Lista 7

Lucas Gualtieri Firace Evangelista

Questão 1

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

class Perceptron:

    def __init__(self, input_size, learning_rate=0.1, epochs=1000):

        self.weights = np.zeros(input_size + 1)
        self.learning_rate = learning_rate
        self.epochs = epochs

    def predict(self, x):

        x = np.insert(x, 0, 1)
        return 1 if np.dot(self.weights, x) >= 0 else 0

    def train(self, X, y):

        errors = []

        for _ in range(self.epochs):

            total_error = 0

            for xi, target in zip(X, y):
                prediction = self.predict(xi)
                error = target - prediction
                total_error += abs(error)
                self.weights += self.learning_rate * error * np.insert(xi, 0, 1)

            errors.append(total_error)

            if total_error == 0: break

        return errors
```

A função descrita inicialmente define os pesos como zeros e os ajusta durante o treinamento, com base no erro entre as previsões e os valores esper-

ados. A função `predict` calcula a soma ponderada das entradas (incluindo o bias, representado por um valor 1 adicional às entradas) e retorna 1 se o resultado for maior ou igual a 0; caso contrário, retorna 0.

O método `train` percorre os dados de entrada (`X`) e as saídas esperadas (`y`) durante um número predefinido de épocas. Durante esse processo, os pesos são ajustados proporcionalmente ao erro e à taxa de aprendizado. O treinamento para quando o erro total em uma época chega a zero. Além disso, os erros de cada época são armazenados para posterior análise e visualização do progresso do aprendizado.

Visualização do limite de decisão

```python
def plot_decision_boundary(X, y, weights, title):

    plt.figure()

    for xi, target in zip(X, y):
        color = 'r' if target == 0 else 'b'
        plt.scatter(xi[0], xi[1], c=color)

    x1 = np.linspace(0, 1, 100)
    x2 = -(weights[0] + weights[1] * x1) / weights[2]

    plt.plot(x1, x2, 'k-')
    plt.title(title)
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.show()
```

A função implementada também permite plotar o limite de decisão.

Testando as portas com 2 inputs

Ao aplicar o perceptron na porta XOR, observa-se que ele não consegue separar os limites de cada grupo. Isso ocorre porque o perceptron é incapaz de lidar com problemas não linearmente separáveis, como demonstrado no gráfico.

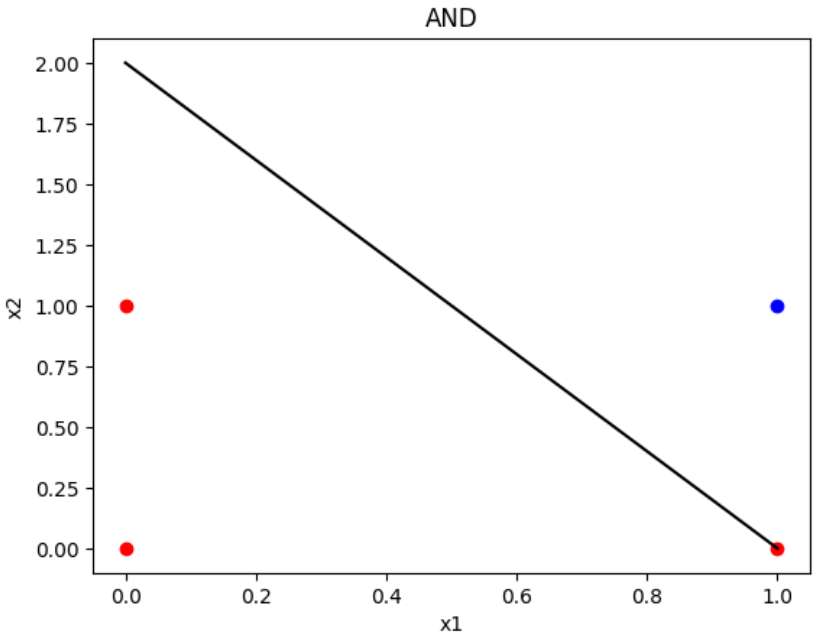
```python
# AND com 2 inputs
X_and = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_and = np.array([0, 0, 0, 1])
perceptron_and = Perceptron(input_size=2)
errors_and = perceptron_and.train(X_and, y_and)
plot_decision_boundary(X_and, y_and, perceptron_and.weights, 'AND')

# OR com 2 inputs
X_or = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_or = np.array([0, 1, 1, 1])
perceptron_or = Perceptron(input_size=2)
errors_or = perceptron_or.train(X_or, y_or)
plot_decision_boundary(X_or, y_or, perceptron_or.weights, 'OR')

# XOR com 2 inputs
X_xor = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_xor = np.array([0, 1, 1, 0])
perceptron_xor = Perceptron(input_size=2)
errors_xor = perceptron_xor.train(X_xor, y_xor)
plot_decision_boundary(X_xor, y_xor, perceptron_xor.weights, 'XOR')

print("Pesos Perceptron para AND:", perceptron_and.weights)
print("Pesos Perceptron para OR:", perceptron_or.weights)
print("Pesos Perceptron para XOR:", perceptron_xor.weights)
```

Questão 2

1. **A importância da taxa de aprendizado**
   A taxa de aprendizado determina o quanto os pesos e o bias são ajustados em cada iteração durante o treinamento. Um valor moderado é ideal, pois equilibra eficiência no aprendizado e estabilidade no processo. Métodos adaptativos, como Adam, podem melhorar significativamente o desempenho.

2. **A importância do bias**
   O bias adiciona um deslocamento ao somatório ponderado das entradas, permitindo à rede aprender funções que não passam pela origem. Isso aumenta a flexibilidade e a capacidade de modelar funções complexas, sendo essencial em problemas práticos.

3. **Funções de ativação e a relevância de suas derivadas**
   Funções de ativação aplicam transformações não lineares às saídas dos neurônios, permitindo que a rede modele relações complexas. A derivada da função de ativação é usada no *backpropagation* para calcular os gra-

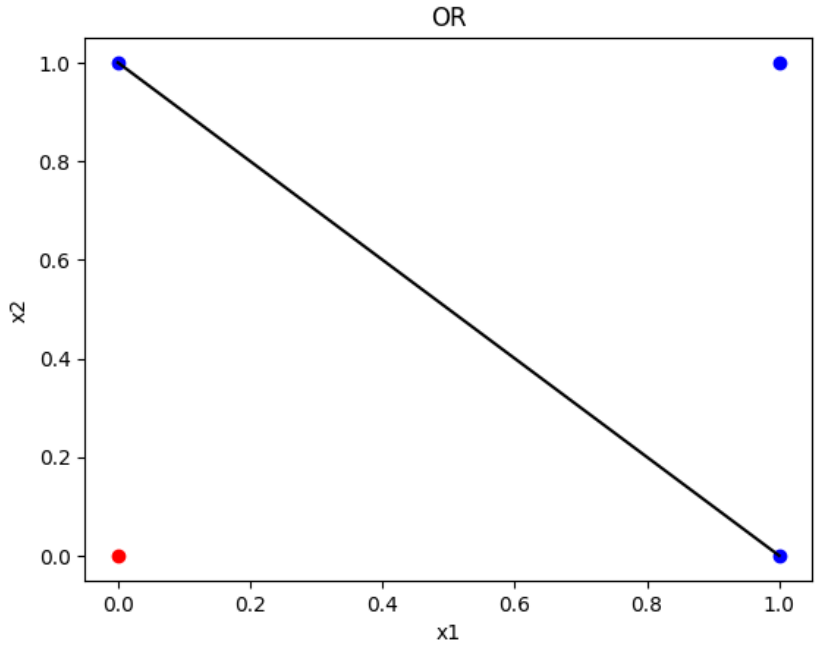


dientes e ajustar os pesos.

- **Sigmoid:** Sofre de *vanishing gradient* em redes profundas, mas é útil para saídas entre 0 e 1.
- **Tanh:** Melhora o aprendizado ao centralizar os valores em torno de 0, sendo uma alternativa superior ao Sigmoid.
- **ReLU:** Funciona bem em redes profundas, mas pode apresentar o problema de "neurônios mortos" (gradiente zero).

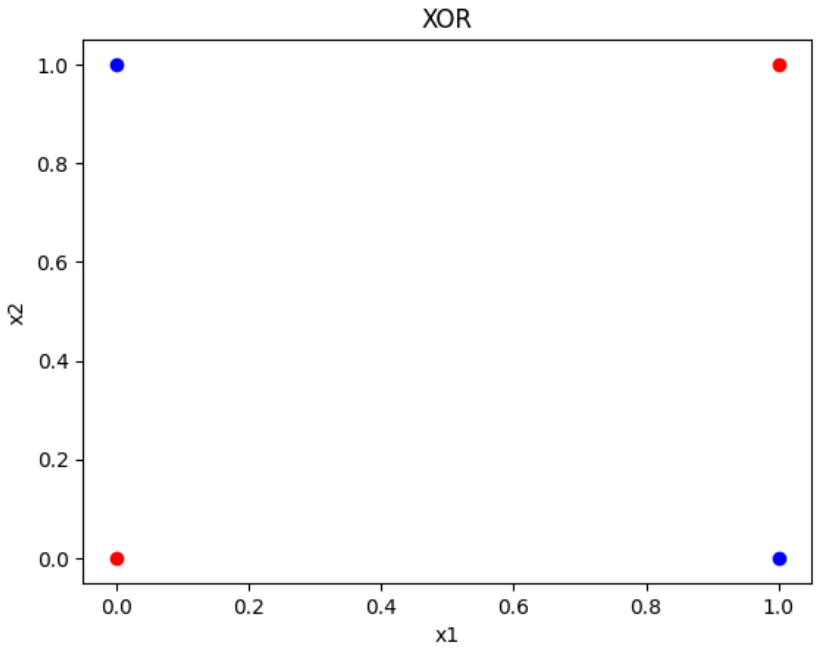
A derivada controla o ajuste dos pesos. Derivadas muito pequenas (como no Sigmoid em valores extremos) podem levar ao *vanishing gradient*, dificultando o aprendizado. Por isso, funções como ReLU são mais robustas.

4. **Por que o *backpropagation* usa funções de ativação não lineares?**
   Sem funções não lineares, a composição de várias camadas resultaria apenas em transformações lineares. Isso limitaria a rede a resolver problemas linearmente separáveis, como o Perceptron. A introdução



de não linearidade permite modelar relações complexas, como o XOR, expandindo significativamente o potencial de aprendizado da rede.

Código: O código completo está disponível no Google Colab.




```python
import numpy as np

class BackPropagation:
    def __init__(self, layers, learning_rate=0.1, epochs=1000, activation='sigmoid', use_bias=True):
        self.layers = layers
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.activation = activation
        self.use_bias = use_bias
        self.weights = []
        self.biases = []
        self.init_weights()

    def init_weights(self):
        for i in range(len(self.layers) - 1):
            weight = np.random.randn(self.layers[i], self.layers[i + 1])
            self.weights.append(weight)
            if self.use_bias:
                bias = np.random.randn(self.layers[i + 1])
                self.biases.append(bias)

    def activation_function(self, x):
        if self.activation == 'sigmoid':
            return 1 / (1 + np.exp(-x))
        elif self.activation == 'tanh':
            return np.tanh(x)
        elif self.activation == 'relu':
            return np.maximum(0, x)

    def activation_derivative(self, x):
        if self.activation == 'sigmoid':
            return x * (1 - x)
        elif self.activation == 'tanh':
            return 1 - x ** 2
        elif self.activation == 'relu':
            return np.where(x > 0, 1, 0)
```

```python
# Função para criar dados de entrada e saída para AND, OR e XOR com n entradas
def create_data(logic_function, n):
    X = np.array([list(map(int, bin(i)[2:].zfill(n))) for i in range(2**n)])
    if logic_function == 'AND':
        y = np.array([[np.prod(x)] for x in X])
    elif logic_function == 'OR':
        y = np.array([[np.max(x)] for x in X])
    elif logic_function == 'XOR':
        y = np.array([[np.sum(x) % 2] for x in X])
    return X, y
```

```python
n = 2
logic_function = 'XOR'
X, y = create_data(logic_function, n)

nn = BackPropagation(layers=[n, 2, 1], learning_rate=0.1, epochs=10000, activation='sigmoid', use_bias=True)
nn.train(X, y)
predictions = nn.predict(X)
print(f"Predictions for {logic_function} with {n} inputs:\n", predictions)
```