
[Link para os Códigos]

Exercício 1 - Implementação do Algoritmo Perceptron para Funções Lógicas AND, OR e XOR

Introdução

Neste projeto, implementamos um algoritmo Perceptron para resolver funções lógicas com **n entradas booleanas** — especificamente as funções **AND**, **OR** e **XOR**. O objetivo é permitir que o usuário escolha o número de entradas (por exemplo, 2, 3 ou 10) e a função lógica desejada para o treinamento do Perceptron.

Além disso, o código inclui a plotagem do hiperplano de decisão que o Perceptron aprende durante o treinamento, adaptando-se para visualizar dados 2D, 3D ou, para dimensões maiores, uma projeção em 2D usando PCA (Análise de Componentes Principais).

Também é mostrado, através dos testes, que o Perceptron é incapaz de aprender a função XOR, evidenciando uma limitação fundamental desse modelo.

Explicação do Código

1. Função `gerar_dados(n_entradas, tipo)`

- Gera todas as combinações possíveis de entradas booleanas para `n_entradas` usando o produto cartesiano (`itertools.product`).
- Cria o vetor de saída `y` de acordo com a função lógica escolhida:
  - **AND**: saída 1 se todas as entradas forem 1 (`np.all(x)`).
  - **OR**: saída 1 se pelo menos uma entrada for 1 (`np.any(x)`).
  - **XOR**: saída 1 se o número de entradas 1 for ímpar (`np.sum(x) % 2`).

Retorna as matrizes `X` (entradas) e `y` (saídas).

---

2. Função `treinar_perceptron(X, y, titulo, plot)`

- Treina um Perceptron (implementação da biblioteca `scikit-learn`) com os dados `X` e `y`.
- Exibe:
  - Os pesos e o bias aprendidos.
  - As saídas esperadas e previstas.
  - A acurácia do modelo (proporção de classificações corretas).
- Chama a função de plotagem do hiperplano de decisão.

---

3. Função `plotar_hiperplano(X, y, clf, titulo)`

- **Para 2 entradas:** plota o hiperplano como uma linha reta separando as classes.
- **Para 3 entradas:** plota o hiperplano como um plano em um gráfico 3D.
- **Para mais de 3 entradas:** aplica PCA para reduzir a dimensão para 2D e plota a projeção dos dados, permitindo visualizar a separação mesmo em espaços de alta dimensão.

Por que o Perceptron NÃO resolve o XOR?

O Perceptron é um modelo linear, ou seja, ele busca um **hiperplano linear** que consiga separar as classes em um espaço vetorial.

- As funções **AND** e **OR** são **linearmente separáveis**, ou seja, existe uma linha (ou plano, em mais dimensões) que pode separar perfeitamente as saídas 0 e 1.
- A função **XOR não é linearmente separável**. Isso significa que não há um único hiperplano que possa dividir os pontos da classe 1 dos da classe 0 sem erro.

Por isso:

- O Perceptron **consegue aprender AND e OR** perfeitamente, obtendo acurácia 1.0.
- Mas o Perceptron **falha ao tentar aprender XOR**, geralmente com acurácia em torno de 0.5, o que equivale a uma classificação aleatória.

Exemplo dos Resultados

Função	Entradas	Acurácia Esperada	Comentário
AND	2	1.0	Linearmente separável
AND	10	1.0	Linearmente separável em alta dimensão
OR	3	1.0	Linearmente separável
XOR	2	~0.5	Não linearmente separável; Perceptron não aprende

Considerações Finais

Este exercício mostra na prática:

- Como o Perceptron funciona para problemas de classificação linear.
- A importância da linearidade dos dados para que o Perceptron funcione.
- O limite do Perceptron para problemas não linearmente separáveis como o XOR.

Para resolver problemas como o XOR, é necessário usar modelos mais complexos, como redes neurais multicamadas (MLPs), que conseguem aprender fronteiras não lineares.

Exercício 2 - Implementação do Backpropagation

Implementação do Algoritmo Backpropagation para Portas Lógicas Booleanas

Este documento detalha a implementação do algoritmo Backpropagation para resolver as funções lógicas **AND**, **OR**, e **XOR** com n entradas booleanas. Além da implementação, investigamos a importância da taxa de aprendizado, do bias e de diferentes funções de ativação.

---

1. Explicação da Implementação

A rede neural implementada é uma **Rede Neural Multicamadas (MLP)** simples, composta por uma camada de entrada, uma camada oculta e uma camada de saída. Utilizamos `numpy` para as operações matemáticas eficientes.

1.1. Estrutura da Rede Neural (`NeuralNetwork` Class)

- **Inicialização (`__init__`)**:
    - Define o número de neurônios nas camadas de entrada, oculta e saída.
    - **Pesos e Biases**: Inicializados aleatoriamente com valores entre -1 e 1. Essa inicialização é crucial para quebrar a simetria e permitir que a rede aprenda. Cada conexão entre neurônios tem um peso associado, e cada neurônio (exceto os da camada de entrada) tem um bias.
    - **Função de Ativação**: A classe é configurada para usar uma função de ativação escolhida (Sigmoide, Tanh ou ReLU) e sua respectiva derivada.

- **Funções de Ativação**:
    - **Sigmoide**: $f(x) = \frac{1}{1 + e^{-x}}$. Comprime a entrada entre 0 e 1.
    - **Tangente Hiperbólica (Tanh)**: $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. Comprime a entrada entre -1 e 1.
    - **ReLU (Rectified Linear Unit)**: $f(x) = \max(0, x)$. Retorna 0 para entradas negativas e a própria entrada para valores positivos.
    - Para cada função de ativação, sua **derivada** é implementada, pois ela é fundamental para o cálculo dos gradientes no Backpropagation.

- **Propagação Direta (`feedforward`)**:
    1. Calcula a entrada líquida para a camada oculta (`net_h`) usando o produto escalar das entradas pelos pesos da camada de entrada-oculta, somando o bias.
    2. Aplica a função de ativação para obter a saída da camada oculta (`hidden_layer_output`).
    3. Repete os passos para a camada de saída, usando a saída da camada oculta como entrada.
    4. Retorna a saída prevista da rede.

- **Backpropagation (`backpropagate`)**:
    1. **Cálculo do Erro**: Determina a diferença entre a saída desejada (`targets`) e a saída prevista (`predicted_output`).
    2. **Gradiente da Camada de Saída**: Multiplica o erro da camada de saída pela derivada da função de ativação da saída prevista. Este `delta_output` representa o quanto os pesos da camada de saída

precisam ser ajustados.
3. **Erro da Camada Oculta**: Propaga o erro da camada de saída de volta para a camada oculta, multiplicando `delta_output` pelos pesos da camada oculta-saída (transpostos).
4. **Gradiente da Camada Oculta**: Multiplica o erro da camada oculta pela derivada da função de ativação da saída da camada oculta. Este `delta_hidden` indica o ajuste necessário para os pesos da camada de entrada-oculta.
5. **Atualização de Pesos e Biases**: Os pesos e biases são ajustados na direção oposta ao gradiente (descida do gradiente), proporcionalmente à **taxa de aprendizado**.

- **Treinamento (`train`)**:
    - Itera por um número definido de épocas.
    - Para cada época, percorre todo o conjunto de treinamento, realizando o `feedforward` e o `backpropagation` para cada par (entrada, saída desejada).
    - Monitora o erro médio absoluto (`MAE`) para acompanhar o progresso do treinamento.

- **Previsão (`predict`)**: Utiliza a fase de `feedforward` para obter a saída da rede para novas entradas.

1.2. Geração de Dados Booleanos

A função `generate_boolean_data` cria todas as combinações de entradas booleanas para n entradas e calcula a saída esperada para as portas **AND**, **OR**, e **XOR**. Isso garante que a rede seja treinada em todos os casos possíveis para a lógica booleana.

---

2. Resultados dos Testes e Investigações

Foram realizados experimentos para as portas AND, OR e XOR com diferentes configurações de entradas, taxas de aprendizado, número de neurônios ocultos e funções de ativação.

2.1. 1) A Importância da Taxa de Aprendizado

A **taxa de aprendizado** (`learning_rate`) é um hiperparâmetro crítico que controla o tamanho dos passos dados durante o ajuste dos pesos da rede.

Configuração Base: Porta AND, 2 entradas, 4 neurônios ocultos, função de ativação Sigmoide, 10000 épocas.

- **Taxa de Aprendizado: 0.1 (Padrão)**

```
--- Experimentando: AND com 2 entradas ---
Taxa de Aprendizado: 0.1, Neurônios Ocultos: 4, Função de Ativação:
sigmoid

Resultados do Teste:
Entrada: [0, 0], Saída Esperada: 0, Saída Prevista: 0.0001 (Classe: 0)
Entrada: [0, 1], Saída Esperada: 0, Saída Prevista: 0.0237 (Classe: 0)
Entrada: [1, 0], Saída Esperada: 0, Saída Prevista: 0.0228 (Classe: 0)
Entrada: [1, 1], Saída Esperada: 1, Saída Prevista: 0.9625 (Classe: 1)
Acurácia: 100.00%
```

    - **Observação**: Com uma taxa de aprendizado de 0.1, a rede convergiu perfeitamente, atingindo 100% de acurácia. Os valores de saída estão bem próximos de 0 ou 1, indicando um bom aprendizado.

- **Taxa de Aprendizado: 0.01 (Menor)**

  ```
  --- Experimentando: AND com 2 entradas ---
  Taxa de Aprendizado: 0.01, Neurônios Ocultos: 4, Função de Ativação:
  sigmoid

  Resultados do Teste:
  Entrada: [0, 0], Saída Esperada: 0, Saída Prevista: 0.0264 (Classe: 0)
  Entrada: [0, 1], Saída Esperada: 0, Saída Prevista: 0.2020 (Classe: 0)
  Entrada: [1, 0], Saída Esperada: 0, Saída Prevista: 0.2024 (Classe: 0)
  Entrada: [1, 1], Saída Esperada: 1, Saída Prevista: 0.7131 (Classe: 1)
  Acurácia: 100.00%
  ```

    - **Observação**: A acurácia ainda é 100%, mas as saídas previstas para 0 ([0,0], [0,1], [1,0]) estão mais afastadas de 0 (e mais próximas de 0.5) do que com `learning_rate=0.1`. A saída para [1,1] também está mais distante de 1 (0.7131). Isso sugere que, embora a classificação esteja correta, a rede está convergindo mais lentamente ou pode precisar de mais épocas para refinar suas previsões e aproximá-las mais dos valores ideais de 0 e 1.

- **Taxa de Aprendizado: 0.5 (Maior)**

  ```
  --- Experimentando: AND com 2 entradas ---
  Taxa de Aprendizado: 0.5, Neurônios Ocultos: 4, Função de Ativação:
  sigmoid

  Resultados do Teste:
  Entrada: [0, 0], Saída Esperada: 0, Saída Prevista: 0.0004 (Classe: 0)
  Entrada: [0, 1], Saída Esperada: 0, Saída Prevista: 0.0093 (Classe: 0)
  Entrada: [1, 0], Saída Esperada: 0, Saída Prevista: 0.0095 (Classe: 0)
  Entrada: [1, 1], Saída Esperada: 1, Saída Prevista: 0.9862 (Classe: 1)
  Acurácia: 100.00%
  ```

    - **Observação**: Para o problema simples da porta AND, mesmo uma taxa de aprendizado de 0.5 levou a 100% de acurácia, com saídas bem próximas dos valores ideais. Isso indica que, para este cenário, a rede é robusta o suficiente para lidar com passos maiores sem oscilar. Em problemas mais complexos ou com ruído, uma taxa de aprendizado tão alta poderia causar divergência ou impedir a convergência.

Conclusão sobre a Taxa de Aprendizado: A taxa de aprendizado é um trade-off. Um valor muito pequeno pode levar a um treinamento lento (apesar de ainda poder convergir para boa acurácia), enquanto um valor muito grande pode causar instabilidade e impedir a convergência para problemas mais complexos. Para problemas simples como AND, a rede pode ser mais tolerante a variações.

2.2. 2) A Importância do Bias

O **bias** (viés) é um termo de ajuste crucial em redes neurais. Ele permite que o neurônio "desloque" sua função de ativação, o que é fundamental para a rede aprender padrões onde o limite de decisão não passa pela origem.

Investigação: Na implementação fornecida, o bias está incluído e inicializado aleatoriamente, sendo atualizado durante o treinamento. A importância do bias é mais conceitual e observável pela **falha da rede em aprender** se ele for removido.

- **Se o Bias fosse removido**: A função de ativação de um neurônio, sem bias, sempre produziria zero se todas as entradas fossem zero. Isso restringe a capacidade do neurônio de se ativar sob certas condições, tornando-o incapaz de modelar relações que exigem um intercepto diferente de zero. Para problemas como XOR, que não são linearmente separáveis e exigem um "limite" de decisão que não passa pela origem, a ausência de bias inviabilizaria o aprendizado.

Conclusão sobre o Bias: O bias é indispensável para a maioria das aplicações de redes neurais, pois adiciona um grau de liberdade que permite que a rede aprenda padrões mais complexos e faça decisões mais flexíveis, independentemente das entradas serem zero.

2.3. 3) A Importância da Função de Ativação

As **funções de ativação** introduzem não-linearidade na rede, permitindo que ela aprenda mapeamentos complexos e não lineares entre entradas e saídas. Sem funções de ativação não-lineares, uma rede neural, mesmo com múltiplas camadas, se comportaria como um modelo linear simples.

Configuração Base: Porta XOR, 2 entradas, 4 neurônios ocultos, taxa de aprendizado 0.1, 10000 épocas. (XOR é um bom teste, pois não é linearmente separável).

- **Função de Ativação: Sigmoide**

  ```
  --- Experimentando: XOR com 2 entradas ---
  Taxa de Aprendizado: 0.1, Neurônios Ocultos: 4, Função de Ativação: sigmoid

  Resultados do Teste:
  Entrada: [0, 0], Saída Esperada: 0, Saída Prevista: 0.0444 (Classe: 0)
  Entrada: [0, 1], Saída Esperada: 1, Saída Prevista: 0.9599 (Classe: 1)
  Entrada: [1, 0], Saída Esperada: 1, Saída Prevista: 0.9518 (Classe: 1)
  Entrada: [1, 1], Saída Esperada: 0, Saída Prevista: 0.0391 (Classe: 0)
  Acurácia: 100.00%
  ```

  - **Observação**: A Sigmoide funcionou muito bem para o XOR com esta configuração, atingindo 100% de acurácia e saídas bem próximas dos valores ideais.

- **Função de Ativação: Tanh (Tangente Hiperbólica)**

  ```
  --- Experimentando: XOR com 2 entradas ---
  Taxa de Aprendizado: 0.1, Neurônios Ocultos: 4, Função de Ativação: tanh
  ```

```
Resultados do Teste:
Entrada: [0, 0], Saída Esperada: 0, Saída Prevista: 0.0000 (Classe: 0)
Entrada: [0, 1], Saída Esperada: 1, Saída Prevista: 0.9908 (Classe: 1)
Entrada: [1, 0], Saída Esperada: 1, Saída Prevista: 0.9911 (Classe: 1)
Entrada: [1, 1], Saída Esperada: 0, Saída Prevista: 0.0001 (Classe: 0)
Acurácia: 100.00%
```

  - **Observação**: A Tanh também atingiu 100% de acurácia, com saídas extremamente próximas dos valores desejados. Suas saídas centradas em zero (-1 a 1) podem, em alguns casos, levar a uma convergência mais rápida e estável em comparação com a Sigmoide. A classificação para 0 ou 1 é feita corretamente com o limiar de 0.5.

- **Função de Ativação: ReLU (Rectified Linear Unit)**

```
--- Experimentando: XOR com 2 entradas ---
Taxa de Aprendizado: 0.1, Neurônios Ocultos: 4, Função de Ativação: relu

Resultados do Teste:
Entrada: [0, 0], Saída Esperada: 0, Saída Prevista: 0.0000 (Classe: 0)
Entrada: [0, 1], Saída Esperada: 1, Saída Prevista: 0.0000 (Classe: 0)
Entrada: [1, 0], Saída Esperada: 1, Saída Prevista: 0.0000 (Classe: 0)
Entrada: [1, 1], Saída Esperada: 0, Saída Prevista: 0.0000 (Classe: 0)
Acurácia: 50.00%
```

  - **Observação**: Com a função ReLU, a rede falhou em aprender o problema XOR, resultando em apenas 50% de acurácia. Isso significa que ela está sempre prevendo a mesma classe (0, neste caso) para todas as entradas, o que para XOR resulta em acertar metade das vezes.
    - **Possíveis Causas para a Falha da ReLU**:
      - **Dying ReLU**: É um problema comum onde neurônios ReLU podem "morrer" (sua saída se torna sempre 0, e o gradiente também 0) se a entrada para eles for constantemente negativa, impedindo que eles aprendam.
      - **Inicialização de Pesos**: A ReLU é mais sensível à inicialização de pesos. Uma má inicialização pode levar a muitos neurônios "mortos" desde o início.
      - **Taxa de Aprendizado**: Para ReLU, uma taxa de aprendizado alta pode empurrar os neurônios para a região de "morte".
      - **Problema Específico**: Para portas lógicas simples com um pequeno número de neurônios, a ReLU pode não ser a melhor escolha ou exigir um ajuste mais fino de hiperparâmetros e/ou um número maior de neurônios ocultos e/ou mais épocas. As funções Sigmoide e Tanh são mais "suaves" e seus gradientes estão sempre presentes, o que pode facilitar o treinamento para este tipo de problema.

Conclusão sobre a Função de Ativação: A escolha da função de ativação é crucial. Para problemas não lineares como XOR, uma função de ativação não-linear na camada oculta é obrigatória. Enquanto Sigmoide e Tanh funcionaram perfeitamente, a ReLU demonstrou sensibilidade, falhando em aprender com as configurações atuais. Isso destaca a importância de testar diferentes funções de ativação e seus hiperparâmetros para cada problema.

2.4. Testes com Diferentes Números de Entradas

Verificamos a generalidade do algoritmo para diferentes números de entradas.

- **Porta AND com 3 Entradas:**

```
--- Experimentando: AND com 3 entradas ---
Taxa de Aprendizado: 0.1, Neurônios Ocultos: 8, Função de Ativação: sigmoid

Resultados do Teste:
Entrada: [0, 0, 0], Saída Esperada: 0, Saída Prevista: 0.0000 (Classe: 0)
Entrada: [0, 0, 1], Saída Esperada: 0, Saída Prevista: 0.0001 (Classe: 0)
Entrada: [0, 1, 0], Saída Esperada: 0, Saída Prevista: 0.0001 (Classe: 0)
Entrada: [0, 1, 1], Saída Esperada: 0, Saída Prevista: 0.0224 (Classe: 0)
Entrada: [1, 0, 0], Saída Esperada: 0, Saída Prevista: 0.0001 (Classe: 0)
Entrada: [1, 0, 1], Saída Esperada: 0, Saída Prevista: 0.0224 (Classe: 0)
Entrada: [1, 1, 0], Saída Esperada: 0, Saída Prevista: 0.0235 (Classe: 0)
Entrada: [1, 1, 1], Saída Esperada: 1, Saída Prevista: 0.9605 (Classe: 1)
Acurácia: 100.00%
```

  - **Observação**: A rede neural conseguiu resolver a porta AND com 3 entradas com 100% de acurácia, com saídas precisas. O aumento do número de neurônios ocultos para 8 foi uma boa escolha para lidar com o espaço de entrada maior ($2^3 = 8$ combinações).

- **Porta OR com 4 Entradas:**

```
--- Experimentando: OR com 4 entradas ---
Taxa de Aprendizado: 0.1, Neurônios Ocultos: 8, Função de Ativação: sigmoid

Resultados do Teste:
Entrada: [0, 0, 0, 0], Saída Esperada: 0, Saída Prevista: 0.0325 (Classe: 0)
Entrada: [0, 0, 0, 1], Saída Esperada: 1, Saída Prevista: 0.9851 (Classe: 1)
Entrada: [0, 0, 1, 0], Saída Esperada: 1, Saída Prevista: 0.9852 (Classe: 1)
Entrada: [0, 0, 1, 1], Saída Esperada: 1, Saída Prevista: 0.9996 (Classe: 1)
Entrada: [0, 1, 0, 0], Saída Esperada: 1, Saída Prevista: 0.9845 (Classe: 1)
Entrada: [0, 1, 0, 1], Saída Esperada: 1, Saída Prevista: 0.9995 (Classe: 1)
Entrada: [0, 1, 1, 0], Saída Esperada: 1, Saída Prevista: 0.9995 (Classe: 1)
Entrada: [0, 1, 1, 1], Saída Esperada: 1, Saída Prevista: 0.9998 (Classe: 1)
Entrada: [1, 0, 0, 0], Saída Esperada: 1, Saída Prevista: 0.9842 (Classe: 1)
```

```
Entrada: [1, 0, 0, 1], Saída Esperada: 1, Saída Prevista: 0.9995 (Classe:
1)
Entrada: [1, 0, 1, 0], Saída Esperada: 1, Saída Prevista: 0.9995 (Classe:
1)
Entrada: [1, 0, 1, 1], Saída Esperada: 1, Saída Prevista: 0.9998 (Classe:
1)
Entrada: [1, 1, 0, 0], Saída Esperada: 1, Saída Prevista: 0.9994 (Classe:
1)
Entrada: [1, 1, 0, 1], Saída Esperada: 1, Saída Prevista: 0.9998 (Classe:
1)
Entrada: [1, 1, 1, 0], Saída Esperada: 1, Saída Prevista: 0.9998 (Classe:
1)
Entrada: [1, 1, 1, 1], Saída Esperada: 1, Saída Prevista: 0.9998 (Classe:
1)
Acurácia: 100.00%
```

  - **Observação**: A rede aprendeu a função OR com 4 entradas perfeitamente, com saídas muito próximas dos valores esperados.

- **Porta XOR com 3 Entradas (Mais Complexo):**

```
--- Experimentando: XOR com 3 entradas ---
Taxa de Aprendizado: 0.1, Neurônios Ocultos: 8, Função de Ativação: tanh

Resultados do Teste:
Entrada: [0, 0, 0], Saída Esperada: 0, Saída Prevista: 0.0001 (Classe: 0)
Entrada: [0, 0, 1], Saída Esperada: 1, Saída Prevista: 0.9943 (Classe: 1)
Entrada: [0, 1, 0], Saída Esperada: 1, Saída Prevista: 0.9935 (Classe: 1)
Entrada: [0, 1, 1], Saída Esperada: 0, Saída Prevista: -0.0000 (Classe: 0)
Entrada: [1, 0, 0], Saída Esperada: 1, Saída Prevista: 0.9946 (Classe: 1)
Entrada: [1, 0, 1], Saída Esperada: 0, Saída Prevista: -0.0000 (Classe: 0)
Entrada: [1, 1, 0], Saída Esperada: 0, Saída Prevista: 0.0000 (Classe: 0)
Entrada: [1, 1, 1], Saída Esperada: 1, Saída Prevista: 0.9953 (Classe: 1)
Acurácia: 100.00%
```

  - **Observação**: O problema XOR se torna mais complexo com mais entradas (o número de combinações dobra). No entanto, a rede com Tanh e 8 neurônios ocultos conseguiu atingir 100% de acurácia, com saídas extremamente precisas (inclusive com valores próximos de 0.0000 e -0.0000, que são interpretados como 0). Isso demonstra a robustez do Backpropagation para problemas não lineares de maior dimensionalidade quando a arquitetura (número de neurônios ocultos) e a função de ativação são bem escolhidas.