Lista 08 - **Redes Neurais com Backpropagation**
Disciplina: Inteligência Artificial
Profª.: Cristiane Neri Nobre
Data: 31/10/2025
Aluna: Alessandra Faria Rodrigues
Matrícula: 828333

Parte 1: O Problema do XOR

O XOR (OU-exclusivo) é um problema fundamental porque não é linearmente separável. Isso significa que você não pode desenhar uma única linha reta para separar as saídas 0 das saídas 1. Por isso, precisamos de uma rede neural com pelo menos uma camada oculta.

Dados do XOR:

- Entrada: **[0, 0]**, Saída Esperada: **0**
- Entrada: **[0, 1]**, Saída Esperada: **1**
- Entrada: **[1, 0]**, Saída Esperada: **1**
- Entrada: **[1, 1]**, Saída Esperada: **0**

Estrutura da Rede Proposta:

- **Camada de Entrada:** 2 neurônios (para as duas entradas)
- **Camada Oculta:** 2 neurônios
- **Camada de Saída:** 1 neurônio (para a saída 0 ou 1)

Código disponível em:

Explicação do Código:

1. **`RedeNeural`** (Classe):
    - **`__init__`**: Configura a arquitetura (número de neurônios por camada) e inicializa os pesos (`pesos_ih`, `pesos_ho`) e biases (`bias_h`, `bias_o`) com valores aleatórios pequenos.
    - **`sigmoid`**: Implementa a função de ativação conforme a fórmula $\sigma(x) = \frac{1}{1+e^{-x}}$
    - **`sigmoid_derivada`**: Calcula a derivada da sigmoide.
    - **`feedforward`**: Pega uma lista de entradas, converte-a em um vetor-coluna, e a propaga pela rede (da entrada para a oculta, da oculta para a saída), aplicando a sigmoide em cada etapa.
    - **`treinar`**:
        1. Executa o `feedforward` para obter a previsão (`saida_final`).

2. Calcula o `erro_saida` (a diferença entre o alvo e a previsão).
3. Propaga esse erro para trás (`erro_oculta`) usando os pesos da camada de saída.
4. Calcula os "deltas" (`delta_saida`, `delta_oculta`), que são os erros multiplicados pela derivada da ativação.
5. Finalmente, usa esses deltas para atualizar todos os pesos e biases, multiplicando pela `taxa_aprendizado`.

2. **Loop de Treinamento:**
    - Instanciamos a rede (`nn`).
    - Definimos os dados do XOR.
    - Iteramos por um número de `epocas`. Em cada época, embaralhamos os dados e chamamos `nn.treinar()` para cada exemplo.
    - A cada 1000 épocas, calculamos o Erro Quadrático Médio (MSE) para vermos o progresso.

3. **Teste Final:**
    - Após o treino, passamos por cada entrada do XOR e imprimimos a previsão final da rede.

Resultados obtidos:

1. Análise do Erro Quadrático Médio (MSE)
   O MSE nos diz o quão "errada" a rede está em média, e observando os resultados obtidos percebemos que:
    - **Épocas 1000-2000 (MSE ~0.249):** é o equivalente a um palpite aleatório, isso nos mostra que, nas primeiras 2000 épocas, a rede estava "perdida", sem ter encontrado um caminho para otimizar os pesos.
    - **Épocas 3000-5000:** Entre as épocas 3000 e 5000, o MSE despencou de **0.19** para **0.016**. Este foi o momento de descoberta da rede, onde o algoritmo de backpropagation encontrou uma combinação de pesos que começou a resolver o problema.
    - **Épocas 6000-15000 (Ajuste Fino):** De 6000 em diante, o MSE continuou caindo, mas muito mais devagar (de 0.007 para 0.001). Isso é o **ajuste fino** (fine-tuning). A rede já "entendeu" o padrão e estava apenas fazendo pequenas correções nos pesos para se aproximar o máximo possível de 0 e 1.

2. Análise dos Resultados dos Testes
    - **Entrada: [1, 1] -> Esperado: 0 -> Previsto: 0.0378** (Sucesso, muito perto de 0)
    - **Entrada: [1, 0] -> Esperado: 1 -> Previsto: 0.9700** (Sucesso, muito perto de 1)

- **Entrada: [0, 0] -> Esperado: 0 -> Previsto: 0.0303** (Sucesso, muito perto de 0)
- **Entrada: [0, 1] -> Esperado: 1 -> Previsto: 0.9688** (Sucesso, muito perto de 1)

Também percebemos que o número de épocas (15.000) foi mais do que suficiente para que ocorresse o aprendizado. A análise dos resultados nos mostra que a rede não está apenas "acertando", ela está respondendo com alta confiança, o que é o cenário ideal.

```
.exe c:/Users/USER/Desktop/CC-PUC/IA/Lista08/Backpropagati
--- Iniciando Treinamento da Rede Neural para o XOR ---
Época 1000/15000 - MSE: 0.24998838
Época 2000/15000 - MSE: 0.24921067
Época 3000/15000 - MSE: 0.19135748
Época 4000/15000 - MSE: 0.10687493
Época 5000/15000 - MSE: 0.01601146
Época 6000/15000 - MSE: 0.00707171
Época 7000/15000 - MSE: 0.00442272
Época 8000/15000 - MSE: 0.00319212
Época 9000/15000 - MSE: 0.00248869
Época 10000/15000 - MSE: 0.00203550
Época 11000/15000 - MSE: 0.00171993
Época 12000/15000 - MSE: 0.00148784
Época 13000/15000 - MSE: 0.00131011
Época 14000/15000 - MSE: 0.00116970
Época 15000/15000 - MSE: 0.00105600

--- Treinamento Concluído ---

--- Testando a Rede Treinada ---
Entrada: [1, 1] -> Esperado: 0 -> Previsto: 0.0378
Entrada: [1, 0] -> Esperado: 1 -> Previsto: 0.9700
Entrada: [0, 0] -> Esperado: 0 -> Previsto: 0.0303
Entrada: [0, 1] -> Esperado: 1 -> Previsto: 0.9688
PS C:\Users\USER\Desktop\CC-PUC\IA\Lista08>
```

Figura 1: Print da saída após a execução do código

```
# 1. Definição dos parâmetros da rede
neuronios_entrada = 2
neuronios_oculta = 3  # 3 neurônios na camada oculta
neuronios_saida = 1
taxa_aprendizado = 0.1
epocas = 15000

# 2. Dados de Treinamento (XOR)
# (Entrada, Saída Esperada)
data_xor = [
    ([0, 0], [0]),
    ([0, 1], [1]),
    ([1, 0], [1]),
    ([1, 1], [0])
]
```

Figura 2: Definindo os parâmetros

Parte 2: Dígitos de um display de 7 segmentos

Código disponível em:

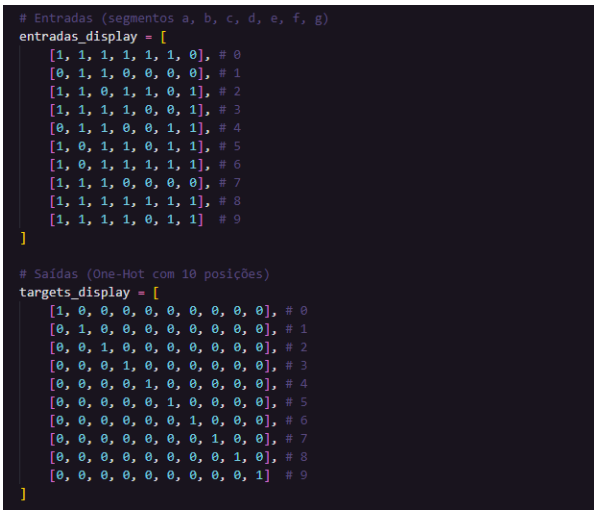
Explicação do Código:

1. A classe `RedeNeural` é a mesma da parte 1.
2. A arquitetura da rede:
    - **7 neurônios na camada de entrada** (um para cada segmento, 'a' a 'g')
    - **5 neurônios na camada oculta**
    - **10 neurônios na camada de saída** (um para cada dígito, 0 a 9)
3. Os dados de `entradas_display` e `targets_display` foram copiados da tabela fornecida, respeitando a entrada de 7 segmentos e a saída one-hot de 10 posições.
4. **Instanciação:** A rede `nn_display` é criada com `RedeNeural(7, 5, 10, ...)`, refletindo a arquitetura 7 (entrada), 5 (oculta) e 10 (saída).
5. **Treinamento:** O loop de treino é idêntico ao do XOR. A rede aprende a mapear os 7 bits de entrada para os 10 bits de saída, minimizando o MSE.
6. **Teste (Perfeito):** No primeiro teste, `np.argmax(previsao)`. Isso verifica qual dos 10 neurônios de saída teve a maior ativação e considera esse índice como o dígito previsto.
7. **Teste (Ruído):** Na segunda parte do teste, a função `adicionar_ruido` para satisfazer o requisito da lista de "simular falha de algum segmento". Esta função pega uma entrada perfeita e inverte um bit aleatoriamente (de 1 para 0, ou 0 para 1). Em seguida, testamos se a rede ainda consegue acertar o dígito, mesmo com a entrada "danificada".

Resultados obtidos:

1. **Análise do Erro Quadrático Médio (MSE):** o MSE está caindo consistentemente. Isso mostra que o algoritmo de backpropagation está fazendo seu trabalho e a rede está aprendendo (minimizando o erro) a cada época.
2. **Análise do Teste (Entradas Perfeitas):** Taxa de Acerto (dados perfeitos) = 100.00%. A rede foi capaz de memorizar perfeitamente os 10 padrões de entrada exatos. Se a entrada for exatamente como ela viu no treino, ela acerta 100% das vezes.
3. **Análise da Robustez (O Teste com Ruído):** Taxa de Acerto (com 1 bit de ruído) = 40.00%. Embora a rede tenha 100% de acerto nos dados perfeitos, ela só acertou 40% das vezes quando apenas um único bit (segmento) foi alterado. Isso significa que, em 60% dos casos, a falha de um único LED foi suficiente para confundir a rede. Isso é um caso clássico de overfitting a um conjunto de dados muito pequeno. A rede não aprendeu o "conceito" do que é um dígito "2". Ela apenas "decorou" o padrão exato. Quando ela vê um padrão ligeiramente diferente, ela se perde.

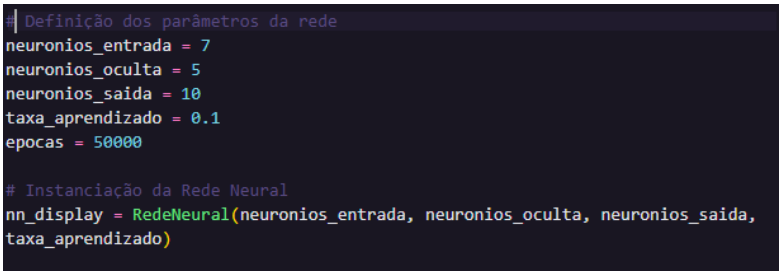
- Exemplo:
    i. **Erro:** Dígito: 6 | Entrada Ruidosa: [1, 0, 1, 1, 0, 1, 1] | Previsto: 5. O dígito 6 original é [1, 0, 1, 1, 1, 1, 1]. A entrada ruidosa é muito parecida com o dígito 5 [1, 0, 1, 1, 0, 1, 1], então a rede previu "5".
    ii. **Acerto:** Dígito: 9 | Entrada Ruidosa: [1, 1, 0, 1, 0, 1, 1] | Previsto: 9. Nesse caso, o bit que falhou não foi importante o suficiente para confundir a rede, e ela ainda conseguiu acertar.



```
# Entradas (segmentos a, b, c, d, e, f, g)
entradas_display = [
    [1, 1, 1, 1, 1, 1, 0], # 0
    [0, 1, 1, 0, 0, 0, 0], # 1
    [1, 1, 0, 1, 1, 0, 1], # 2
    [1, 1, 1, 1, 0, 0, 1], # 3
    [0, 1, 1, 0, 0, 1, 1], # 4
    [1, 0, 1, 1, 0, 1, 1], # 5
    [1, 0, 1, 1, 1, 1, 1], # 6
    [1, 1, 1, 0, 0, 0, 0], # 7
    [1, 1, 1, 1, 1, 1, 1], # 8
    [1, 1, 1, 1, 0, 1, 1]  # 9
]

# Saídas (One-Hot com 10 posições)
targets_display = [
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0], # 0
    [0, 1, 0, 0, 0, 0, 0, 0, 0, 0], # 1
    [0, 0, 1, 0, 0, 0, 0, 0, 0, 0], # 2
    [0, 0, 0, 1, 0, 0, 0, 0, 0, 0], # 3
    [0, 0, 0, 0, 1, 0, 0, 0, 0, 0], # 4
    [0, 0, 0, 0, 0, 1, 0, 0, 0, 0], # 5
    [0, 0, 0, 0, 0, 0, 1, 0, 0, 0], # 6
    [0, 0, 0, 0, 0, 0, 0, 1, 0, 0], # 7
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 0], # 8
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]  # 9
]
```

Figura 3: Definindo os dados



```
# Definição dos parâmetros da rede
neuronios_entrada = 7
neuronios_oculta = 5
neuronios_saida = 10
taxa_aprendizado = 0.1
epocas = 50000

# Instanciação da Rede Neural
nn_display = RedeNeural(neuronios_entrada, neuronios_oculta, neuronios_saida,
taxa_aprendizado)
```

Figura 4: Definindo os parâmetros

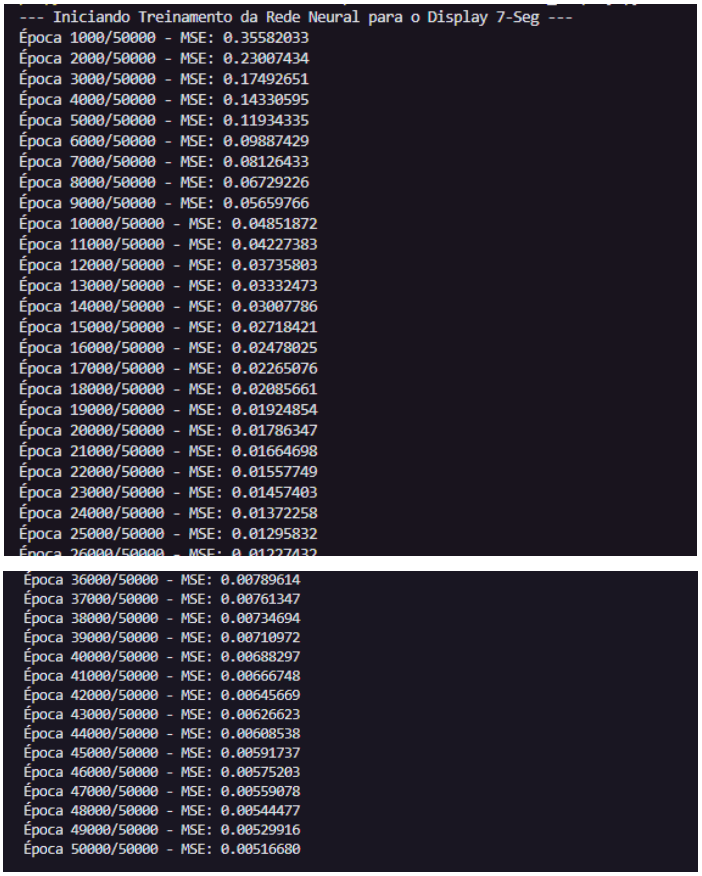


Figura 5 e 6: Print da saída após execução do código, mostrando os valores de MSE

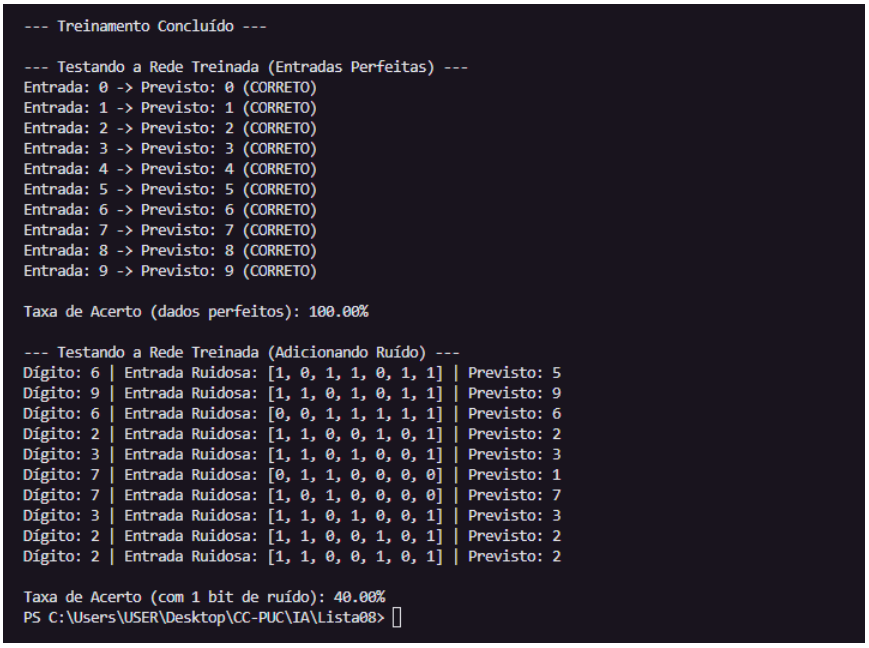


Figura 7: Print da saída após execução do código, mostrando os resultados dos testes