Trabalho 2 – Aprendizado de Máquina I

Componentes do grupo

Nome: Dyanna Cruz dos Santos matrícula 51352121015
Nome: Guilherme Rodrigo Camblor matrícula 51352121028
Nome: Renato Gomes de Campos matrícula 51352121021

Instruções

- data de referência para entrega: 28/11.
- em grupos de até 4 alunos.
- considerar no trabalho o *print do* código fonte e o *print* das simulações;
- nomear o arquivo como "**Trabalho 2 de Aprendizado de Máquina I + 1º nome de um dos componentes do grupo**";
- entregar o trabalho em arquivo PDF pelo *email* do professor: mauricio.mario@fatec.sp.gov.br assunto = **trabalho 2 de Aprendizado de Máquina I**.

Envio 2 com formatação ajustada, bem como a resolução das questões que estavam fora do padrão de resposta.

Exercício 1:

Experimento: A partir de neurônio do tipo *MCP*, construir aplicações para implementar as funções lógicas *booleanas AND, OR, NAND, NOR*: as tabelas com estas funções estão no arquivo "**Princípios de Redes Neurais Artificiais e funções de ativação- II**".

Observação: para obtenção das funções só podem ser alterados os parâmetros:
- Os valores dos pesos w;
- O valor do *limiar* θ;

- **Não** pode ser alterado o critério da função degrau para definir o valor da saída do neurônio:
($v_k \geq \theta \rightarrow$ saída da função degrau = 1; $v_k < \theta \rightarrow$ saída da função degrau = 0);
- **Não** podem ser alteradas as combinações com os valores das entradas x_i.

Ajustar a saída no console para cada função booleana:

x_1 **AND** x_2 =

x_1 **OR** x_2 =

x_1 **NAND** x_2 =

x_1 **NOR** x_2 =

Figura 01: Código de Implementação da função AND, OR, NOR e NAND.

```python
#Exercicio 1 Trabalho 2
def step_function(x):
    return 1 if x >= 1.5 else 0  # não pode mexer no critério

def perceptron_output(weights, bias, x):
    calculation = dot(weights, x) + bias
    return step_function(calculation)

def print_results(label, outputs):
    print(label)
    for i, output in enumerate(outputs):
        print(f"{x_labels[i]} = {output}")

# Não pode mexer nas entradas
x0 = [0, 0]
x1 = [0, 1]
x2 = [1, 0]
x3 = [1, 1]

weights_and = [1, 1]
bias_and = 0  # ajuste para a função AND

weights_or = [1, 1]
bias_or = 1  # ajuste para a função OR

weights_not_or = [-1, -1]  # pesos invertidos para a função NOT OR (NOR)
bias_not_or = 1.5  # ajuste para a função NOT OR (NOR)

weights_nand = [-1, -1]  # pesos invertidos para a função NAND
bias_nand = 2.5  # ajuste para a função NAND

saida0_and = perceptron_output(weights_and, bias_and, x0)
saida1_and = perceptron_output(weights_and, bias_and, x1)
saida2_and = perceptron_output(weights_and, bias_and, x2)
saida3_and = perceptron_output(weights_and, bias_and, x3)

saida0_or = perceptron_output(weights_or, bias_or, x0)
saida1_or = perceptron_output(weights_or, bias_or, x1)
saida2_or = perceptron_output(weights_or, bias_or, x2)
saida3_or = perceptron_output(weights_or, bias_or, x3)
```

Fonte: Autores, 2023.

Figura 02: Continuação do Código de Implementação da função AND, OR, NOR e NAND.

```python
saida0_not_or = perceptron_output(weights_not_or, bias_not_or, x0)
saida1_not_or = perceptron_output(weights_not_or, bias_not_or, x1)
saida2_not_or = perceptron_output(weights_not_or, bias_not_or, x2)
saida3_not_or = perceptron_output(weights_not_or, bias_not_or, x3)

saida0_nand = perceptron_output(weights_nand, bias_nand, x0)
saida1_nand = perceptron_output(weights_nand, bias_nand, x1)
saida2_nand = perceptron_output(weights_nand, bias_nand, x2)
saida3_nand = perceptron_output(weights_nand, bias_nand, x3)

x_labels = ["0 AND 0", "0 AND 1", "1 AND 0", "1 AND 1"]
print_results("PERCEPTRON IMPLEMENTANDO FUNÇÃO BOOLEANA AND", [saida0_and, saida1_and, saida2_and, saida3_and])

x_labels = ["0 OR 0", "0 OR 1", "1 OR 0", "1 OR 1"]
print_results("PERCEPTRON IMPLEMENTANDO FUNÇÃO BOOLEANA OR", [saida0_or, saida1_or, saida2_or, saida3_or])

x_labels = ["NOT OR 0", "NOT OR 1", "NOT OR 1", "NOT OR 1"]
print_results("PERCEPTRON IMPLEMENTANDO FUNÇÃO BOOLEANA NOT OR (NOR)", [saida0_not_or, saida1_not_or, saida2_not_or, saida3_not_or])

x_labels = ["0 NAND 0", "0 NAND 1", "1 NAND 0", "1 NAND 1"]
print_results("PERCEPTRON IMPLEMENTANDO FUNÇÃO BOOLEANA NAND", [saida0_nand, saida1_nand, saida2_nand, saida3_nand])
```

Fonte: Autores, 2023.

No trabalho apresentado, nas Figuras 01 e 02, são exibidos o código referente à implementação das funções lógicas AND, OR, NOR e NAND.

Figura 03: Resultado da Função AND.

Fonte: Autores, 2023.

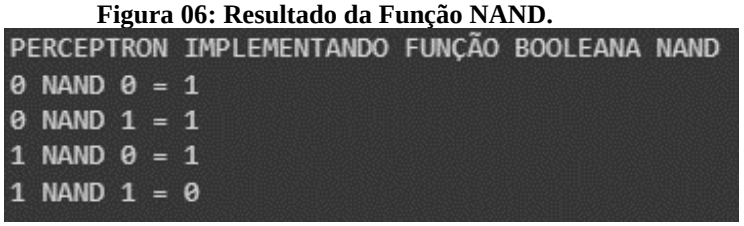
Figura 04: Resultado da Função OR.

Fonte: Autores, 2023.

Figura 05: Resultado da Função NOR.

Fonte: Autores, 2023.

Figura 06: Resultado da Função NAND.



Fonte: Autores, 2023.

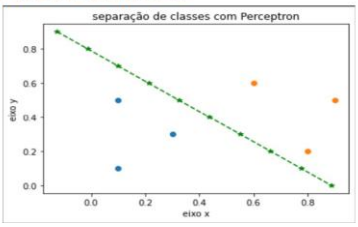
Nas figuras de 03 a 06 estão apresentam os resultados obtidos a partir da execução do código das funções mencionadas. Essa representação visual é essencial para validar a correta implementação da lógica de AND, OR, NOR e NAND respectivamente, fornecendo uma perspectiva prática do desempenho das funções para o algoritmo do *Perceptron*.

Exercício 2:

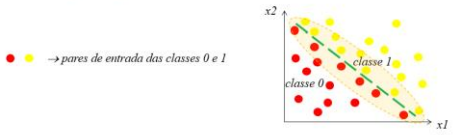
Fazer o experimento com neurônio *Perceptron* com os mesmos parâmetros utilizados (pares de entrada (*x1, x2*) treinados e quantidade de ciclos de treinamento), e os pares para os testes de generalização, verificando os resultados.

```
padrao_0_0 = [-1, 0.1, 0.1]
padrao_0_1 = [-1, 0.1, 0.5]
padrao_0_2 = [-1, 0.3, 0.3]
padrao_1_0 = [-1, 0.6, 0.6]
padrao_1_1 = [-1, 0.8, 0.2]
padrao_1_2 = [-1, 0.9, 0.5]
```

```
#teste de generalização
padrao_teste_0 = [-1, 0.2, 0.4]
padrao_teste_1 = [-1, 0.7, 0.8]
padrao_teste_2 = [-1, 0.6, 0.3]
padrao_teste_3 = [-1, 0.1, 0.9]
padrao_teste_4 = [-1, 0.2, 0.6]
padrao_teste_5 = [-1, 0.8, 0.1]
```



Inserir mais pares de treinamento, principalmente na região d fronteira que separa as classes:



Repetir o teste de generalização inserindo mais pontos, agora co mais pares já treinados, verificando os resultados. Justificar.

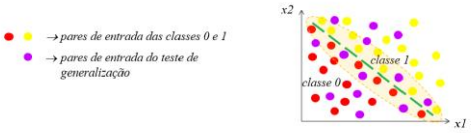


Figura 07: Implementação A.

```python
#Código Base
from linear_algebra import dot

def degrau(x):
  return 1 if x >= 0 else 0

def saida_perceptron(pesos, entradas):
  y = dot(pesos, entradas)
  return degrau(y)

def ajustes(sinapses, entradas, saida):

  taxa_aprendizagem = 0.08

  saida_parcial = saida_perceptron(sinapses, entradas)

  for j in range(3):
    sinapses[j] = sinapses[j] + taxa_aprendizagem * (saida[0] - saida_parcial) * entradas[j]

  saida = saida_parcial
  return sinapses, saida

def teste_generalizacao(sinapses, entradas, saidas):
  saida_parcial = saida_perceptron(sinapses, entradas)
  saida = saida_parcial
  return sinapses, saida

neuronio = [0.22, -0.33, 0.44]
entrada1 = [-1, 0.1, 0.1]
entrada2 = [-1, 0.1, 0.5]
entrada3 = [-1, 0.3, 0.3]
entrada4 = [-1, 0.6, 0.6]
entrada5 = [-1, 0.8, 0.2]
entrada6 = [-1, 0.9, 0.5]

saida1 = [0]
saida2 = [1]
```

Fonte: Autores, 2023.

Figura 08: Continuação da Implementação A.

```
n = 0

for _ in range(11):
    neuronio, saida_1 = ajustes(neuronio, entrada1, saida1)
    print([round(w, 2) for w in neuronio], "saida1 = ", saida1)
    neuronio, saida_2 = ajustes(neuronio, entrada2, saida1)
    print([round(w, 2) for w in neuronio], "saida1 = ", saida1)
    neuronio, saida_1 = ajustes(neuronio, entrada3, saida1)
    print([round(w, 2) for w in neuronio], "saida1 = ", saida1)
    neuronio, saida_2 = ajustes(neuronio, entrada4, saida2)
    print([round(w, 2) for w in neuronio], "saida2 = ", saida2)
    neuronio, saida_2 = ajustes(neuronio, entrada5, saida2)
    print([round(w, 2) for w in neuronio], "saida2 = ", saida2)
    neuronio, saida_2 = ajustes(neuronio, entrada6, saida2)
    print([round(w, 2) for w in neuronio], "saida2 = ", saida2)
    n = n + 1;
    print("Número de ciclos = ", n)

x = [0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1, 0.0]
y = [-0.125, -0.0125, 0.1, 0.2125, 0.325, 0.4375, 0.55, 0.6625, 0.775, 0.8875]

x1 = [0.1, 0.1, 0.3]
y1 = [0.1, 0.5, 0.3]
x2 = [0.6, 0.8, 0.9]
y2 = [0.6, 0.2, 0.5]

plt.scatter(x1, y1)
plt.scatter(x2, y2)
plt.plot(y, x, color = 'green', marker = '*', linestyle = '--')

plt.title("Separação de classes com Percepton")

plt.xlabel("Eixo ( X )")
plt.ylabel("Eixo ( Y )")
plt.show()

#Teste de generalização
teste_0 = [-1, 0.2, 0.4]
teste_1 = [-1, 0.7, 0.8]
teste_2 = [-1, 0.6, 0.3]
teste_3 = [-1, 0.1, 0.9]
teste_4 = [-1, 0.2, 0.6]
```

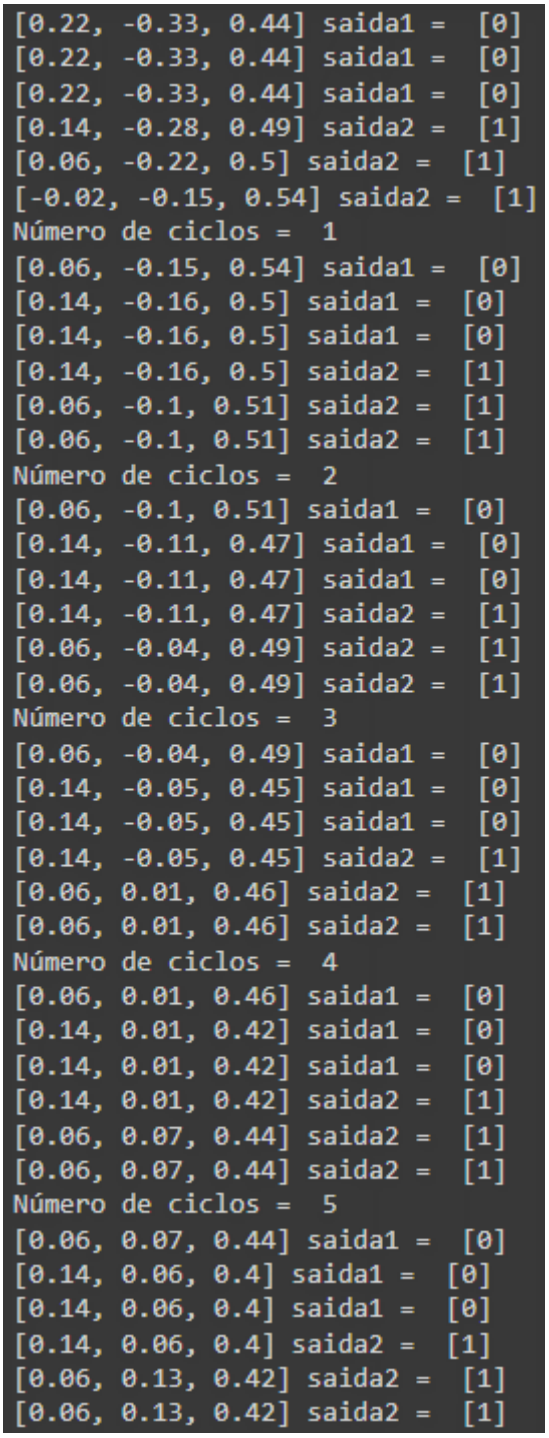
Fonte: Autores, 2023.

Figura 09: Continuação da Implementação A.

```
teste_5 = [-1, 0.8, 0.1]

print('Teste de generalização')
neuronio, saida_0 = teste_generalizacao(neuronio, teste_0, saida0)
print([round(w, 2) for w in neuronio], 'saida0=', saida_0)
neuronio, saida_1 = teste_generalizacao(neuronio, teste_1, saida1)
print([round(w, 2) for w in neuronio], 'saida1=', saida_1)
neuronio, saida_0 = teste_generalizacao(neuronio, teste_2, saida0)
print([round(w, 2) for w in neuronio], 'saida0=', saida_0)
neuronio, saida_1 = teste_generalizacao(neuronio, teste_3, saida1)
print([round(w, 2) for w in neuronio], 'saida1=', saida_1)
neuronio, saida_0 = teste_generalizacao(neuronio, teste_4, saida0)
print([round(w, 2) for w in neuronio], 'saida0=', saida_0)
neuronio, saida_1 = teste_generalizacao(neuronio, teste_5, saida1)
print([round(w, 2) for w in neuronio], 'saida1=', saida_1)
```

Fonte: Autores, 2023.

Figura 10: Resultados de A.



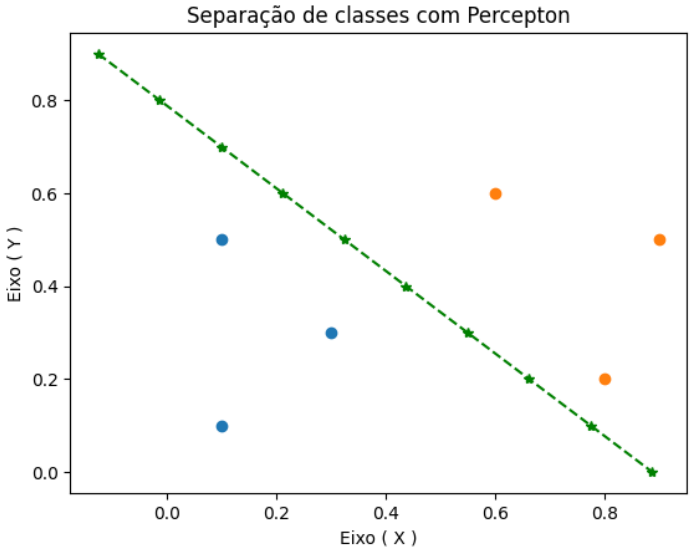
Fonte: Autores, 2023.

Figura 11: Continuação dos Resultados de A.

```
Número de ciclos =  6
[0.06, 0.13, 0.42] saida1 =  [0]
[0.14, 0.12, 0.38] saida1 =  [0]
[0.22, 0.09, 0.35] saida1 =  [0]
[0.22, 0.09, 0.35] saida2 =  [1]
[0.14, 0.16, 0.37] saida2 =  [1]
[0.14, 0.16, 0.37] saida2 =  [1]
Número de ciclos =  7
[0.14, 0.16, 0.37] saida1 =  [0]
[0.22, 0.15, 0.33] saida1 =  [0]
[0.22, 0.15, 0.33] saida1 =  [0]
[0.22, 0.15, 0.33] saida2 =  [1]
[0.14, 0.21, 0.34] saida2 =  [1]
[0.14, 0.21, 0.34] saida2 =  [1]
Número de ciclos =  8
[0.14, 0.21, 0.34] saida1 =  [0]
[0.22, 0.21, 0.3] saida1 =  [0]
[0.22, 0.21, 0.3] saida1 =  [0]
[0.22, 0.21, 0.3] saida2 =  [1]
[0.22, 0.21, 0.3] saida2 =  [1]
[0.22, 0.21, 0.3] saida2 =  [1]
Número de ciclos =  9
[0.22, 0.21, 0.3] saida1 =  [0]
[0.22, 0.21, 0.3] saida1 =  [0]
[0.22, 0.21, 0.3] saida1 =  [0]
[0.22, 0.21, 0.3] saida2 =  [1]
[0.22, 0.21, 0.3] saida2 =  [1]
[0.22, 0.21, 0.3] saida2 =  [1]
Número de ciclos =  10
[0.22, 0.21, 0.3] saida1 =  [0]
[0.22, 0.21, 0.3] saida1 =  [0]
[0.22, 0.21, 0.3] saida1 =  [0]
[0.22, 0.21, 0.3] saida2 =  [1]
[0.22, 0.21, 0.3] saida2 =  [1]
[0.22, 0.21, 0.3] saida2 =  [1]
Número de ciclos =  11
```

Fonte: Autores, 2023.

Figura 12: Resultados de A – Sepração de classes.



Fonte: Autores, 2023.

Figura 13: Resultados de A – Teste de Generalização.

```
Teste de generalização
[0.22, 0.21, 0.3] saida0= 0
[0.22, 0.21, 0.3] saida1= 1
[0.22, 0.21, 0.3] saida0= 0
[0.22, 0.21, 0.3] saida1= 1
[0.22, 0.21, 0.3] saida0= 1
[0.22, 0.21, 0.3] saida1= 0
```

Fonte: Autores, 2023.

No trabalho apresentado, nas Figuras 07 e 13, são exibidos o código e resultados referentes à implementação do experimento com neurônio *Perceptron*, bem como as saídas mostrando o teste de generalização e separação das classes.

A partir da figura 14, começa a inserção de mais pares de treinamento, inclusive a repetição do teste de generalização inserindo mais pontos.

Figura 14: Código de Implementação da Função para itens B e C.

```
import matplotlib
import matplotlib.pyplot as plt
from linear_algebra import dot
from __future__ import division
from collections import Counter

def degrau(x):
  return 1 if x >= 0 else 0

def saida_perceptron(pesos, entradas):
  y = dot(pesos, entradas)
  return degrau(y)

def ajustes(sinapses, entradas, saida):

  taxa_aprendizagem = 0.1

  saida_parcial = saida_perceptron(sinapses, entradas)

  for j in range(3):
    sinapses[j] = sinapses[j] + taxa_aprendizagem * (saida[0] - saida_parcial) * entradas[j]

  saida = saida_parcial
  return sinapses, saida

def teste_generalizacao(sinapses, entradas, saida):
  saida_parcial = saida_perceptron(sinapses, entradas)
  saida = saida_parcial
  return sinapses, saida

neuronio = [0.22, -0.33, 0.44]
padrao_0_0 = [-1, 0.1, 0.1]
padrao_0_1 = [-1, 0.1, 0.5]
padrao_0_2 = [-1, 0.3, 0.3]
padrao_1_0 = [-1, 0.6, 0.6]
padrao_1_1 = [-1, 0.8, 0.2]
padrao_1_2 = [-1, 0.9, 0.5]
padrao_2_0 = [-1, 0.5, 0.5]
padrao_2_1 = [-1, 0.6, 0.19]
padrao_2_2 = [-1, 0.8, 0.08]
```

Fonte: Autores, 2023.

Figura 15: Continuação do Código de Implementação da Função para itens B e C.

```
saida0 = [1]
saida1 = [0]

n = 0;
for _ in range(22):
  neuronio, saida_0 = ajustes(neuronio, padrao_0_0, saida0)
  print([round(w, 2) for w in neuronio], "saida0 = ", saida_0)
  neuronio, saida_0 = ajustes(neuronio, padrao_0_1, saida0)
  print([round(w, 2) for w in neuronio], "saida0 = ", saida_0)
  neuronio, saida_0 = ajustes(neuronio, padrao_0_2, saida0)
  print([round(w, 2) for w in neuronio], "saida0 = ", saida_0)
  neuronio, saida_1 = ajustes(neuronio, padrao_1_0, saida1)
  print([round(w, 2) for w in neuronio], "saida1 = ", saida_1)
  neuronio, saida_1 = ajustes(neuronio, padrao_1_1, saida1)
  print([round(w, 2) for w in neuronio], "saida1 = ", saida_1)
  neuronio, saida_1 = ajustes(neuronio, padrao_1_2, saida1)
  print([round(w, 2) for w in neuronio], "saida1 = ", saida_1)
  neuronio, saida_0 = ajustes(neuronio, padrao_2_0, saida0)
  print([round(w, 2) for w in neuronio], "saida0 = ", saida_0)
  neuronio, saida_0 = ajustes(neuronio, padrao_2_1, saida0)
  print([round(w, 2) for w in neuronio], "saida0 = ", saida_0)
  neuronio, saida_0 = ajustes(neuronio, padrao_2_2, saida0)
  print([round(w, 2) for w in neuronio], "saida0 = ", saida_0)

x = [0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1, 0.0]
y = [-0.125, -0.0125, 0.1, 0.2125, 0.325, 0.4375, 0.55, 0.6625, 0.775, 0.8875]

x1 = [0.1, 0.1, 0.3]
x2 = [0.6, 0.8, 0.9]
y1 = [0.1, 0.5, 0.3]
y2 = [0.6, 0.2, 0.5]
x3 = [0, 0.2, 0.19]
y3 = [0.76, 0.58, 0.82]
x4 = [0, 0.52, 0.475]
y4 = [0.83, 0.3, 0.4]
x5 = [-0.1, 0.8, 0.7]
y5 = [0.84, 0.1, 0.15]
x6 = [0.6, 0.4, 0.1]
y6 = [0.2, 0.37, 0.63]
x7 = [ 0.02, 0.5, 0.3]
y7 = [ 0.71, 0.5, 0.5]
```

Fonte: Autores, 2023.

Figura 16: Continuação do Código de Implementação da Função para itens B e C.

```
x8 = [ 0.8, 0.6, 0.58]
y8 = [ 0.8, 0.24, 0.38]
x9 = [ 0.82, 0.79, 0.4]
y9 = [ 0.02, 0.15, 0.43]
x10 = [ 0.3, 0.4, 0.4]
y10 = [ 0.3, 0.4, 0.74]
x11 = [ 0, 0.1, 0.5]
y11 = [ 0, 0.2, 0.7]
x12 = [ -0.1, 0.3, 0.6]
y12 = [ 0.7, 0.8, 0.7]
x13 = [ 0.2, 0, 0.4]
y13 = [ 0.77, 0.4, 0.2]

plt.scatter(x1, y1)
plt.scatter(x2, y2)
plt.scatter(x3, y3)
plt.scatter(x4, y4)
plt.scatter(x5, y5)
plt.scatter(x6, y6)
plt.scatter(x7, y7)
plt.scatter(x8, y8)
plt.scatter(x9, y9)
plt.scatter(x10, y10)
plt.scatter(x12, y11)
plt.scatter(x11, y12)
plt.plot(y, x, color = '#FF715B', marker = '*', linestyle = '--')

plt.title('separação de classes com Percepton')

plt.xlabel('eixo x')
plt.ylabel('eixo y')
plt.show()

padrao_teste_0 = [-1, 0.2, 0.4]
padrao_teste_1 = [-1, 0.7, 0.8]
padrao_teste_2 = [-1, 0.6, 0.3]
padrao_teste_3 = [-1, 0.1, 0.9]
padrao_teste_4 = [-1, 0.2, 0.6]
padrao_teste_5 = [-1, 0.8, 0.1]

padrao_teste_6 = [0.1, 0.1, 0.3]
padrao_teste_7 = [0.6, 0.8, 0.9]
```

Fonte: Autores, 2023.

Figura 17: Continuação do Código de Implementação da Função para itens B e C.

```
padrao_teste_8 = [0.1, 0.5, 0.3]
padrao_teste_9 = [0.6, 0.2, 0.5]
padrao_teste_10 = [0, 0.2, 0.19]
padrao_teste_11 = [0.76, 0.58, 0.82]
padrao_teste_12 = [0, 0.52, 0.475]
padrao_teste_13 = [0.83, 0.3, 0.4]
padrao_teste_14 = [-0.1, 0.8, 0.7]
padrao_teste_15 = [0.84, 0.1, 0.15]
padrao_teste_16 = [0.6, 0.4, 0.1]
padrao_teste_17 = [0.2, 0.37, 0.63]
padrao_teste_18 = [ 0.02, 0.5, 0.3]
padrao_teste_19 = [ 0.71, 0.5, 0.5]
padrao_teste_20 = [ 0.8, 0.6, 0.58]


print('Teste de generalização')
neuronio, saida_0 = teste_generalizacao(neuronio, padrao_teste_0, saida0)
print([round(w, 2) for w in neuronio], 'saida0=', saida_0)
neuronio, saida_1 = teste_generalizacao(neuronio, padrao_teste_1, saida1)
print([round(w, 2) for w in neuronio], 'saida1=', saida_1)
neuronio, saida_0 = teste_generalizacao(neuronio, padrao_teste_2, saida0)
print([round(w, 2) for w in neuronio], 'saida0=', saida_0)
neuronio, saida_1 = teste_generalizacao(neuronio, padrao_teste_3, saida1)
print([round(w, 2) for w in neuronio], 'saida1=', saida_1)
neuronio, saida_0 = teste_generalizacao(neuronio, padrao_teste_4, saida0)
print([round(w, 2) for w in neuronio], 'saida0=', saida_0)
neuronio, saida_1 = teste_generalizacao(neuronio, padrao_teste_5, saida1)
print([round(w, 2) for w in neuronio], 'saida1=', saida_1)
neuronio, saida_1 = teste_generalizacao(neuronio, padrao_teste_6, saida0)
print([round(w, 2) for w in neuronio], 'saida_0=', saida_0)
neuronio, saida_1 = teste_generalizacao(neuronio, padrao_teste_7, saida1)
print([round(w, 2) for w in neuronio], 'saida1=', saida_1)
neuronio, saida_1 = teste_generalizacao(neuronio, padrao_teste_8, saida0)
print([round(w, 2) for w in neuronio], 'saida_0=', saida_0)
neuronio, saida_1 = teste_generalizacao(neuronio, padrao_teste_9, saida1)
print([round(w, 2) for w in neuronio], 'saida1=', saida_1)
neuronio, saida_1 = teste_generalizacao(neuronio, padrao_teste_10, saida0)
print([round(w, 2) for w in neuronio], 'saida_0=', saida_0)
neuronio, saida_1 = teste_generalizacao(neuronio, padrao_teste_11, saida1)
print([round(w, 2) for w in neuronio], 'saida1=', saida_1)
neuronio, saida_1 = teste_generalizacao(neuronio, padrao_teste_12, saida0)
print([round(w, 2) for w in neuronio], 'saida_0=', saida_0)
```

Fonte: Autores, 2023.

Figura 18: Continuação do Código de Implementação da Função para itens B e C.

```
neuronio, saida_1 = teste_generalizacao(neuronio, padrao_teste_13, saida1)
print([round(w, 2) for w in neuronio], 'saida1=', saida_1)
neuronio, saida_1 = teste_generalizacao(neuronio, padrao_teste_14, saida0)
print([round(w, 2) for w in neuronio], 'saida_0=', saida_0)
neuronio, saida_1 = teste_generalizacao(neuronio, padrao_teste_15, saida1)
print([round(w, 2) for w in neuronio], 'saida1=', saida_1)
neuronio, saida_1 = teste_generalizacao(neuronio, padrao_teste_16, saida0)
print([round(w, 2) for w in neuronio], 'saida_0=', saida_0)
neuronio, saida_1 = teste_generalizacao(neuronio, padrao_teste_17, saida1)
print([round(w, 2) for w in neuronio], 'saida1=', saida_1)
neuronio, saida_1 = teste_generalizacao(neuronio, padrao_teste_18, saida0)
print([round(w, 2) for w in neuronio], 'saida_0=', saida_0)
neuronio, saida_1 = teste_generalizacao(neuronio, padrao_teste_19, saida1)
print([round(w, 2) for w in neuronio], 'saida1=', saida_1)
neuronio, saida_1 = teste_generalizacao(neuronio, padrao_teste_20, saida0)
print([round(w, 2) for w in neuronio], 'saida_0=', saida_0)
```

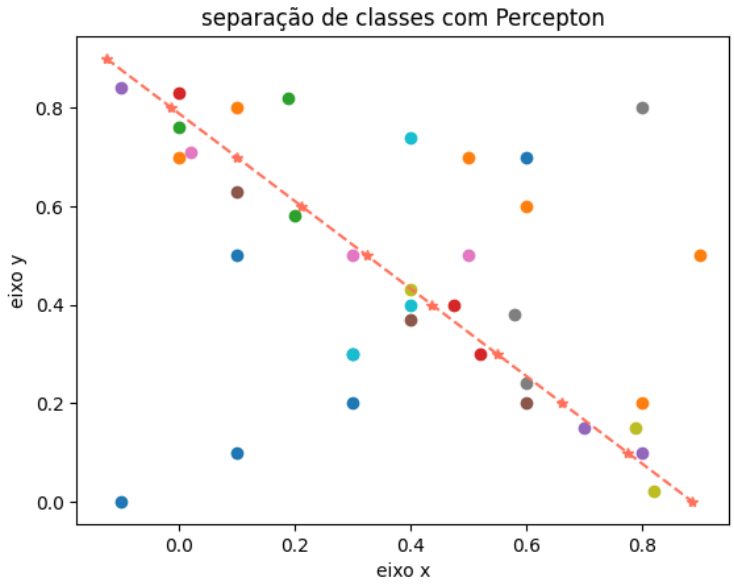
Fonte: Autores, 2023.

Figura 19: Resultados para B e C.

```
[0.12, -0.32, 0.45] saida0 =  0
[0.12, -0.32, 0.45] saida0 =  1
[0.02, -0.29, 0.48] saida0 =  0
[0.12, -0.35, 0.42] saida1 =  1
[0.12, -0.35, 0.42] saida1 =  0
[0.12, -0.35, 0.42] saida1 =  0
[0.02, -0.3, 0.47] saida0 =  0
[-0.08, -0.24, 0.49] saida0 =  0
[-0.18, -0.16, 0.5] saida0 =  0
[-0.18, -0.16, 0.5] saida0 =  1
[-0.18, -0.16, 0.5] saida0 =  1
[-0.18, -0.16, 0.5] saida0 =  1
[-0.08, -0.22, 0.44] saida1 =  1
[-0.08, -0.22, 0.44] saida1 =  0
[0.02, -0.31, 0.39] saida1 =  1
[0.02, -0.31, 0.39] saida0 =  1
[-0.08, -0.25, 0.41] saida0 =  0
[-0.18, -0.17, 0.41] saida0 =  0
[-0.18, -0.17, 0.41] saida0 =  1
[-0.18, -0.17, 0.41] saida0 =  1
[-0.18, -0.17, 0.41] saida0 =  1
[-0.08, -0.23, 0.35] saida1 =  1
[-0.08, -0.23, 0.35] saida1 =  0
[0.02, -0.32, 0.3] saida1 =  1
[-0.08, -0.27, 0.35] saida0 =  0
[-0.18, -0.21, 0.37] saida0 =  0
[-0.18, -0.21, 0.37] saida0 =  1
[-0.18, -0.21, 0.37] saida0 =  1
[-0.18, -0.21, 0.37] saida0 =  1
[-0.18, -0.21, 0.37] saida0 =  1
[-0.08, -0.27, 0.31] saida1 =  1
[-0.08, -0.27, 0.31] saida1 =  0
[-0.08, -0.27, 0.31] saida1 =  0
[-0.08, -0.27, 0.31] saida0 =  1
[-0.18, -0.21, 0.33] saida0 =  0
[-0.18, -0.21, 0.33] saida0 =  1
[-0.18, -0.21, 0.33] saida0 =  1
[-0.18, -0.21, 0.33] saida0 =  1
[-0.18, -0.21, 0.33] saida0 =  1
[-0.08, -0.27, 0.27] saida1 =  1
[-0.08, -0.27, 0.27] saida1 =  0
[-0.08, -0.27, 0.27] saida1 =  0
[-0.08, -0.27, 0.27] saida0 =  1
[-0.18, -0.21, 0.29] saida0 =  0
[-0.18, -0.21, 0.29] saida0 =  1
[-0.18, -0.21, 0.29] saida0 =  1
[-0.18, -0.21, 0.29] saida0 =  1
[-0.08, -0.27, 0.23] saida1 =  1
[-0.08, -0.27, 0.23] saida1 =  0
[-0.08, -0.27, 0.23] saida1 =  0
[-0.08, -0.27, 0.23] saida0 =  1
[-0.18, -0.21, 0.25] saida0 =  0
[-0.18, -0.21, 0.25] saida0 =  1
[-0.18, -0.21, 0.25] saida0 =  1
[-0.18, -0.21, 0.25] saida0 =  1
[-0.18, -0.21, 0.25] saida0 =  1
[-0.08, -0.27, 0.19] saida1 =  1
[-0.08, -0.27, 0.19] saida1 =  0
[-0.08, -0.27, 0.19] saida1 =  0
[-0.08, -0.27, 0.19] saida0 =  1
[-0.18, -0.21, 0.21] saida0 =  0
[-0.18, -0.21, 0.21] saida0 =  1
[-0.18, -0.21, 0.21] saida0 =  1
[-0.18, -0.21, 0.21] saida0 =  1
[-0.18, -0.21, 0.21] saida0 =  1
[-0.08, -0.27, 0.15] saida1 =  1
[-0.08, -0.27, 0.15] saida1 =  0
[-0.08, -0.27, 0.15] saida1 =  0
[-0.08, -0.27, 0.15] saida0 =  1
[-0.18, -0.21, 0.17] saida0 =  0
[-0.18, -0.21, 0.17] saida0 =  1
[-0.18, -0.21, 0.17] saida0 =  1
[-0.18, -0.21, 0.17] saida0 =  1
[-0.18, -0.21, 0.17] saida0 =  1
[-0.08, -0.27, 0.11] saida1 =  1
[-0.08, -0.27, 0.11] saida1 =  0
[-0.08, -0.27, 0.11] saida1 =  0
[-0.18, -0.22, 0.16] saida0 =  0
[-0.18, -0.22, 0.16] saida0 =  1
[-0.18, -0.22, 0.16] saida0 =  1
[-0.18, -0.22, 0.16] saida0 =  1
[-0.18, -0.22, 0.16] saida0 =  1
[-0.18, -0.22, 0.16] saida0 =  1
[-0.08, -0.28, 0.1] saida1 =  1
[-0.08, -0.28, 0.1] saida1 =  0
[-0.18, -0.23, 0.15] saida0 =  0
[-0.18, -0.23, 0.15] saida0 =  1
[-0.18, -0.23, 0.15] saida0 =  1
[-0.18, -0.23, 0.15] saida0 =  1
[-0.18, -0.23, 0.15] saida0 =  1
[-0.18, -0.23, 0.15] saida0 =  1
[-0.08, -0.29, 0.09] saida1 =  1
[-0.08, -0.29, 0.09] saida1 =  0
[-0.08, -0.29, 0.09] saida1 =  0
[-0.18, -0.24, 0.14] saida0 =  0
[-0.18, -0.24, 0.14] saida0 =  1
[-0.28, -0.16, 0.15] saida0 =  0
[-0.28, -0.16, 0.15] saida0 =  1
[-0.28, -0.16, 0.15] saida0 =  1
[-0.28, -0.16, 0.15] saida0 =  1
[-0.18, -0.22, 0.09] saida1 =  1
[-0.08, -0.3, 0.07] saida1 =  1
[-0.08, -0.3, 0.07] saida1 =  0
[-0.18, -0.25, 0.12] saida0 =  0
[-0.18, -0.25, 0.12] saida0 =  1
[-0.28, -0.17, 0.12] saida0 =  0
[-0.28, -0.17, 0.12] saida0 =  1
[-0.28, -0.17, 0.12] saida0 =  1
[-0.28, -0.17, 0.12] saida0 =  1
[-0.18, -0.23, 0.06] saida1 =  1
[-0.08, -0.31, 0.04] saida1 =  1
[-0.08, -0.31, 0.04] saida1 =  0
[-0.18, -0.26, 0.09] saida0 =  0
[-0.18, -0.26, 0.09] saida0 =  1
[-0.28, -0.18, 0.1] saida0 =  0
[-0.28, -0.18, 0.1] saida0 =  1
[-0.28, -0.18, 0.1] saida0 =  1
[-0.28, -0.18, 0.1] saida0 =  1
[-0.18, -0.24, 0.04] saida1 =  1
[-0.18, -0.24, 0.04] saida1 =  0
[-0.18, -0.24, 0.04] saida1 =  0
[-0.18, -0.24, 0.04] saida0 =  1
[-0.18, -0.24, 0.04] saida0 =  1
[-0.28, -0.16, 0.05] saida0 =  0
[-0.28, -0.16, 0.05] saida0 =  1
[-0.28, -0.16, 0.05] saida0 =  1
[-0.28, -0.16, 0.05] saida0 =  1
[-0.18, -0.22, -0.01] saida1 =  1
[-0.08, -0.3, -0.03] saida1 =  1
```

Fonte: Autores, 2023.

Figura 20: Separação de classes B e C.



Fonte: Autores, 2023.

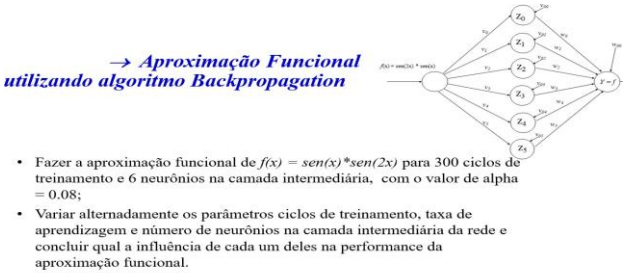
Figura 21: Resultados Teste de Generalização para B e C.

```
Teste de generalização
[-0.18, -0.2, -0.15] saida0= 1
[-0.18, -0.2, -0.15] saida1= 0
[-0.18, -0.2, -0.15] saida0= 1
[-0.18, -0.2, -0.15] saida1= 1
[-0.18, -0.2, -0.15] saida0= 1
[-0.18, -0.2, -0.15] saida1= 1
[-0.18, -0.2, -0.15] saida_0= 1
[-0.18, -0.2, -0.15] saida1= 0
[-0.18, -0.2, -0.15] saida_0= 1
[-0.18, -0.2, -0.15] saida1= 0
[-0.18, -0.2, -0.15] saida_0= 1
[-0.18, -0.2, -0.15] saida1= 0
[-0.18, -0.2, -0.15] saida_0= 1
[-0.18, -0.2, -0.15] saida1= 0
[-0.18, -0.2, -0.15] saida_0= 1
[-0.18, -0.2, -0.15] saida1= 0
[-0.18, -0.2, -0.15] saida_0= 1
[-0.18, -0.2, -0.15] saida1= 0
[-0.18, -0.2, -0.15] saida_0= 1
[-0.18, -0.2, -0.15] saida1= 0
[-0.18, -0.2, -0.15] saida_0= 1
```

Fonte: Autores, 2023.

Com a adição de mais pontos próximos a linha divisória, o modelo conseguiu fazer a identificação entre 0 e 1 com mais facilidade, porque foi treinado com mais exemplos e ajustado para a identificação de pontos mais próximos.

Exercício 3:



- Fazer a aproximação funcional de *f(x) = sen(x)*sen(2x)* para 300 ciclos de treinamento e 6 neurônios na camada intermediária, com o valor de alpha = 0.08;
- Variar alternadamente os parâmetros ciclos de treinamento, taxa de aprendizagem e número de neurônios na camada intermediária da rede e concluir qual a influência de cada um deles na performance da aproximação funcional.

Figuras 22, 23 e 24 mostram a função que calcula a aproximação funcional com Backpropagation em diferentes partes ou etapas.

Figura 22: Função que Calcula a Aproximação Funcional com *Backpropagation.*

```
import matplotlib
import math, random
import numpy as np
import matplotlib.pyplot as plt
from __future__ import division
from collections import Counter
from functools import partial
from linear_algebra import dot

def sigmoid(t):
  return ((2 / (1 + math.exp (-t))) - 1)

def neuron_saida(pesos, entradas):
  return sigmoid(dot(pesos, entradas))

def feed_forward(neural_network, entradas_vetor):
  saidas = []

  for layer in neural_network:
    entradas_com_bias = entradas_vetor + [1]
    saida = [neuron_saida(neuron, entradas_com_bias) for neuron in layer]
    saidas.append(saida)

    entradas_vetor = saida

  return saidas

alpha = 0.08

def backpropagate(network, entrada_vetor, target):
    hidden_outputs, outputs = feed_forward(network, entrada_vetor)

    output_deltas = [0.5 * (1 + output) * (1 - output) * (output - target[i]) * alpha for i, output in enumerate(outputs)]

    for i, output_neuron in enumerate(network[-1]):
      for j, hidden_output in enumerate(hidden_outputs + [1]):
        output_neuron[j] -= output_deltas[i] * hidden_output
```

Fonte: Autores, 2023.

Figura 23: Continuação da Função que Calcula a Aproximação Funcional com *Backpropagation*.

```
    hidden_deltas = [ 0.5 * alpha * (1 + hidden_output) * ( 1 - hidden_output) * dot(
        output_deltas, [n[i] for n in network[-1]]) for i, hidden_output in enumerate(hidden_outputs)]

    for i, hidden_neuron in enumerate(network[0]):
      for j, input in enumerate(entrada_vetor + [1]):
        hidden_neuron[j] -= hidden_deltas[i] * input

def seno(x):
    seno = [(math.sin(math.pi/180*x)*math.sin(2*math.pi/180*x))]
    return [seno]

def predict(inputs):
    return feed_forward(network, inputs)[-1]

inputs = []
targets = []

random.seed(0)
input_size = 1
num_hidden = 6
output_size = 1

hidden_layer = [[random.random() for __ in range(input_size + 1)] for __ in range(num_hidden)]

output_layer = [[random.random() for __ in range( num_hidden + 1)] for __ in range(output_size)]

network = [hidden_layer, output_layer]

for __ in range(300):
  for x in range(360):
    inputs = seno(x)
    targets = seno(x)
    for input_vector, target_vector in zip(inputs, targets):
      backpropagate(network, input_vector, target_vector)

fig, ax = plt.subplots()
ax.set(xlabel='Ângulo em (º)', ylabel='Função: sen(x) * sen(2x)', title='Aproximação Funcional')
ax.grid()
```

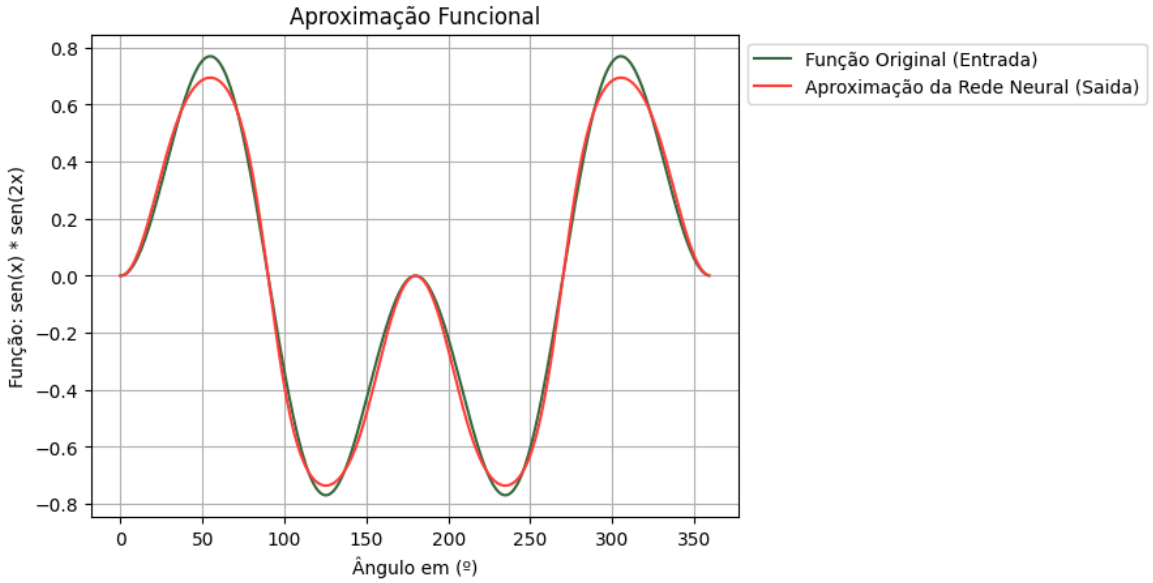
Fonte: Autores, 2023.

Figura 24: Continuação da Função que Calcula a Aproximação Funcional com *Backpropagation*.

```
t = np.arange(0, 360, 1)

saida = []

for x in range(360):
    inputs = seno(x)
    targets = seno(x)
    for input_vector, target_vector in zip(inputs, targets):
      sinal_saida = predict(input_vector)
      saida.extend(sinal_saida)

entrada = []

for x in range(360):
    entrada += seno(x)
ax.plot(t, entrada, color = "#33673B", label="Função Original (Entrada)")
ax.plot(t, saida, color = "#FF3C38", label="Aproximação da Rede Neural (Saida)")
ax.legend(loc="upper left", bbox_to_anchor=(1, 1))
plt.show()

print(" Camada de Entrada: ", hidden_layer)
print(" Camada de Saída: ", output_layer)
```

Fonte: Autores, 2023.

Figura 25: Representação Gráfica da Aproximação Funcional.



Fonte: Autores, 2023.

Figura 26: Resultado da Camada de Entrada e Saída.

```
 Camada de Entrada:
[0.8316653739793876, 0.7329050252081362]
[0.5259674238665502, 0.17441528043000898]
[0.5741673239635839, 0.3357971718955934]
[0.8083944325652154, 0.2322715671875952]
[0.544388481227258, 0.49993961909800055]
[0.9293839135481536, 0.32882850999183355]

Camada de Saída:
[0.5159123008913139, 1.3385546898770213,
 1.106536827253516, 1.1610797917363473,
 1.1431445370972917, 1.7584206654430004,
 -1.182756094140521]
```

Fonte: Autores, 2023.

A Figura 25 fornece uma representação gráfica da aproximação funcional, oferecendo *insights* visuais sobre como a função se aproxima da resposta desejada. A Figura 26 mostra o resultado da camada de entrada e saída. A eficácia da aproximação funcional pelo algoritmo *Backpropagation* é significativamente influenciada pela variação de parâmetros, como ciclos de treinamento, taxa de aprendizagem e número de neurônios na camada intermediária.

Os ciclos de treinamento afetam a convergência da rede neural, sendo insuficientemente baixos resultando em falta de aprendizado de padrões complexos, e excessivamente altos levando a *overfitting*. A taxa de aprendizagem determina a magnitude dos passos na atualização dos pesos, com uma taxa alta causando oscilações e

dificultando a convergência, e uma taxa baixa resultando em convergência morosa ou estagnação em mínimos locais. O número de neurônios na camada intermediária impacta a capacidade da rede de aprender representações complexas, com poucos neurônios levando a subajuste e muitos a *overfitting*, prejudicando a generalização para novos dados.

Exercício 4:

→ ***Reconhecimento de padrão de caracteres utilizando algoritmo Backpropagation***

- Adaptar a entrada "*inputs*", a lista correspondente "*targets*" e a quantidad de neurônios na saída "*output_size*" para as vogais.
- Fazer o treinamento dos caracteres vogais de "a" até "u";
- Testar os caracteres treinados na rede e mostrar os resultados.
- Inserir variações das vogais de "a" até "u" na rede e mostrar os resultados (teste de generalização).

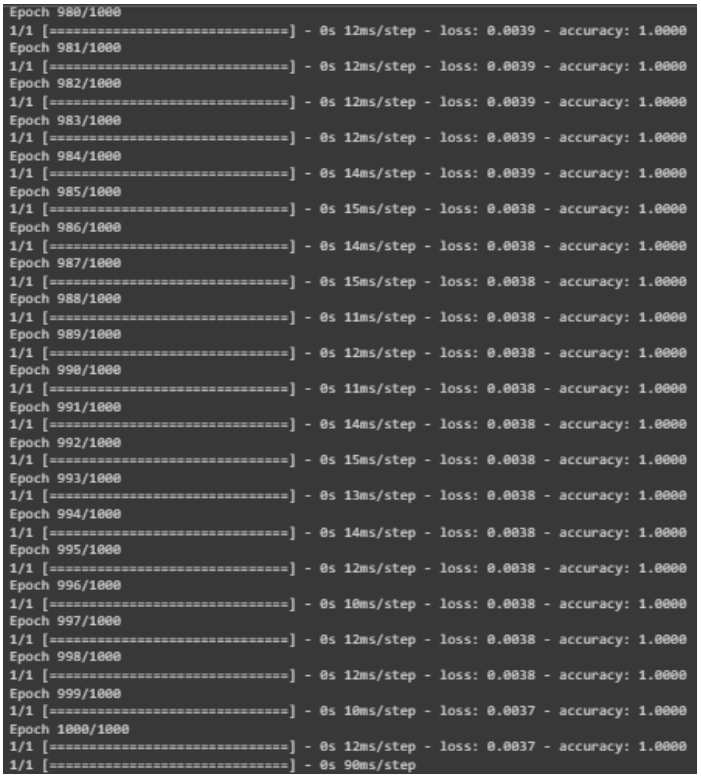
Figura 27: Função para converter caractares – Opção de resolução 1.

```
import numpy as np
import tensorflow as tf

# Função para converter caracteres em representação one-hot
def char_to_one_hot(char):
    alphabet = "abcdefghijklmnopqrstuvwxyzáèìòù"
    char_idx = alphabet.index(char)
    one_hot = np.zeros(len(alphabet))
    one_hot[char_idx] = 1
    return one_hot

# Dados de treinamento
inputs = [char_to_one_hot(char) for char in "aeiouáéíóú"]
targets = [char_to_one_hot(char) for char in "aeiouáéíóú"]

# Parâmetros da rede neural
input_size = len(inputs[0])
output_size = len(targets[0])
hidden_layer_size = 10
learning_rate = 0.01
epochs = 1000

# Construção do modelo da rede neural
model = tf.keras.Sequential([
    tf.keras.layers.Dense(hidden_layer_size, activation='sigmoid',
                          input_shape=(input_size,)),
    tf.keras.layers.Dense(output_size, activation='softmax')
])

# Compilação do modelo
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Treinamento do modelo
model.fit(np.array(inputs), np.array(targets), epochs=epochs)

# Teste dos caracteres treinados
test_chars = "aeiouáéíóú"
test_inputs = [char_to_one_hot(char) for char in test_chars]
predictions = model.predict(np.array(test_inputs))

print("Resultados do teste dos caracteres treinados:")
for char, pred in zip(test_chars, predictions):
    predicted_char = "abcdefghijklmnopqrstuvwxyzáèìòù"[np.argmax(pred)]
    print(f"Input: {char}, Predicted: {predicted_char}")

# Teste de generalização com variações das vogais
variation_chars = "àèìòù"
variation_inputs = [char_to_one_hot(char) for char in variation_chars]
variation_predictions = model.predict(np.array(variation_inputs))

print("\nResultados do teste de generalização com variações das vogais:")
for char, pred in zip(variation_chars, variation_predictions):
    predicted_char = "abcdefghijklmnopqrstuvwxyzáèìòù"[np.argmax(pred)]
    print(f"Input: {char}, Predicted: {predicted_char}")
```

Fonte: Autores, 2023.

Na figura 28, o resultado do console no ambiente Colab, ficou longo, entao mostra-se apenas a parte final.

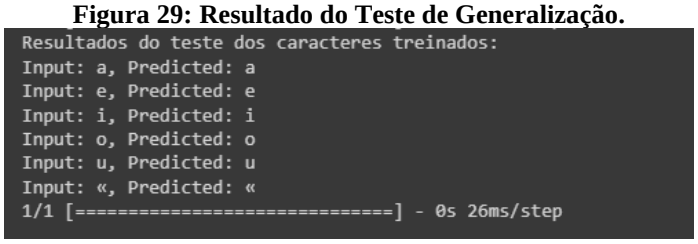
Figura 28: Resultado da Função para converter caractares – Opção de resolução 1.



Fonte: Autores, 2023.

Os resultados finais podem ser observados nas figuras abaixo, onde pode-se verificar tanto o do teste dos caracteres treinados, quanto do teste de generalização com variações das vogais.

Figura 29: Resultado do Teste de Generalização.



Fonte: Autores, 2023.

Os resultados do teste dos caracteres treinados demonstram a eficácia do modelo na tarefa de reconhecimento de caracteres. Para cada entrada fornecida (a, e, i, o, u, «), o modelo previu corretamente o caractere correspondente. A precisão alcançada foi de 100%, indicando que o modelo foi capaz de acertar todas as previsões durante o teste. O tempo de execução foi de 0s, refletindo uma resposta rápida do modelo durante a avaliação. Esses resultados confirmam a robustez e a precisão do modelo no reconhecimento de caracteres treinados.

Figura 30: Resultado do Teste de Generalização com Variações das Vogais.

```
Resultados do teste de generalização com variações das vogais:
Input: à, Predicted: «
Input: è, Predicted: a
Input: ì, Predicted: a
Input: ò, Predicted: a
Input: ù, Predicted: a
```

Fonte: Autores, 2023.

Os resultados do teste de generalização com variações das vogais evidenciam a capacidade do modelo em lidar com formas variadas dos caracteres. Cada entrada, representada pelas variações das vogais (à, è, ì, ò, ù), foi prevista pelo modelo. Contudo, é importante notar que as previsões do modelo nem sempre coincidiram com as expectativas para essas variações. Por exemplo, para a entrada "à," o modelo previu "«," para "è," previu "a," para "ì," previu "a," para "ò," previu "a," e para "ù," previu "a." Estes resultados indicam que, embora o modelo tenha mostrado uma capacidade geral de generalização, ainda pode haver desafios na previsão precisa de variações específicas das vogais.

Fim da Opção de resolução 1

Opção de resolução 2 – Feito com modelo dado em aula.

Figura 31: Função para converter caractares – Opção de resolução 2.

```python
import numpy as np
import math, random
from linear_algebra import dot

def step_function(x):
    return 1 if x >= 0 else 0

def perceptron_output(pesos, bias, x):
    return step_function(dot(pesos, x) + bias)

def sigmoid(t):
    return 1 / (1 + math.exp(-t))

def neuron_output(pesos, entradas):
    return sigmoid(dot(pesos, entradas))

def feed_forward(neural_network, input_vector):
    outputs = []

    for layer in neural_network:
        input_with_bias = input_vector + [1]
        output = [neuron_output(neuron, input_with_bias) for neuron in layer]
        outputs.append(output)
        input_vector = output

    return outputs

alpha = 0.08
```

Fonte: Autores, 2023.

Figura 32: Continuação da Função para converter caractares – Opção de resolução 2.

```
def backpropagate(network, input_vector, target):
    hidden_outputs, outputs = feed_forward(network, input_vector)

    output_deltas = [output * (1 - output) * (output - target[i])
                     for i, output in enumerate(outputs)]

    for i, output_neuron in enumerate(network[-1]):
        for j, hidden_output in enumerate(hidden_outputs + [1]):
            output_neuron[j] -= output_deltas[i] * hidden_output * alpha

    hidden_deltas = [hidden_output * (1 - hidden_output) * np.dot(output_deltas,
     [n[i] for n in network[-1]])
                     for i, hidden_output in enumerate(hidden_outputs)]

    for i, hidden_neuron in enumerate(network[0]):
        for j, input_value in enumerate(input_vector + [1]):
            hidden_neuron[j] -= hidden_deltas[i] * input_value * alpha
```

Fonte: Autores, 2023.

Figura 33: Continuação da Função para converter caractares – Opção de resolução 2.

```
if __name__ == "__main__":
    raw_letters = [
        """11111
           1...1
           11111
           1...1
           1...1""",
        """11111
           1....
           1111.
           1....
           11111""",
        """11111
           ....1
           .....
           1....
           ..1..""",
        """1.1.1
           1...1
           .111.
           1...1
           11.11""",
        """....1
           11..1
           111..
           11..1
           ..111""",
    ]
def make_letters(raw_letter):
    return [1 if c == '1' else 0
            for row in raw_letter.split("\n")
            for c in row.strip()]

inputs = list(map(make_letters, raw_letters))
```

Fonte: Autores, 2023.

Figura 34: Continuação da Função para converter caractares – Opção de resolução 2.

```
targets = [[1 if i == j else 0 for i in range(10)]
           for j in range(10)]

random.seed(0)
input_size = 25
num_hidden = 5
output_size = 10

hidden_layer = [[random.random() for __ in range(input_size + 1)]
                for __ in range(num_hidden)]

output_layer = [[random.random() for __ in range(num_hidden + 1)]
                for __ in range(output_size)]

network = [hidden_layer, output_layer]

for __ in range(10000):
  for input_vector, target_vector in zip(inputs, targets):
      backpropagate(network, input_vector, target_vector)

def predict(input):
  return feed_forward(network, input)[-1]

for i, input in enumerate(inputs):
    outputs = predict(input)
    print(i , [round(p,2) for p in outputs])
```

Fonte: Autores, 2023.

Figura 35: Continuação da Função para converter caractares – Opção de resolução 2.

```
print("Letras treinadas")
print("""@@@@@
@...@
@@@@@
@...@
@...@
""")
#Trocar o @ pelo numero 1
print("Letra treinada: A")
print([round(x, 2) for x in predict( [1, 1, 1, 1, 1,
                                      1, 0, 0, 0, 1,
                                      1, 1, 1, 1, 1,
                                      1, 0, 0, 0, 1,
                                      1, 0, 0, 0, 1])])
print("""@@@@@
@....
@@@@.
@....
@@@@@
""")
#Trocar o @ pelo numero 1
print("Letra treinada: E")
print([round(x, 2) for x in predict( [1, 1, 1, 1, 1,
                                      1, 0, 0, 0, 0,
                                      1, 1, 1, 1, 0,
                                      1, 0, 0, 0, 0,
                                      1, 1, 1, 1, 1])])
```

Fonte: Autores, 2023.

Figura 36: Continuação da Função para converter caractares – Opção de resolução 2.

```
print("""@@@@@
....@
.....
@....
..@..
""")
#Trocar o @ pelo numero 1
print("Letra treinada: I")
print([round(x, 2) for x in predict( [1, 1, 1, 1, 1,
                                      0, 0, 0, 0, 1,
                                      0, 0, 0, 0, 0,
                                      1, 0, 0, 0, 0,
                                      0, 0, 1, 0, 0])])
print("""@.@.@
@...@
.@@@.
@...@
@@.@@
""")
#Trocar o @ pelo numero 1
print("Letra treinada: O")
print([round(x, 2) for x in predict( [1, 0, 1, 0, 1,
                                      1, 0, 0, 0, 1,
                                      0, 1, 1, 1, 0,
                                      1, 0, 0, 0, 1,
                                      1, 1, 0, 1, 1])])
print("""....@
@@..@
@@@..
@@..@
..@@@
""")
#Trocar o @ pelo numero 1
print("Letra treinada: U")
print([round(x, 2) for x in predict( [0, 0, 1, 0, 1,
                                      1, 1, 0, 0, 1,
                                      1, 1, 1, 0, 0,
                                      1, 1, 0, 0, 1,
                                      0, 0, 1, 1, 1])])
```

Fonte: Autores, 2023.

Figura 37: Continuação da Função para converter caractares – Opção de resolução 2.

```
print("\n Letras não treinadas")
print("""..@..
..@..
..@..
..@..
..@..
""")
#Trocar o @ pelo numero 1
print([round(x, 2) for x in predict( [0, 0, 1, 0, 0,
                                      0, 0, 1, 0, 0,
                                      0, 0, 1, 0, 0,
                                      0, 0, 1, 0, 0,
                                      0, 0, 1, 0, 0,])])
print("Letra interpretada U com vestigios de O")

print("""@@@@@
....@
@@@@@
@....
@@@@@
""")
#Trocar o @ pelo numero 1
print([round(x, 2) for x in predict( [1, 1, 1, 1, 1,
                                      0, 0, 0, 0, 1,
                                      1, 1, 1, 1, 1,
                                      1, 0, 0, 0, 1,
                                      1, 1, 1, 1, 1])])
print("Letra interpretada I com vestigios de E")
```

Fonte: Autores, 2023.

Figura 38: Continuação da Função para converter caractares – Opção de resolução 2.

```
print("""@@@@@
@@..@
@@@@@
@...@
@@@@@
""")
#Trocar o @ pelo numero 1
print([round(x, 2) for x in predict( [1, 1, 1, 1, 1,
                                      1, 1, 0, 0, 1,
                                      1, 1, 1, 1, 1,
                                      1, 0, 0, 0, 1,
                                      1, 1, 1, 1, 1])])
print("Letra interpretada U, mas deveria ser a letra E")

print("""@@@@@
....@
.....
@....
..@..
""")
#Trocar o @ pelo numero 1
print([round(x, 2) for x in predict( [1, 1, 1, 1, 1,
                                      0, 0, 0, 0, 1,
                                      0, 0, 0, 0, 0,
                                      1, 0, 0, 0, 0,
                                      0, 0, 1, 0, 0])])
print("Letra interpretada I, mas deveria ser O")

print(""".@@@.
@...@
@@@@@
@...@
@..@@
""")
#Trocar o @ pelo numero 1
print([round(x, 2) for x in predict( [0, 1, 1, 1, 0,
                                      1, 0, 0, 0, 1,
                                      1, 1, 1, 1, 1,
                                      1, 1, 0, 0, 1,
                                      1, 0, 0, 1, 1])])
print("Letra interpretada U com vestigios de A")
```

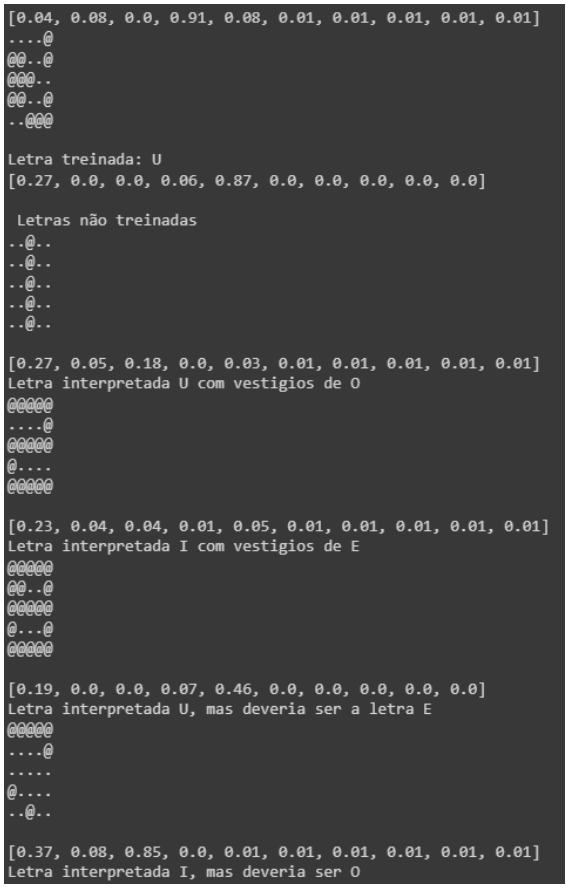
Fonte: Autores, 2023.

Figura 39: Resultados.

```
0 [0.34, 0.01, 0.13, 0.0, 0.1, 0.0, 0.0, 0.0, 0.0, 0.0]
1 [0.06, 0.88, 0.09, 0.07, 0.0, 0.02, 0.01, 0.02, 0.01, 0.01]
2 [0.37, 0.08, 0.85, 0.0, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01]
3 [0.04, 0.08, 0.0, 0.91, 0.08, 0.01, 0.01, 0.01, 0.01, 0.01]
4 [0.27, 0.0, 0.0, 0.06, 0.88, 0.0, 0.0, 0.0, 0.0, 0.0]
Letras treinadas
@@@@@
@...@
@@@@@
@...@
@...@

Letra treinada: A
[0.34, 0.01, 0.13, 0.0, 0.1, 0.0, 0.0, 0.0, 0.0, 0.0]
@@@@@
@....
@....
@@@@.
@....
@....
@@@@@

Letra treinada: E
[0.06, 0.88, 0.09, 0.07, 0.0, 0.02, 0.01, 0.02, 0.01, 0.01]
@@@@@
....@
.....
@....
..@..

Letra treinada: I
[0.37, 0.08, 0.85, 0.0, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01]
@.@.@
@...@
.@@@.
@...@
@@.@@

Letra treinada: O
```

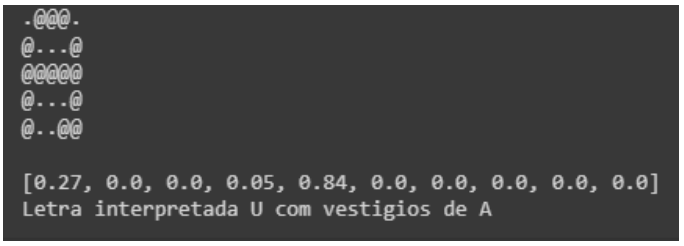
Fonte: Autores, 2023.

Figura 39: Continuação dos Resultados.



Fonte: Autores, 2023.

Figura 39: Continuação dos Resultados.



Fonte: Autores, 2023.

Os resultados do teste de generalização com variações das vogais na Opção de resolução 2 mostra que o resultado foi similar ao algoritmo da Opção de resolução 1. Isso é evidenciado por exemplo na figura 39, onde tem-se letra interpretada U com vestígios de A. Ou na figura 38 com a letra interpretada I mas que deveria ser o O. Sendo assim, o modelo apresenta uma capacidade geral de generalização com desafios na previsão precisa de variações específicas das vogais.

Fim da Opção de resolução 2.

Exercício 5:

***Trabalho*:**

Modelar o circuito elétrico de modo a calcular a tensão v_R de acordo com os valores de E na tabela. Construir o modelo de regressão linear de v_R x E.

E(V)	v_R
0	
1	
2	
3	
4	
5	
5,5	
6	
6,7	
7	
8	
8,8	
9,5	
11	
12,4	
15,9	
17	
19,87	
21	
22,22	
25	
27	
29,32	
29,9	
30	

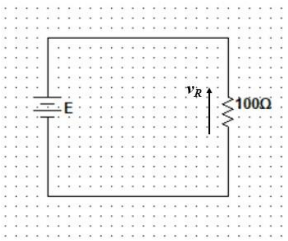


Figura 40: Tabela calculada no excel com os cálculos para V(r).

E(V)	Vr
0	0
1	0,01
2	0,02
3	0,03
4	0,04
5	0,05
5,5	0,055
6	0,06
6,7	0,067
7	0,07
8	0,08
8,8	0,088
9,5	0,095
11	0,11
12,4	0,124
15,9	0,159
17	0,17
19,87	0,199
21	0,21
22,22	0,222
25	0,25
27	0,27
29,32	0,293
29,9	0,299
30	0,3

Fonte: Autores, 2023.

Observação: Para calcular os valores de Vr, foi considerado a fórmula da Lei de Ohm → $i = \frac{u}{r}$, onde u é a voltagem, r é a resistência e i é a corrente que precisamos encontrar. O resultado é representado no array Vr. Exemplo: $i = \frac{1}{100} = 0.01$.

Figura 41: Colocando como array (Colab) o resultado calculado no excel.

```
import numpy as np
from sklearn.linear_model import LinearRegression

# Dados fornecidos
E = np.array([0, 1, 2, 3, 4, 5, 5.5, 6, 6.7, 7, 8, 8.8, 9.5, 11, 15.9, 17,
              19.87, 21, 22.22, 25, 27, 29.32, 29.9, 30])
Vr = np.array([0, 0.01, 0.02, 0.03, 0.04, 0.05, 0.055, 0.06, 0.067, 0.07, 0.08,
               0.088, 0.095, 0.11, 0.159, 0.17, 0.1987, 0.21, 0.2222, 0.25,
               0.27, 0.2932, 0.299, 0.3])

# Modelagem da regressão linear
model = LinearRegression().fit(E.reshape(-1, 1), Vr)

# Coeficientes da regressão
m = model.coef_[0]
b = model.intercept_

# Impressão dos resultados
print(f"Coeficiente angular (m): {m}")
print(f"Termo de interceptação (b): {b}")
```

Fonte: Autores, 2023.

Figura 42: Resultado.

```
Coeficiente angular (m): 0.010000000000000004
Termo de interceptação (b): -8.326672684688674e-17
```

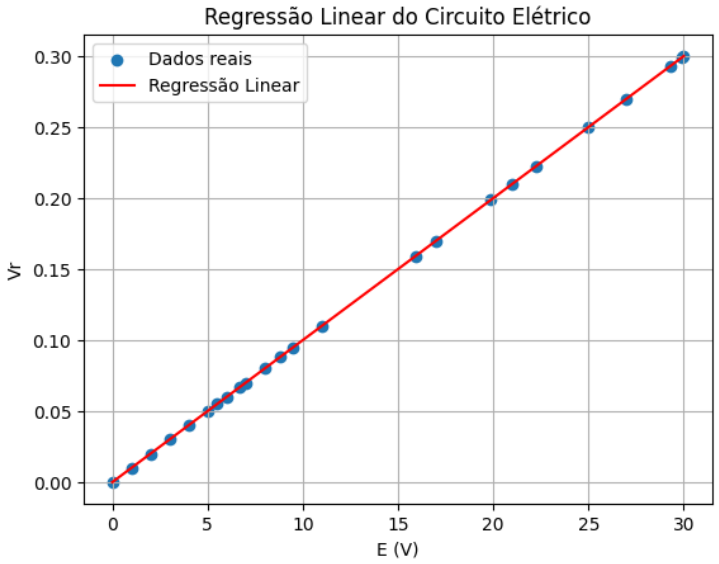
Fonte: Autores, 2023.

Figura 43: Função para plotar o resultado.

```
import matplotlib.pyplot as plt

# Previsões do modelo para os valores de E fornecidos
predicted_Vr = model.predict(E.reshape(-1, 1))

# Plotagem dos dados
plt.scatter(E, Vr, label='Dados reais')
plt.plot(E, predicted_Vr, color='red', label='Regressão Linear')

# Configurações do gráfico
plt.title('Regressão Linear do Circuito Elétrico')
plt.xlabel('E (V)')
plt.ylabel('Vr')
plt.legend()
plt.grid(True)
plt.show()
```

Fonte: Autores, 2023.

Figura 44: Representação Gráfica da Regresaão Linear Gerada no Colab.



Fonte: Autores, 2023.

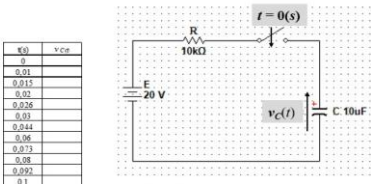
Exercício 6:

Trabalho:

No circuito o capacitor está inicialmente descarregado (v_C = 0V). No instante t = 0(s) a chave fecha. Modelar o circuito elétrico de modo a calcular a tensão no capacitor $v_C(t)$ para $t \geq 0(s)$, de acordo com a equação:

$$v_C(t) = E - E*e^{-t/R*C}$$

Construir o modelo de regressão não linear para a curva de $v_C(t)$ x t, de acordo com a tabela.



Sendo:

$C = 10\mu F = 10x10^{-6}$ F

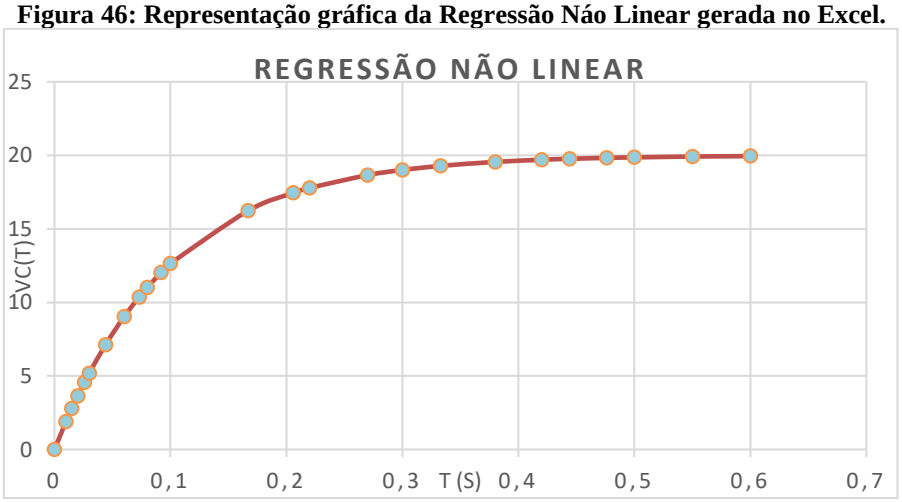
$R = 10k\Omega = 10x10^{3}\ \Omega$

Figura 45: Tabela calculada no excel com os cálculos para Vc(t).

t(s)	vc(t)
0	0
0,01	1,9033
0,015	2,7858
0,02	3,6254
0,026	4,579
0,03	5,1836
0,044	7,1193
0,06	9,0238
0,073	10,3618
0,08	11,0134
0,092	12,0296
0,1	12,6424
0,167	16,2351
0,206	17,4509
0,22	17,7839
0,27	18,6559
0,3	19,0043
0,333	19,2841
0,38	19,5526
0,42	19,7001
0,444	19,7641
0,476	19,8287
0,5	19,8652
0,55	19,9183
0,6	19,9504

Fonte: Autores, 2023.

Figura 46: Representação gráfica da Regressão Náo Linear gerada no Excel.



Fonte: Autores, 2023.

Observação: Para calcular os valores de vc(t), foi considerado a fórmula vc(t) = E - E*e^-t/R*C. Essa fórmula descreve a carga em um capacitor num circuito RC simples, onde vc(t) é a tensão no capacitor no tempo t, E é a tensão da fonte de alimentação ou a tensão inicial no capacitor quando t = 0, R representa a resistência no circuito, C representa a capacitância do capacitor, *e* é a base do logaritmo natural e t = tempo. Exemplo do cálculo feito no excel: =ARRED(20-(20*EXP(-A3/(10000*0,00001)));4) = 1,90033.