Lista 10

Aluna: Caroline Freitas Alvernaz

Questão 1)

Código utilizado para resolver a questão: Perceptron.ipynb - Colab

Perceptron é um modelo de aprendizado supervisionado usado para resolver problemas de classificação binária, como as funções lógicas AND e OR. O usuário pode escolher o número de entradas booleanas e observar o treinamento do Perceptron passo a passo. É um modelo de aprendizado supervisionado, ou seja, aprende a partir de exemplos com saídas desejadas conhecidas. O Perceptron é treinado com entradas booleanas (valores 0 e 1), com o objetivo de aprender a fornecer a saída correta de acordo com a função lógica escolhida.

O processo de treinamento consiste em aplicar o Perceptron sobre todos os exemplos da base. A cada nova passada completa, o algoritmo ajusta os pesos sempre que encontra erros. O treinamento continua até que todas as classificações estejam corretas ou até atingir o número máximo de tentativas definido no código.

O programa também plota graficamente a linha de separação que o Perceptron está formando entre as duas classes (valores de saída 0 e 1). Isso permite visualizar como o modelo vai ajustando sua capacidade de separação ao longo das atualizações de pesos.

A classe Perceptron é estruturada a partir de um construtor que possui o número de entradas, a taxa de aprendizado que, neste caso, foi estabelecida como 0,5, o número máximo de correção de pesos para averiguar que não haja ciclo infinito de tentativas de aprendizado, e o vetor de pesos, inicialmente zerados mais o peso do bias.

```
def __init__(self, n_inputs, learning_rate=0.5, max_epochs=100):
    self.n_inputs = n_inputs
    self.learning_rate = learning_rate
    self.max_epochs = max_epochs
    self.weights = np.zeros(n_inputs + 1)
```

Ainda, tem-se a função de ativação que retorna 1 se a soma ponderada das entradas ($u = \sum w_i \cdot x_i$) for >= 0 e 0, se < 0.

```
def activation_function(self, u):
    return 1 if u >= 0 else 0
```

A função predict faz a previsão da saída para uma entrada x e, para isso, insere o bias no vetor de entrada, calcula o produto escalar entre os pesos e as entradas e aplica a função de ativação.

```
def predict(self, x):
    x_aug = np.insert(x, 0, 1)
    u = np.dot(self.weights, x_aug)
    return self.activation_function(u)
```

Na função train é onde ocorre o ajuste de pesos da rede com base nos erros, caracterizando o aprendizado supervisionado, utilizando a fórmula:

$$W\ t+1 = Wt+(\text{taxa de aprendizado} \times \text{erro} \times \text{entrada})$$

O processo se repete até que todos os exemplos estejam corretos ou até atingir o valor máximo de correções.

```
def train(self, X, y):
    for epoch in range(self.max_epochs):
        error_occurred = False
        for xi, di in zip(X, y):
            xi_aug = np.insert(xi, 0, 1)
            yi = self.predict(xi)
            error = di - yi
            if error != 0:
                self.weights += self.learning_rate * error * xi_aug
                error_occurred = True
        if self.n_inputs == 2:
            self.plot_decision_boundary(X, y, epoch)
        if not error_occurred:
            break
```

E, para o caso de duas entradas, plota os pontos de entrada com cores diferentes conforme a saída esperada e traça a reta que representa a separação entre as classes (a fronteira de decisão), calculada com base nos pesos atuais.

```
def plot_decision_boundary(self, X, y, epoch):
    plt.figure()
    for xi, target in zip(X, y):
        if target == 1:
            plt.plot(xi[0], xi[1], 'bo')
        else:
            plt.plot(xi[0], xi[1], 'rx')

    x_vals = np.array(plt.gca().get_xlim())
    if self.weights[2] != 0:
        y_vals = -(self.weights[1] * x_vals + self.weights[0]) / self.weights[2]
        plt.plot(x_vals, y_vals, 'k--')
    plt.title(f"Época {epoch + 1}")
    plt.grid(True)
    plt.show()
```

A função generate logical data gera todas as combinações possíveis de n valores booleanos (0 ou 1) e aplica a função lógica escolhida (AND ou OR) para determinar a saída desejada para cada combinação.

```python
def generate_logical_data(n, logic_function):
    X = np.array(list(product([0, 1], repeat=n)))
    if logic_function == 'AND':
        y = np.array([int(np.all(xi)) for xi in X])
    elif logic_function == 'OR':
        y = np.array([int(np.any(xi)) for xi in X])
    else:
        raise ValueError("Função lógica inválida. Use AND ou OR.")
    return X, y
```

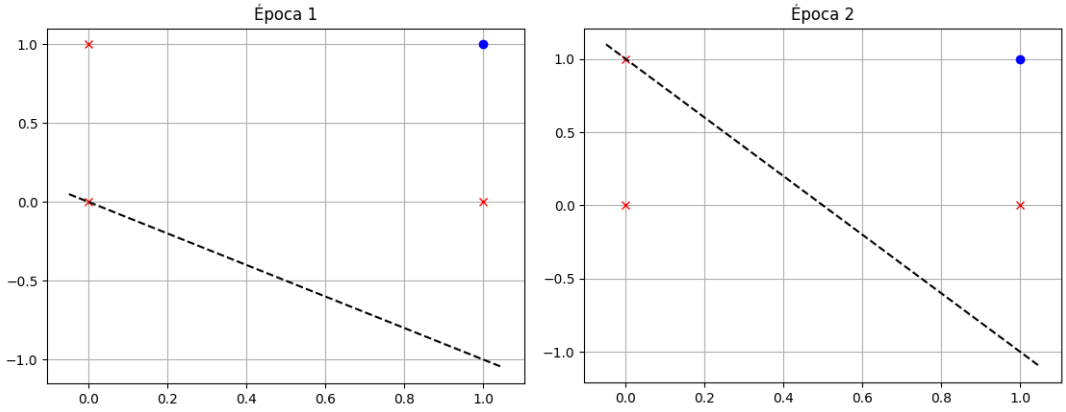
Ao executar o código, é solicitado ao usuário que escolha qual função lógica será aplicada:

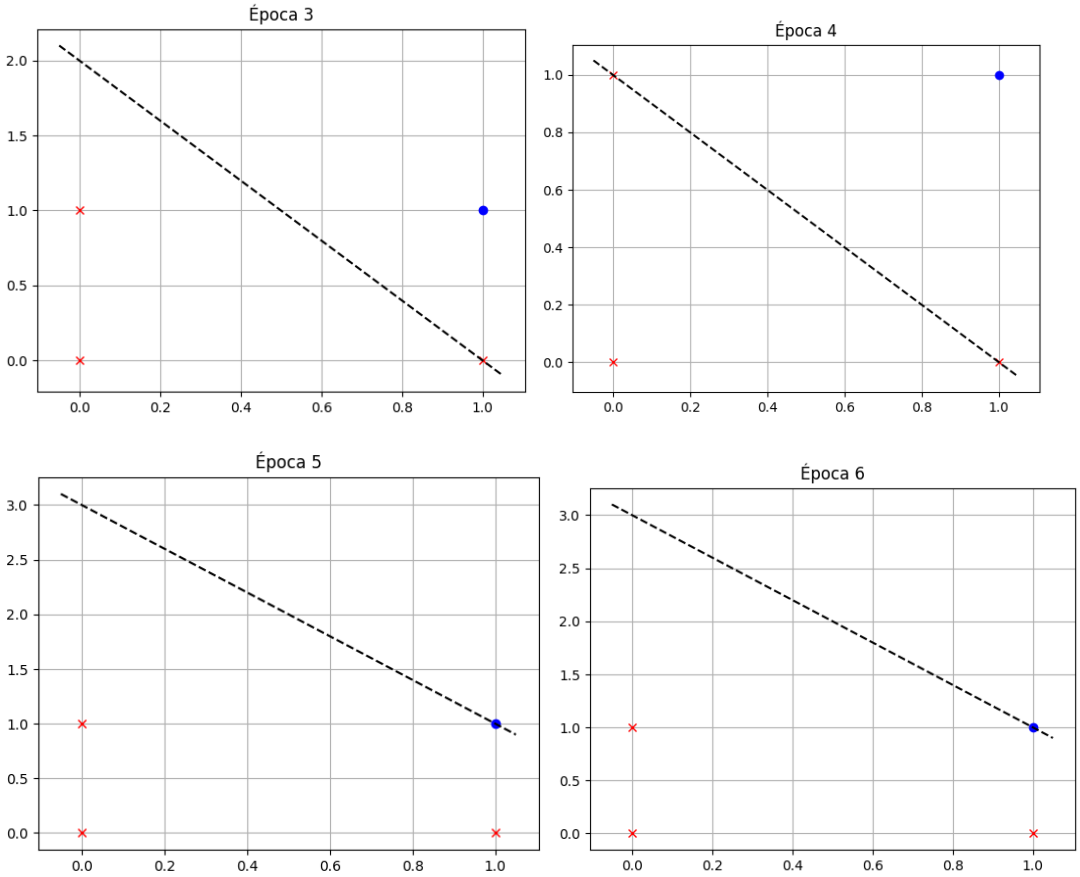
```
Digite a função lógica (AND ou OR):
```

E, após, o usuário deve digitar o número de entradas booleanas desejado:

```
Digite o número de entradas booleanas (ex: 2, 3, 10):
```

Para a opção de função lógica AND para duas entradas, tem-se o seguinte:






```
Resultados finais:
Entrada: [0 0], Esperado: 0, Previsto: 0
Entrada: [0 1], Esperado: 0, Previsto: 0
Entrada: [1 0], Esperado: 0, Previsto: 0
Entrada: [1 1], Esperado: 1, Previsto: 1
```

Essa saída mostra que o Perceptron aprendeu corretamente a função lógica AND com duas entradas booleanas.

Para a opção de função lógica OR com 4 entradas, tem-se a seguinte saída:

```
Resultados finais:
Entrada: [0 0 0 0], Esperado: 0, Previsto: 0
Entrada: [0 0 0 1], Esperado: 1, Previsto: 1
Entrada: [0 0 1 0], Esperado: 1, Previsto: 1
Entrada: [0 0 1 1], Esperado: 1, Previsto: 1
Entrada: [0 1 0 0], Esperado: 1, Previsto: 1
Entrada: [0 1 0 1], Esperado: 1, Previsto: 1
Entrada: [0 1 1 0], Esperado: 1, Previsto: 1
Entrada: [0 1 1 1], Esperado: 1, Previsto: 1
Entrada: [1 0 0 0], Esperado: 1, Previsto: 1
Entrada: [1 0 0 1], Esperado: 1, Previsto: 1
Entrada: [1 0 1 0], Esperado: 1, Previsto: 1
Entrada: [1 0 1 1], Esperado: 1, Previsto: 1
Entrada: [1 1 0 0], Esperado: 1, Previsto: 1
Entrada: [1 1 0 1], Esperado: 1, Previsto: 1
Entrada: [1 1 1 0], Esperado: 1, Previsto: 1
Entrada: [1 1 1 1], Esperado: 1, Previsto: 1
```

O perceptron não consegue resolver a função xor, porque ele só consegue aprender funções cuja saída possa ser separada por um hiperplano linear, ou seja, deve haver uma única linha (ou plano) que consegue separar perfeitamente as classes. Como a função XOR não é linearmente separável, o Perceptron nunca vai convergir para uma solução correta.

Questão 2)

Código utilizado para resolução da questão: Backpropagation.ipynb - Colab

O Backpropagation é um algoritmo de aprendizado supervisionado utilizado para treinar redes neurais multicamadas. Seu objetivo é minimizar o erro entre a saída real da rede e a saída desejada. Ele faz isso ajustando os pesos sinápticos da rede a partir da propagação do erro, de trás para frente.

Diferindo-se do Perceptron, o Backpropagation possui a função gerar_dados cria todos os vetores possíveis de entradas booleanas (como [0, 1, 1]) com n posições, e calcula a saída esperada com base na função lógica escolhida, AND (1 somente se todas as entradas forem 1), OR (1 se pelo menos uma for 1) e XOR (1 se a soma das entradas for ímpar).

```
def gerar_dados(n, funcao):
    X = np.array(list(product([0, 1], repeat=n)))
    if funcao == 'AND':
        y = np.array([[int(np.all(xi))] for xi in X])
    elif funcao == 'OR':
        y = np.array([[int(np.any(xi))] for xi in X])
    elif funcao == 'XOR':
        y = np.array([[int(np.sum(xi) % 2)] for xi in X])
    else:
        raise ValueError("Função lógica inválida. Use AND, OR ou XOR.")
    return X, y
```

A função MLP representa a rede neural treinável e é composta pela sua inicialização, definindo a quantidade de neurônios na camada de entrada e na camada oculta e, se o usuário não definir o tamanho da camada oculta, o código usa a regra de Kolmogorov, definida por: Noculta = (2N + 1)

```
def __init__(self, n_entradas, n_oculta=None, taxa_aprendizado=0.5):
    self.n_entradas = n_entradas
    self.n_oculta = n_oculta or (2 * n_entradas + 1)
    self.n_saida = 1
    self.taxa_aprendizado = taxa_aprendizado

    self.w_entrada_oculta = np.random.uniform(-1, 1, (self.n_entradas, self.n_oculta))
    self.b_oculta = np.zeros((1, self.n_oculta))

    self.w_oculta_saida = np.random.uniform(-1, 1, (self.n_oculta, self.n_saida))
    self.b_saida = np.zeros((1, self.n_saida))
```

Além disso, tem-se a fase de propagação em que calcula-se a entrada e saída da camada oculta com a função sigmoide, propaga para a camada de saída, também usando a sigmoide e retorna a saída final da rede.

```
def forward(self, X):
    self.entrada = X
    self.in_oculta = np.dot(X, self.w_entrada_oculta) + self.b_oculta
    self.out_oculta = sigmoid(self.in_oculta)

    self.in_saida = np.dot(self.out_oculta, self.w_oculta_saida) + self.b_saida
    self.saida = sigmoid(self.in_saida)
    return self.saida
```

Após, a fase de retropropagação calcula o erro da saída (desejado - obtido), propaga esse erro de volta para a camada oculta e ajusta os pesos de ambas as camadas segundo a fórmula:

$$w_{jl}(t+1) = w_{jl}(t) + \eta x^j \delta_l$$

```
def backward(self, y):
    erro_saida = y - self.saida
    delta_saida = erro_saida * sigmoid_derivada(self.saida)

    erro_oculta = delta_saida.dot(self.w_oculta_saida.T)
    delta_oculta = erro_oculta * sigmoid_derivada(self.out_oculta)

    self.w_oculta_saida += self.taxa_aprendizado * self.out_oculta.T.dot(delta_saida)
    self.b_saida += self.taxa_aprendizado * np.sum(delta_saida, axis=0, keepdims=True)

    self.w_entrada_oculta += self.taxa_aprendizado * self.entrada.T.dot(delta_oculta)
    self.b_oculta += self.taxa_aprendizado * np.sum(delta_oculta, axis=0, keepdims=True)

    return np.mean(np.abs(erro_saida))
```

Na fase de treinamento, executa-se o ciclo de propagação + retropropagação repetidamente e interrompe-se o treinamento se o erro médio ficar abaixo da tolerância.

```
def treinar(self, X, y, max_iter=10000, tolerancia=1e-4):
    for _ in range(max_iter):
        self.forward(X)
        erro = self.backward(y)
        if erro < tolerancia:
            break
```

A predição é caracterizada pela execução da fase forward e transforma a saída contínua (entre 0 e 1) em 0 ou 1 (limite 0.5).

```
def prever(self, X):
    return (self.forward(X) > 0.5).astype(int)
```

O usuário deve fornecer a função lógica desejada:

```
Escolha a função lógica (AND, OR, XOR):
```

E o número de entradas booleanas:

```
Digite o número de entradas booleanas (ex: 2, 3, 10):
```

Como exemplo, foi escolhida a função lógica XOR e 5 entradas. O resultado obtido foi o seguinte:

```
Resultados obtidos pela rede MLP com backpropagation:
Entrada: [0 0 0 0], Esperado: 0, Previsto: 0
Entrada: [0 0 0 1], Esperado: 1, Previsto: 1
Entrada: [0 0 1 0], Esperado: 1, Previsto: 1
Entrada: [0 0 1 1], Esperado: 0, Previsto: 0
Entrada: [0 1 0 0], Esperado: 1, Previsto: 1
Entrada: [0 1 0 1], Esperado: 0, Previsto: 0
Entrada: [0 1 1 0], Esperado: 0, Previsto: 0
Entrada: [0 1 1 1], Esperado: 1, Previsto: 1
Entrada: [1 0 0 0], Esperado: 1, Previsto: 1
Entrada: [1 0 0 1], Esperado: 0, Previsto: 0
Entrada: [1 0 1 0], Esperado: 0, Previsto: 0
Entrada: [1 0 1 1], Esperado: 1, Previsto: 1
Entrada: [1 1 0 0], Esperado: 0, Previsto: 0
Entrada: [1 1 0 1], Esperado: 1, Previsto: 1
Entrada: [1 1 1 0], Esperado: 1, Previsto: 1
Entrada: [1 1 1 1], Esperado: 0, Previsto: 0
```

A rede foi capaz de prever corretamente todas as saídas esperadas para as combinações possíveis de entradas. A função XOR retorna 1 quando o número de bits iguais a 1 é ímpar, e 0 caso contrário e a rede aprendeu exatamente esse padrão.

Através da implementação do Backpropagation, é possível entender que a taxa de aprendizado (neste caso taxa_aprendizado = 0.5) é um parâmetro fundamental no treinamento de redes neurais. Ela controla o tamanho do passo que os pesos darão em direção ao erro mínimo. Se a taxa for muito baixa, o treinamento será muito lento, pois os ajustes de pesos serão pequenos, porém se a taxa for muito alta, o modelo pode ultrapassar o ponto ótimo (mínimo do erro), oscilar ou até divergir, tornando o aprendizado instável.

Além disso, nota-se que o bias, um termo adicional somado ao cálculo dos neurônios, tanto na camada oculta quanto na camada de saída, é essencial, uma vez que permite que a função de ativação seja deslocada horizontalmente, ou seja, que o neurônio

dispare (ative) mesmo quando todas as entradas forem zero, sem o bias, a rede só aprenderia padrões que passam pela origem (0,0,...), o que limitaria severamente sua capacidade de aprendizagem ele age como um ajuste fino que dá mais flexibilidade à rede para modelar relações mais complexas.

Ao implementar as funções de ativação Sigmoide, Tangente Hiperbólica e ReLU para as entradas [-2. -1. 0. 1. 2.], obteve-se:

Sigmoide: [0.1192 0.2689 0.5 0.7311 0.8808]

Tanh (tangente hiperbólica): [-0.9640 -0.7616 0. 0.7616 0.9640]

ReLU: [0. 0. 0. 1. 2.]

A função sigmoide gera saídas suavizadas no intervalo **(0, 1)**. Valores negativos resultam em saídas próximas de 0, e positivos se aproximam de 1. O valor 0 gera saída 0.5 e é útil quando se deseja uma saída entre 0 e 1, como em classificações binárias. No entanto, pode sofrer com o problema do vanishing gradient, pois seus valores saturam em extremos.

A função tanh retorna valores no intervalo (-1, 1), centrada em 0. Para entradas negativas, a saída é negativa; para entradas positivas, a saída é positiva, com transição suave e é semelhante à sigmoide, mas sua saída é centrada em zero (de -1 a 1), o que melhora a simetria e acelera o aprendizado em muitos casos.

A função ReLU (Rectified Linear Unit) retorna 0 para valores negativos e o próprio valor para positivos. É uma função não linear, mas com forma simples e eficiente e hoje a função mais usada em redes profundas por ser computacionalmente simples e evitar o problema do gradiente pequeno. Ela ativa apenas neurônios com valores positivos, tornando o modelo mais eficiente e seletivo.