1)

Perceptron

```
import random

def perceptron(x, y, epochs=100, lr=0.1, random_state=None):
    # Inicialização
    n = len(x[0])
    if random_state is not None:
        random.seed(random_state)

    weights = [random.uniform(-1, 1) for _ in range(n)]
    b = random.uniform(-1, 1)
    epoch = 0
    output = []

    while output != y and epoch < epochs:
        output = []

        for i in range(len(x)):
            # Soma ponderada + bias
            sum_w = sum(x[i][j] * weights[j] for j in range(n)) + b

            # Função de ativação
            found = 1 if sum_w > 0 else 0
            output.append(found)

            # Cálculo do erro
            error = y[i] - found

            # Ajuste de pesos
            for j in range(n):
                weights[j] += lr * error * x[i][j]
            b += lr * error

        epoch += 1

        print(f"\nÉpoca {epoch}")
        print(f"Pesos: {weights}")
        print(f"Bias: {b}")
# perceptron
```

Resolvendo AND, OR e XOR

```
entradas = [[0, 0], [0, 1], [1, 0], [1, 1]]
```

AND

```
saidas_and = [0, 0, 0, 1]
perceptron(entradas, saidas_and, epochs=50, lr=0.1, random_state=42)
```

```
    Época 1
    Pesos: [0.3788535969157675, -0.8499784895546662]
    Bias: -0.3499413632617615

    Época 2
    Pesos: [0.3788535969157675, -0.7499784895546662]
    Bias: -0.3499413632617615

    Época 3
    Pesos: [0.3788535969157675, -0.6499784895546662]
    Bias: -0.3499413632617615

    Época 4
    Pesos: [0.3788535969157675, -0.5499784895546662]
    Bias: -0.3499413632617615

    Época 5
    Pesos: [0.3788535969157675, -0.44997848955466624]
    Bias: -0.3499413632617615

    Época 6
    Pesos: [0.3788535969157675, -0.34997848955466626]
    Bias: -0.3499413632617615

    Época 7
    Pesos: [0.3788535969157675, -0.24997848955466626]
    Bias: -0.3499413632617615

    Época 8
    Pesos: [0.3788535969157675, -0.14997848955466625]
    Bias: -0.3499413632617615

    Época 9
    Pesos: [0.3788535969157675, -0.049978489554666244]
    Bias: -0.3499413632617615

    Época 10
    Pesos: [0.3788535969157675, 0.05002151044533376]
    Bias: -0.3499413632617615

    Época 11
    Pesos: [0.3788535969157675, 0.15002151044533377]
    Bias: -0.3499413632617615

    Época 12
```

```
Pesos: [0.3788535969157675, 0.25002151044533377]
Bias: -0.3499413632617615

Época 13
Pesos: [0.2788535969157675, 0.25002151044533377]
Bias: -0.4499413632617615

Época 14
Pesos: [0.2788535969157675, 0.25002151044533377]
Bias: -0.4499413632617615
```

OR

```
saidas_or = [0, 1, 1, 1]
perceptron(entradas, saidas_or, epochs=50, lr=0.1, random_state=42)
```

```
Época 1
Pesos: [0.47885359691576745, -0.7499784895546662]
Bias: -0.1499413632617615

Época 2
Pesos: [0.5788535969157674, -0.5499784895546662]
Bias: 0.050058636738238516

Época 3
Pesos: [0.5788535969157674, -0.44997848955466624]
Bias: 0.050058636738238516

Época 4
Pesos: [0.5788535969157674, -0.34997848955466626]
Bias: 0.050058636738238516

Época 5
Pesos: [0.5788535969157674, -0.24997848955466626]
Bias: 0.050058636738238516

Época 6
Pesos: [0.5788535969157674, -0.14997848955466625]
Bias: 0.050058636738238516

Época 7
Pesos: [0.5788535969157674, -0.049978489554666244]
Bias: 0.050058636738238516

Época 8
Pesos: [0.5788535969157674, 0.05002151044533376]
Bias: 0.050058636738238516

Época 9
Pesos: [0.5788535969157674, 0.05002151044533376]
Bias: -0.04994136326176149
```

```
Epoca 10
Pesos: [0.5788535969157674, 0.05002151044533376]
Bias: -0.04994136326176149
```

XOR

```
saidas_xor = [0, 1, 1, 0]
perceptron(entradas, saidas_xor, epochs=50, lr=0.1, random_state=42)
```

```
Época 1
Pesos: [0.3788535969157675, -0.8499784895546662]
Bias: -0.2499413632617615

Época 2
Pesos: [0.3788535969157675, -0.7499784895546662]
Bias: -0.1499413632617615

Época 3
Pesos: [0.3788535969157675, -0.6499784895546662]
Bias: -0.04994136326176149

Época 4
Pesos: [0.3788535969157675, -0.5499784895546662]
Bias: 0.050058636738238516

Época 5
Pesos: [0.3788535969157675, -0.44997848955466624]
Bias: 0.050058636738238516

Época 6
Pesos: [0.2788535969157675, -0.44997848955466624]
Bias: -0.04994136326176149

Época 7
Pesos: [0.2788535969157675, -0.34997848955466626]
Bias: 0.050058636738238516

Época 8
Pesos: [0.1788535969157675, -0.34997848955466626]
Bias: -0.04994136326176149

Época 9
Pesos: [0.1788535969157675, -0.24997848955466626]
Bias: 0.050058636738238516

Época 10
Pesos: [0.07885359691576749, -0.24997848955466626]
Bias: -0.04994136326176149

Época 11
Pesos: [0.07885359691576749, -0.14997848955466625]
Bias: 0.050058636738238516
```

```
Época 12
Pesos: [-0.021146403084232518, -0.14997848955466625]
Bias: -0.04994136326176149

Época 13
Pesos: [-0.021146403084232518, -0.049978489554666244]
Bias: 0.050058636738238516

Época 14
Pesos: [-0.12114640308423252, -0.049978489554666244]
Bias: -0.04994136326176149

Época 15
```

O problema do XOR não pode ser resolvido pelo perceptron porque este é um problema **não** linearmente separável.

Explicação do Código

- Objetivo

  A função `perceptron` implementa o algoritmo do perceptron simples, uma técnica clássica de aprendizado supervisionado usada para classificação binária linearmente separável.

- Etapas do Algoritmo

  1. **Inicialização**:
     - Gera pesos (`weights`) e bias (`b`) aleatórios entre -1 e 1.
     - O número de pesos corresponde ao número de atributos da entrada.
  2. **Treinamento**:
     - Para cada época:
       - Percorre todas as entradas do conjunto de dados.
       - Calcula a soma ponderada dos atributos mais o bias.
       - Aplica a **função de ativação degrau binária**: `1` se a soma for maior que 0, `0` caso contrário.
       - Compara a saída prevista com a saída esperada e calcula o erro.
       - Atualiza os pesos e o bias com base no erro e na taxa de aprendizado.
  3. **Condição de parada**:
     - Para se a saída prevista for igual à saída desejada para todos os exemplos, ou se atingir o número máximo de épocas.

2)

Backpropagation

```
import math
import random

# Função de ativação sigmoide e sua derivada
def sigmoid(x):
    return 1 / (1 + math.exp(-x))

def sigmoid_derivada(x):
    return x * (1 - x)  # x já é a saída da sigmoide

def inicializar_pesos(n_entrada, n_oculta, n_saida):
    random.seed(42)  # reprodutibilidade

    pesos_input_hidden = [[random.uniform(-1, 1) for _ in range(n_entrada)] for _
    bias_hidden = [random.uniform(-1, 1) for _ in range(n_oculta)]

    pesos_hidden_output = [random.uniform(-1, 1) for _ in range(n_oculta)]
    bias_output = random.uniform(-1, 1)

    return pesos_input_hidden, bias_hidden, pesos_hidden_output, bias_output

def mlp_backprop(x, y, n_oculta=2, epochs=10000, lr=0.5):
    n_entrada = len(x[0])
    n_saida = 1

    # Inicialização dos pesos
    w_ih, b_h, w_ho, b_o = inicializar_pesos(n_entrada, n_oculta, n_saida)

    for epoca in range(epochs):
        for i in range(len(x)):
            # Forward Pass
            entrada = x[i]
            alvo = y[i]

            # Camada oculta
            soma_hidden = [sum(entrada[k] * w_ih[j][k] for k in range(n_entrada))
            saida_hidden = [sigmoid(v) for v in soma_hidden]

            # Camada de saída
            soma_output = sum(saida_hidden[j] * w_ho[j] for j in range(n_oculta))
            saida_output = sigmoid(soma_output)

            # Erro da saída
            erro_saida = alvo - saida_output
```

```
            delta_saida = erro_saida * sigmoid_derivada(saida_output)

            # Erro da camada oculta
            delta_hidden = [
                sigmoid_derivada(saida_hidden[j]) * w_ho[j] * delta_saida
                for j in range(n_oculta)
            ]

            # Atualização dos pesos (output)
            for j in range(n_oculta):
                w_ho[j] += lr * delta_saida * saida_hidden[j]
            b_o += lr * delta_saida

            # Atualização dos pesos (input -> hidden)
            for j in range(n_oculta):
                for k in range(n_entrada):
                    w_ih[j][k] += lr * delta_hidden[j] * entrada[k]
                b_h[j] += lr * delta_hidden[j]

        # Imprimir erro médio a cada 1000 épocas
        if epoca % 1000 == 0 or epoca == epochs - 1:
            saidas_pred = []
            for entrada in x:
                soma_h = [sum(entrada[k] * w_ih[j][k] for k in range(n_entrada))
                saida_h = [sigmoid(v) for v in soma_h]
                soma_o = sum(saida_h[j] * w_ho[j] for j in range(n_oculta)) + b_o
                saida_o = sigmoid(soma_o)
                saidas_pred.append(round(saida_o))
            print(f"Época {epoca + 1} - Saída prevista: {saidas_pred}")

    # Resultados finais
    print("\n--- Resultado Final ---")
    for i in range(len(x)):
        entrada = x[i]
        soma_h = [sum(entrada[k] * w_ih[j][k] for k in range(n_entrada)) + b_h[j]
        saida_h = [sigmoid(v) for v in soma_h]
        soma_o = sum(saida_h[j] * w_ho[j] for j in range(n_oculta)) + b_o
        saida_o = sigmoid(soma_o)
        print(f"Entrada: {entrada} -> Saída prevista: {round(saida_o)} (Esperado:
```

Resolvendo AND, OR e XOR

```
entradas = [[0, 0], [0, 1], [1, 0], [1, 1]]
```

AND

```
saidas_and = [0, 0, 0, 1]
mlp_backprop(entradas, saidas_and, n_oculta=2, epochs=10000, lr=0.5)
```

```
mlp_backprop(entradas, saidas_and, n_oculta=2, epochs=10000, lr=0.5)
```

```
Época 1 - Saída prevista: [0, 0, 0, 0]
Época 1001 - Saída prevista: [0, 0, 0, 1]
Época 2001 - Saída prevista: [0, 0, 0, 1]
Época 3001 - Saída prevista: [0, 0, 0, 1]
Época 4001 - Saída prevista: [0, 0, 0, 1]
Época 5001 - Saída prevista: [0, 0, 0, 1]
Época 6001 - Saída prevista: [0, 0, 0, 1]
Época 7001 - Saída prevista: [0, 0, 0, 1]
Época 8001 - Saída prevista: [0, 0, 0, 1]
Época 9001 - Saída prevista: [0, 0, 0, 1]
Época 10000 - Saída prevista: [0, 0, 0, 1]

--- Resultado Final ---
Entrada: [0, 0] -> Saída prevista: 0 (Esperado: 0)
Entrada: [0, 1] -> Saída prevista: 0 (Esperado: 0)
Entrada: [1, 0] -> Saída prevista: 0 (Esperado: 0)
Entrada: [1, 1] -> Saída prevista: 1 (Esperado: 1)
```

OR

```
saidas_or = [0, 1, 1, 1]
mlp_backprop(entradas, saidas_or, n_oculta=2, epochs=10000, lr=0.5)
```

```
Época 1 - Saída prevista: [1, 0, 1, 1]
Época 1001 - Saída prevista: [0, 1, 1, 1]
Época 2001 - Saída prevista: [0, 1, 1, 1]
Época 3001 - Saída prevista: [0, 1, 1, 1]
Época 4001 - Saída prevista: [0, 1, 1, 1]
Época 5001 - Saída prevista: [0, 1, 1, 1]
Época 6001 - Saída prevista: [0, 1, 1, 1]
Época 7001 - Saída prevista: [0, 1, 1, 1]
Época 8001 - Saída prevista: [0, 1, 1, 1]
Época 9001 - Saída prevista: [0, 1, 1, 1]
Época 10000 - Saída prevista: [0, 1, 1, 1]

--- Resultado Final ---
Entrada: [0, 0] -> Saída prevista: 0 (Esperado: 0)
Entrada: [0, 1] -> Saída prevista: 1 (Esperado: 1)
Entrada: [1, 0] -> Saída prevista: 1 (Esperado: 1)
Entrada: [1, 1] -> Saída prevista: 1 (Esperado: 1)
```

XOR

```
saidas_xor = [0, 1, 1, 0]
mlp_backprop(entradas, saidas_xor, n_oculta=2, epochs=10000, lr=0.5)
```

```
Época 1 - Saída prevista: [0, 0, 1, 0]
Época 1001 - Saída prevista: [0, 1, 1, 0]
Época 2001 - Saída prevista: [0, 1, 1, 0]
Época 3001 - Saída prevista: [0, 1, 1, 0]
```

```
Época 3001 – Saída prevista: [0, 1, 1, 0]
Época 4001 – Saída prevista: [0, 1, 1, 0]
Época 5001 – Saída prevista: [0, 1, 1, 0]
Época 6001 – Saída prevista: [0, 1, 1, 0]
Época 7001 – Saída prevista: [0, 1, 1, 0]
Época 8001 – Saída prevista: [0, 1, 1, 0]
Época 9001 – Saída prevista: [0, 1, 1, 0]
Época 10000 – Saída prevista: [0, 1, 1, 0]

--- Resultado Final ---
Entrada: [0, 0] -> Saída prevista: 0 (Esperado: 0)
Entrada: [0, 1] -> Saída prevista: 1 (Esperado: 1)
Entrada: [1, 0] -> Saída prevista: 1 (Esperado: 1)
Entrada: [1, 1] -> Saída prevista: 0 (Esperado: 0)
```

Investigando

1. Por que a taxa de aprendizado é tão importante?

A taxa de aprendizado é um hiperparâmetro essencial durante o treinamento de redes neurais. Ela determina o quão grande será o passo dado pelo otimizador (como o gradiente descendente) ao atualizar os pesos com base nos erros. • Se a taxa for muito baixa: • O treinamento será demorado, já que as atualizações nos pesos são mínimas. • Pode acabar preso em mínimos locais ou demorar tanto que não converge em tempo hábil. • Se a taxa for muito alta: • O modelo pode ultrapassar o ponto ótimo, causando oscilações nos resultados. • Em casos extremos, pode até divergir e se tornar instável.

———

2. Qual o papel do bias (viés)?

O bias é um termo adicional aplicado em cada neurônio. Ele desloca a função de ativação, dando mais flexibilidade à rede para ajustar os limites de decisão. • Sem o bias: • A saída depende apenas da combinação linear entre pesos e entradas. • A função de ativação sempre passa pela origem, o que limita a capacidade da rede de aprender padrões complexos. • Com o bias: • A rede pode representar funções mais complexas, mesmo com uma arquitetura simples. • Isso melhora a adaptação da rede a diferentes tipos de dados.

———

3. Por que a função de ativação é essencial?

A função de ativação introduz não-linearidade na rede, o que é crucial. Sem ela, mesmo redes profundas se comportariam como uma única função linear — incapazes de lidar com problemas como classificação não-linear ou funções como XOR.

Explicação do Código

O código implementa o treinamento de uma rede neural simples com uma camada oculta, utilizando o algoritmo de backpropagation. O objetivo é aprender a função lógica XOR, que não pode ser resolvida por um perceptron de camada única, exigindo ao menos uma camada oculta para capturar a não-linearidade do problema.

———

Estrutura da Rede Neural

A rede tem a seguinte configuração: 1. Camada de entrada com 2 neurônios, representando os valores binários de entrada (0 ou 1); 2. Camada oculta com 2 neurônios, responsável por aprender representações intermediárias; 3. Camada de saída com 1 neurônio, que fornece a saída final da rede.

———

Funcionamento do Algoritmo 1. Inicialização de pesos e biases: • Os pesos (w1, w2) e os biases (b1, b2) são atribuídos aleatoriamente, com valores entre -1 e 1. • O número de neurônios na camada oculta é fixado em 2. • Pode-se usar uma random_state para garantir reprodutibilidade nos testes. 2. Escolha da função de ativação: • É possível alternar entre sigmoide e tangente hiperbólica (tanh). • Ambas são funções contínuas, não-lineares e com derivadas bem definidas — características ideais para o backpropagation. 3. Treinamento (forward e backpropagation): • O processo é repetido por várias épocas, percorrendo cada exemplo da base XOR. Passo forward: • Calcula-se a combinação linear das entradas com os pesos da camada oculta, somando os biases. • Aplica-se a função de ativação para obter a saída da camada oculta. • Com a saída da camada oculta, realiza-se o mesmo processo para gerar a saída final da rede. Cálculo do erro e backpropagation: • Calcula-se a diferença entre a saída da rede e o valor esperado. • Esse erro é propagado de volta para ajustar os pesos da camada de saída e depois da oculta. • As atualizações são feitas com base no gradiente da função de ativação e escaladas pela taxa de aprendizado. 4. Teste após o treinamento: • Por fim, a rede é testada com todas as combinações de entrada da função XOR. • Os resultados são impressos, mostrando as previsões feitas pela rede.