

## ✓ Utils - Classe Para Gerar .gif

```
!pip install imageio
```

```
Requirement already satisfied: imageio in /usr/local/lib/python3.11/dist-packages (2.37.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages (from imageio) (2.0.2)
Requirement already satisfied: pillow>=8.3.2 in /usr/local/lib/python3.11/dist-packages (from imageio) (11.2.1)
```

```
!pip install matplotlib
```

```
Requirement already satisfied: matplotlib in /usr/local/lib/python3.11/dist-packages (3.10.0)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (1.3.2)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (4.58.0)
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (1.4.8)
Requirement already satisfied: numpy>=1.23 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (2.0.2)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (24.2)
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (11.2.1)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (3.2.3)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.11/dist-packages (from matplotlib) (2.9.0.post0)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages (from python-dateutil>=2.7->matplotlib) (1.17.0)
```

```
import os
import imageio.v2 as imageio
import numpy as np
import matplotlib.pyplot as plt

class GifLogger:
    def __init__(self, inputs, expected_outputs, perceptron_model,
                 x_margin=1, y_margin=1, frames_per_second=2):
        """
        Inicializa o logger para gerar GIFs durante o treinamento do perceptron.

        :param inputs: Conjunto de dados de entrada
        :param expected_outputs: Saídas esperadas (rótulos)
        :param perceptron_model: Instância do perceptron
        :param x_margin: Margem extra no eixo X para visualização
        :param y_margin: Margem extra no eixo Y
        :param frames_per_second: Taxa de quadros por segundo para o GIF
        """
        self.inputs = inputs
        self.labels = expected_outputs
        self.perceptron = perceptron_model
        self.frames_folder = "frames"
        self.gif_folder = "gifs"
        self.saved_frame_paths = []
        self.fps = frames_per_second

        input_array = np.array(inputs)
        self.x_min = input_array[:, 0].min() - x_margin
        self.x_max = input_array[:, 0].max() + x_margin
        self.y_min = input_array[:, 1].min() - y_margin
        self.y_max = input_array[:, 1].max() + y_margin

        os.makedirs(self.frames_folder, exist_ok=True)
        os.makedirs(self.gif_folder, exist_ok=True)
    # __init__

    def save_frame(self, epoch):
        plt.figure(figsize=(8, 6))

        for i, point in enumerate(self.inputs):
            color = 'red' if self.labels[i] == 0 else 'blue'
            label = f'Class {self.labels[i]}' if i == 0 else ""
            plt.scatter(point[0], point[1], color=color, label=label)

        x_values = np.linspace(self.x_min, self.x_max, 100)

        if abs(self.perceptron.weights[1]) < 1e-6:
            x_constant = -self.perceptron.bias_weight / (self.perceptron.weights[0] if abs(self.perceptron.weights[0]) > 1e-6 else 1e-6)
            plt.axvline(x=x_constant, color='green', label='Decision Boundary')
        else:
```

```

        slope = -self.perceptron.weights[0] / self.perceptron.weights[1]
        intercept = -self.perceptron.bias_weight / self.perceptron.weights[1]
        y_values = slope * x_values + intercept
        plt.plot(x_values, y_values, label='Decision Boundary', color='green')

    plt.title(f'Epoch {epoch + 1}')
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.legend()
    plt.grid(True)

    plt.xlim(self.x_min, self.x_max)
    plt.ylim(self.y_min, self.y_max)

    frame_path = os.path.join(self.frames_folder, f"frame_{epoch}.png")
    plt.savefig(frame_path)
    plt.close()

    self.saved_frame_paths.append(frame_path)
# save_frame

def generate_gif(self, output_filename="perceptron_training.gif"):
    output_path = os.path.join(self.gif_folder, output_filename)
    images = [imageio.imread(frame) for frame in self.saved_frame_paths]
    imageio.mimsave(output_path, images, fps=self.fps)

    for frame in self.saved_frame_paths:
        os.remove(frame)
    if os.path.exists(self.frames_folder):
        os.rmdir(self.frames_folder)

    print(f"GIF saved as {output_path}")
# generate_gif
# GifLogger

```

## ▼ Questão 1

---

### ▼ Implementação do Perceptron

```

import random
# from gif_generator import GifLogger

class Perceptron:
    def __init__(self, num_inputs: int, learning_rate: float = 0.1, random_seed: int = None) -> None:
        """
        Inicializa o perceptron com o número de entradas especificado.

        :param num_inputs: Número de entradas
        :param learning_rate: Taxa de aprendizado
        :param random_seed: Semente para geração de números aleatórios
        """
        if random_seed is not None:
            random.seed(random_seed)

        self.num_inputs = num_inputs
        self.learning_rate = learning_rate
        self.weights = [0] * num_inputs
        self.bias_weight = random.uniform(-1, 1)

        self._initialize_weights()
# __init__

    def _initialize_weights(self) -> None:
        """
        Inicializa os pesos com valores aleatórios entre -1 e 1.
        """
        for i in range(self.num_inputs):
            self.weights[i] = random.uniform(-1, 1)
# _initialize_weights

    def _activation_function(self, value: float) -> int:
        """
        Função de ativação do perceptron

        :param value: Valor de entrada
        :return: 1 se valor > 0, caso contrário 0
        """
        return 1 if value > 0 else 0

```

```

# _activation_function

def _compute_weighted_sum(self, input_vector) -> float:
    """
    Calcula a soma ponderada das entradas mais o bias.

    :param input_vector: Vetor de entrada
    :return: Soma ponderada
    """
    weighted_sum = sum(input_vector[i] * self.weights[i] for i in range(self.num_inputs))
    weighted_sum += self.bias_weight
    return weighted_sum
# _compute_weighted_sum

def _compute_error(self, expected: float, predicted: float) -> float:
    """
    Calcula o erro como a diferença entre esperado e previsto.

    :param expected: Valor esperado
    :param predicted: Valor previsto
    :return: Erro
    """
    return expected - predicted
# _compute_error

def _adjust_weights(self, input_vector, error: float) -> None:
    """
    Ajusta os pesos com base no erro.

    :param input_vector: Vetor de entrada
    :param error: Erro de previsão
    """
    for i in range(self.num_inputs):
        self.weights[i] += self.learning_rate * error * input_vector[i]
    self.bias_weight += self.learning_rate * error
# _adjust_weights

def _predict_single(self, input_vector) -> float:
    """
    Faz uma previsão para uma única entrada.

    :param input_vector: Vetor de entrada
    :return: Previsão (0 ou 1)
    """
    weighted_sum = self._compute_weighted_sum(input_vector)
    return self._activation_function(weighted_sum)
# _predict_single

def train(self, input_data, expected_outputs, num_epochs: int,
          verbose: bool = False,
          generate_gif: bool = False, gif_name: str = "perceptron_training.gif") -> list:
    """
    Treina o perceptron usando os dados fornecidos.

    :param input_data: Lista de vetores de entrada
    :param expected_outputs: Lista de saídas esperadas
    :param num_epochs: Número de épocas de treinamento
    :param verbose: Mostra informações do treinamento se True
    :param generate_gif: Gera gif do processo de treinamento se True
    :param gif_name: Nome do arquivo GIF
    :return: Instância do perceptron treinado
    """
    epoch = 0
    predictions = []
    logger = GifLogger(input_data, expected_outputs, self) if generate_gif else None

    while predictions != expected_outputs and epoch < num_epochs:
        predictions = []
        for i in range(len(input_data)):
            predicted = self._predict_single(input_data[i])
            error = self._compute_error(expected_outputs[i], predicted)
            predictions.append(predicted)

            if error != 0: # Atualiza apenas se houver erro
                self._adjust_weights(input_data[i], error)

        epoch += 1

    if verbose:
        print(f'\nÉpoca {epoch}')
        print("-" * 20)
        print(f'Pesos: {self.weights}')
        print(f'Peso do Bias: {self.bias_weight}')

```

```

        print(f'Saída esperada: {expected_outputs}')
        print(f'Saída prevista: {predictions}')

    if generate_gif:
        logger.save_frame(epoch)

    if generate_gif:
        logger.generate_gif(output_filename=gif_name)

    return self
# train
# Perceptron

```

## ✓ Explicação da Implementação

- A classe `Perceptron` implementa um modelo de rede neural bem simples, o perceptron de camada única. Ele é usado para resolver problemas de classificação binária, ou seja, onde o objetivo é decidir entre duas classes (por exemplo, "sim" ou "não").
- O perceptron tenta encontrar uma reta (ou uma linha de decisão) que separa corretamente os exemplos de uma classe dos exemplos da outra classe.

- Passo a Passo:

### 1. Inicialização (`__init__`)

- O construtor define:
  - Quantas entradas (inputs) o modelo deve esperar.
  - Qual será a taxa de aprendizado (ou seja, o quanto os pesos mudam a cada erro).
  - E se uma semente aleatória foi passada, ele usa isso para garantir que os resultados sejam reproduzíveis.
- Os pesos são números que representam a importância de cada entrada. Eles são iniciados com valores aleatórios.
- Também existe um bias — um peso extra que não depende das entradas, mas ajuda o modelo a ajustar melhor a linha de separação.

### 2. Função de ativação

- Ela é chamada sempre que o perceptron precisa tomar uma decisão.
- O modelo soma os produtos de cada entrada pelo seu peso, adiciona o bias e passa o resultado por uma função que devolve:
  - 1 se o valor for positivo,
  - 0 caso contrário.

### 3. Treinamento

- O perceptron é treinado comparando suas previsões com os resultados esperados.
- Se a previsão estiver errada, os pesos são atualizados de acordo com a diferença entre o valor previsto e o valor real.
- Esse processo se repete por várias "épocas" — rodadas completas sobre os dados de entrada.
- O treinamento para quando o perceptron acerta todos os exemplos ou quando atinge o número máximo de épocas.

### 4. Visualização (opcional)

- Se o parâmetro `generate_gif` for `True`, o perceptron vai capturar imagens do processo de treinamento para gerar um GIF ao final. Isso ajuda a visualizar como a linha de decisão evolui com o tempo.


## ✓ Resolvendo a Função AND

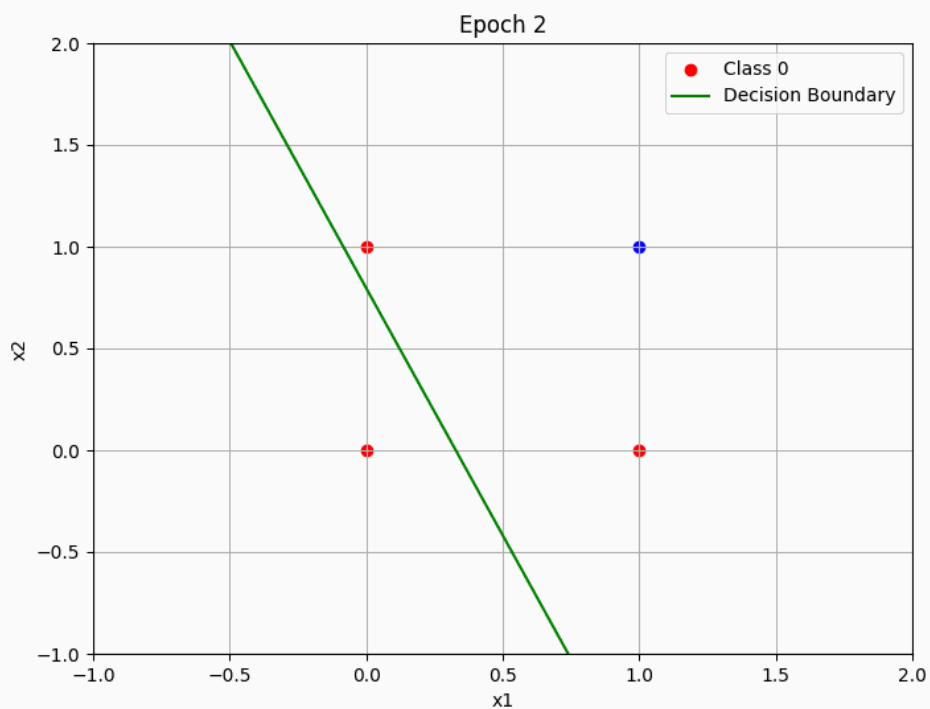
```

entradas = [[0, 0],
            [0, 1],
            [1, 0],
            [1, 1]]
saidas_esperadas = [0, 0, 0, 1]

perceptron_and = Perceptron(num_inputs=2, learning_rate=0.1, random_seed=42)
perceptron_and.train(entradas, saidas_esperadas, num_epochs=30, verbose=False, generate_gif=True, gif_name="AND_perceptron.gif")

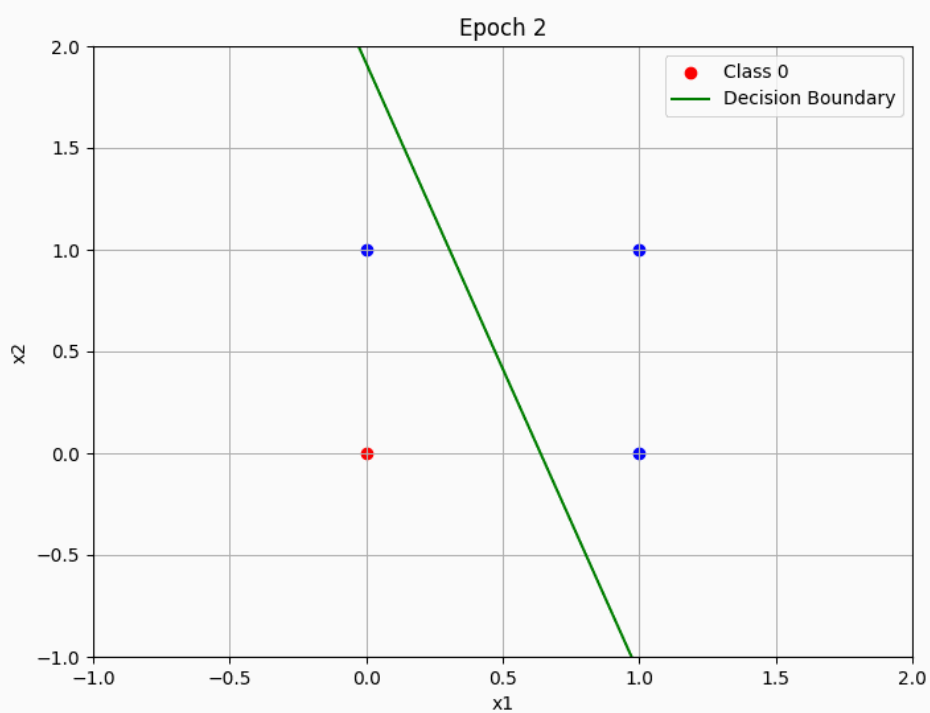
```

 GIF saved as gifs/AND\_perceptron.gif  
 <\_\_main\_\_.Perceptron at 0x7e3420de0550>



### Resolvendo a Função OR


```
entradas = [[0, 0],  
            [0, 1],  
            [1, 0],  
            [1, 1]]  
saidas_esperadas = [0, 1, 1, 1]  
  
perceptron_or = Perceptron(num_inputs=2, learning_rate=0.1, random_seed=42)  
perceptron_or.train(entradas, saidas_esperadas, num_epochs=30, verbose=False, generate_gif=True, gif_name="OR_perceptron.gif")  
  
GIF saved as gifs/OR_perceptron.gif  
<__main__.Perceptron at 0x7e34342e0dd0>
```

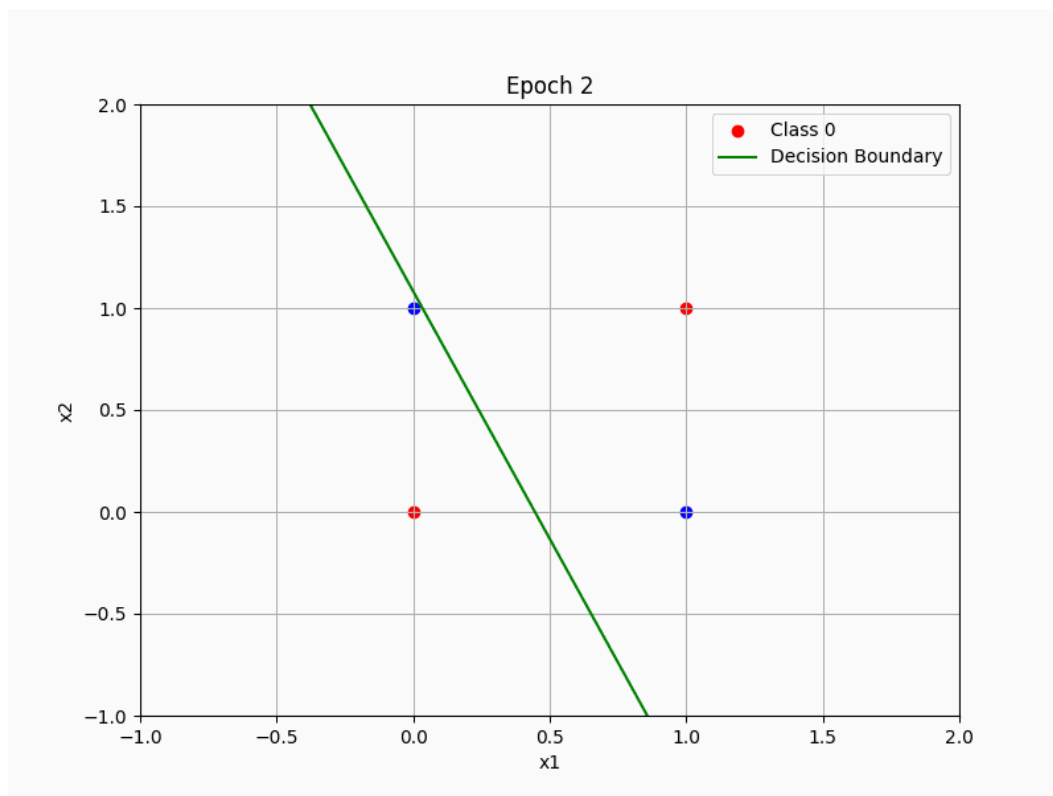


## ✓ Resolvendo a Função XOR

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

perceptron_xor = Perceptron(num_inputs=2, learning_rate=0.1, random_seed=42)
perceptron_xor.train(entradas, saidas_esperadas, num_epochs=30, verbose=False, generate_gif=True, gif_name="XOR_perceptron.gif")
```

 GIF saved as gifs/XOR\_perceptron.gif  
<\_\_main\_\_.Perceptron at 0x7e3420467610>



O Perceptron não resolve o XOR, pois não é um problema linearmente separável.

## ✓ Questão 02

### ✓ Implementação do BackPropagation

```
import numpy as np

class BackPropagation:
    def __init__(self, num_inputs, num_hidden_neurons, num_outputs,
                 learning_rate=0.1, activation_function='sigmoid', use_bias=True,
                 random_seed=None):
        """
        Inicializa a rede neural com os parâmetros fornecidos.
        :param num_inputs: Número de neurônios na camada de entrada
        :param num_hidden_neurons: Número de neurônios na camada oculta
        :param num_outputs: Número de neurônios na camada de saída
        :param learning_rate: Taxa de aprendizado para atualização dos pesos
        :param activation_function: Função de ativação a ser usada ('sigmoid', 'tanh', 'relu')
        :param use_bias: Se True, inclui bias nas camadas
        :param random_seed: Semente para reprodutibilidade (opcional)
        """
        if random_seed is not None:
            np.random.seed(random_seed)
        self.num_inputs = num_inputs
        self.num_hidden_neurons = num_hidden_neurons
        self.num_outputs = num_outputs
        self.learning_rate = learning_rate
        self.use_bias = use_bias
```

```

self.activation_function_name = activation_function

self._initialize_weights()

self.training_loss_history = []

self._initialize_activation_functions()
# __init__

def _initialize_weights(self):
    self.weights_input_to_hidden = np.random.uniform(-1, 1, (self.num_inputs, self.num_hidden_neurons))
    self.weights_hidden_to_output = np.random.uniform(-1, 1, (self.num_hidden_neurons, self.num_outputs))

    if self.use_bias:
        self.bias_hidden_layer = np.random.uniform(-1, 1, (1, self.num_hidden_neurons))
        self.bias_output_layer = np.random.uniform(-1, 1, (1, self.num_outputs))
    else:
        self.bias_hidden_layer = np.zeros((1, self.num_hidden_neurons))
        self.bias_output_layer = np.zeros((1, self.num_outputs))
# _initialize_weights

def _initialize_activation_functions(self):
    if self.activation_function_name == 'sigmoid':
        self.activation = self._sigmoid
        self.activation_derivative = self._sigmoid_derivative
    elif self.activation_function_name == 'tanh':
        self.activation = self._tanh
        self.activation_derivative = self._tanh_derivative
    elif self.activation_function_name == 'relu':
        self.activation = self._relu
        self.activation_derivative = self._relu_derivative
    else:
        raise ValueError(f"Função de ativação '{self.activation_function_name}' não suportada.")
# _initialize_activation_functions

def _sigmoid(self, x):
    return 1 / (1 + np.exp(-np.clip(x, -500, 500)))

def _sigmoid_derivative(self, x):
    sig = self._sigmoid(x)
    return sig * (1 - sig)

def _tanh(self, x):
    return np.tanh(x)

def _tanh_derivative(self, x):
    return 1 - np.tanh(x) ** 2

def _relu(self, x):
    return np.maximum(0, x)

def _relu_derivative(self, x):
    return (x > 0).astype(float)

def _compute_loss(self, predictions, expected_output):
    return np.mean((predictions - expected_output) ** 2)

def _compute_gradients(self, input_data, expected_output):
    """Calcula os gradientes para atualização dos pesos"""
    num_samples = input_data.shape[0]

    output_error = self.predicted_output - expected_output
    output_gradient = output_error # derivada da MSE * sigmoide já aplicada na saída

    grad_weights_hidden_output = (1 / num_samples) * np.dot(self.hidden_output.T, output_gradient)
    grad_bias_output = (1 / num_samples) * np.sum(output_gradient, axis=0, keepdims=True)

    hidden_error = np.dot(output_gradient, self.weights_hidden_to_output.T)
    hidden_gradient = hidden_error * self.activation_derivative(self.hidden_input)

    grad_weights_input_hidden = (1 / num_samples) * np.dot(input_data.T, hidden_gradient)
    grad_bias_hidden = (1 / num_samples) * np.sum(hidden_gradient, axis=0, keepdims=True)

    return grad_weights_input_hidden, grad_bias_hidden, grad_weights_hidden_output, grad_bias_output
# _compute_gradients

def _adjust_weights(self, grad_w_ih, grad_b_h, grad_w_ho, grad_b_o):
    """
    Ajusta os pesos e bias da rede neural usando os gradientes calculados.
    :param grad_w_ih: Gradiente dos pesos da camada de entrada para a camada oculta
    :param grad_b_h: Gradiente do bias da camada oculta
    :param grad_w_ho: Gradiente dos pesos da camada oculta para a camada de saída
    :param grad_b_o: Gradiente do bias da camada de saída
    """

```

```

self.weights_input_to_hidden -= self.learning_rate * grad_w_ih
self.weights_hidden_to_output -= self.learning_rate * grad_w_ho

if self.use_bias:
    self.bias_hidden_layer -= self.learning_rate * grad_b_h
    self.bias_output_layer -= self.learning_rate * grad_b_o
# _adjust_weights

def _forward_pass(self, input_data):
    """
    Executa a passagem direta pela rede neural.
    :param input_data: Dados de entrada (numpy array)
    :return: Saída prevista pela rede neural
    """
    self.hidden_input = np.dot(input_data, self.weights_input_to_hidden) + self.bias_hidden_layer
    self.hidden_output = self.activation(self.hidden_input)

    self.output_input = np.dot(self.hidden_output, self.weights_hidden_to_output) + self.bias_output_layer
    self.predicted_output = self._sigmoid(self.output_input)

    return self.predicted_output
# _forward_pass

def _backward_pass(self, input_data, expected_output):
    """
    Executa a passagem reversa para calcular os gradientes e atualizar os pesos.
    :param input_data: Dados de entrada (numpy array)
    :param expected_output: Saída esperada (numpy array)
    """
    gradients = self._compute_gradients(input_data, expected_output)
    self._adjust_weights(*gradients)
# _backward_pass

def train(self, input_data, expected_output, num_epochs=1000, verbose=False):
    """
    Treina a rede neural usando o algoritmo de retropropagação.
    :param input_data: Dados de entrada (numpy array)
    :param expected_output: Saída esperada (numpy array)
    :param num_epochs: Número de épocas para treinamento
    :param verbose: Se True, imprime o progresso do treinamento
    """
    for epoch in range(num_epochs):
        predictions = self._forward_pass(input_data)
        loss = self._compute_loss(predictions, expected_output)
        self.training_loss_history.append(loss)

        self._backward_pass(input_data, expected_output)

        if verbose and epoch % 100 == 0:
            print(f"Época {epoch} | Erro: {loss:.6f}")
            # print("-" * 50)
            # print(f"Pesos entrada-oculta:\n{self.weights_input_to_hidden}")
            # print(f"Pesos oculta-saída:\n{self.weights_hidden_to_output}")
            # print(f"Bias oculta:\n{self.bias_hidden_layer}")
            # print(f"Bias saída:\n{self.bias_output_layer}")

    if verbose:
        print(f"Treinamento concluído. Erro final: {self.training_loss_history[-1]:.6f}")
# train

def predict(self, input_data):
    """
    Faz previsões com a rede neural treinada.
    :param input_data: Dados de entrada (numpy array)
    :return: Previsões (numpy array)
    """
    predictions = self._forward_pass(input_data)
    return (predictions > 0.5).astype(int)
# predict
# BackPropagation

```

## ✎ Explicação da Implementação

- Visão Geral

O código implementa uma rede neural artificial simples, com uma camada oculta, função de ativação configurável e treinamento via backpropagation com erro quadrático médio.

- Estrutura da Rede

- Camada de entrada: recebe os dados (num\_inputs).



- Camada oculta: processa sinais com pesos ajustáveis e função de ativação (`num_hidden_neurons`).
- Camada de saída: produz a saída final com ativação sigmoide (ideal para problemas de classificação binária).
- Componentes Importantes
  - Pesos: `weights_input_to_hidden` e `weights_hidden_to_output` são matrizes que conectam os neurônios entre camadas.
  - Biases: valores adicionados ao cálculo de cada neurônio, opcionais.
  - Funções de ativação: controlam a não-linearidade da rede. Suporta `sigmoid`, `tanh` e `relu`.
- Fluxo do Treinamento
  1. Forward Pass: propaga os sinais da entrada até a saída, gerando uma predição.
  2. Erro: calcula a diferença entre a predição e o valor esperado.
  3. Backward Pass (Backpropagation): calcula os gradientes e ajusta os pesos para minimizar o erro.
  4. Repetição: isso é feito por várias épocas até o erro diminuir.

## ✓ Resolvendo a Função AND

```
X_and = np.array([[0, 0],
                  [0, 1],
                  [1, 0],
                  [1, 1]])
y_and = np.array([[0], [0], [0], [1]])

bp_and = BackPropagation(num_inputs=2, num_hidden_neurons=2, num_outputs=1,
                          learning_rate=0.1, activation_function='sigmoid', use_bias=True, random_seed=42)
bp_and.train(X_and, y_and, num_epochs=10000, verbose=True)

predictions_and = bp_and.predict(X_and)
print("Previsões AND:")
print(predictions_and)
```

↻ Época 4800 | Erro: 0.000461

```

época 9700 | Erro: 0.000063
Época 9800 | Erro: 0.000062
Época 9900 | Erro: 0.000060
Treinamento concluído. Erro final: 0.000059
Previsões AND:
[[0]
 [0]
 [0]
 [1]]

```

## Resolvendo a Função OR

```

X_or = np.array([[0, 0],
                 [0, 1],
                 [1, 0],
                 [1, 1]])
y_or = np.array([[0], [1], [1], [1]])

bp_or = BackPropagation(num_inputs=2, num_hidden_neurons=2, num_outputs=1,
                        learning_rate=0.1, activation_function='sigmoid', use_bias=True, random_seed=42)
bp_or.train(X_or, y_or, num_epochs=10000, verbose=True)

predictions_or = bp_or.predict(X_or)
print("Previsões OR:")
print(predictions_or)

```

```

↔ Época 4800 | Erro: 0.000086
Época 4900 | Erro: 0.000081
Época 5000 | Erro: 0.000077
Época 5100 | Erro: 0.000073
Época 5200 | Erro: 0.000069
Época 5300 | Erro: 0.000065
Época 5400 | Erro: 0.000062
Época 5500 | Erro: 0.000059
Época 5600 | Erro: 0.000056
Época 5700 | Erro: 0.000053
Época 5800 | Erro: 0.000051
Época 5900 | Erro: 0.000049
Época 6000 | Erro: 0.000046
Época 6100 | Erro: 0.000044
Época 6200 | Erro: 0.000042
Época 6300 | Erro: 0.000041
Época 6400 | Erro: 0.000039
Época 6500 | Erro: 0.000037
Época 6600 | Erro: 0.000036
Época 6700 | Erro: 0.000035
Época 6800 | Erro: 0.000033
Época 6900 | Erro: 0.000032
Época 7000 | Erro: 0.000031
Época 7100 | Erro: 0.000030
Época 7200 | Erro: 0.000029
Época 7300 | Erro: 0.000028
Época 7400 | Erro: 0.000027
Época 7500 | Erro: 0.000026
Época 7600 | Erro: 0.000025
Época 7700 | Erro: 0.000024
Época 7800 | Erro: 0.000023
Época 7900 | Erro: 0.000023
Época 8000 | Erro: 0.000022
Época 8100 | Erro: 0.000021
Época 8200 | Erro: 0.000020
Época 8300 | Erro: 0.000020
Época 8400 | Erro: 0.000019
Época 8500 | Erro: 0.000019
Época 8600 | Erro: 0.000018
Época 8700 | Erro: 0.000018
Época 8800 | Erro: 0.000017
Época 8900 | Erro: 0.000017
Época 9000 | Erro: 0.000016
Época 9100 | Erro: 0.000016
Época 9200 | Erro: 0.000015
Época 9300 | Erro: 0.000015
Época 9400 | Erro: 0.000015
Época 9500 | Erro: 0.000014
Época 9600 | Erro: 0.000014
Época 9700 | Erro: 0.000013
Época 9800 | Erro: 0.000013
Época 9900 | Erro: 0.000013
Treinamento concluído. Erro final: 0.000012
Previsões OR:
[[0]
 [1]
 [1]
 [1]]

```

## Resolvendo a Função XOR

```
X = np.array([[0, 0],
              [0, 1],
              [1, 0],
              [1, 1]])
y = np.array([[0], [1], [1], [0]])

bp = BackPropagation(num_inputs=2, num_hidden_neurons=2, num_outputs=1,
                    learning_rate=0.1, activation_function='sigmoid', use_bias=True, random_seed=42)
bp.train(X, y, num_epochs=10000, verbose=True)

predictions = bp.predict(X)
print("Previsões:")
print(predictions)
```

```
↳ Época 4800 | Erro: 0.163741
Época 4900 | Erro: 0.157625
Época 5000 | Erro: 0.150277
Época 5100 | Erro: 0.141440
Época 5200 | Erro: 0.130919
Época 5300 | Erro: 0.118748
Época 5400 | Erro: 0.105354
Época 5500 | Erro: 0.091553
Época 5600 | Erro: 0.078284
Época 5700 | Erro: 0.066288
Época 5800 | Erro: 0.055935
Época 5900 | Erro: 0.047267
Época 6000 | Erro: 0.040128
Época 6100 | Erro: 0.034285
Época 6200 | Erro: 0.029503
Época 6300 | Erro: 0.025572
Época 6400 | Erro: 0.022320
Época 6500 | Erro: 0.019611
Época 6600 | Erro: 0.017338
Época 6700 | Erro: 0.015416
Época 6800 | Erro: 0.013781
Época 6900 | Erro: 0.012379
Época 7000 | Erro: 0.011171
Época 7100 | Erro: 0.010123
Época 7200 | Erro: 0.009210
Época 7300 | Erro: 0.008409
Época 7400 | Erro: 0.007704
Época 7500 | Erro: 0.007081
Época 7600 | Erro: 0.006527
Época 7700 | Erro: 0.006034
Época 7800 | Erro: 0.005592
Época 7900 | Erro: 0.005195
Época 8000 | Erro: 0.004838
Época 8100 | Erro: 0.004515
Época 8200 | Erro: 0.004222
Época 8300 | Erro: 0.003956
Época 8400 | Erro: 0.003714
Época 8500 | Erro: 0.003492
Época 8600 | Erro: 0.003289
Época 8700 | Erro: 0.003103
Época 8800 | Erro: 0.002932
Época 8900 | Erro: 0.002774
Época 9000 | Erro: 0.002628
Época 9100 | Erro: 0.002493
Época 9200 | Erro: 0.002368
Época 9300 | Erro: 0.002252
Época 9400 | Erro: 0.002144
Época 9500 | Erro: 0.002044
Época 9600 | Erro: 0.001950
Época 9700 | Erro: 0.001862
Época 9800 | Erro: 0.001780
Época 9900 | Erro: 0.001703
Treinamento concluído. Erro final: 0.001632
Previsões:
[[0]
 [1]
 [1]
 [0]]
```

## ▼ Investigando Tópicos

### 1. A importância da taxa de aprendizado (learning rate)

A taxa de aprendizado (learning rate) é um hiperparâmetro fundamental no treinamento de redes neurais. Ela define o tamanho do passo que o algoritmo de otimização (como o gradiente descendente) dá ao ajustar os pesos durante o processo de backpropagation.

- Se for muito pequena:
  1. O treinamento será extremamente lento, pois os pesos mudam muito pouco a cada iteração.
  2. Pode ficar preso em mínimos locais ou até não convergir em um número razoável de épocas.

- Se for muito grande:
    1. A rede pode “saltar” o ótimo local, oscilando ou divergir, tornando o modelo instável.
- 

## 2. A importância do bias

O bias (viés) é um parâmetro adicional adicionado a cada neurônio que desloca a função de ativação, permitindo que o neurônio tenha maior flexibilidade na modelagem das fronteiras de decisão.

- Sem bias:
    1. A saída do neurônio depende exclusivamente da combinação linear dos pesos e entradas.
    2. A função de ativação sempre passará pela origem (0,0), o que limita a capacidade de aprendizado da rede.
  - Com bias:
    1. Permite que a rede aprenda funções mais complexas e não-linearidades, mesmo com poucos neurônios.
    2. A rede consegue se ajustar melhor a diferentes distribuições dos dados.
- 

## 3. A importância da função de ativação

A função de ativação é o componente que traz não-linearidade à rede. Sem ela, mesmo com várias camadas, a rede neural se comportaria como uma função linear composta, ou seja, incapaz de resolver problemas como XOR ou classificação não-linear.