University of Tehran

Electrical and Computer Engineering Department

Neural Networks and Deep Learning

Homework 1

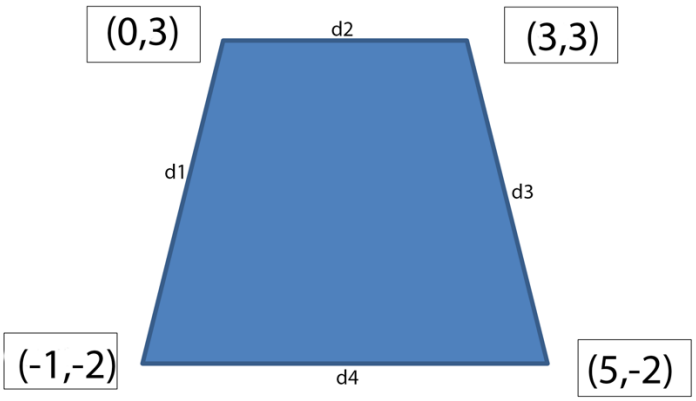
Name	Danial Saeedi
Student No	810198571
Date	Tuesday, October 26, 2021

Table of contents

1. McCulloch-Pitts Neuron

1.1 Task

The task is to design a McCulloch-Pitts Neural Network such that it can separate inside point from outside of this Trapezoid.



1.2 Model Architecture

Let's choose $\Theta = 0$ for the neuron's threshold. The activation function of each output neuron is:

$$d_i(net_i) = sign(net_i) = \begin{cases} 1 & net_i > \Theta \\ 0 & net_i = \Theta \\ -1 & net_i < \Theta \end{cases}$$

$$net_i = w_{i1}x + w_{i2}y + b_j$$

Each output neuron represents one of these lines:

$$d_1: y - 5x - 3 = 0$$

$$d_2: y - 3 = 0$$

$$d_3: y + 2.5x - 10.5 = 0$$

$$d_4: y + 2 = 0$$

The weights and biases of each output neuron are adjusted according to their coefficient in the line equation.

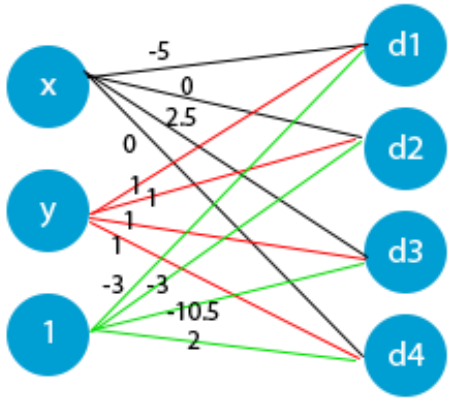


Figure 1.1 The AdaLine Neural Network

1.2.1 How to determine whether a 2D Point $A(x, y)$ is within the Trapezoid

We know that if the net_i of each output neuron is greater than 0, that means A is above the line, otherwise it's below the line. If $net_i = 0$ then A is exactly on line d_i

If A is below d_1, d_2, d_3 and above d_4 then A is inside the Trapezoid. Otherwise it is outside of this area. So the output of d_1, d_2, d_3 should be less than or equal to 0 and he output of d_4 should be greater or equal to 0.

1.2.2 Forward Propagation

$$\begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \end{bmatrix} = sign(\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix})$$

1.3 Implementation

The notebook for this question can be found in notebooks/1.McCulloch-Pitts.ipynb

```
class NeuralNetwork() :

    def __init__(self, weights, biases, in_features, out_features) :
        #Seed
        np.random.seed(42)

        self.weights = weights
        self.biases = biases
        self.in_features = in_features
        self.out_features = out_features

    def forward(self, x) :
        mult = np.sign(np.matmul(self.weights,x)+self.biases)

        return mult

    """
    Checks if a the given point is inside or not.
    If the point is inside the output is 1 and otherwise 0.
    d1 : arr[0]
    d2 : arr[1]
    d3 : arr[3]
    d4 : arr[4]
    """
    def predict(self,x) :
        result = []
        for point in x :
            arr = self.forward(point)
            result.append(arr[0] <= 0 and arr[1] <= 0 and arr[2] <= 0 and arr
[3] >= 0)

        return result

weights = np.array([
    [-5,1],
    [0,1],
    [2.5,1],
    [0,1]
])
biases = [-3,-3,-10.5,2]
in_features = 2
out_feature = 4

model = NeuralNetwork(weights,biases, in_features, out_feature)
```

1.4 Testing some sample points

```
sample_point = [
    [10,10],
    [2,2]
]

model.predict(np.array(sample_point))
```

Output : [False, True]

2. Binary Classification Using AdaLine Algorithm

2.1.1. Making the Data

- The first category has 100 points such that x ~N(μ=2,σ2=0.25) and y ~N(μ=0,σ2=0.04).
- The second category has 30 points such that x~N(μ=0,σ2=0.01) and y~N(μ=1,σ2=0.49).

Figure 2.1 The generated data points

	x	y	label
0	1.882923	-0.032257	-1
1	1.882932	0.080810	-1
2	2.789606	0.377237	-1
3	2.383717	0.034916	-1
4	1.765263	0.051510	-1
...	...	...	...
25	0.067960	1.542244	1
26	-0.073037	0.351149	1
27	0.021646	0.958332	1
28	0.004557	-1.268887	1
29	-0.065160	0.282929	1

130 rows × 3 columns

2.1.2. Plotting the Dataset

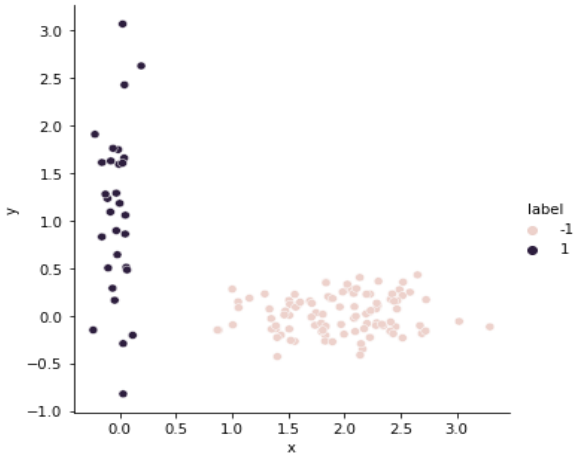


Figure 2.2 Plotting the Dataset

2.2 Implementation

The notebook for this question can be found in notebooks/2.AdaLine-BinaryClassification.ipynb

```
class AdaLineNetwork() :

    def __init__(self, in_features, out_features, seed = 42) :
        #Seed
        np.random.seed(seed)

        self.in_features = in_features
        self.out_features = out_features
        # Initialize Weights & Biases
        self.init_wad()

    # Initialize Weights & Biases
    def init_wad(self) :
        self.weights = np.random.uniform(low=-1, high=1, size=(in_features*ou
t_features,)).reshape(out_features,-1)
        self.biases = np.random.uniform(low=-1, high=1, size=(out_features,))

        #Learning Rate
        self.lr = 0.01

    def h(self,num) :
        if num >= 0 :
            return 1
        else :
            return -1
    def forward(self, x) :
        net = np.matmul(self.weights,x)+self.biases
        prediction = pd.Series(np.matmul(self.weights,x)+self.biases)
        prediction = prediction.apply(self.h).values
        return prediction,net

    def backward(self,t, net, xi) :
        self.weights = self.weights + self.lr*(t-net)*xi
        self.biases = self.biases + self.lr*(t-net)

    def train(self,df,epochs = 1) :
        error_per_epoch = []
        for epoch in range(0,epochs):
            # Evaluate
            error_per_epoch.append(self.evaluate(df))

            for index, row in df.iterrows():
                # Forward Propagation
                prediction,net = self.forward(row[['x','y']].values)
                # Backpropagation
                self.backward(row['label'],net,row[['x','y']].values)
```

```
        return error_per_epoch

    def calc_error(self,target,net) :
        return (np.square(target-net)/2)

    def evaluate(self,df) :
        error = 0
        for index, row in df.iterrows():
            #Forward Propagation
            t,net = self.forward(row[['x','y']].values)
            error += model.calc_error(t,net)[0]

        return error

    def predict(self,df) :
        result = []
        for index, row in df.iterrows() :
            target,net =  self.forward(row[['x','y']].values)
            result.append(target[0])

        return result
```

2.2.2. Initializing the model

This is a binary classification problem. So we need only 1 output neuron.

```
in_features = 2
out_features = 1
model = AdaLineNetwork(in_features, out_features)
```

2.3.1. Part A) Training

The model was trained for 5 epochs. As you can see, after 3 epochs the error stays the same.

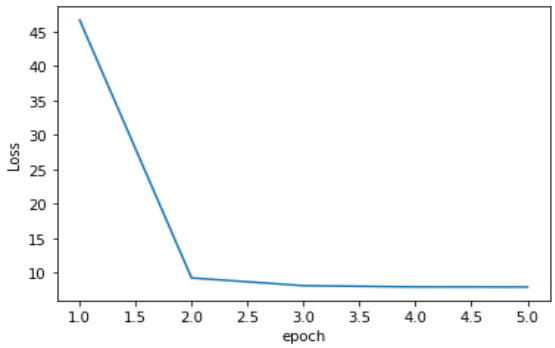


Figure2.3 Loss per epoch

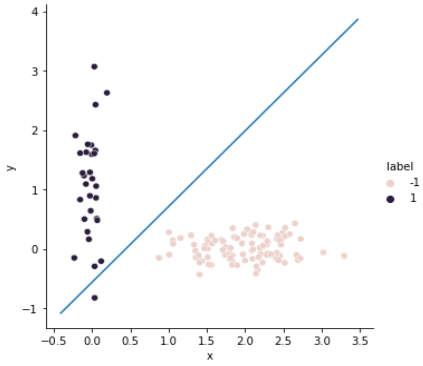


Figure 2.4 Separator Line

2.4. Evaluating the Model

	precision	recall	f1-score	support
-1	0.99	1.00	1.00	100
1	1.00	0.97	0.98	30
accuracy			0.99	130
macro avg	1.00	0.98	0.99	130
weighted avg	0.99	0.99	0.99	130

Figure 2.5 Classification Report

2.5. Part B : Is AdaLine Algorithm good for seperating the two categories?

The F1-Score of this model is 0.99, which is reasonable. But a perfect lines exists and this model is unable to find it. So AdaLine is not a good choice for this question.

The solution is to replace the “sign” function with a “soft sign” such as “tanh”. Unlike “sign”, “tanh” is smooth and one can compute the new updating rules by Gradient Descent.

$$J_p(\boldsymbol{w}, b) = 0.5\big(t(p) - \tanh(\gamma net(x(p), w, b)\big)^2$$

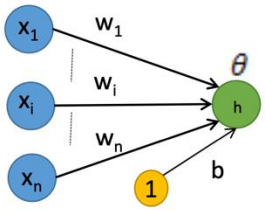
$$\delta w_i = w_i^+ - w_i^- = -\alpha \frac{\partial J_p(\boldsymbol{w}, b)}{\partial w_i}\bigg|_{w^-, b^-} = \alpha\big(1 - h^2\big)\gamma(t - h)x_i$$

$$\delta b = b^+ - b^- = -\alpha \frac{\partial J_p(\boldsymbol{w}, b)}{\partial b}\bigg|_{w^-, b^-} = \alpha\big(1 - h^2\big)\gamma(t - h)$$

γ should be chosen big because “tanh” will be near to a “sign” function. Now, by decreasing the loss function, the output of network will converge to the target value.

3. Perceptron

3.1. Linear Perceptron Network



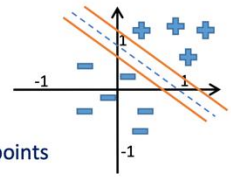
$$\text{net} = w_1x_1 + \cdots . + w_ix_i + \cdots + w_nx_n + b$$

$$(\#)\quad h = \begin{cases} 1 & \text{if} \quad \text{net} > \theta \quad \text{Active} \\ 0 & \text{if} \quad |\,\text{net}\,| < \theta \quad \text{non - descision} \\ -1 & \text{if} \quad \text{net} < -\theta \quad \text{Passive} \end{cases}$$

$$\theta : \text{non} - \text{negative threshold}$$

Example $\quad \text{net} = x_1 + x_2 - 1$

$\theta = 0.2$



- In linear perceptron network, two parallel lines (with a distance depended to θ) separate feature points of two classes.
- Bigger threshold (θ) makes larger distance between lines.
- The considered gap 2θ between lines forces a conservative margin in separation of two classes

3.2. Part A: Updating rule for the weights and bias in Perceptron

- Weights : $w_i(new) = w_i(old) + \alpha . x_i . t$
- Bias : $b(new) = b(old) + \alpha . t$
- $0 < \alpha \leq 1$

Where x_i denotes ith input, t denotes the target value for output and α denotes the learning rate.

Step 0- initialize the weights, the bias term, the learning rate and the threshold

$$w_i = 0, \quad b = 0, \quad \theta = 0 \quad \text{and} \quad \alpha = 1$$

Step 1- for each (input , target) training set, {(s,t)}, do steps 2, 3 and 4.

Step 2- Assign $s_i \rightarrow x_i$ for $i = 1, 2, \cdots, n$

Step 3- Compute the output vector, h, according to equation (#)

Step 4- if the error, (h-t), is not zero update the weights and the bias term according to equations (*),(**)

Step 5- if the computed error terms for all training pairs (s, t) are zero go to "End" else go to Step1.

Step 6- End.

3.3. Part B

Suppose a perceptron network has 3 inputs. The weights and bias of this network is $w_1 = 0.1, w_2 = 0.6, w_3 = 0.3, b = -0.5$ respectively. And the learning rate is $\alpha = 0.2$.

The threshold of the activation function is $\theta = 0$

Train the network with this input (0,3,3) twice and update weights and bias. The output of this input is -1.

First epoch:

$$w_1 = 0.1 + 0.2 * 0 * (-1) = 0.1$$

$$w_2 = 0.6 + 0.2 * 3 * (-1) = 0$$

$$w_3 = 0.3 + 0.2 * 3 * (-1) = -0.3$$

$$b = -0.5 + 0.2 * (-1) = -0.7$$

Second epoch:

$$w_1 = 0.1 + 0.2 * 0 * (-1) = 0.1$$

$$w_2 = 0 + 0.2 * 3 * (-1) = -0.6$$

$$w_3 = -0.3 + 0.2 * 3 * (-1) = -0.9$$

$$b = -0.7 + 0.2 * (-1) = -0.9$$

3.4. Implementation

The notebook for this question can be found in notebooks/3.Perceptron.ipynb

```
class PerceptronNetwork() :

    def __init__(self, weights,bias, in_features, out_features, seed = 42) :
        #Seed
        np.random.seed(seed)

        self.in_features = in_features
        self.out_features = out_features
        # Initialize Weights & Biases
        self.weights = weights
        self.bias = bias

        #Learning Rate
        self.lr = 0.2

    # Activation Function
    def h(self,num, theta = 0) :
        if num >= theta :
            return 1
        else :
            return -1

    def backward(self,t, xi) :
        coefficient = self.lr*t
        self.weights = np.round(self.weights + coefficient*xi,decimals = 2)
        self.bias = round(self.bias + coefficient,2)

    def train(self,t,x, epochs) :
        for epoch in range(0,epochs):

            self.backward(t,x)
            print("Epoch:",epoch+1)
            print(f'w1 : {self.weights[0]}, w2 : {self.weights[1]} , w3 :
{self.weights[2]} , b : {self.bias}')
```

3.5. Initializing the model

```
in_features = 3
out_features = 1
weights = np.array([0.1,0.6,0.3]).reshape(-1,1)
bias = -0.5
model = PerceptronNetwork(weights, bias, in_features, out_features)
```

3.6. Training the model

```
epochs_ = 2

model.train(-1,np.array([0,3,3]).reshape(-1,1),epochs_)
```

```
Epoch: 1
w1 : [0.1], w2 : [-0.] , w3 : [-0.3] , b : -0.7
Epoch: 2
w1 : [0.1], w2 : [-0.6] , w3 : [-0.9] , b : -0.9
```

4. MadaLine

The notebook for this question can be found in notebooks/4.MadaLine.ipynb

In MRI Algorithm only the weights and biases for the hidden AdaLines are adjusted. The weights for the output unit are fixed.

4.1. Model Architecture

The input of our neuron is x and y so we need 2 neurons as input. And we need 6 lines to separate 3 different regions thus we need 6 neurons in the hidden layer. There are 3 distinctive classes we should classify so we need 2 output neurons.

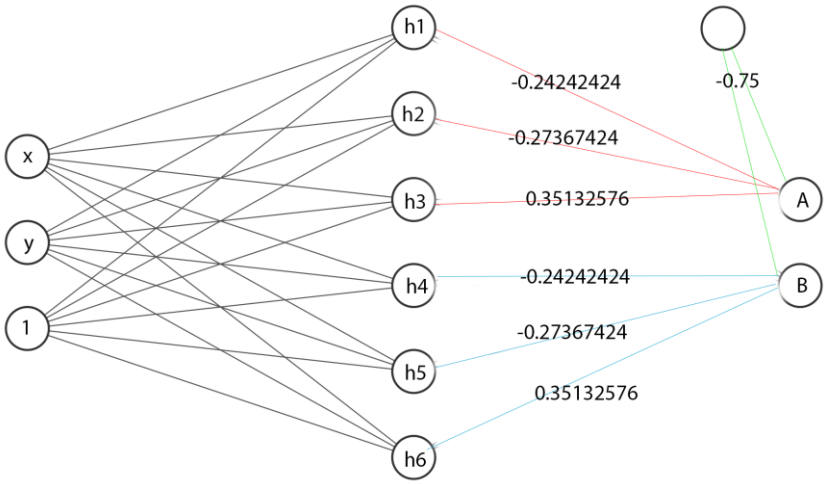


Figure 4.1 Model Architecture

The boolean function of neuron A and B is:

$$f_A(h_1, h_2, h_3) = \overline{h_1}.\overline{h_2}.h_3$$

$$f_B(h_4, h_5, h_6) = \overline{h_4}.\overline{h_5}.h_6$$

The weights and biases of these two output neurons are learned separately using AdaLine's Learning Algorithm.

4.2 Generating the data points

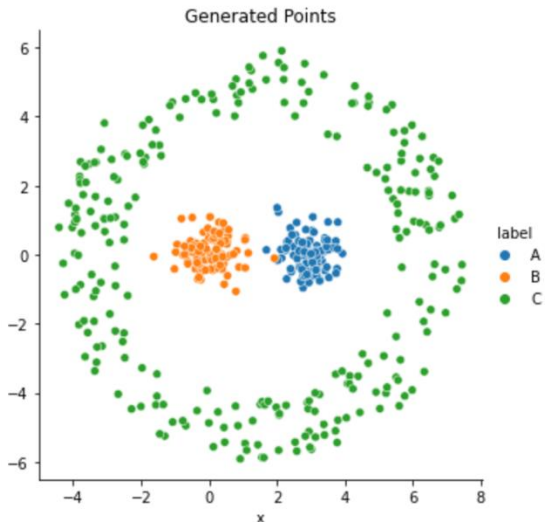


Figure 4.2 Generated Points

4.3. Calculating the output weights & biases for output layer

We are trying to calculate the weights and biases for this Boolean functions :

$$f_A(h_1, h_2, h_3) = \overline{h_1}.\overline{h_2}.h_3$$

$$f_B(h_4, h_5, h_6) = \overline{h_4}.\overline{h_5}.h_6$$

The truth table of this function is stored at notebooks/data/CustomGate.csv.

	x	y	z	label
0	1	1	1	-1
1	1	1	-1	-1
2	1	-1	1	-1
3	1	-1	-1	-1
4	-1	1	1	-1
5	-1	1	-1	-1
6	-1	-1	1	1
7	-1	-1	-1	-1

Figure 4.2 Truth table of the boolean function

After 10 epochs of training, the weights and biases are learned :

```
In [339]: from sklearn.metrics import classification_report
          y_pred = model.predict(train_custom_gate)
          y_true = train_custom_gate['label'].values
          print(classification_report(y_true, y_pred, labels=[-1,1]))
```

```
              precision    recall  f1-score   support

          -1       1.00      1.00      1.00         7
           1       1.00      1.00      1.00         1

    accuracy                           1.00         8
   macro avg       1.00      1.00      1.00         8
weighted avg       1.00      1.00      1.00         8
```

Figure 3.4 Classification Report

4.5. MadaLine Implementation

For Learning the hidden layer's weights and biases, I used MRI Algorithm.

Summary of MRI Algorithm

Set learning parameter α //Assume bipolar units and outputs. Only 1 hidden layer.
while stopping condition is false do
for each bipolar training pair s:t do

- Set activation of input units i = 1 to n { xi = si }
- Set activation of input units i = 1 to n { xi = si }
- Compute net input to hidden units, e.g., zin1 = b1 + x1 w11 + x2 w21
- Determine output of each hidden ADALINE, e.g., z1 = f (z_in1)
- Determine output of net: yin = b3 + z1 v1 + z2 v2; y = f(yin)
- if t=y, then no updates are performed //no errror
- else if t=1, //error, the expected output is 1, the computed output is -1; at least one of the Z's should be 1 then update weights on Z_J, the unit whose net input is closest to 1 (or closest 0, both are the same)
  b_J (new) = b_J (old) + α (1 – z_inJ)
  w_iJ (new) = w_iJ (old) + α (1 – z_inJ) xi
- if t=-1, then update weights on all units Z_k that have posi$ve net input//error

```
    """
    MRI Algorithm
    """
    def backward(self,y_true,x,prediction2,net1) :
        index = 0
        for pred in prediction2 :
            # Don't update the weights if the output was correct.
            if y_true[index] == pred:
                index += 1
                continue
            else :
                if y_true[index] == 1:
                    net1 = self.update_wab_for_positive_1(x,net1)
                else :
                    net1 = self.update_wab_for_negative_1(x,net1)
                index += 1

    def update_wab_for_positive_1(self,x,net1):
        idx, net_i = find_nearest(net1,0)

        old_weights = self.first_layer.weights[idx]
        old_biases = self.first_layer.biases[idx]

        new_weights = old_weights + self.lr*(1+(-1)*net_i[0])*x.reshape(1,2)
        new_biases = old_biases + self.lr*(1+(-1)*net_i[0])

        self.first_layer.weights[idx] = new_weights
        self.first_layer.biases[idx] = new_biases

        new_predictions2,new_net2, new_net1 = self.forward(x)

        return new_net1



    def update_wab_for_negative_1(self,x,net1):
        index = 0
        for n in np.nditer(net1):
            if n > 0 :
                old_weights = self.first_layer.weights[index]
                old_biases = self.first_layer.biases[index]
                new_weights =old_weights + self.lr*(-1+(-1)*n)*x.reshape(1,2)
                new_biases = old_biases + self.lr*(-1+(-1)*n)
                self.first_layer.weights[index] = new_weights
                self.first_layer.biases[index] = new_biases
```

4.6. Plotting the separator lines (Convex-hyper Polygons)

After 50 epochs of training, These are the lines that the model learned:

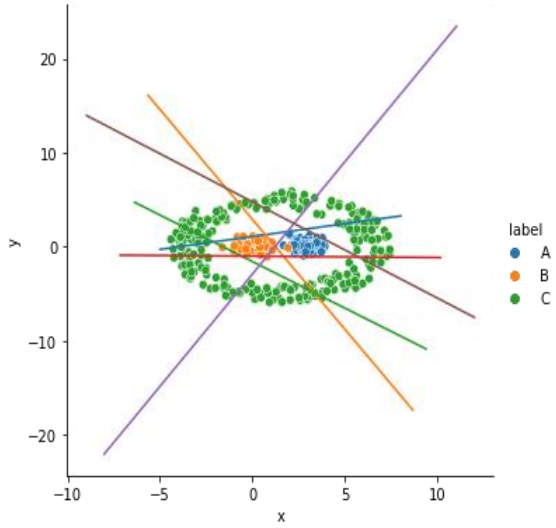


Figure 4.5 Plotting the Separator Lines

4.7. Trying Different Learning Rate

Through experiment, I found $\alpha = 0.000001$ to be the best learning rate.