# P6 - Códigos

## Parte A

Função que separa o plano cartesiano em duas regiões a partir de um discriminante linear.

```
def discriminator(x, y, wx, wy, c, a):

    xx, yy = np.meshgrid(x, y, sparse=True)
    neuron = linearDiscriminator(xx, yy, wx, wy, c, a)
    line = equalZero(x, wx, wy, c)

    plt.plot(x, line, linewidth=3, color="#ffc121", label="equalZero")
    plt.fill_between(x, line, max(y), color="#3a3a3a", label = "+")
    plt.fill_between(x, min(y), line, color="gray", label = "-")
    plt.xlim([min(x), max(x)])
    plt.ylim([min(y), max(y)])
    plt.legend()
    plt.title("Descriminador Linear: " + "Wx = " + str(wx) + " " + "Wy = "
    + str(wy) + " " + "c = " + str(c) + " " + "A = " + str(a))
    plt.show()

def linearDiscriminator(x, y, wx, wy, c, a):

    mat =  a*(x*wx + y*wy + c)

    return mat

def equalZero(x, wx, wy, c):

    y = []

    for i in range(len(x)):
        value = calcZero(x[i], wx, wy, c)
        y.append(value)

    return y
```

Função que gera distrinuições de frutas (uvas e bananas) fictícias seguindo valores típicos de peso e tamanho.

```
def simulateFruit():
    std = 3
    number = 500
    grapes = np.zeros(number)
    bananas = np.zeros(number)

    grapeSize = abs(np.random.normal(3, std, number))
    grapeWeigh = np.random.normal(5.1, std, number)

    bananaSize = abs(np.random.normal(19, std, number))
    bananaWeigh = np.random.normal(113.3, std, number)

    data = np.zeros((2*number, 3))
    ###Grape == 0, Banana ==1
    grapeFlag = np.zeros(number)
    bananaFlag = np.ones(number)

    data[:, 0] = np.hstack((grapeSize, grapeWeigh))#[:,0]
    data[:, 1] = np.hstack((bananaSize, bananaWeigh))
    data[:number, 2] = grapeFlag
    data[number:, 2] = bananaFlag

    frame = pd.DataFrame(data = data, columns=["Size", "Weigh", "Flag"])

    print(frame)

    plt.scatter(grapeSize, grapeWeigh, color='purple', label="Uva")
    plt.scatter(bananaSize, bananaWeigh, color = "orange", label="Banana")
    plt.xlabel("Tamanho")
    plt.ylabel("Peso")
    plt.legend()
    plt.show()
    grape  = [grapeSize, grapeWeigh]
    banana = [bananaSize, bananaWeigh]

    return [grape, banana]
```

Função que gera a separação entre as uvas e bananas produzidas a partir da variação de parâmetros do discriminate linear.

```
def fruitDesc(x, y, wx, wy, c, a, grape, banana):

    xx, yy = np.meshgrid(x, y, sparse=True)
    neuron = linearDiscriminator(xx, yy, wx, wy, c, a)
    line = equalZero(x, wx, wy, c)

    plt.plot(x, line, linewidth=3, color="#ffc121", label="Limite")
    plt.fill_between(x, line, max(y), color="#3a3a3a", label = "Banana")
    plt.fill_between(x, min(y), line, color="gray", label = "Uva")
    plt.xlim([min(x), max(x)])
    plt.ylim([min(y), max(y)])
    plt.scatter(grape[0], grape[1], color='purple', label = "Uva")
    plt.scatter(banana[0], banana[1], color = "yellow",  label = "Banana")
    plt.xlabel("Tamanho(cm)")
    plt.ylabel("Peso(g)")
    plt.legend()
    plt.title("Descriminador Linear: " + "Wx = " + str(wx) + " " + "Wy = " + str(wy) + " " + "c = " + str(c) + " " + "A = " + str(a))
    plt.show()
```

## Parte B

Função para carregamento das imagens e rótulos da base de dados MNIST

```
def loadSet(names):
    images, labels = loadlocal_mnist(
                images_path=names[0],
                labels_path=names[1])

    images = images.reshape(60000, 28, 28)

    imagesTest = images[int(images.shape[0]/2): -1]
    labelsTest = labels[int(labels.shape[0]/2): -1]

    imagesTrain = images[0:int(images.shape[0]/2)]
    labelsTrain = labels[0:int(labels.shape[0]/2)]

    digitsTrain, numberOfEachDigitTrain = np.unique(labelsTrain, return_counts=1)
    digitsTest, numberOfEachDigitTest = np.unique(labelsTest, return_counts=1)

    digitsSortedTrain = np.argsort(labelsTrain)
    digitsSortedTest = np.argsort(labelsTest)

    sortedTrain = imagesTrain[digitsSortedTrain]
    sortedTest = imagesTest[digitsSortedTest]


    return [sortedTrain, labelsTrain[digitsSortedTrain], sortedTest,
   labelsTest[digitsSortedTest],  digitsTrain, numberOfEachDigitTrain,
   digitsTest, numberOfEachDigitTest ]

###LoadDataset and extract info
names = ["train-images.idx3-ubyte", "train-labels.idx1-ubyte"]

sortedTrain, labelsTrain, sortedTest, labelsTest, digitsTrain,
numberOfEachDigitTrain, digitsTest, numberOfEachDigitTest  = loadSet(names)
```

Determinação do vetor de pesos de cada dígito a partir da escolha de uma única imagem.

```
def chooseOne(sortedTrain, numberOfEachDigitTest):

    choosen = []
    last = 0

    for i in range(len(numberOfEachDigitTrain)):
        index = 0

        if(i == 0):
            choosen.append(sortedTrain[i])

        else:
            for j in range(0, i):
                index += numberOfEachDigitTrain[j]
            choosen.append(sortedTrain[index])

        # plt.imshow(choosen[-1])
```

```
        # plt.show()

    return choosen
```

Criação dos vetores de peso a partir de cada uma das imagens escolhidas

```
def createWeights(general):
    weights = []
    for i in range(len(general)):
        weights.append(np.hstack(general[i]))

    return weights
```

Criação dos vetores de estímulos a partir de cada imagem do conjunto de teste de nossa rede.

```
def createStimulus(images):
    stim = []
    for i in range(len(images)):
        stim.append(np.hstack(images[i]))

    return stim
```

Cálculo da projeção (produto escalar) do estímulo em cada um dos vetores de pesos.

```
def projection(weights, stim, labelsTest):

    guess = []

    for i in range(len(stim)):
        proj = []
        for j in range(len(weights)):
            proj.append(np.dot(stim[i], weights[j]))
        #print(proj)
        ansatz = np.argmax(proj)
        guess.append([ansatz, labelsTest[i]])
    #print(guess)
    return guess
```

Criação da matriz de confusão a partir dos valores inferidos pela projeção e dos rótulos corretos de cada imagem.

```
def createConfusionMat(guessed,numberOfEachDigitTest):
    a = np.zeros((10,10))
    for i in range(len(guessed)):

        a[guessed[i][0], guessed[i][1]] += 1
    # a = np.round(a, decimals=1)
    # print(numberOfEachDigitTest[8])
    # for i in range(len(a)):
    #     a[i:] = a[i:]/numberOfEachDigitTest


    # aMax = a.max()
    # aMin = a.min()

    # aNorm = (a - aMin ) / (aMax - aMin)

    df_cm = pd.DataFrame(a, index = [i for i in "0123456789"],columns = [i for i in "0123456789"])
    #plt.figure(figsize = (10,7))
    sns.heatmap(df_cm, annot=True, cmap="gist_gray",  fmt='g', linewidths=.5, cbar=False)
    plt.xlabel("Real")
    plt.ylabel("Inferido")
    plt.tick_params(axis='both', which='major', labelsize=10, labelbottom = False, bottom=False, top = False, labeltop=True)
    plt.title("title")
    plt.show()
    '''
    To normalize:
    zi=xi−min(x)/max(x)−min(x)
    '''
```

Criação de novos vetores de pesos a partir da média de todas as imagens correspondentes a um dígito no conjunto de treinamento

```
def meanSet(digits, numberOfEachDigit ,sortedTrain):
    general = []

    last = 0

    lastIndex = 0

    for i in range(len(digits)):
        mean = np.zeros(sortedTrain[0].shape)
        lastIndex = last
        # print("lastIndex " + str(lastIndex))
        for j in range(lastIndex, lastIndex+numberOfEachDigit[i]):
            mean += sortedTrain[j]
            last = j
        general.append(mean/numberOfEachDigit[i])
        #if(i==0):
        #    erosion_size = 1
        #    element = cv2.getStructuringElement(cv2.MORPH_CROSS, (2 * erosion_size + 1, 2 * erosion_size + 1),
  #   (erosion_size, erosion_size))
        #    general[i] = cv2.erode(general[i], element)
        # plt.imshow(general[-1], cmap="gist_gray")
        # plt.show()
    print("Average calculated with " + str(len(sortedTrain)) + " images")
    return general
```

Erosão da imagem média dos zeros.

```
if(i==0):
  erosion_size = 1
        element = cv2.getStructuringElement(cv2.MORPH_CROSS, (2 * erosion_size + 1, 2 * erosion_size + 1),
    (erosion_size, erosion_size))
        general[i] = cv2.erode(general[i], element)
```

Simulação das curvas da função sigmóide a partir da variação do parâmetro $\beta$.

```
def sigmoid(x, b):
    #a = 1/(1+np.exp(-b*x))

    for i in range(0, 20, 3):
        a = (1/(1+np.exp(-i*x)) - 0.5) #* 1/110
        plt.plot(x, a, label=r'$\beta = $' + str(i))
    #plt.xlim([-5, 5])
    #plt.ylim([-5, 5])
    plt.legend()
    plt.xlabel("In (S)")
    plt.ylabel("Out")
    plt.title("Sigmoid "r'$P = A_{(s)} = \frac{1}{1+\exp^{-s\beta}} - 0.5$')
    plt.grid()
    plt.show()
```

Determinação da ativação de um neurônio a partir dos valores da função sigmóide para cada produto escalar calculado.

```
def sigmoidDiscrim(weights, stim, labelsTest, img):

    guess = []

    for i in range(len(stim)):
        proj = []
        s = []
        out = []

        for j in range(len(weights)):
            s.append(np.dot(stim[i], weights[j])/1000000)

        s = np.array(s)
        sMax = s.max()
        sMin = s.min()

        s = (s-sMin) / (sMax-sMin)

        #print(s)

        sigB = 15
        for k in range(len(s)):
            out.append(sigmoidFunc(s[k], sigB))
```

```
            #print(out[-1])

        # if(labelsTest[i] == 0):
        #     x = np.round(np.arange(-1.1, 1.1, 0.01), 2)
        #     p = sigmoidFunc(x, sigB)
        #     plt.plot(x, p, color='gray',)
        #     plt.title("Sigmoid "r'$P = A_{(s)} = \frac{1}{1+\exp^{-s\beta}} - 0.5$')
        #     plt.scatter(s, out, color="orange", label = "Inferido")
        #     plt.imshow(img[i], extent=[-0.25, (-0.25+.28), 0.7, 1], cmap='gray',)
        #     for l in range(len(out)):
        #         plt.annotate(str(l), (s[l], out[l]))
        #     plt.ylim(-0.25, 1)
        #     plt.xlim(-0.25, 1.1)
        #     plt.legend()
        #     plt.show()
        # print(out)
        ansatz = np.argmax(out)
        guess.append([ansatz, labelsTest[i]])

    #print(guess)

    return guess

def sigmoidFunc(dot, b):

    return (1/(1+np.exp(-b*dot)) - 0.5)
```