Autora: Mirla Borges Costa

O presente trabalho é composto por códigos desenvolvidos em Python no *Anaconda Spyder*. O código visa montar um perceptron para aprender o comportamento de uma porta lógica AND. Essa porta lógica recebe 2 valores de entrada binários e tem uma saída binária correspondente para cada entrada seguindo a Tabela 1.

Tabela 1: Porta AND

A	B	SAIDA
0	0	0
0	1	0
1	0	0
1	1	1

O código inicia com a instalação das bibliotecas numpy, para promover os cálculos matemáticos, matplotlib.pyplot e seaborn para a plotagem dos gráficos de saída.

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

Após a instalação das bibliotecas, foram definidas as bases de dados. Como a Tabela 1 é a referência de base de dados, ela é replicada com *arrays numpy*. Na qual, o objeto *entradas* corresponde às entradas A e B e *saidas* corresponde a coluna SAÍDA.

```
# AND
entradas = np.array([[0,0],[0,1], [1,0], [1,1]])
saidas = np.array([0,0,0,1])
```

Depois, foi feita a inicialização de variáveis do perceptron e criação dos arrays que receberão os dados do modelo. Portanto foi criado o array *d* que corresponde aos valores de saída dados pelo modelo durante seu aprendizado, por conta disso, é um array de mesma dimensão do array *saidas* (1x4). Além dele, os pesos das duas entradas foram inicializados em 0 dentro do array *pesos*, foi definido também a taxa de aprendizagem do perceptron como 0,1 na variável *taxaAprendizagem*, o valor de bias (viés) foi inicializado em 0 em *b*, o número de iterações (épocas) do modelo foi definido como 10 na variável *iteracoes*, e por fim, foram estabelecidos dois arrays para coletar os erros do modelo. O primeiro array é denominado *erros* e coletará os erros do perceptron por iterações, facilitando o cálculo de ajuste de pesos, já o segundo é dado por *erros_iter* e servirá para armazenar os erros gerados a cada rodada para um futuro plot.

```
d = np.zeros([4])
pesos = np.array([0.0, 0.0])
taxaAprendizagem = 0.1
b=0
```

```
iteracoes = 10

erros = np.zeros([4])
erros_iter = np.zeros([iteracoes,4]) #10x4
```

Com as variáveis inicializadas foi dado início a montagem de funções recorrentes no modelo. A primeira função construída foi a função de ativação escolhida para a aplicação, a ***Step Function*** (Função Degrau). A escolha dessa função para ativação se deu pelo fato de que as saídas e entradas do banco de dados são valores binários, que são os mesmos retornos que a função de ativação nos permite receber.

```
# função de ativação: step function
def stepFunction(soma):
    if (soma >= 1):
        return 1
    return 0
```

A segunda função foi construída pensando na plotagem da Decision Boundary (Limitar/Reta de Decisão). Foi escolhida uma plotagem que indicasse de maneira bem visual a reta de separação, também indicando por cores os lados de decisão do modelo. Para que isso fosse possível de ser realizado, foi preciso, primeiro, ser realizada uma coleta de todos os pontos presentes no plano de plotagem e, depois, com os pontos do plano é feito o teste do modelo e coletado os resultados da análise. Portanto, as variáveis *xx* e *yy* são todos os pontos em malha 2D do plano cartesiano de plotagem do modelo, e *Z* é a variável que recebe os valores resultantes de cada ponto do plano, essas 3 variáveis são os retornos da função *gerar_espaço* e que serão úteis para a próxima função.

```
#Função para gerar a decision boundary
def gerar_espaco(pesos,b):
    pixels = 100
    eixo_x = np.arange(-0.5, 1.8, (1.8 + 0.5)/pixels)
    eixo_y = np.arange(-0.5, 1.8, (1.8 + 0.5)/pixels)
    xx, yy = np.meshgrid(eixo_x, eixo_y)
    pontos = np.c_[xx.ravel(), yy.ravel()]

    Z = np.zeros([pontos.shape[0]])
    for i in range(Z.shape[0]):
        U = pontos[i].dot(pesos)
        Z[i] = stepFunction(U + b)
    Z = Z.reshape(xx.shape)
    return xx,yy,Z
```

Com os resultados que podem ser gerados na função *gerar_espaco* foi criada a função *sub_plots*, essa função será responsável por catalogar e agrupar todos os plots da decision boundary por iteração. Na função *sub_plots*, os valores *xx*, *yy* e *Z* são aplicados ao método

contourf da biblioteca *matplotlib* que traça contornos preenchidos, segundo diz a documentação. Além da reta, também é plotado o resultado da saída do modelo em pontos através do método *scatterplot* da biblioteca *seaborn*.

```
# função de subplots
fig = plt.figure(figsize=(15, 5))

def sub_plots(d,pesos,n,b):
    xx, yy, Z = gerar_espaco(pesos,b)
    plt.contourf(xx, yy, Z, alpha=0.3)
    sns.scatterplot(x=[0,0,1,1], y=[0,1,0,1], hue=d, s=90)
    plt.xlim(-0.15,1.2)
    plt.ylim(-0.15,1.2)
    plt.title(str(n+1) + '° Iteração ')
```

Após a definição das funções recorrentes do modelo, foi dado início ao treinamento do perceptron. O código de treinamento inicia-se com um laço de repetição que roda todas as iterações, depois é feito outro laço de repetição que se repete por todas as quantidades de entradas. No segundo laço é feita a leitura de todas as entradas, para os cálculos de aprendizagem do perceptron. O primeiro cálculo foi dado pela multiplicação entre matrizes e a soma do bias, atribuindo o resultado a *U*, de acordo com a equação 1:

$$u_k = \sum_{j=0}^{m} (w_{kj} \cdot x_j) \tag{1}$$

Depois, é feita a aplicação de *U* à função de ativação (*stepFunction*) e a saída é atribuída ao array *d* conforme a equação 2.

$$y_k = \varphi(u_k + b) \tag{2}$$

Logo após é feito o cálculo do erro a partir da saida real dos dados e com isso é feito o ajuste de pesos e bias segundo a equação 3:

$$w_{i(t+1)} = w_{i(t)} + taxaAprendizagem \cdot erro_i \cdot x_i \tag{3}$$

Após cada iteração é feita a coleta dos erros obtidos e do gráfico da decision boundary. Ao final das iterações o plot com das retas de decisão de cada iteração é gerado.

```
# treinamento
for it in range(iteracoes):
    for i in range(entradas.shape[0]):
        U = entradas[i].dot(pesos)
        d[i] = stepFunction(U + b)
        erros[i] = saidas[i] - d[i]
        for j in range(pesos.shape[0]):
            pesos[j] = pesos[j] + (taxaAprendizagem * entradas[i][j]
* erros[i])
        b += taxaAprendizagem*erros[i]
    fig.add_subplot(2,5,it+1)
    sub_plots(d,pesos,it,b)
```

```
    plt.tight_layout()
    ##salvar os erros por iteração
    erros_iter[it] = erros
plt.show()
```

Por fim, é feita a plotagem dos erros do modelo por iterações com os dados do array *erros_iter*.

```
# Plot de erros por iteração

fig = plt.figure(figsize=(15, 3))
for i in range(iteracoes):
    fig.add_subplot(2,5,i+1)
    plt.plot(['[0,0]','[0,1]','[1,0]','[1,1]'],erros_iter[i])
    plt.title(str(i+1) + '° Iteração ')
    plt.tight_layout()
plt.show()
```

Como resultados foram obtidos os gráficos da reta de decisão com os valores de saída dados pelo modelo, o qual podem ser vistos na Figura 1, e os gráficos do erro de cada iteração na Figura 2.

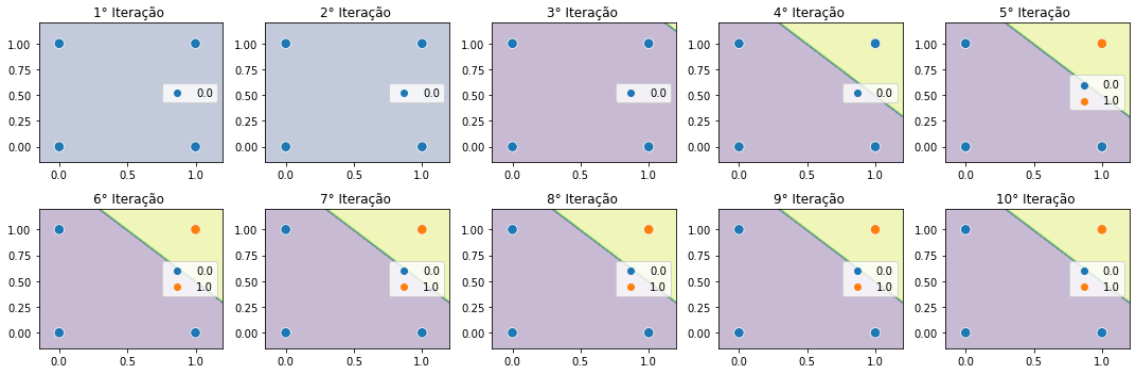

Figura 1: Plotagens da reta de decisão e dos valores de saída do perceptron a cada iteração com entrada referente a porta lógica AND.

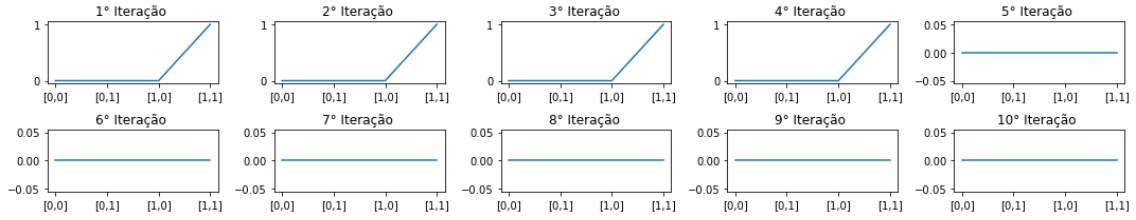

Figura 2: Plotagens dos erros encontrados a cada iteração durante o aprendizado do perceptron com entrada referente a porta lógica AND.

É perceptível a movimentação da *decision boundary* a cada iteração, mostrando que o modelo está conseguindo fazer os ajustes dos seus pesos e vieses a cada iteração. Além disso, observando os plots de erros (Figura 2) o modelo mostrou-se bastante rápido na aprendizagem de modo que na 5° iteração já atingiu o resultado ideal de aprendizado, ou seja com 0 erros.

Por último, foi feito o teste de aprendizado também com entradas referentes a porta lógica OR visando apenas verificar a diferença de desempenho de aprendizado entre ambas as portas no modelo construído.

Tabela 2: Porta OR

A	B	SAIDA
0	0	0
0	1	1
1	0	1
1	1	1

Assim, foram alteradas apenas as entradas e saídas do modelo para seguir a Tabela 2:

```
# OR
entradas = np.array([[0,0],[0,1], [1,0], [1,1]])
saidas = np.array([0,1,1,1])
```

Com os dados de entradas da porta lógica OR, o perceptron obteve 100% de acertos na 4° iteração. O resultado obtido para essa aplicação também se mostrou bastante interessante conforme mostram as Figuras 3 e 4:

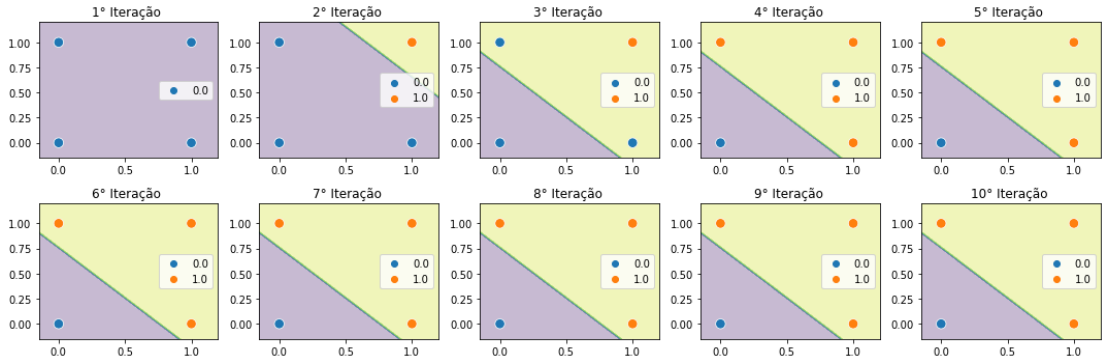

Figura 3: Plotagens da reta de decisão e dos valores de saída do perceptron a cada iteração com entrada referente a porta lógica OR.

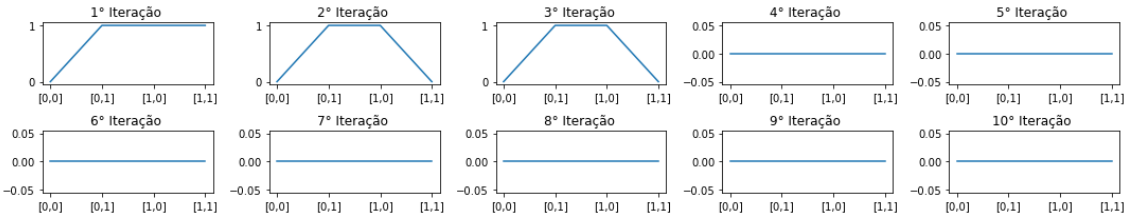

Figura 3: Plotagens dos erros encontrados a cada iteração durante o aprendizado do perceptron com entrada referente a porta lógica OR.

Código final implementado:

```
import numpy as np
```

```
import matplotlib.pyplot as plt
import seaborn as sns

# AND
entradas = np.array([[0,0],[0,1], [1,0], [1,1]])
saidas = np.array([0,0,0,1])
# OR
#entradas = np.array([[0,0],[0,1], [1,0], [1,1]])
#saidas = np.array([0,1,1,1])

d = np.zeros([4])
pesos = np.array([0.0, 0.0])
taxaAprendizagem = 0.1
b=0

iteracoes = 10

erros = np.zeros([4])
erros_iter = np.zeros([iteracoes,4]) #10x4

# função de ativação: step function
def stepFunction(soma):
    if (soma >= 1):
        return 1
    return 0

#Função para gerar a decision boundary
def gerar_espaco(pesos,b):
    pixels = 100
    eixo_x = np.arange(-0.5, 1.8, (1.8 + 0.5)/pixels)
    eixo_y = np.arange(-0.5, 1.8, (1.8 + 0.5)/pixels)
    xx, yy = np.meshgrid(eixo_x, eixo_y)
    pontos = np.c_[xx.ravel(), yy.ravel()]

    Z = np.zeros([pontos.shape[0]])
    for i in range(Z.shape[0]):
        U = entradas[i].dot(pesos)
        Z[i] = stepFunction(U + b)
    Z = Z.reshape(xx.shape)
    return xx,yy,Z

# função de subplots
fig = plt.figure(figsize=(15, 5))

def sub_plots(d,pesos,n,b):
    xx, yy, Z = gerar_espaco(pesos,b)
    plt.contourf(xx, yy, Z, alpha=0.3)
    sns.scatterplot(x=[0,0,1,1], y=[0,1,0,1], hue=d, s=90)
    plt.xlim(-0.15,1.2)
    plt.ylim(-0.15,1.2)
```

```
    plt.title(str(n+1) + '° Iteração ')

# treinamento
for it in range(iteracoes):
    for i in range(entradas.shape[0]):
        U = entradas[i].dot(pesos)
        d[i] = stepFunction(U + b)
        erros[i] = saidas[i] - d[i]
        for j in range(pesos.shape[0]):
            pesos[j] = pesos[j] + (taxaAprendizagem * entradas[i][j]
* erros[i])
        b += taxaAprendizagem*erros[i]
    fig.add_subplot(2,5,it+1)
    sub_plots(d,pesos,it,b)
    plt.tight_layout()
    ##salvar os erros por iteração
    erros_iter[it] = erros
plt.show()

# Plot de erros por iteração

fig = plt.figure(figsize=(15, 3))
for i in range(iteracoes):
    fig.add_subplot(2,5,i+1)
    plt.plot(['[0,0]','[0,1]','[1,0]','[1,1]'],erros_iter[i])
    plt.title(str(i+1) + '° Iteração ')
    plt.tight_layout()
plt.show()
```