Computational Intelligence First assignment

Mohammad ali farahat

97521423

---

1) In this question, I created a Perceptron class that name is NOR.
This class has a method for training the perceptron, the method gets two inputs, one of them is training data and another one is the label of that data.
I used gradient descent algorithm to train my perceptron.
First, it calculates the sum of multiply of weights and inputs (dots) with NumPy. (the first weight is for bios)
After that calculates the error (difference of dots and label). Then update the weights with this formula :

$$w(n+1) = w(n) + \eta x(n) e(n)$$

We had four inputs to train a NOR perceptron:

[-1, -1] → 1
[-1, 1] → -1
[1, -1] → -1
[1, 1] → -1

I use -1 instead of 0, because weights won't update if they multiply by zero.
In each iteration, we pass all of this data to perceptron and its weights become more accurate each time.
This algorithm works until the largest weight change in one iteration becomes less than a specific limit (in this case I set the limit to 0. 000000001)

- One problem that I had was the learning rate, when learning rate was constant, the algorithm never ends. Actually, it jumps over it and never ends. So I decrease it every time learn method calls. And problem solved!

- Comments in codes shows everything clearly

- To test the program, I print some information during running, and to increase performance I commented them now , but they print something like this in each iteration:

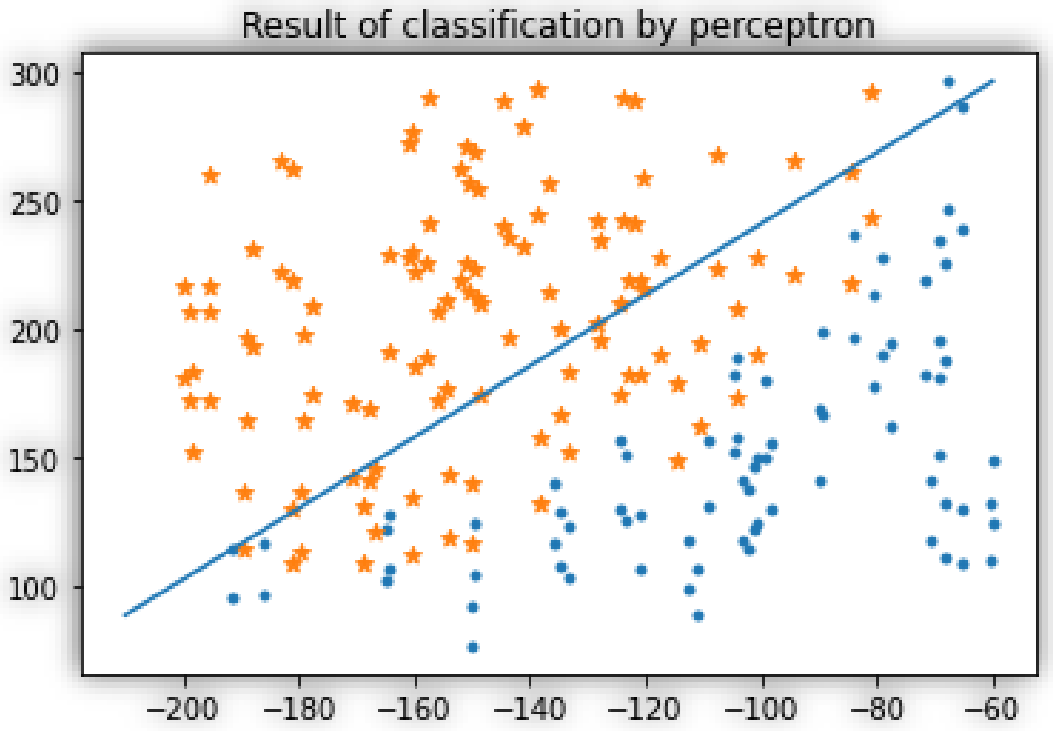
```
**********
[-1, -1] ==> 1
error:  0.5118076223298117
weight before:  [-0.50000403 -0.49606547 -0.49213094]
learning rate:  0.01535180506892185
max_change :  0.007873094910403333
weight after:  [-0.49213899 -0.5039305  -0.49999597]
**********
[-1, 1] ==> -1
error:  -0.5117955359098711
weight before:  [-0.49213899 -0.5039305  -0.49999597]
learning rate:  0.015336453263852929
max_change :  0.007865035886682359
weight after:  [-0.49999598 -0.49607352 -0.50785296]
**********
[1, -1] ==> -1
error:  -0.5117834621468244
weight before:  [-0.49999598 -0.49607352 -0.50785296]
learning rate:  0.015321116810589076
max_change :  0.007856985302432784
weight after:  [-0.50784492 -0.50392246 -0.50000402]
**********
[1, 1] ==> -1
error:  0.511771401026833
weight before:  [-0.50784492 -0.50392246 -0.50000402]
learning rate:  0.015305795693778487
max_change :  0.007848943148427656
weight after:  [-0.50000401 -0.49608155 -0.49216311]
```

- Finally, after 4430 iterations, the output is like below and if you run the program, you'll see this:

```
iterations:  4430
final weights:  [-0.5 -0.5 -0.5]
```

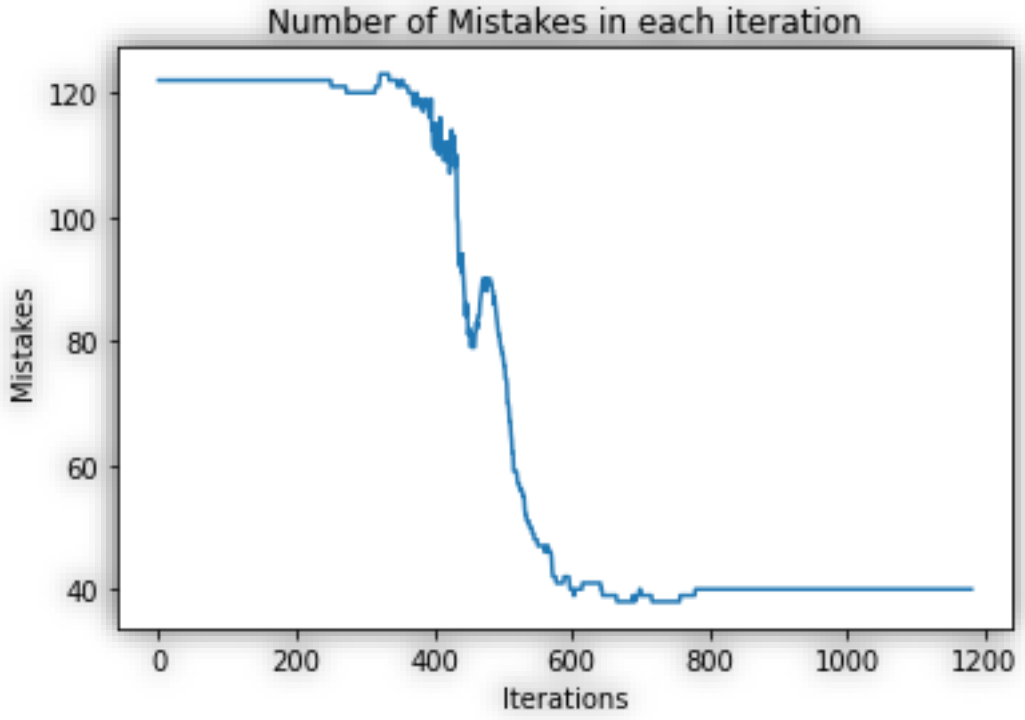
2) In this question we have two class of points and we want to classify them by perceptron algorithm.
Like Q1, I design a perceptron class (I said details in last part).
First, I open data file and read it (probably you cannot run program because you don't have data file in correct directory), and then parse it.
I changed all 0 labels to -1.
There are 200 input data and we train perceptron with them in each iteration.
In my case, after 1182 iteration algorithm stopped (because it reached maximum weight change limit) and after that I print number of iterations and two plots:

- Result of classification that have points and separator Line



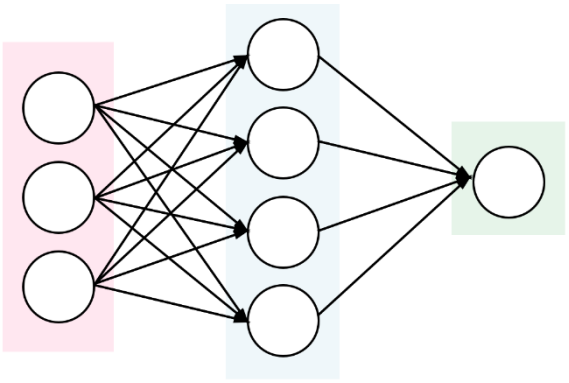
We see that perceptron works fine and separated points in 2 class

- plot of mistakes for each iteration

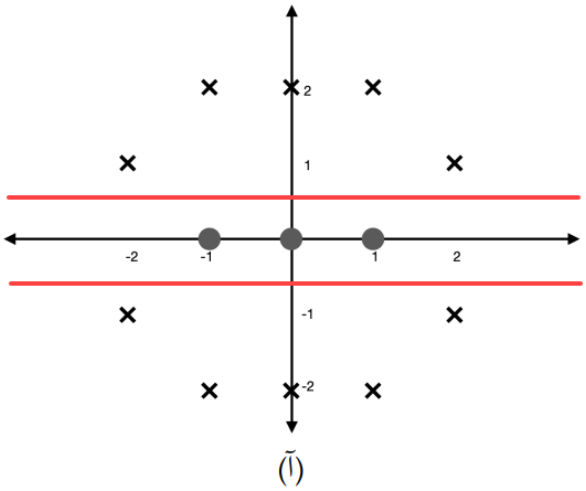


At first mistakes are 120, but after 300 iteration mistakes decreased and after about 600 iterations, they become about 40. And after that, they are stabled there.

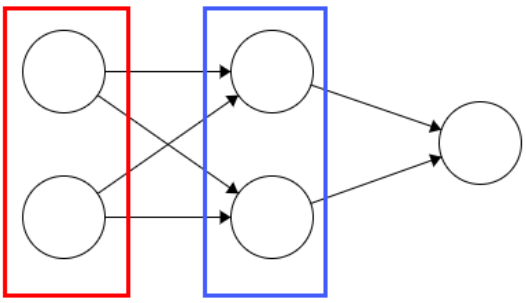
3) Madaline is several Adaline in parallel. Madaline has one hidden layer and actually it separates one convex shape from others. Madaline has one output (like Adaline). So, if our data points are separatable by one convex shape (the lines does not have to be closed, they can be two line) , Madaline can do it.



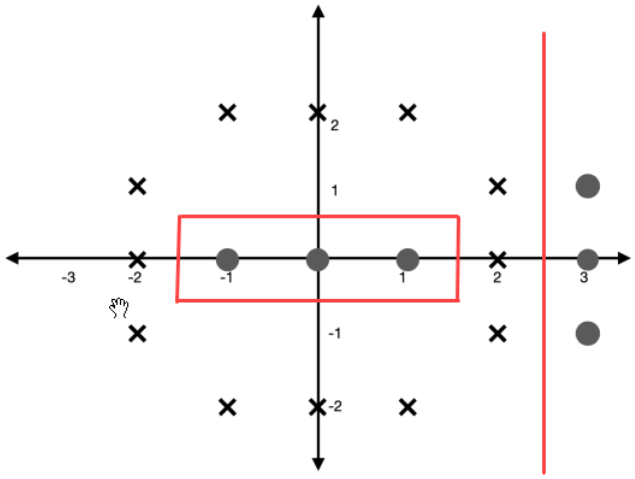
A) ✓ this example can be solved by Madaline. neural network has two input neurons in first layer, two neurons in hidden layer (for each red line in left plot) and one final neuron for combine these two lines.



(ĩ)



B) ✗ This example cannot solve by Madaline, because we need one convex for three inside point and one more line for three points on the right, so we need more than one hidden layer. So, Madaline cannot work here.



(ب)

4) In this question we used Keras library to create and train our MLP and test mnist dataset.
First of all, we extract our mnist dataset from Keras library itself and reshape it to fit in Keras model.
For optimization I normalized inputs by dividing them by 255 (inputs becomes between 0 and 1)

Then, we create a sequential model. Keras library describes sequential model like this:

> "A Sequential model is appropriate for a plain stack of layers where each layer has exactly one input tensor and one output tensor."

After that we add layers, for that we use "Dense" method. Inputs of this method are number of neurons, activation function of each layer and …

I used 2 layer (except inputs), hidden layer activation function is relu and output layer activation function is softmax. The input layer has 784 (28*28) neurons, the hidden layer has 85 neurons and the output has 10 neurons for 10 digits. The accuracy reached 100% that is amazing! And loss becomes 3.9051e-05 . the whole training and testing process completed in 2m 26s.

After that we compile model (for getting ready and optimization).
Then our model is ready to train and after that for tests.

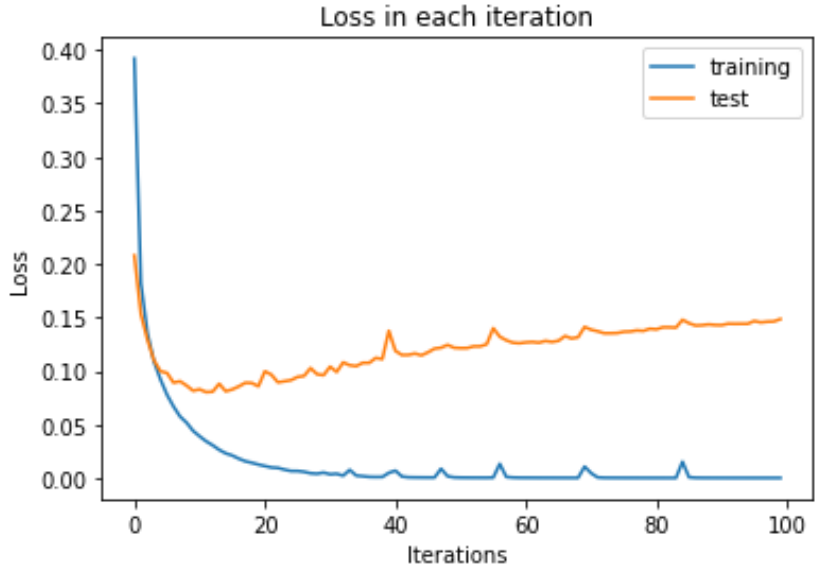
```
Model: "sequential_10"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_27 (Dense)             (None, 85)                66725
_________________________________________________________________
dense_28 (Dense)             (None, 10)                860
=================================================================
Total params: 67,585
Trainable params: 67,585
Non-trainable params: 0
```

You can see some of output here:

```
Epoch 1/100
469/469 [==============================] - 2s 3ms/step - loss: 0.6874 - acc: 0.8078 - val_loss: 0.2080 - val_acc: 0.9420
Epoch 2/100
469/469 [==============================] - 1s 3ms/step - loss: 0.1919 - acc: 0.9450 - val_loss: 0.1531 - val_acc: 0.9555
Epoch 3/100
469/469 [==============================] - 2s 3ms/step - loss: 0.1423 - acc: 0.9586 - val_loss: 0.1293 - val_acc: 0.9620
Epoch 4/100
469/469 [==============================] - 1s 3ms/step - loss: 0.1067 - acc: 0.9696 - val_loss: 0.1101 - val_acc: 0.9686
Epoch 5/100
469/469 [==============================] - 1s 3ms/step - loss: 0.0915 - acc: 0.9732 - val_loss: 0.0997 - val_acc: 0.9711
Epoch 6/100
469/469 [==============================] - 1s 3ms/step - loss: 0.0793 - acc: 0.9778 - val_loss: 0.0982 - val_acc: 0.9705
Epoch 7/100
469/469 [==============================] - 1s 3ms/step - loss: 0.0647 - acc: 0.9815 - val_loss: 0.0891 - val_acc: 0.9742
Epoch 8/100
469/469 [==============================] - 1s 3ms/step - loss: 0.0540 - acc: 0.9850 - val_loss: 0.0904 - val_acc: 0.9729
Epoch 9/100
469/469 [==============================] - 1s 3ms/step - loss: 0.0499 - acc: 0.9857 - val_loss: 0.0864 - val_acc: 0.9736
```

*As you can see, accuracy is increasing, and loss is decreasing.

We can see results here:



5) In this question I build the MLP myself from scratch and everything totally done by myself. But it does not run very well. So here is description of what I did:

- I created a class of MLP and implemented forward and backward methods.
- This class has 3 layers (just like Q4), and the hidden layer has 85 neurons.
- w1, w2, b1, b2 are weights of between input and hidden layer, weights of between hidden layer and output layer, bios of first layer, bios of second layer.
- net is result of dot of wights of last layer and output of each neuron in last layer.
- Out is activationFunction(net) :)
- Forward method: this method calculates net of each layer, step by step. And I used numpy for calculate of net in each layer (not for loops).
- After calculating output of last layer, we have to calculate error:

$$E(n) = \frac{1}{2} \sum_{j \text{ output node}} e_j^2(n)$$

- Now it's time to back propagate the error and update the weights.
  - for this part I learned back propagation algorithm by one example from this site.
  - we calculate the error and now we use it to update the weights by back propagation And gradient descent algorithm.
  - this formula is used to update weights :

$$w_{ji} = w_{ji} + \Delta w_{ji}$$

$$\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}}$$

- now we have to calculate gradient :

$$\delta_j = \begin{cases} \varphi'(v_j)(d_j - y_j) & \text{IF j output node} \\ \varphi'(v_j) \sum_{k \text{ of next layer}} \delta_k w_{kj} & \text{IF j hidden node} \end{cases}$$

when we are coming back from last layer to update w2 , we have to calculate 3 part , and multiply them (vectorize) , first part is derivative of error in last layer, it is: $out - target$

second part is derivative of activation function that we use in last layer, and it is softmax: $\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1})$

third part is derivative of net2, and it is always out1 (because the only coefficient of w2 is out1, in this equation) :

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial w_5} = 1 * out_{h1} * w_5^{(1-1)} + 0 + 0 = out_{h1}$$

I calculate this parts and new weights here:

```
# calculate gradient for first layer back
t1 = np.subtract(self.out2, self.Y)
t2 = np.multiply(self.out2, np.subtract(1, self.out2))
a = np.multiply(t1, t2)
wtf = np.dot(a,self.out1.T)
changes = wtf * self.etha

# calculate new w2
w2_n = np.subtract(self.w2, changes)
```

For update weights of next layer (w1), we have to calculate these parts again, plus, we have to sum errors that propagated from last layer.
So, first of all we dot part1*part2 of last layer into w2 (we must not have update weights until end, after all calculations we'll update them), this is first part of this layer

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

The second part and the third parts are like last layer,
Derivative of activation function of hidden layer, we used sigmoid here, so the derivative is like part 2 of last layer:

$$\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1})$$

And third part is inputs, because they are actually output of first layer.
* we update bios by part1 and part2 in both layers.

Here is code of updating weights of w1:

```
# calculate gradient for second layer back
x1 = np.dot(self.w2.T, a)
x2 = np.multiply(self.out1, np.subtract(1, self.out1))
a2 = np.multiply(x1, x2)
wtf2 = np.dot(a2, self.input.T)
changes2 = wtf2 * self.etha

# calculate new w1 and update it
self.w1 = np.subtract(self.w1, changes2)
```

- Now we can update the weights, so we start training the dataset.

  I parse input data and read it line by line, and traing my MLP (look at my main)

  The accuracy is about 10%

Output should look like this :

```
 1 --> ****************************************  loss: 0.5424740593780816   acc: 0.0997
 2 --> ****************************************  loss: 0.5423615766377766   acc: 0.09965
 3 --> ****************************************  loss: 0.5423699597680387   acc: 0.0995
 4 --> ****************************************  loss: 0.5423533264957455   acc: 0.0996
 5 --> ****************************************  loss: 0.5423726840564175   acc: 0.0995
 6 --> ****************************************  loss: 0.5423906508209957   acc: 0.09955
 7 --> ****************************************  loss: 0.5423932748744666   acc: 0.09955
 8 --> ****************************************  loss: 0.5423818596070799   acc: 0.09955
 9 --> ****************************************  loss: 0.5424571373963769   acc: 0.0996
10 --> ****************************************  loss: 0.5423836060948708   acc: 0.09965
11 --> ****************************************  loss: 0.5423664704727614   acc: 0.0996
12 --> ****************************************  loss: 0.542385195908452   acc: 0.0996
13 --> ****************************************  loss: 0.5423841647245811   acc: 0.09955
14 --> ****************************************  loss: 0.5423482160132768   acc: 0.0997
15 --> ****************************************  loss: 0.5423493801406153   acc: 0.0996
16 --> ****************************************  loss: 0.5423267227925135   acc: 0.0996
17 --> ****************************************  loss: 0.542378360446598   acc: 0.09955
18 --> ****************************************  loss: 0.5423268802365636   acc: 0.0994
19 --> ****************************************  loss: 0.5423636544781425   acc: 0.09955
20 --> ****************************************  loss: 0.5423807560201229   acc: 0.0997
tests acc=  0.0962
```