rates. Let's take the average in both scenarios.

$$Avg(one\ at\ a\ time) = \frac{[\sum_{n=1}^{10} all\ iteration]}{10} = 90.5 = \mathbf{90}$$

$$Avg(many\ at\ a\ time) = \frac{[\sum_{n=1}^{10} all\ iteration]}{10} = 101.2 = \mathbf{101}$$

So many at a time for initial weight vector ones needs more iteration for convergence than one at time.So for all initial weight vector ones the case, one at a time performs better than many at a time.

C. Initial Weight Vector Random Values with Fixed Seed

For this case, the single update with varies learning rate algorithm has a variety of iterative results between 87 to 114 before converging the weight vector. For batch updates, the results have also a variation for different learning rates. Let's take the average in both scenarios.

$$Avg(one\ at\ a\ time) = \frac{[\sum_{n=1}^{10} all\ iteration]}{10} = 100.5 = \mathbf{100}$$

$$Avg(many\ at\ a\ time) = \frac{[\sum_{n=1}^{10} all\ iteration]}{10} = 109.2 = \mathbf{109}$$

So many at a time for initial weight vector random with seed fixed needs more iteration for convergence than one at time.So for all initial weight vector random values with fixed seed the case, one at a time performs better than many at a time. Below the bar chart diagram and table the results are shown:

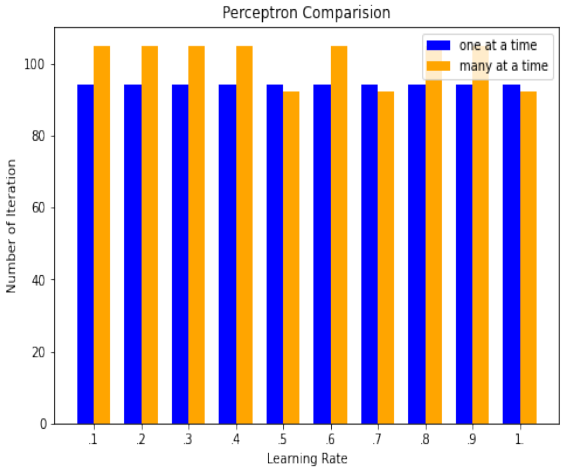

Fig. 2. Perceptron Comparison with initials weight zeros diagram

Initial Weight Vector All Zeros

	learning rate	one at a time	many at a time
0	0.1	94	105
1	0.2	94	105
2	0.3	94	105
3	0.4	94	105
4	0.5	94	92
5	0.6	94	105
6	0.7	94	92
7	0.8	94	105
8	0.9	94	105
9	1.0	94	92

Fig. 3. Perceptron Comparison with initials weight zeros table chart

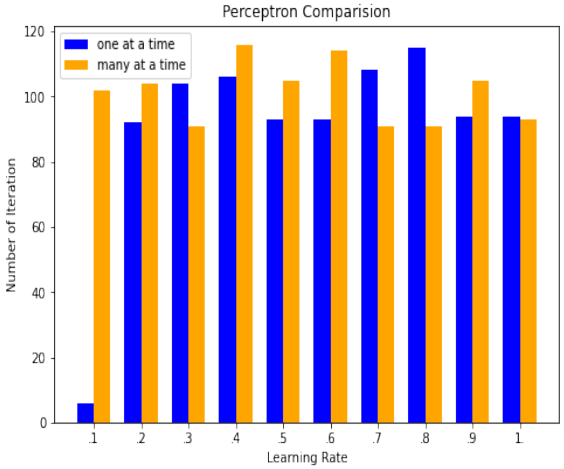

Fig. 4. Perceptron Comparison with initials weight ones diagram

Initial Weight Vector All Ones

	learning rate	one at a time	many at a time
0	0.1	6	102
1	0.2	92	104
2	0.3	104	91
3	0.4	106	116
4	0.5	93	105
5	0.6	93	114
6	0.7	108	91
7	0.8	115	91
8	0.9	94	105
9	1.0	94	93

Fig. 5. Perceptron Comparison with initials weight ones table chart

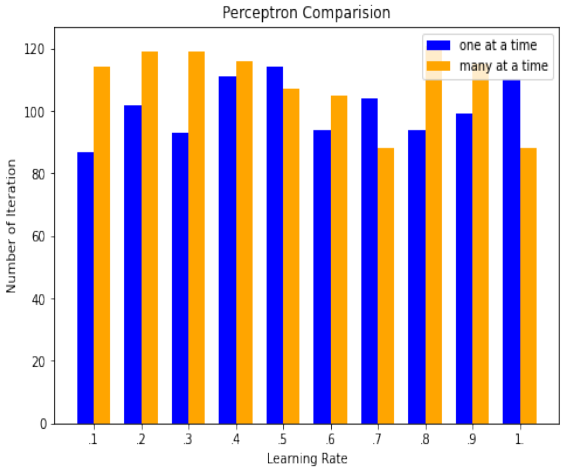

Fig. 6. Perceptron Comparison with initials weight random values with fixed seed diagram

Initial Weight Vector Random Seed Values

	learning rate	one at a time	many at a time
0	0.1	87	114
1	0.2	102	119
2	0.3	93	119
3	0.4	111	116
4	0.5	114	107
5	0.6	94	105
6	0.7	104	88
7	0.8	94	121
8	0.9	99	115
9	1.0	110	88

Fig. 7. Perceptron Comparison with initials weight random values with fixed seed table chart

IV. Conclusion

From my experiment, it is clear with different learning rate and all three cases(initial weight vector zeros, ones, and random values with fixed seed) that taking the average in both update case i.e, one at a time(single update) performs better than many at a time(batch update). Also, I took only six data points of two classes. So it is not enough data to say that all-time single performs better than batch update and **in such a small dataset more than a hundred iterations are needed. So a large dataset will need more and more iteration to converge, also may need an infinite amount of time to iterate which not possible in real life scenarios and results in miss classifications of data points**. Besides, **with a specific learning rate in all three cases(zeros, ones, and random) batch update performs better than a single update. So finally in overall most of the cases, single update needs less iteration and performs well in my experiment.**

V. Algorithm Implementation / Code

In this section snapshot of the codes are given:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from random import seed
from random import random
```

```
traindataset=pd.read_csv('train-perceptron.txt',sep=" ",header=None)
trn=traindataset.to_numpy();
#print(trn)
```

```
class1 =[([i[0],i[1],i[2]]) for i in trn if i[2] == 1]
class1 = np.array(class1)
class2 =[([i[0],i[1],i[2]]) for i in trn if i[2] == 2]
class2 = np.array(class2)
Trainx = np.concatenate((class1, class2))
#print(Trainx)
```

```
fig, axis = plt.subplots()
fig.set_figheight(10)
fig.set_figwidth(15)

#scatting train values
axis.scatter(class1[:,0],class1[:,1],marker='x',color='orange',s=70,label='Class 1')
axis.scatter(class2[:,0],class2[:,1],marker='x',color='blue',s=70,label='Class 2')
#Labeling decoration
legend = axis.legend(loc='upper center',fontsize='large',labelspacing=1.0)
legend.get_frame().set_facecolor('yellow')
```

Fig. 8. Code Snap1

```
y=np.ones([6,6])
y[:,0]=Trainx[:,0]**2
y[:,1]=Trainx[:,1]**2
y[:,2]=Trainx[:,0]*Trainx[:,1]
y[:,3]=Trainx[:,0]
y[:,4]=Trainx[:,1]
y[3:6,:]*=-1
#print(a)
```

```
wz  =  np.zeros([6])
wo  =  np.ones([6])
seed(2)
ra  =  []
for  _  in  range(6):
     value  =  random()
     ra.append(value)

wr  =np.array(ra)
#print(wr)

TrainX  =  np.concatenate((class1,  class2))
w  =np.concatenate((wz,wo,wr))
w  =  w.reshape(3,6)
#print(w.ndim)
#print(w)
alp  =np.round(np.arange(.1,1.1,.1),1)
#print(alp)
totalconverge=[]
for  i  in  w:
     converge=[]
     #print(i)
```

Fig. 9. Code Snap2

```
        cc=0
        itr=0
        while itr<200 and cc<len(y):
            itr += 1
            cc=0
            for ix in y:

                g =np.dot(ix,np.transpose(temp).reshape(6,1))
                #print(temp)
                if g <= 0:
                    temp = temp +j*ix
                    # print(temp)
                else:
                    cc = cc +1
        converge.append(itr)
        #print(converge)
    totalconverge.append(converge)
#print(totalconverge)
```

```
totalconverge1 = []
for i in w:
    converge1=[]
    #print(i)
    for j in alp:
        #print(j)
        temp = i
        itr=0
        flag=False
        while itr<200 and flag!=True:
            itr += 1
            g =np.dot(y,np.transpose(temp)).reshape(6,1)
                #print(temp)
            mc=np.array(np.where(g<=0.0))
            temp = temp+np.sum(y[mc[0],:],axis=0)*j
            #print(temp)
            if len(mc[0]) == 0 :
                flag = True
        converge1.append(itr)
        #print(converge1)
    totalconverge1.append(converge1)
#print(totalconverge1)
```

Fig. 10. Code Snap3

```
x=np.arange(10)
print(x)
width = 0.35
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
ax.bar(x-width/2,totalconverge[0],width,color="blue")
ax.bar(x+width/2,totalconverge1[0],width,color="orange")
plt.xticks(x, ('.1', '.2', '.3', '.4','.5','.6','.7','.8','.9','1.'))
ax.legend(labels=['one at a time', 'many at a time'])
ax.set_xlabel('Learning Rate')
ax.set_ylabel('Number of Iteration')
ax.set_title('Perceptron Comparision')



#ax.bar(alp,tconverge[1],width)
plt.show()
```

```
x=np.arange(10)
#print(x)
width = 0.35
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
ax.bar(x-width/2,totalconverge[1],width,color="blue")
ax.bar(x+width/2,totalconverge1[1],width,color="orange")
plt.xticks(x, ('.1', '.2', '.3', '.4','.5','.6','.7','.8','.9','1.'))
ax.legend(labels=['one at a time', 'many at a time'])
ax.set_xlabel('Learning Rate')
ax.set_ylabel('Number of Iteration')
ax.set_title('Perceptron Comparision')

#ax.bar(alp,tconverge[1],width)
plt.show()
```

Fig. 11. Code Snap4

```
x=np.arange(10)
#print(x)
width = 0.35
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
ax.bar(x-width/2,totalconverge[2],width,color="blue")
ax.bar(x+width/2,totalconverge1[2],width,color="orange")
plt.xticks(x, ('.1', '.2', '.3', '.4','.5','.6','.7','.8','.9','1.'))
ax.legend(labels=['one at a time', 'many at a time'])
ax.set_xlabel('Learning Rate')
ax.set_ylabel('Number of Iteration')
ax.set_title('Perceptron Comparision')
#ax.bar(alp,tconverge[1],width)
plt.show()
```

```
print('\033[1m' + 'Initial Weight Vector All Zeros' + '\033[0m')
plotdata = pd.DataFrame({
    "learning rate":alp.tolist(),
     "one at a time":totalconverge[0],
    "many at a time":totalconverge1[0]
    },

)
plotdata
```

```
print('\033[1m' + 'Initial Weight Vector All Ones' + '\033[0m')
plotdata = pd.DataFrame({
    "learning rate":alp.tolist(),
     "one at a time":totalconverge[1],
    "many at a time":totalconverge1[1]
    },

)
plotdata
```

```
print('\033[1m' + 'Initial Weight Vector Random Seed Values' + '\033[0m')
plotdata = pd.DataFrame({
    "learning rate":alp.tolist(),
     "one at a time":totalconverge[2],
    "many at a time":totalconverge1[2]
    },

)
plotdata
```

Fig. 12. Code Snap5