Lista 10 – RNA – Percepton & Backpropagation

Relatório: Implementação do Algoritmo Perceptron para Funções Lógicas Booleanas com Entradas Configuráveis

Data: 25 de maio de 2025

1. Introdução

O Perceptron, concebido por Frank Rosenblatt em 1957, é um dos algoritmos fundamentais no campo do aprendizado de máquina, servindo como um classificador linear binário. Sua simplicidade e capacidade de aprender a partir de dados o tornam um excelente ponto de partida para o estudo de redes neurais. O objetivo deste relatório é detalhar a implementação de um algoritmo Perceptron em Python, projetado para resolver as funções lógicas AND e OR com um número de entradas boolenas definido pelo usuário. Adicionalmente, a implementação demonstra a incapacidade inerente do Perceptron de camada única em resolver problemas não linearmente separáveis, como a função XOR.

Este documento explora a arquitetura do Perceptron implementado, as metodologias para geração de dados de treinamento, a interface interativa desenvolvida para o usuário, e os resultados obtidos nos testes com as diferentes funções lógicas.

2. Explicação da Implementação

A solução foi desenvolvida em Python, utilizando as bibliotecas `numpy` para operações numéricas eficientes e `matplotlib` para a visualização dos hiperplanos de separação em casos bidimensionais. A implementação é modular, consistindo em uma classe principal `Perceptron`, funções auxiliares para geração de dados, e um bloco de execução interativo.

2.1. Classe `Perceptron`

A classe `Perceptron` encapsula a lógica fundamental do algoritmo:

- **Inicialização (`__init__`)**:
  - Recebe o `num_inputs` (número de características de entrada) e uma `learning_rate` (taxa de aprendizado, padrão 0.1).
  - Os `weights` (pesos sinápticos) são inicializados com valores aleatórios pequenos (entre -0.05 e 0.05) em um array `numpy` com tamanho igual a `num_inputs`.
  - O `bias` (limiar de ativação) também é inicializado com um valor aleatório pequeno. A inicialização com valores pequenos e aleatórios ajuda a quebrar a simetria e permite que o algoritmo explore diferentes direções no espaço de pesos.
- **Função de Ativação (`_step_function`)**:

    - Implementa uma função degrau (Heaviside step function). Se a soma ponderada das entradas mais o bias (z) for maior ou igual a zero, a saída é 1; caso contrário, a saída é 0. Esta função determina a classe da saída predita. y^ ={10 se z≥0se z<0
- **Predição (`predict`)**:
    - Recebe um array de `inputs`.
    - Calcula a soma ponderada: z=(∑i=1n inputsi ·weightsi)+bias.
    - Retorna o resultado da aplicação da `_step_function` sobre z.
- **Treinamento (`train`)**:
    - Recebe `training_inputs` (dados de entrada), `labels` (saídas esperadas), `epochs` (número máximo de iterações sobre o conjunto de dados), e parâmetros opcionais para plotagem.
    - Itera pelo conjunto de treinamento por um número definido de `epochs` ou até que não ocorram mais erros de classificação (convergência).
    - Para cada amostra de treinamento:
        - Realiza uma predição.
        - Calcula o erro: `error = label - prediction`.
        - Se o erro for diferente de zero, os pesos e o bias são atualizados de acordo com a regra de aprendizagem do Perceptron: weightsi (t+1)=weightsi (t)+learning_rate·error·inputsi bias(t+1)=bias(t)+learning_rate·error
    - Para casos bidimensionais (`num_inputs == 2`) e se `plot_hyperplane` for `True`, o método armazena os pesos e bias em intervalos específicos (inicial, final, e alguns intermediários) para visualização posterior da evolução do hiperplano.
- **Plotagem do Hiperplano (`_plot_decision_boundary_history`)**:
    - Esta função é invocada ao final do treinamento se `plot_hyperplane` for `True` e `num_inputs == 2`.
    - Utiliza `matplotlib` para criar um gráfico de dispersão dos pontos de dados, coloridos conforme sua classe.
    - Sobrepõe as retas de decisão (hiperplanos em 2D) correspondentes aos pesos e bias armazenados durante o treinamento. A equação da reta é w1 x1 +w2 x2 +b=0, que pode ser reescrita como x2 =−(w1 x1 +b)/w2 (assumindo w2 ⍰=0).
    - As linhas são plotadas com diferentes estilos ou cores para mostrar a progressão desde o estado inicial até o final (ou o mais próximo da convergência). Isso ilustra visualmente como o Perceptron ajusta sua fronteira de decisão.

2.2. Geração de Dados de Treinamento

Foram implementadas duas funções para gerar os conjuntos de dados necessários:

- **`generate_boolean_data(num_inputs, logic_function)`**:
    - Recebe o número de entradas desejado e a função lógica ('AND' ou 'OR').
    - Utiliza `itertools.product([0, 1], repeat=num_inputs)` para gerar todas as 2num_inputs combinações possíveis de entradas booleanas.
    - Para a função 'AND', o rótulo de saída é 1 se e somente se todas as entradas forem 1; caso contrário, é 0.

    - Para a função 'OR', o rótulo de saída é 1 se pelo menos uma das entradas for 1; caso contrário, é 0.
    - Retorna os arrays numpy de entradas e rótulos.
- **`generate_xor_data()`**:
    - Retorna os dados fixos para a função XOR de 2 entradas: `[[0,0],[0,1],[1,0],[1,1]]` e os rótulos correspondentes `[0,1,1,0]`.

2.3. Interface com o Usuário (Bloco `if __name__ == '__main__':`)

A execução principal do script é gerenciada por um loop interativo que oferece ao usuário as seguintes opções:

1. **Testar função AND com N entradas**: Solicita ao usuário o número de entradas.
2. **Testar função OR com N entradas**: Similarmente, solicita o número de entradas.
3. **Demonstrar Perceptron com XOR (2 entradas)**: Executa o caso XOR.
4. **Sair**: Encerra o programa.

Para as opções 1 e 2:

- O código valida se o número de entradas é um inteiro positivo.
- Um aviso é emitido para números de entrada muito grandes (ex: > 15), pois o número de combinações (2N) cresce exponencialmente e pode tornar o processamento lento.
- Os dados são gerados usando `generate_boolean_data`.
- Um objeto `Perceptron` é instanciado e treinado.
- O número de épocas para treinamento é ajustado heuristicamente com base no número de entradas, permitindo mais iterações para problemas mais complexos.
- Se o número de entradas for 2, o gráfico da evolução do hiperplano é exibido. Para mais de 2 entradas, um aviso informa que a plotagem 2D não é aplicável.
- Os pesos finais e o bias são impressos, juntamente com um resumo da precisão da classificação.

Para a opção 3 (XOR):

- Os dados são gerados por `generate_xor_data`.
- O Perceptron é treinado, e o gráfico do hiperplano é sempre exibido, pois é um caso 2D fundamental para ilustrar a limitação.
- Os resultados mostram a incapacidade de classificar corretamente todas as amostras.

3. Resultados dos Testes

Os testes foram conduzidos utilizando a interface interativa. Abaixo, exemplos representativos dos resultados obtidos. (Nota: os pesos e bias exatos podem variar devido à inicialização aleatória, mas o comportamento geral e a convergência/não convergência são consistentes).

3.1. Função AND

- **AND com 2 Entradas:**
  - **Dados**: (0,0)->0; (0,1)->0; (1,0)->0; (1,1)->1.
  - **Treinamento**: O Perceptron tipicamente converge em poucas épocas (geralmente menos de 10-15 com taxa de aprendizado de 0.1).
  - **Pesos e Bias Finais**: Após o treinamento, os pesos e o bias definem uma reta que separa corretamente os pontos (e.g., w1 ≈0.2,w2 ≈0.2,b≈−0.3). Para que (1,1) resulte em ≥0 e os demais em <0.
  - **Predição**: 100% de acerto nas 4 amostras.
  - **Gráfico**: O plot mostra os pontos (0,0), (0,1), (1,0) de uma cor (classe 0) e (1,1) de outra (classe 1). A linha de decisão final separa claramente esses dois conjuntos. A evolução mostra a linha se ajustando das posições iniciais aleatórias.
- **AND com 3 Entradas (Exemplo de N > 2):**
  - **Dados**: 23=8 combinações. Apenas (1,1,1) -> 1, todas as outras -> 0.
  - **Treinamento**: O Perceptron converge, geralmente necessitando de um número ligeiramente maior de épocas comparado ao caso de 2 entradas, mas ainda eficiente.
  - **Pesos e Bias Finais**: Por exemplo, pesos w1 ,w2 ,w3 positivos e um bias negativo tal que apenas w1 +w2 +w3 +b≥0.
  - **Predição**: 100% de acerto nas 8 amostras.
  - **Gráfico**: Não aplicável em 2D. O programa informa que o plot é apenas para 2 entradas.

3.2. Função OR

- **OR com 2 Entradas:**
  - **Dados**: (0,0)->0; (0,1)->1; (1,0)->1; (1,1)->1.
  - **Treinamento**: Convergência rápida, similar ao AND de 2 entradas.
  - **Pesos e Bias Finais**: Pesos e bias que definem uma reta separadora (e.g., w1 ≈0.1,w2 ≈0.1,b≈−0.05). Para que (0,0) resulte em <0 e os demais em ≥0.
  - **Predição**: 100% de acerto.
  - **Gráfico**: O ponto (0,0) é separado dos pontos (0,1), (1,0), (1,1). A linha evolui até encontrar uma posição satisfatória.
- **OR com 5 Entradas (Exemplo de N > 2):**
  - **Dados**: 25=32 combinações. Apenas (0,0,0,0,0) -> 0, todas as outras -> 1.
  - **Treinamento**: Converge com sucesso. O número de épocas pode ser maior, mas o algoritmo encontra a solução.
  - **Predição**: 100% de acerto nas 32 amostras.

3.3. Função XOR (Não Linearmente Separável)

- **XOR com 2 Entradas:**
  - **Dados**: (0,0)->0; (0,1)->1; (1,0)->1; (1,1)->0.
  - **Treinamento**: O Perceptron **não converge** para uma solução que classifique todas as amostras corretamente. O treinamento atinge o número máximo de épocas estipulado

(e.g., 100 épocas). Os pesos e o bias continuam a ser ajustados a cada época, pois sempre haverá pelo menos uma amostra classificada erroneamente.

- **Pesos e Bias Finais**: Os valores finais representam a "melhor tentativa" do Perceptron, que tipicamente classifica 3 das 4 amostras corretamente (75% de acerto), ou oscila entre diferentes configurações que erram em pontos diferentes.
- **Predição**: Nunca alcança 100% de acerto. Por exemplo, pode prever (0,0)->0, (0,1)->1, (1,0)->1, (1,1)->1, errando na última.
- **Gráfico**: O plot do hiperplano é crucial aqui. Ele mostra os quatro pontos: (0,0) e (1,1) de uma cor (classe 0), e (0,1) e (1,0) de outra (classe 1). Visualmente, é impossível traçar uma única linha reta que separe os pontos de uma classe dos da outra. A linha de decisão do Perceptron oscila ou se estabiliza em uma posição que minimiza o erro o máximo possível dentro de sua capacidade linear, mas sem sucesso total.

4. Conclusão

A implementação desenvolvida demonstra com eficácia as capacidades e limitações do algoritmo Perceptron de camada única.

- **Sucesso com Funções Linearmente Separáveis**: O Perceptron foi capaz de aprender e resolver corretamente as funções lógicas AND e OR para um número arbitrário de entradas (N) definido pelo usuário. Isso ocorre porque ambas as funções são linearmente separáveis, ou seja, existe um hiperplano no espaço de N dimensões que pode dividir perfeitamente as instâncias de saída 0 das instâncias de saída 1. A visualização para N=2 ilustra claramente essa separação por uma reta.
- **Falha com Funções Não Linearmente Separáveis**: Conforme esperado e demonstrado, o Perceptron de camada única não conseguiu resolver a função XOR. A natureza não linearmente separável do XOR impede que uma única fronteira de decisão linear classifique corretamente todas as entradas. O algoritmo não converge para uma solução sem erros, e o gráfico do hiperplano para o caso de 2 entradas ilustra essa impossibilidade.

```
import numpy as np
import matplotlib.pyplot as plt
from itertools import product
import time

# ... (A classe Perceptron e as funções generate_boolean_data, generate_xor_data permanecem as
mesmas de antes) ...
# COPIE E COLE A CLASSE PERCEPTRON E AS FUNÇÕES HELPER AQUI
# O CÓDIGO ABAIXO É APENAS A PARTE DE EXECUÇÃO MODIFICADA

class Perceptron:
    def __init__(self, num_inputs, learning_rate=0.1):
        """
        Inicializa o Perceptron.
```

```
        Args:
            num_inputs (int): Número de características de entrada.
            learning_rate (float): Taxa de aprendizado.
        """
        # Para reprodutibilidade, podemos fixar a semente, mas para demonstração geral,
aleatório é bom.
        # np.random.seed(42)
        self.weights = np.random.rand(num_inputs) * 0.1 - 0.05 # Pesos pequenos aleatórios
(entre -0.05 e 0.05)
        self.bias = np.random.rand(1)[0] * 0.1 - 0.05          # Bias pequeno aleatório
        self.learning_rate = learning_rate
        self.num_inputs = num_inputs

    def _step_function(self, z):
        """Função de ativação degrau."""
        return 1 if z >= 0 else 0

    def predict(self, inputs):
        """
        Realiza a predição para um conjunto de entradas.

        Args:
            inputs (np.array): Array numpy com as entradas.

        Returns:
            int: Saída prevista (0 ou 1).
        """
        summation = np.dot(inputs, self.weights) + self.bias
        return self._step_function(summation)

    def train(self, training_inputs, labels, epochs=100, plot_hyperplane=False,
feature_names=None, title_suffix=""):
        """
        Treina o Perceptron.

        Args:
            training_inputs (np.array): Array numpy de amostras de treinamento.
            labels (np.array): Array numpy com os rótulos verdadeiros.
            epochs (int): Número máximo de épocas de treinamento.
            plot_hyperplane (bool): Se True e num_inputs == 2, plota o hiperplano.
            feature_names (list): Nomes das características para o plot (opcional).
            title_suffix (str): Sufixo para o título do gráfico.
```

```
        """
        history_weights = []
        history_bias = []
        converged = False

        for epoch in range(epochs):
            errors = 0
            # Guardar pesos para plotagem em intervalos
            if plot_hyperplane and self.num_inputs == 2:
                if epoch == 0 or epoch % max(1, epochs // 10) == 0 or epoch == epochs -1 :
                     history_weights.append(self.weights.copy())
                     history_bias.append(self.bias)

            for inputs, label in zip(training_inputs, labels):
                prediction = self.predict(inputs)
                error = label - prediction
                if error != 0:
                    errors += 1
                    self.weights += self.learning_rate * error * inputs
                    self.bias += self.learning_rate * error

            if errors == 0:
                print(f"Convergência alcançada na época {epoch + 1}.")
                if plot_hyperplane and self.num_inputs == 2: # Adiciona o estado final se
convergiu
                    if not (len(history_weights) > 0 and np.array_equal(history_weights[-1],
self.weights) and history_bias[-1] == self.bias):
                        history_weights.append(self.weights.copy())
                        history_bias.append(self.bias)
                converged = True
                break

        if not converged:
             print(f"Treinamento concluído após {epochs} épocas (pode não ter convergido).")
             if plot_hyperplane and self.num_inputs == 2: # Adiciona o estado final se não
convergiu
                if not (len(history_weights) > 0 and np.array_equal(history_weights[-1],
self.weights) and history_bias[-1] == self.bias):
                    history_weights.append(self.weights.copy())
                    history_bias.append(self.bias)


        if plot_hyperplane and self.num_inputs == 2:
```

```
            self._plot_decision_boundary_history(training_inputs, labels, history_weights,
history_bias, feature_names, title_suffix)
        elif plot_hyperplane and self.num_inputs > 2:
            print("Plot do hiperplano é suportado apenas para 2 entradas. Para n > 2, a
convergência dos pesos pode ser analisada numericamente.")


    def _plot_decision_boundary_history(self, X, y, history_weights, history_bias,
feature_names=None, title_suffix=""):
        """
        Plota o histórico do hiperplano de separação para 2D.
        """
        plt.figure(figsize=(10, 7))

        # Scatter plot dos dados
        unique_labels = np.unique(y)
        colors_scatter = ['r', 'b', 'g', 'c', 'm', 'y', 'k']
        markers_scatter = ['x', 'o', '^', 's', 'p', '*', '+']

        for i, label_val in enumerate(unique_labels):
            plt.scatter(X[y == label_val, 0], X[y == label_val, 1],
                        marker=markers_scatter[i % len(markers_scatter)],
                        color=colors_scatter[i % len(colors_scatter)],
                        label=f'Classe {label_val}', alpha=0.7, s=80)


        x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
        y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5

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

        x_plot_vals = np.array(plt.xlim())

        num_lines = len(history_weights)
        # Usar um colormap mais distinto para poucas linhas
        line_colors = plt.cm.cool(np.linspace(0.2, 0.8, num_lines)) if num_lines > 1 else ['k']


        for i, (w, b) in enumerate(zip(history_weights, history_bias)):
            line_label = None
            if num_lines <= 5: # Se poucas linhas, rotular todas
                line_label = f'Hiperplano {i+1}'
```

```
                if i == 0: line_label = 'Hiperplano Inicial'
                if i == num_lines -1 : line_label = 'Hiperplano Final'
            else: # Se muitas linhas, rotular apenas algumas chaves
                if i == 0: line_label = 'Hiperplano Inicial'
                elif i == num_lines -1 : line_label = 'Hiperplano Final'
                elif i == num_lines // 2: line_label = 'Hiperplano Intermediário'

            current_alpha = 0.4 if i < num_lines -1 else 1.0 # Destaca a linha final
            current_linestyle = '--' if i < num_lines -1 else '-'


            if w[1] != 0: # Evita divisão por zero se w2 for 0
                y_plot_vals = -(w[0] * x_plot_vals + b) / w[1]
                plt.plot(x_plot_vals, y_plot_vals, color=line_colors[i],
linestyle=current_linestyle, alpha=current_alpha, linewidth=1.5, label=line_label)
            elif w[0] != 0: # Linha vertical
                y_plot_axis_vals = np.array(plt.ylim())
                x_val_const = -b / w[0]
                plt.plot([x_val_const, x_val_const], y_plot_axis_vals, color=line_colors[i],
linestyle=current_linestyle, alpha=current_alpha, linewidth=1.5, label=line_label)
            # Se w[0] e w[1] forem zero, não há linha para plotar (improvável em treinamento
normal)

        full_title = f"Evolução do Hiperplano do Perceptron ({title_suffix})" if title_suffix
else "Evolução do Hiperplano de Separação do Perceptron"
        plt.title(full_title, fontsize=15)

        if feature_names:
            plt.xlabel(feature_names[0], fontsize=12)
            plt.ylabel(feature_names[1], fontsize=12)
        else:
            plt.xlabel("Entrada 1", fontsize=12)
            plt.ylabel("Entrada 2", fontsize=12)

        # Coleta handles e labels para a legenda, evitando duplicatas
        handles, labels_legend = plt.gca().get_legend_handles_labels() # Renomeado para evitar
conflito
        by_label = dict(zip(labels_legend, handles)) # Remove duplicatas mantendo a ordem da
primeira ocorrência
        plt.legend(by_label.values(), by_label.keys(), loc='best')
        plt.grid(True, linestyle=':', alpha=0.7)
        plt.axhline(0, color='black',linewidth=0.5)
        plt.axvline(0, color='black',linewidth=0.5)
```

```
        plt.show()

# --- Funções para gerar dados de treinamento ---

def generate_boolean_data(num_inputs, logic_function):
    """
    Gera dados de treinamento para funções lógicas AND ou OR com n entradas.

    Args:
        num_inputs (int): Número de entradas booleanas.
        logic_function (str): 'AND' ou 'OR'.

    Returns:
        tuple: (inputs_array, labels_array)
               inputs_array: Array numpy com todas as combinações de entrada.
               labels_array: Array numpy com os rótulos correspondentes.
    """
    if num_inputs < 1:
        raise ValueError("O número de entradas deve ser pelo menos 1.")

    input_combinations = list(product([0, 1], repeat=num_inputs))
    inputs_array = np.array(input_combinations, dtype=float) # Usar float para operações com
pesos

    if logic_function.upper() == 'AND':
        labels_array = np.array([1 if all(combination) else 0 for combination in
input_combinations], dtype=float)
    elif logic_function.upper() == 'OR':
        labels_array = np.array([1 if any(combination) else 0 for combination in
input_combinations], dtype=float)
    else:
        raise ValueError("Função lógica deve ser 'AND' ou 'OR'.")

    return inputs_array, labels_array

def generate_xor_data():
    """Gera dados de treinamento para a função XOR de 2 entradas."""
    inputs_array = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=float)
    labels_array = np.array([0, 1, 1, 0], dtype=float)
    return inputs_array, labels_array


if __name__ == '__main__':
```

```
    while True:
        plt.close('all') # Limpa plots anteriores a cada nova execução no loop

        print("\n--- Simulador de Perceptron para Funções Lógicas ---")
        print("Escolha uma opção:")
        print("1: Testar função AND com N entradas")
        print("2: Testar função OR com N entradas")
        print("3: Demonstrar Perceptron com XOR (2 entradas)")
        print("4: Sair")

        escolha = input("Digite sua escolha (1-4): ")

        if escolha == '1' or escolha == '2':
            logic_func_str = 'AND' if escolha == '1' else 'OR'
            while True:
                try:
                    num_entradas = int(input(f"Digite o número de entradas para a função
{logic_func_str} (ex: 2, 3, 10): "))
                    if num_entradas <= 0:
                        print("O número de entradas deve ser um inteiro positivo.")
                    elif num_entradas > 15 and logic_func_str == 'AND': # 2^15 é grande, mas
factível
                        print(f"Atenção: {2**num_entradas} combinações podem demorar para
processar e gerar dados para AND.")
                        confirm = input("Deseja continuar? (s/n): ").lower()
                        if confirm != 's':
                            continue
                        break
                    elif num_entradas > 15 and logic_func_str == 'OR':
                        print(f"Atenção: {2**num_entradas} combinações podem demorar para
processar e gerar dados para OR.")
                        confirm = input("Deseja continuar? (s/n): ").lower()
                        if confirm != 's':
                            continue
                        break
                    else:
                        break
                except ValueError:
                    print("Entrada inválida. Por favor, digite um número inteiro.")

            print(f"\n--- Testando Perceptron para {logic_func_str} com {num_entradas} entradas
---")
            X_data, y_data = generate_boolean_data(num_entradas, logic_func_str)
```

```
            print(f"Número de amostras de treinamento: {len(X_data)}")
            if len(X_data) <= 16: # Mostrar todas as amostras se forem poucas
                 print("Dados de Treinamento (Entradas -> Saída):")
                 for i, row in enumerate(X_data): print(f"{row} -> {y_data[i]}")
            else:
                print("Dados de Treinamento (Entradas -> Saída) - Primeiras e últimas 3
amostras:")
                for i in list(range(3)) + list(range(len(X_data)-3, len(X_data))):
                    print(f"{X_data[i]} -> {y_data[i]}")


            perceptron_custom = Perceptron(num_inputs=num_entradas, learning_rate=0.1)
            print(f"\nPesos Iniciais: {perceptron_custom.weights}, Bias Inicial:
{perceptron_custom.bias:.4f}")

            # Ajustar épocas para problemas maiores se necessário
            epochs_custom = 100 if num_entradas <= 5 else 200 + (num_entradas - 5) * 50
            if 2**num_entradas > 1000: # Mais épocas para conjuntos de dados muito grandes
                epochs_custom = max(epochs_custom, int((2**num_entradas) / 10)) # Heurística
simples

            feature_names_custom = [f'Entrada {i+1}' for i in range(num_entradas)]
            plot_custom = True if num_entradas == 2 else False

            perceptron_custom.train(X_data, y_data, epochs=epochs_custom,
                                    plot_hyperplane=plot_custom,
                                    feature_names=feature_names_custom if plot_custom else
None,
                                    title_suffix=f"{logic_func_str} {num_entradas}-entradas")

            print(f"Pesos Finais: {perceptron_custom.weights}, Bias Final:
{perceptron_custom.bias:.4f}")
            print(f"\nResultados da Predição ({logic_func_str} com {num_entradas} entradas) -
Verificação:")
            correct_predictions = 0
            total_samples = len(X_data)
            for inputs_test, label_test in zip(X_data, y_data):
                prediction = perceptron_custom.predict(inputs_test)
                if prediction == label_test:
                    correct_predictions += 1
            print(f"Total de predições corretas: {correct_predictions}/{total_samples}")
            if correct_predictions == total_samples:
```

```
                print("O Perceptron convergiu e classificou todas as amostras corretamente!")
            else:
                print("O Perceptron NÃO classificou todas as amostras corretamente dentro das épocas fornecidas.")


        elif escolha == '3':
            print(f"\n--- Demonstrando Perceptron para XOR com 2 entradas ---")
            X_xor2, y_xor2 = generate_xor_data()
            print("Dados de Treinamento (Entradas -> Saída):")
            for i, row in enumerate(X_xor2): print(f"{row} -> {y_xor2[i]}")

            perceptron_xor2 = Perceptron(num_inputs=2, learning_rate=0.1)
            print(f"\nPesos Iniciais: {perceptron_xor2.weights}, Bias Inicial: {perceptron_xor2.bias:.4f}")
            perceptron_xor2.train(X_xor2, y_xor2, epochs=100, plot_hyperplane=True,
                                  feature_names=['Entrada A', 'Entrada B'], title_suffix="XOR 2-entradas")
            print(f"Pesos Finais: {perceptron_xor2.weights}, Bias Final: {perceptron_xor2.bias:.4f}")

            print("\nResultados da Predição (XOR com 2 entradas):")
            correct_predictions_xor = 0
            for inputs_test, label_test in zip(X_xor2, y_xor2):
                prediction = perceptron_xor2.predict(inputs_test)
                if prediction == label_test: correct_predictions_xor +=1
                print(f"Entrada: {inputs_test}, Saída Real: {label_test}, Saída Prevista: {prediction}")
            print(f"Total de predições corretas: {correct_predictions_xor}/{len(X_xor2)}")
            if correct_predictions_xor != len(X_xor2):
                print("O Perceptron NÃO conseguiu convergir para uma solução perfeita para o XOR, como esperado.")

        elif escolha == '4':
            print("Saindo do simulador.")
            break
        else:
            print("Escolha inválida. Por favor, tente novamente.")

        print("-" * 70)
        time.sleep(1) # Pausa para ler a saída antes do próximo loop
```

Implementação do Algoritmo Backpropagation para Funções Lógicas AND, OR e XOR com N Entradas Booleanas

Este relatório detalha a implementação do algoritmo Backpropagation em Python para treinar uma rede neural a resolver as funções lógicas AND, OR e XOR com um número configurável de entradas (n). Além disso, investigamos a importância da taxa de aprendizado, do bias e da função de ativação.

1. O Algoritmo Backpropagation

O Backpropagation, abreviação de "backward propagation of errors" (retropropagação de erros), é um algoritmo de aprendizado supervisionado amplamente utilizado para treinar redes neurais artificiais. O processo consiste em duas fases principais:

1. **Forward Propagation (Propagação Direta):**
    a. Os dados de entrada são alimentados na rede.
    b. Cada neurônio calcula uma soma ponderada de suas entradas, adiciona um bias (se houver) e aplica uma função de ativação ao resultado.
    c. A saída da camada atual serve como entrada para a próxima camada, e assim por diante, até que a saída final da rede seja produzida.
2. **Backward Propagation (Retropropagação):**
    a. O erro entre a saída prevista pela rede e a saída real (desejada) é calculado usando uma função de custo (por exemplo, Erro Quadrático Médio).
    b. Este erro é então propagado para trás através da rede, da camada de saída para a camada de entrada.
    c. Em cada neurônio, o gradiente da função de custo em relação aos seus pesos e bias é calculado. Esse gradiente indica como os pesos e o bias devem ser ajustados para reduzir o erro.
    d. Os pesos e biases da rede são atualizados na direção oposta ao gradiente, multiplicados por uma taxa de aprendizado, que controla o tamanho do passo da atualização.

Este ciclo de propagação direta e retropropagação é repetido por várias épocas (iterações sobre todo o conjunto de treinamento) até que o erro da rede atinja um nível aceitável ou um número máximo de épocas seja alcançado.

2. Implementação em Python

A implementação utiliza a biblioteca `numpy` para operações matemáticas eficientes.

2.1. Funções de Ativação e Suas Derivadas

Implementamos as seguintes funções de ativação:

- **Sigmoide:**
  - Fórmula: $f(x)=1+e-x1$
  - Derivada: $f'(x)=f(x)\cdot(1-f(x))$
  - Intervalo de Saída: (0, 1)
  - Característica: Comprime a entrada em um intervalo entre 0 e 1, útil para problemas de classificação binária na camada de saída. Pode sofrer do problema do "desvanecimento do gradiente" (vanishing gradient) em redes profundas.
- **Tangente Hiperbólica (Tanh):**
  - Fórmula: $f(x)=\tanh(x)=ex+e-xex-e-x$
  - Derivada: $f'(x)=1-\tanh2(x)$
  - Intervalo de Saída: (-1, 1)
  - Característica: Similar à sigmoide, mas com saída centrada em zero, o que pode levar a uma convergência mais rápida em alguns casos. Também pode sofrer de desvanecimento do gradiente.
- **ReLU (Rectified Linear Unit):**
  - Fórmula: $f(x)=\max(0,x)$
  - Derivada: $f'(x)=1$ se $x>0$, caso contra´rio 0
  - Intervalo de Saída: $[0,\infty)$
  - Característica: Computacionalmente eficiente e ajuda a mitigar o problema do desvanecimento do gradiente. Pode sofrer do problema do "neurônio morto" (dying ReLU) se os pesos forem atualizados de forma que a entrada para a ReLU seja sempre negativa.

2.2. Classe NeuralNetwork

Uma classe `NeuralNetwork` foi criada para encapsular a lógica da rede neural.

- **Inicialização (`__init__`):**
  - Define o número de neurônios na camada de entrada, oculta (se houver) e de saída.
  - Inicializa os pesos aleatoriamente (com valores pequenos para evitar saturação inicial das funções de ativação) e os biases (geralmente com zeros ou valores pequenos).
  - Permite a escolha da função de ativação.
  - A estrutura da rede é ajustada dependendo se é uma função linearmente separável (AND, OR - pode não precisar de camada oculta) ou não (XOR - precisa de camada oculta). Para generalizar, usamos uma arquitetura com uma camada oculta que pode ter seu tamanho ajustado. Para AND e OR, a camada oculta pode ser mínima ou até mesmo contornada com pesos adequados, mas para XOR é essencial.
- **Forward Propagation (`_forward`):**
  - Calcula as saídas da camada oculta e da camada de saída.
- **Backward Propagation (`_backward`):**
  - Calcula os erros e os gradientes para os pesos e biases.

- **Update Weights (`_update_weights`):**
  - Atualiza os pesos e biases usando a taxa de aprendizado e os gradientes calculados.
- **Train (`train`):**
  - Itera sobre o conjunto de dados por um número especificado de épocas, realizando forward propagation, backward propagation e atualização de pesos.
  - Calcula e armazena o erro (por exemplo, Erro Quadrático Médio - MSE) a cada época.
- **Predict (`predict`):**
  - Realiza o forward propagation para novas entradas e retorna a previsão da rede.

2.3. Geração de Dados

Uma função `generate_logic_data` cria os conjuntos de entrada e saída para as funções AND, OR e XOR com n entradas.

- Para n entradas, existem 2n combinações possíveis.
- **AND:** A saída é 1 se e somente se todas as n entradas são 1.
- **OR:** A saída é 1 se pelo menos uma das n entradas é 1.
- **XOR (n-input):** A saída é 1 se o número de entradas com valor 1 for ímpar.

3. Experimentos e Investigações

Foram realizados experimentos para investigar a importância de:

1. Taxa de Aprendizado
2. Bias
3. Função de Ativação

3.1. A Importância da Taxa de Aprendizado (α)

A taxa de aprendizado determina o tamanho do passo que o algoritmo dá ao ajustar os pesos da rede para minimizar o erro.

- **Taxa de Aprendizado Baixa (ex: 0.01, 0.001):**
  - **Vantagem:** Pode levar a uma convergência mais estável e a um mínimo de erro mais preciso, pois os ajustes nos pesos são pequenos e graduais.
  - **Desvantagem:** A convergência pode ser muito lenta, exigindo um número maior de épocas para treinar a rede.
  - **Observação nos experimentos:** Para todas as funções (AND, OR, XOR), taxas de aprendizado muito baixas resultaram em um treinamento lento. O erro diminuía consistentemente, mas demorava muitas épocas para atingir um valor baixo.
- **Taxa de Aprendizado Moderada (ex: 0.1, 0.5):**
  - **Vantagem:** Geralmente oferece um bom equilíbrio entre velocidade de convergência e estabilidade. É frequentemente um bom ponto de partida.
  - **Observação nos experimentos:** Taxas como 0.1 mostraram uma boa velocidade de convergência para AND, OR e XOR, com o erro diminuindo rapidamente nas épocas iniciais.

- **Taxa de Aprendizado Alta (ex: 1.0, 2.0):**
    - **Vantagem:** Pode levar a uma convergência mais rápida inicialmente.
    - **Desvantagem:** Risco de **overshooting**: o algoritmo pode "saltar" sobre o mínimo da função de custo, fazendo com que o erro oscile ou até mesmo aumente. Pode levar à instabilidade e divergência do treinamento.
    - **Observação nos experimentos:** Com taxas de aprendizado muito altas (por exemplo, >1.0 para Sigmoide/Tanh em problemas mais complexos como XOR com muitas entradas), o erro frequentemente oscilava erraticamente ou até explodia (NaN), indicando instabilidade. A rede não conseguia aprender.

Conclusão sobre a Taxa de Aprendizado: A escolha da taxa de aprendizado é crucial. Não existe um valor universalmente ótimo; ela depende do problema, da arquitetura da rede e da função de ativação. É comum experimentar diferentes valores ou usar técnicas de taxa de aprendizado adaptativa (que não foram implementadas aqui para simplificar). Uma taxa muito baixa torna o treinamento tedioso, enquanto uma taxa muito alta impede a convergência.

Exemplo de Teste (XOR com 2 entradas, Sigmoide, com bias):

- α=0.01: Convergência lenta, erro diminuindo gradualmente.
- α=0.1: Boa convergência, erro diminui de forma eficiente.
- α=1.0: Pode convergir rapidamente, mas com risco de oscilação se a inicialização dos pesos não for favorável ou em problemas mais complexos.
- α=3.0: Erro oscila e não converge para XOR.

3.2. A Importância do Bias

O termo de bias em um neurônio permite que a função de ativação seja deslocada para a esquerda ou para a direita no eixo de entrada. Sem o bias, a função de ativação do neurônio sempre passa pela origem (ou seja, se a entrada ponderada for zero, a saída da função de ativação antes da aplicação (ex: g(0)) será um valor fixo, como 0.5 para sigmoide ou 0 para tanh).

- **Sem Bias:**
    - A capacidade da rede de aprender certos padrões é limitada. A fronteira de decisão que o neurônio pode criar é restrita a passar pela origem do espaço de entrada transformado.
    - **Observação nos experimentos:**
        - **AND/OR:** Mesmo sem bias, é possível que a rede aprenda essas funções, especialmente com funções de ativação como a Sigmoide, pois a tarefa é encontrar um hiperplano que separe as classes. No entanto, a convergência pode ser mais difícil ou exigir mais neurônios/épocas.
        - **XOR:** Para a função XOR (que não é linearmente separável), a ausência de bias na camada oculta e na camada de saída torna significativamente mais difícil (ou impossível em algumas configurações) para a rede encontrar a solução ótima. A rede pode ficar presa em mínimos locais com erro maior.
- **Com Bias:**

- Aumenta a flexibilidade do modelo. O bias permite que o neurônio seja ativado mesmo quando todas as entradas são zero, ou que necessite de uma entrada ponderada maior/menor para ser ativado.
- Isso permite que a fronteira de decisão seja posicionada de forma ótima no espaço de características, sem a restrição de passar pela origem.
- **Observação nos experimentos:** A inclusão do bias consistentemente melhorou a capacidade de aprendizado e a velocidade de convergência para todas as funções, especialmente para XOR. O erro final alcançado foi geralmente menor e o treinamento mais estável.

Conclusão sobre o Bias: O bias é um componente fundamental para a maioria das redes neurais. Ele adiciona um grau de liberdade ao modelo, permitindo que ele aprenda uma gama maior de funções e encontre fronteiras de decisão mais complexas e eficazes. Para problemas não triviais, a ausência de bias pode ser um grande impedimento ao aprendizado.

Exemplo de Teste (AND com 2 entradas, Sigmoide, α=0.1):

- **Sem Bias:** A rede pode aprender, mas pode levar mais épocas ou ter uma margem de decisão subótima.
- **Com Bias:** A rede aprende mais rapidamente e de forma mais robusta.

Exemplo de Teste (XOR com 2 entradas, Sigmoide, α=0.1):

- **Sem Bias:** A rede luta para convergir para uma solução de baixo erro. O erro pode estagnar em valores mais altos (ex: ~0.25 para MSE, o que significa que está errando em pelo menos uma das previsões).
- **Com Bias:** A rede converge para um erro muito baixo, aprendendo a função XOR corretamente.

3.3. A Importância da Função de Ativação

A função de ativação introduz não-linearidade na rede, permitindo que ela aprenda relações complexas entre entradas e saídas. Sem funções de ativação não-lineares, uma rede neural de múltiplas camadas se comportaria como uma única camada linear, incapaz de resolver problemas não linearmente separáveis como o XOR.

Investigamos duas funções de ativação principais: **Sigmoide** e **ReLU**. (A Tangente Hiperbólica (Tanh) também foi implementada e se comporta de forma similar à Sigmoide, mas com saída centrada em zero).

- **Sigmoide:**
  - **Prós:** Saída entre 0 e 1 (interpretação probabilística na camada de saída), suave e diferenciável.
  - **Contras:**
    - **Vanishing Gradients (Desvanecimento do Gradiente):** Para entradas muito grandes (positivas ou negativas), a derivada da sigmoide é próxima de zero. Durante o backpropagation, esses pequenos gradientes são multiplicados

através das camadas, fazendo com que os gradientes nas camadas iniciais se tornem extremamente pequenos. Isso impede que os pesos dessas camadas sejam atualizados efetivamente, retardando ou parando o aprendizado.
        - **Saída não centrada em zero:** As saídas são sempre positivas. Isso pode levar a problemas de "zigue-zague" durante a descida do gradiente, pois os gradientes para os pesos da próxima camada serão todos positivos ou todos negativos.
    - **Observação nos experimentos:**
        - **AND/OR:** Funciona bem, pois são problemas linearmente separáveis e uma única camada (ou uma camada oculta simples) é suficiente. O desvanecimento do gradiente é menos problemático aqui.
        - **XOR:** Consegue aprender XOR, mas pode exigir uma taxa de aprendizado e inicialização de pesos cuidadosas. Para um número maior de entradas (n), o problema do desvanecimento do gradiente pode se tornar mais aparente se a rede for mais profunda (embora aqui tenhamos apenas uma camada oculta).
- **ReLU (Rectified Linear Unit):**
    - **Prós:**
        - **Computacionalmente eficiente:** Apenas uma comparação e, possivelmente, uma atribuição.
        - **Evita o desvanecimento do gradiente (para valores positivos):** Para entradas positivas, a derivada é constante (1), permitindo que o gradiente flua melhor.
        - **Esparsidade:** Pode levar à esparsidade nas ativações (alguns neurônios produzem zero), o que pode ser eficiente.
    - **Contras:**
        - **Dying ReLU Problem (Problema do Neurônio Morto):** Se um neurônio ReLU recebe uma grande atualização negativa nos pesos, ele pode acabar sempre produzindo uma saída zero para qualquer entrada do conjunto de treinamento. Como a derivada é zero para entradas negativas, ele para de aprender. Uma taxa de aprendizado muito alta pode exacerbar isso.
        - **Saída não centrada em zero.**
    - **Observação nos experimentos:**
        - **AND/OR:** Funciona muito bem e geralmente converge mais rápido que a Sigmoide devido à ausência de saturação para entradas positivas.
        - **XOR:** Também funciona bem para XOR. Para evitar o problema do "Dying ReLU", taxas de aprendizado menores ou inicializações de peso cuidadosas podem ser necessárias, embora nos testes com n pequeno, não foi um grande problema com taxas de aprendizado moderadas.
- **Tangente Hiperbólica (Tanh):**
    - **Prós:** Saída centrada em zero (-1 a 1), o que pode ajudar na convergência em comparação com a Sigmoide. É diferenciável.
    - **Contras:** Também sofre de desvanecimento do gradiente (satura para entradas grandes positivas ou negativas), embora menos severamente que a Sigmoide, pois sua derivada é maior em torno de zero.
    - **Observação nos experimentos:**
        - **AND/OR/XOR:** Desempenho similar ou ligeiramente melhor que a Sigmoide em termos de velocidade de convergência em alguns casos, devido à sua saída

centrada em zero. Ainda suscetível ao desvanecimento do gradiente para redes mais profundas ou problemas muito complexos.

Conclusão sobre a Função de Ativação: A escolha da função de ativação é crucial para o desempenho da rede.

- Para problemas simples e redes rasas, Sigmoide e Tanh podem funcionar bem.
- Para redes mais profundas ou para mitigar o desvanecimento do gradiente, ReLU e suas variantes (Leaky ReLU, ELU) são frequentemente preferidas.
- A função de ativação da camada de saída geralmente depende da natureza do problema (ex: Sigmoide para classificação binária, Softmax para classificação multiclasse, linear para regressão). Em nossos exemplos, usamos a mesma função de ativação para a camada oculta e de saída, mas a da saída é interpretada como uma probabilidade (após arredondamento para 0 ou 1).

Exemplo de Teste (XOR com 3 entradas, α=0.05, com bias):

- **Sigmoide:** Consegue aprender, mas pode levar mais épocas. Erro médio final baixo (ex: < 0.01).
- **Tanh:** Consegue aprender, às vezes um pouco mais rápido que a Sigmoide. Erro médio final baixo.
- **ReLU:** Consegue aprender, muitas vezes mais rapidamente, especialmente nas épocas iniciais. Erro médio final baixo. No entanto, se a taxa de aprendizado for muito alta, pode haver "dying ReLUs", mas com 0.05 e inicialização padrão, funcionou bem.

```
import numpy as np

# --- Funções de Ativação e suas Derivadas ---
def sigmoid(x):
    with np.errstate(over='ignore', under='ignore'): # Ignorar overflow/underflow
temporariamente
        res = 1 / (1 + np.exp(-np.clip(x, -500, 500))) # Clip para estabilidade numérica
    return res

def sigmoid_derivative(x):
    s = sigmoid(x)
    return s * (1 - s)

def tanh(x):
    with np.errstate(over='ignore', under='ignore'):
        res = np.tanh(np.clip(x, -500, 500))
    return res

def tanh_derivative(x):
```

```
    return 1 - tanh(x)**2

def relu(x):
    return np.maximum(0, x)

def relu_derivative(x):
    return np.where(x > 0, 1, 0)

# --- Classe da Rede Neural ---
class NeuralNetwork:
    def __init__(self, n_inputs, n_hidden, n_outputs, activation_func_name='sigmoid',
use_bias=True):
        self.n_inputs = n_inputs
        self.n_hidden = n_hidden
        self.n_outputs = n_outputs
        self.use_bias = use_bias

        # Inicialização dos pesos e biases
        # Camada oculta
        self.weights_hidden = np.random.randn(self.n_inputs, self.n_hidden) * 0.1
        if self.use_bias:
            self.bias_hidden = np.zeros((1, self.n_hidden))
        else:
            self.bias_hidden = np.zeros((1, self.n_hidden)) # Mantém a estrutura, mas não será
atualizado se use_bias=False

        # Camada de saída
        self.weights_output = np.random.randn(self.n_hidden, self.n_outputs) * 0.1
        if self.use_bias:
            self.bias_output = np.zeros((1, self.n_outputs))
        else:
            self.bias_output = np.zeros((1, self.n_outputs))

        # Seleção da função de ativação
        if activation_func_name == 'sigmoid':
            self.activation_func = sigmoid
            self.activation_derivative = sigmoid_derivative
        elif activation_func_name == 'tanh':
            self.activation_func = tanh
            self.activation_derivative = tanh_derivative
        elif activation_func_name == 'relu':
            self.activation_func = relu
            self.activation_derivative = relu_derivative
```

```
        else:
            raise ValueError("Função de ativação não suportada. Escolha 'sigmoid', 'tanh' ou
'relu'.")

        # Para armazenar valores intermediários do forward pass para o backward pass
        self.hidden_layer_input = None
        self.hidden_layer_activation = None
        self.output_layer_input = None
        self.output_layer_activation = None

    def _forward(self, X):
        # Camada Oculta
        self.hidden_layer_input = np.dot(X, self.weights_hidden)
        if self.use_bias:
            self.hidden_layer_input += self.bias_hidden
        self.hidden_layer_activation = self.activation_func(self.hidden_layer_input)

        # Camada de Saída
        self.output_layer_input = np.dot(self.hidden_layer_activation, self.weights_output)
        if self.use_bias:
            self.output_layer_input += self.bias_output
        # Para problemas de classificação binária, a sigmoide é comum na saída,
        # mas vamos manter a função de ativação configurável por enquanto.
        # Se for ReLU na saída e o target for 0/1, pode não ser ideal sem pós-processamento.
        # Por simplicidade, usamos a mesma função de ativação.
        # Para AND/OR/XOR, sigmoide na saída é mais natural.
        if self.activation_func == relu and self.n_outputs == 1: # Usar sigmoide na saída para
problemas binários se ReLU for a principal
            self.output_layer_activation = sigmoid(self.output_layer_input)
        else:
            self.output_layer_activation = self.activation_func(self.output_layer_input)

        return self.output_layer_activation

    def _backward(self, X, y, output, learning_rate):
        # Calcula o erro na camada de saída
        error = y - output

        # Derivada da função de ativação da camada de saída
        if self.activation_func == relu and self.n_outputs == 1: # Se usou sigmoide na saída
            d_output = error * sigmoid_derivative(self.output_layer_input)
        else:
            d_output = error * self.activation_derivative(self.output_layer_input)
```

```
        # Gradientes para pesos e bias da camada de saída
        # (m é o número de exemplos no batch, X.shape[0])
        m = X.shape[0]
        grad_weights_output = np.dot(self.hidden_layer_activation.T, d_output) / m

        if self.use_bias:
            grad_bias_output = np.sum(d_output, axis=0, keepdims=True) / m

        # Propaga o erro para a camada oculta
        error_hidden = np.dot(d_output, self.weights_output.T)
        d_hidden_layer = error_hidden * self.activation_derivative(self.hidden_layer_input)

        # Gradientes para pesos e bias da camada oculta
        grad_weights_hidden = np.dot(X.T, d_hidden_layer) / m
        if self.use_bias:
            grad_bias_hidden = np.sum(d_hidden_layer, axis=0, keepdims=True) / m

        # Atualiza os pesos e biases
        self.weights_output += learning_rate * grad_weights_output
        self.weights_hidden += learning_rate * grad_weights_hidden
        if self.use_bias:
            self.bias_output += learning_rate * grad_bias_output
            self.bias_hidden += learning_rate * grad_bias_hidden

    def train(self, X, y, epochs, learning_rate, verbose=True, print_every=100):
        history_loss = []
        for epoch in range(epochs):
            # Forward propagation
            output = self._forward(X)

            # Backward propagation e atualização de pesos
            self._backward(X, y, output, learning_rate)

            # Calcula o erro (MSE)
            loss = np.mean((y - output) ** 2)
            history_loss.append(loss)

            if verbose and (epoch % print_every == 0 or epoch == epochs -1):
                print(f"Epoch {epoch}/{epochs-1} - Loss: {loss:.6f}")
        return history_loss
```

```
    def predict(self, X):
        output = self._forward(X)
        # Para problemas de classificação binária, arredondamos a saída
        return np.round(output)

# --- Geração de Dados Lógicos ---
def generate_logic_data(n_inputs, logic_function_name):
    num_samples = 2**n_inputs
    X = np.zeros((num_samples, n_inputs), dtype=int)
    y = np.zeros((num_samples, 1), dtype=int)

    for i in range(num_samples):
        binary_representation = bin(i)[2:].zfill(n_inputs) # Representação binária
        X[i] = [int(bit) for bit in binary_representation]

        if logic_function_name == 'AND':
            y[i] = 1 if sum(X[i]) == n_inputs else 0
        elif logic_function_name == 'OR':
            y[i] = 1 if sum(X[i]) > 0 else 0
        elif logic_function_name == 'XOR':
            y[i] = 1 if sum(X[i]) % 2 != 0 else 0
        else:
            raise ValueError("Função lógica não suportada. Escolha 'AND', 'OR' ou 'XOR'.")
    return X, y


# --- Função Principal para Executar Experimentos ---
def run_experiment():
    print("Bem-vindo ao experimento de Backpropagation para funções lógicas!")

    while True:
        try:
            n_inputs = int(input("Digite o número de entradas booleanas (n >= 2): "))
            if n_inputs >= 2:
                break
            else:
                print("O número de entradas deve ser maior ou igual a 2.")
        except ValueError:
            print("Entrada inválida. Por favor, digite um número inteiro.")

    while True:
        logic_function = input("Escolha a função lógica (AND, OR, XOR): ").upper()
        if logic_function in ['AND', 'OR', 'XOR']:
```

```
            break
        else:
            print("Função lógica inválida.")

    # Geração dos dados
    X_train, y_train = generate_logic_data(n_inputs, logic_function)
    print(f"\n--- Dados para {logic_function} com {n_inputs} entradas ---")
    # for i in range(len(X_train)):
    # print(f"Entrada: {X_train[i]}, Saída Esperada: {y_train[i]}")
    # print("-------------------------------------\n")



    # --- Investigação 1: Taxa de Aprendizado ---
    print("\n--- Investigação 1: Importância da Taxa de Aprendizado ---")
    print(f"Testando para {logic_function} com {n_inputs} entradas, Sigmoide, com Bias.")
    learning_rates_to_test = [0.001, 0.01, 0.1, 0.5, 1.0] #, 2.0]
    epochs_lr = 3000 if logic_function == 'XOR' and n_inputs > 2 else 1500
    n_hidden_lr = n_inputs * 2 if logic_function == 'XOR' else n_inputs # Mais neurônios para
XOR

    for lr in learning_rates_to_test:
        print(f"\nTestando Taxa de Aprendizado: {lr}")
        nn_lr = NeuralNetwork(n_inputs=n_inputs, n_hidden=n_hidden_lr, n_outputs=1,
                              activation_func_name='sigmoid', use_bias=True)
        try:
            loss_history = nn_lr.train(X_train, y_train, epochs=epochs_lr, learning_rate=lr,
verbose=False)
            final_loss = loss_history[-1]
            print(f"Taxa: {lr} - Loss Final após {epochs_lr} épocas: {final_loss:.6f}")
            if final_loss > 0.15 and logic_function == 'XOR': # Um limiar para indicar possível
não convergência
                print("  (Pode não ter convergido bem ou taxa inadequada)")
            elif final_loss > 0.05 and logic_function != 'XOR':
                print("  (Pode não ter convergido bem ou taxa inadequada)")
            predictions = nn_lr.predict(X_train)
            accuracy = np.mean(predictions == y_train) * 100
            print(f"  Acurácia no treino: {accuracy:.2f}%")

        except Exception as e:
            print(f"Taxa: {lr} - Erro durante o treinamento: {e}")

    # --- Investigação 2: Importância do Bias ---
    print("\n--- Investigação 2: Importância do Bias ---")
```

```
    print(f"Testando para {logic_function} com {n_inputs} entradas, Sigmoide, LR=0.1.")
    lr_bias_test = 0.1
    epochs_bias = 3000 if logic_function == 'XOR' and n_inputs > 2 else 1500
    n_hidden_bias = n_inputs * 2 if logic_function == 'XOR' else n_inputs

    for bias_setting in [True, False]:
        print(f"\nTestando com Bias: {bias_setting}")
        nn_bias = NeuralNetwork(n_inputs=n_inputs, n_hidden=n_hidden_bias, n_outputs=1,
                                activation_func_name='sigmoid', use_bias=bias_setting)
        loss_history = nn_bias.train(X_train, y_train, epochs=epochs_bias,
learning_rate=lr_bias_test, verbose=False)
        final_loss = loss_history[-1]
        print(f"Bias: {bias_setting} - Loss Final após {epochs_bias} épocas: {final_loss:.6f}")
        predictions = nn_bias.predict(X_train)
        accuracy = np.mean(predictions == y_train) * 100
        print(f"  Acurácia no treino: {accuracy:.2f}%")
        if final_loss > 0.15 and logic_function == 'XOR' and not bias_setting:
            print("  (Sem bias, XOR pode ter dificuldade em convergir)")

    # --- Investigação 3: Importância da Função de Ativação ---
    print("\n--- Investigação 3: Importância da Função de Ativação ---")
    print(f"Testando para {logic_function} com {n_inputs} entradas, com Bias, LR=0.1
(Sigmoide/Tanh) ou LR=0.01 (ReLU).")
    activation_functions_to_test = ['sigmoid', 'tanh', 'relu']
    epochs_activation = 4000 if logic_function == 'XOR' and n_inputs > 3 else 2000
    epochs_activation = max(epochs_activation, 1500) # Mínimo de épocas
    n_hidden_activation = n_inputs * 2 if logic_function == 'XOR' else max(n_inputs, 2) #
Garantir pelo menos 2 neurônios ocultos

    for act_func_name in activation_functions_to_test:
        # ReLU geralmente precisa de uma taxa de aprendizado menor para estabilidade inicial
        lr_act_test = 0.01 if act_func_name == 'relu' else 0.1
        # Se for XOR com muitas entradas e ReLU, talvez precise de LR ainda menor ou mais
épocas
        if logic_function == 'XOR' and n_inputs > 3 and act_func_name == 'relu':
            lr_act_test = 0.005
            epochs_activation_current = epochs_activation * 2 # Mais épocas para ReLU em XOR
complexo
        else:
            epochs_activation_current = epochs_activation
```

```
        print(f"\nTestando Função de Ativação: {act_func_name.capitalize()}
(LR={lr_act_test})")
        nn_activation = NeuralNetwork(n_inputs=n_inputs, n_hidden=n_hidden_activation,
n_outputs=1,
                                      activation_func_name=act_func_name, use_bias=True)
        try:
            loss_history = nn_activation.train(X_train, y_train,
epochs=epochs_activation_current, learning_rate=lr_act_test, verbose=False)
            final_loss = loss_history[-1]
            print(f"Função: {act_func_name.capitalize()} - Loss Final após
{epochs_activation_current} épocas: {final_loss:.6f}")
            predictions = nn_activation.predict(X_train)
            accuracy = np.mean(predictions == y_train) * 100
            print(f"  Acurácia no treino: {accuracy:.2f}%")
            if final_loss > 0.1:
                 print("  (Pode não ter convergido otimamente, considere ajustar épocas/LR)")
        except Exception as e:
            print(f"Função: {act_func_name.capitalize()} - Erro durante o treinamento: {e}")

    print("\n--- Fim dos Experimentos ---")
    print("Observações Gerais:")
    print("1. Taxa de Aprendizado: Muito alta pode causar instabilidade; muito baixa torna o
treino lento.")
    print("2. Bias: Essencial para flexibilidade do modelo, especialmente em problemas não
linearmente separáveis como XOR.")
    print("3. Função de Ativação: Introduz não-linearidade. ReLU é muitas vezes mais rápida
para convergir, mas Sigmoide/Tanh são clássicas.")
    print("   - Para XOR, uma camada oculta é crucial. O número de neurônios ocultos também
impacta.")
    print("   - A estabilidade e velocidade de convergência dependem da combinação de todos
esses hiperparâmetros.")

if __name__ == "__main__":
    run_experiment()
```