Andrés Eduardo Marchetti Colognesi - 9838161

Relatório Atividade 3 - Inteligência Computacional

1 Questão 1)

1.1 Item a)

O Grafo representando o fluxo de sinais para a estrutura dada no enunciado está colocado na Figura 1.

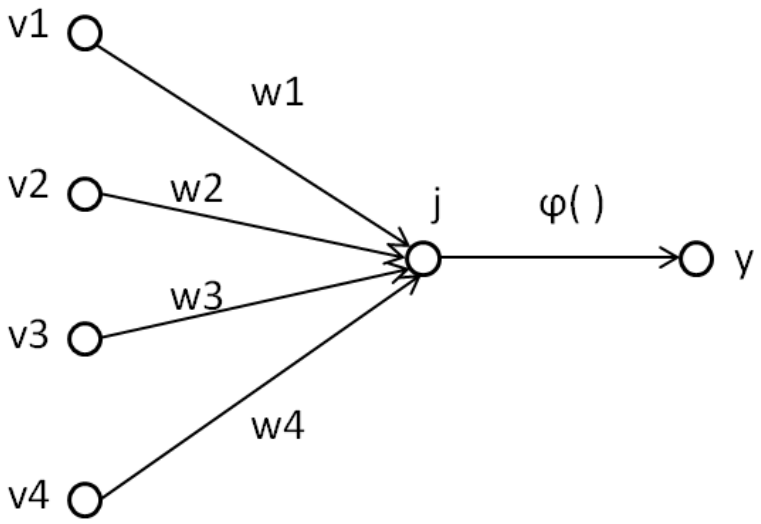

Figura 1: Grafo representando o fluxo de sinais da rede proposta.

1.2 Item b.1)

Considerando a ativação linear para o neurônio j, a saída obtida considerando os valores do enunciado é $y = 1.80$. O código que implementa este neurônio está colocado no Algoritmo 1.

```
# Andres E. M. Colognesi - 9838161

#                    ################################
#                    ## Atividade 3 - Int. Comp. ##
#                    ################################

## Importing libraries:
```

```
import numpy as np
import matplotlib.pyplot as plt


## Exercise 1:

# b.1)
v = np.array([10, -20, 4, -2]) #activation levels
w = np.array([0.8, 0.2, -1.0, -0.9]) #wheights of neuron j
b = 0 #bias

y = np.sum(w*v) + b #linear neuron
print("The output for neuron 'j' with linear activation function
   is: %.2f \n" %(y))
```

Algoritmo 1: Código em Python que implementa o neurônio com ativação linear.

1.3 Item b.2)

Considerando a ativação do neurônio j como no neurônio de *McCulloch-Pitts*, a saída obtida foi $y = 1.00$. O código que implementa este neurônio está colocado no Algoritmo 2. Note que o Algoritmo 2 é continuação do Algoritmo 1.

```
# b.2)
y = (np.sum(w*v) > b).astype(int) #M-P neuron
print("The output for neuron 'j' with M-P activation function is:
   %.2f \n" %(y))
```

Algoritmo 2: Código em Python que implementa o neurônio com ativação semelhanta à do neurônio de McCulloch-Pitts. Note que este trecho de código é continuação do Algoritmo 1.

2 Questão 2)

Aplicando a função de ativação sigmoid ao neurônio, foi obtido um resultado de $y = 0.86$. O código que implementa este neurônio está colocado no Algoritmo 3. Note que este é continuação do Algoritmo 2.

```
## Exercise 2:

g = np.sum(w*v) + b #neuron sum
```

```
y = 1/(1+np.exp(-g)) #sigmoid activation neuron
print("The output for neuron 'j' with sigmoid activation function
   is: %.2f \n" %(y))
```

Algoritmo 3: Código em Python que implementa o neurônio com ativação semelhante à do neurônio de McCulloch-Pitts. Note que este trecho de código é continuação do Algoritmo 2.

3 Questão 3)

Os itens deste exercício foram desenvolvidos com base no uso da função e das variáveis criadas no Algoritmo 4. A classificação é feita de forma semelhante ao neurônio de McCulloch-Pitts: para uma soma de entradas positivas no neurônio, classifica em uma das classes; para negativa, classifica na outra. Foi arbitrado que uma classe seria representada pelo número -1 e a outra pelo número 1. Foi adotada a função de custo MSE para monitorar o aprendizado. Arbitrou-se que a convergência é dada por uma tolerância de 0.001 no erro.

```
## Exercise 3:

# Importing libraries:
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Class points
w1 = np.array([[ 0.1, 1.1],
               [ 6.8, 7.1],
               [-3.5,-4.1],
               [ 2.0, 2.7],
               [ 4.1, 2.8],
               [ 3.1, 5.0],
               [-0.8,-1.3],
               [ 0.9, 1.2],
               [ 5.0, 6.4],
               [ 3.9, 4.0]])

w2 = np.array([[ 7.1, 4.2],
               [-1.4,-4.3],
               [ 4.5, 0.0],
               [ 6.3, 1.6],
               [ 4.2, 1.9],
               [ 1.4,-3.2],
               [ 2.4,-4.0],
```

```
                [ 2.5,-6.1],
                [ 8.4, 3.7],
                [ 4.1,-2.2]])

w3 = np.array([[-3.0,-2.9],
                [ 0.5, 8.7],
                [ 2.9, 2.1],
                [-0.1, 5.2],
                [-4.0, 2.2],
                [-1.3, 3.7],
                [-3.4, 6.2],
                [-4.1, 3.4],
                [-5.3, 1.6],
                [ 1.9, 5.1]])

# Defining functions:

def learn(X, d, alpha, tol=1e-3, limit=100):
    '''
    Implements the Rosenblatt Perceptron algorithm for 2D input,
       with zero
    initialization of wheigts.

    Input:
        X : training data numpy array
        d : label numpy array
        alpha : learning rate
        tol : error tolerance
        limit : maximum number of learning epochs
    Output:
        w : wheights numpy array
        MSE : MSE cost values, evolving with epochs
        data : used training data and labels as a numpy array
    '''
    MSE = [] #vector to store the MSE for each epoch
    w =  np.array([1, 0, 0]) #initialize wheights with 1 for bias
       and zeroes
    X_ext = np.insert(X, 0, 1, 1) #insert columns of 1 for bias
    epoch = 0 #number of epochs
    go_on = True #boolean to stop learning
    while epoch < limit and go_on:
        for i in range(X_ext.shape[0]):
            v = X_ext[i,:] #selects a data point
            y = (np.sum(w*v) > 0).astype(int)*2 - 1 #M-P neuron to
```

```
                classify as 1 or -1

            e = y - d[i] #calculates output error
            w = w - alpha*e*v #updates wheigts

        #Calculating cost for current epoch:
        V = np.sum(w.T*X_ext, axis=1) #overall input
        Y = (V > 0).astype(int)*2 - 1 #overall output
        cost = np.mean( (d-Y)**2 ) #MSE cost

        MSE.append(cost) #add cost of current epoch to MSE vector
        if cost < tol: #the algorithm converged
            go_on = False #end learning loop
        epoch += 1
    data = np.concatenate([X, d.reshape(-1,1)], axis=1)
    return w, MSE, data
```

Algoritmo 4: Código em Python que implementa o treinamento do Perceptron de Rosenblatt com ativação semelhante à do neurônio de McCulloch-Pitts, utilizando os dados da Questão 3.

3.1 Item a)

Para a resolução deste exercício, utilizou-se o código do Algoritmo 5, que toma como base o Algoritmo 4. Note que neste utiliza-se uma taxa de aprendizado $\alpha = 0.2$.

A evolução da função de custo ao longo das *epochs* de treinamento está representada na Figura 2. A partir desta, pode-se extrair facilmente que o algoritmo necessitou de 10 iterações/*epochs* para convergir.

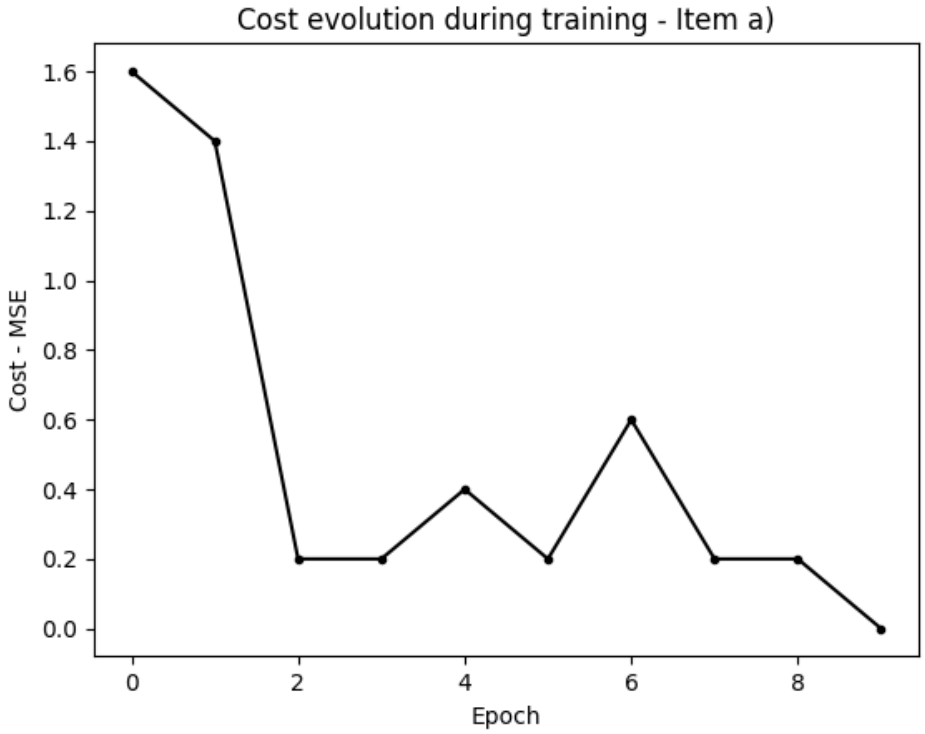

Figura 2: Evolução da função de custo ao longo das *epochs* de treinamento do algoritmo - Item a).

```
# Item a)

X = np.concatenate([w1, w2]) #creating our input vector
ones = np.ones(10)
d = np.concatenate([-1*ones, ones]) #label vector: w1 is class -1
   and w2 is 1
alpha = 0.2 #learning rate

w, MSE, data = learn(X, d, alpha) #training

#Plot cost evolution:
plt.figure(1)
plt.plot(np.arange(len(MSE)), MSE, marker='.', c='black')
plt.xlabel("Epoch")
plt.ylabel("Cost - MSE")
plt.title("Cost evolution during training - Item a)")
plt.show()

#Plot classes:
x1 = [data[:,0].min(), data[:,0].max()]
```

```
x2 = [(-w[0] - w[1]*x1[0])/w[2], (-w[0] - w[1]*x1[1])/w[2]]
plt.figure(2)
plt.scatter(data[:10,0], data[:10,1], c='red', label='w1')
plt.scatter(data[10:,0], data[10:,1], c='blue', label='w2')
plt.plot(x1, x2, color='purple', label='Perceptron',
   linestyle='--', alpha=0.7)
plt.title("Classifications - Item a)")
plt.xlabel("x1")
plt.ylabel("x2")
plt.legend()
plt.show()
```

Algoritmo 5: Código em Python que implementa resolve o item a) da Questão 3. Note que este é continuação do Algoritmo 4.

3.2 Item b)

A resolução deste item é análoga ao item anterior, apenas altera-se os dados de entrada. O código para este item está dado no Algoritmo 6, que toma como base o Algoritmo 4.

A evolução da função de custo ao longo das *epochs* de treinamento está representada na Figura 3. A partir desta, pode-se extrair facilmente que o algoritmo necessitou de 4 iterações/*epochs* para convergir.

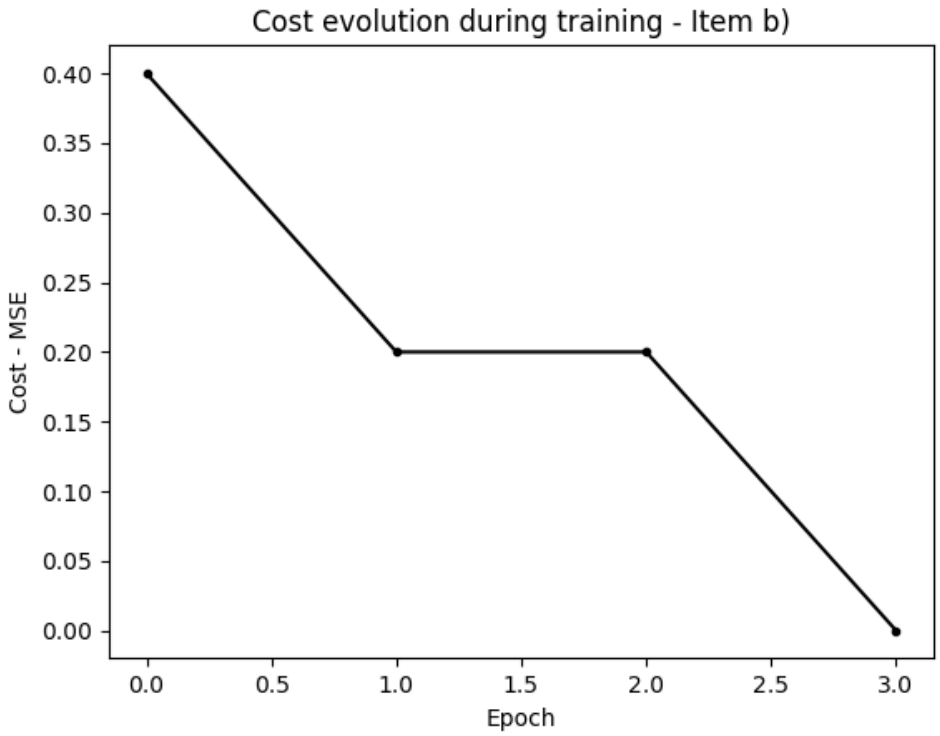

Figura 3: Evolução da função de custo ao longo das *epochs* de treinamento do algoritmo - Item b).

```
# Item b)

X = np.concatenate([w2, w3]) #creating our input vector
ones = np.ones(10)
d = np.concatenate([-1*ones, ones]) #label vector: w2 is class -1
   and w3 is 1
alpha = 0.2 #learning rate

w, MSE, data = learn(X, d, alpha) #training

#Plot cost evolution:
plt.figure(3)
plt.plot(np.arange(len(MSE)), MSE, marker='.', c='black')
plt.xlabel("Epoch")
plt.ylabel("Cost - MSE")
plt.title("Cost evolution during training - Item b)")
plt.show()

#Plot classes:
x1 = [data[:,0].min(), data[:,0].max()]
```

```
x2 = [(-w[0] - w[1]*x1[0])/w[2], (-w[0] - w[1]*x1[1])/w[2]]
plt.figure(2)
plt.scatter(data[:10,0], data[:10,1], c='red', label='w2')
plt.scatter(data[10:,0], data[10:,1], c='blue', label='w3')
plt.plot(x1, x2, color='purple', label='Perceptron',
   linestyle='--', alpha=0.7)
plt.title("Classifications - Item b)")
plt.xlabel("x1")
plt.ylabel("x2")
plt.legend()
plt.show()
```

Algoritmo 6: Código em Python que implementa resolve o item b) da Questão 3. Note que este é continuação do Algoritmo 4.

3.3 Item c)

A diferença de número de iterações para convergir pode ser melhor visualizado ao plotar a função discriminante que é obtida pelo nosso perceptron. Isso é feito pela parte final do código dos Algoritmos 5 e 6. O resultado está colocado nas Figuras 4 e 5.

Para o caso do item a) (Figura 4), os pontos das duas classes estão bastante próximos, obrigando o algoritmo a ajustar a reta neste espaço menor/com mais dificuldade. Já no caso do item b) (Figura 5), os pontos das duas classes estão mais distantes, deixando uma faixa maior para o perceptron ajustar a curva discriminante, sendo assim mais fácil o ajuste. Por isso, no item a) foram necessárias mais iterações para o treinamento do que no item b).

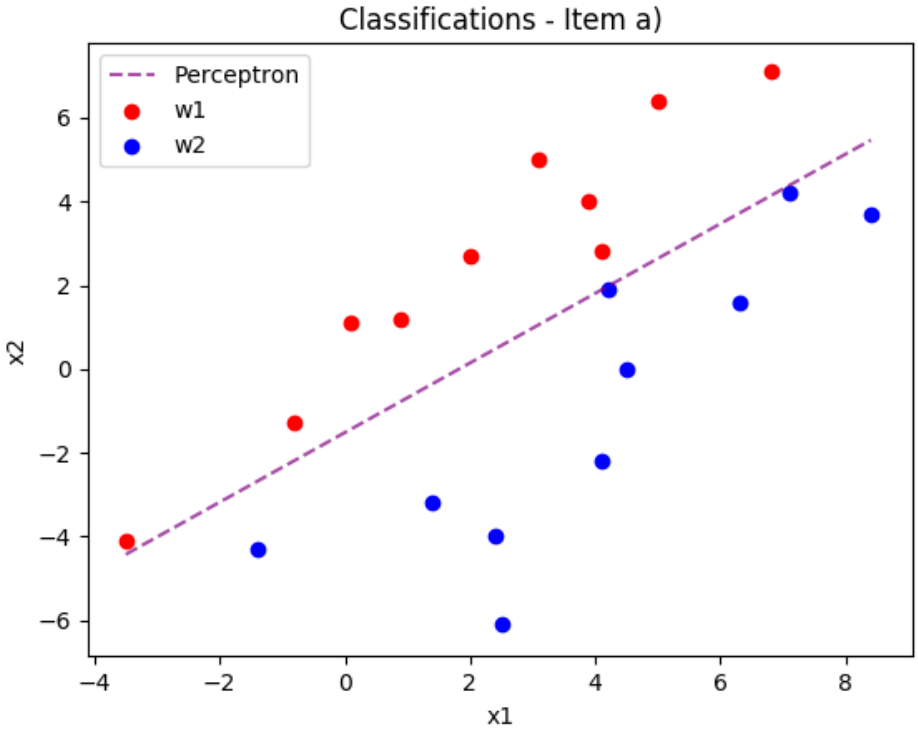

Figura 4: Pontos de treinamento e reta discriminante obtida com o perceptron - Item a).

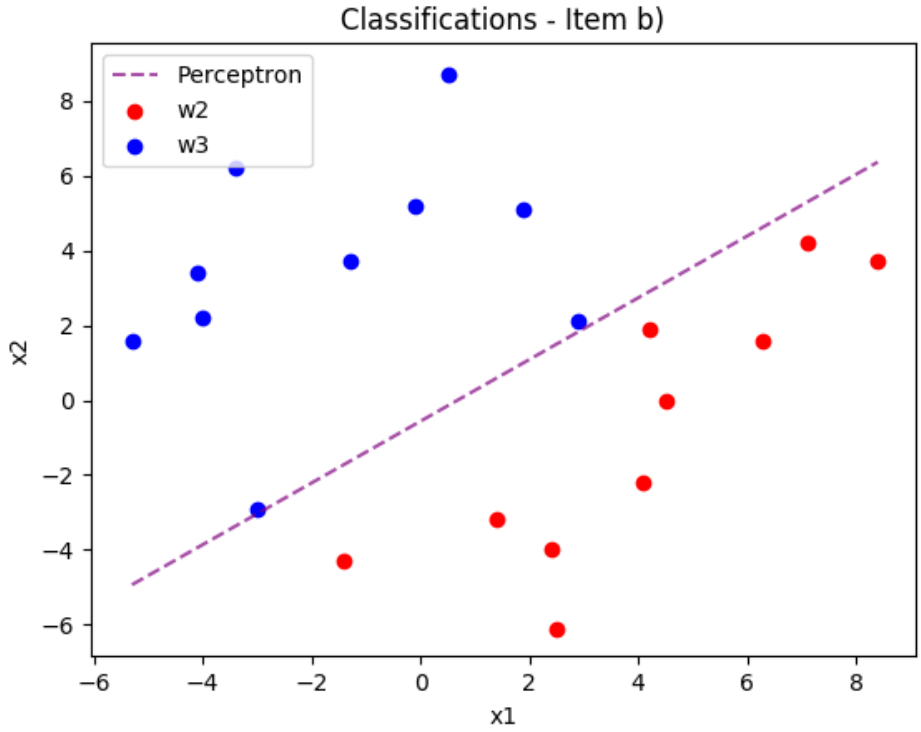

Figura 5: Pontos de treinamento e reta discriminante obtida com o perceptron - Item b).