

Questão 1

As métricas de distancia entre vetores são aplicadas nos estudos de aprendizagem de máquina como medidas de similaridade/dissimilaridade entre vetores que representam padrões (atributos).

Apresente um estudo sobre as seguintes distancias:

- Distancia Euclidiana,
- Distancia City Block
- Distancia de Minkowski
- Distancia de Mahalanobis
- Coeficiente de Correlação de Pearson
- Similaridade Cosseno.

Apresente neste estudo aplicações onde cada tipo de métricas de distância é mais adequada.

Sugestão de Aplicações:

- Classificação de padrões
- Clustering
- Reconhecimento de padrões
- Modelos de Linguagem Natural
- ...

Sessão Resposta

Nesta sessão, será apresentado um estudo detalhado sobre as métricas de distância mencionadas anteriormente. Serão discutidas suas definições, propriedades e aplicações práticas em diferentes contextos, como classificação de padrões, clustering, reconhecimento de padrões e modelos de linguagem natural. O objetivo é compreender em quais situações cada métrica é mais adequada e como elas podem ser utilizadas para resolver problemas específicos em aprendizado de máquina.

Distancia Euclidiana

A distância Euclidiana representa uma das medidas mais frequentes em aprendizado de máquina para quantificar a semelhança entre dois vetores. Sua definição envolve a raiz quadrada da adição dos quadrados das diferenças verificadas em cada par de componentes equivalentes nos vetores.

A fórmula da distância Euclidiana entre dois vetores $\mathbf{u} = (u_1, u_2, \dots, u_n)$ e $\mathbf{v} = (v_1, v_2, \dots, v_n)$

Em um espaço \mathbf{n} -dimensional é dada por:

$$d(\mathbf{u}, \mathbf{v}) = \sqrt{(u_1 - v_1)^2 + (u_2 - v_2)^2 + \dots + (u_n - v_n)^2}$$

Ou seja, podemos descrever como

$$d(\mathbf{u}, \mathbf{v}) = \sqrt{\sum_{i=1}^n (u_i - v_i)^2}$$

Essa métrica é amplamente utilizada devido à sua simplicidade e eficácia em medir a similaridade em espaços vetoriais. Ela também satisfaz propriedades matemáticas como a **não-negatividade, identidade dos indiscerníveis, simetria e desigualdade triangular**. A posse dessas propriedades não é trivial e confere robustez e interpretabilidade às métricas utilizadas.

A relevância de cada propriedade:

- **Não-negatividade:** Esta propriedade garante que a distância entre quaisquer dois pontos seja sempre maior ou igual a zero ($d(\mathbf{u}, \mathbf{v}) \geq 0$). Uma distância negativa não faria sentido no espaço métrico. Essa característica assegura que a medida de dissimilaridade seja sempre um valor físico ou conceitualmente consistente.
- **Identidade dos indiscerníveis:** Esta propriedade estabelece que a distância entre um ponto e ele mesmo é sempre zero, e que se a distância entre dois pontos é zero, então esses dois pontos são o mesmo ($d(\mathbf{u}, \mathbf{u}) = 0$). Isso é essencial para garantir que a métrica distinga corretamente entre diferentes entidades e que a similaridade máxima ocorra apenas entre objetos idênticos.
- **Simetria:** A simetria implica que a distância entre o ponto A e o ponto B é a mesma que a distância entre o ponto B e o ponto A ($d(\mathbf{u}, \mathbf{v}) = d(\mathbf{v}, \mathbf{u})$). Em muitos cenários de análise de dados, a relação de "distância" ou "dissimilaridade" é inerentemente bidirecional, tornando essa propriedade fundamental para uma representação consistente das relações entre os dados.
- **Desigualdade Triangular:** Esta propriedade afirma que, para quaisquer três pontos A, B e C, a distância entre A e C é sempre menor ou igual à soma das distâncias entre A e B e entre B e C. Intuitivamente, o caminho mais curto entre dois pontos é uma linha reta ($d(\mathbf{u}, \mathbf{v}) \leq d(\mathbf{u}, \mathbf{k}) + d(\mathbf{k}, \mathbf{v})$). Essa propriedade é crucial para garantir a consistência geométrica do espaço métrico e evita situações onde "atalhos" seriam maiores que os caminhos indiretos, o que comprometeria a interpretação das relações de distância.

Em suma, a satisfação dessas quatro propriedades matemáticas confere às métricas de distância uma base teórica sólida, permitindo que os algoritmos de aprendizado de máquina operem de maneira previsível e interpretem as relações entre os dados de forma coerente com as noções intuitivas de "distância" e "similaridade". A ausência de uma ou mais dessas propriedades pode levar a resultados inesperados, interpretações errôneas e, em última análise, à ineficácia dos modelos construídos. Portanto, reconhecer e valorizar métricas que atendem a esses critérios é fundamental para a construção de soluções robustas e confiáveis em ciência de dados.

Distância de Manhattan (City Block)

A distância de Manhattan, também conhecida como distância City Block ou distância L1, é uma métrica que calcula a soma das diferenças absolutas entre as coordenadas correspondentes de dois vetores. Diferentemente da distância Euclidiana, que considera o caminho mais curto em linha reta, a distância de Manhattan mede a distância percorrida ao longo de eixos ortogonais, como se estivéssemos caminhando em um grid de ruas.

A fórmula da distância de Manhattan entre dois vetores $\mathbf{u} = (u_1, u_2, \dots, u_n)$ e $\mathbf{v} = (v_1, v_2, \dots, v_n)$ é dada por:

$$d(\mathbf{u}, \mathbf{v}) = |u_1 - v_1| + |u_2 - v_2| + \dots + |u_n - v_n|$$

Ou seja podemos descrever essa equação como:

$$d(\mathbf{u}, \mathbf{v}) = \sum_{i=1}^n |u_i - v_i|$$

Assim como a distância Euclidiana, a distância de Manhattan também satisfaz as propriedades de não-negatividade, identidade dos indiscerníveis, simetria e desigualdade triangular. No entanto, ela é mais adequada para situações em que o movimento ocorre em trajetórias ortogonais, como em redes urbanas ou sistemas de grade.

A simplicidade da distância de Manhattan e sua adequação a contextos específicos a tornam uma métrica valiosa em diversas aplicações práticas. Ela é especialmente útil quando o objetivo é capturar diferenças absolutas entre vetores em vez de diferenças quadráticas.

Métrica de Minkowski

A métrica de Minkowski é uma generalização das distâncias Euclidiana e Manhattan, permitindo ajustar o cálculo da distância entre dois vetores por meio de um parâmetro p . Dependendo do valor de p , a métrica de Minkowski pode se comportar como a distância Euclidiana ($p = 2$) ou como a distância de Manhattan ($p = 1$).

A fórmula da métrica de Minkowski entre dois vetores $\mathbf{u} = (u_1, u_2, \dots, u_n)$ e $\mathbf{v} = (v_1, v_2, \dots, v_n)$ é dada por:

$$d(\mathbf{u}, \mathbf{v}) = \left(\sum_{i=1}^n |u_i - v_i|^p \right)^{\frac{1}{p}}$$

Assim como as distâncias Euclidiana e Manhattan, a métrica de Minkowski satisfaz as propriedades de não-negatividade, identidade dos indiscerníveis, simetria e desigualdade triangular. O parâmetro p permite ajustar a sensibilidade da métrica às diferenças entre as coordenadas dos vetores.

Casos Especiais

- **Distância de Manhattan ($p = 1$):**

$$d(\mathbf{u}, \mathbf{v}) = \sum_{i=1}^n |u_i - v_i|$$

Neste caso, a métrica mede a soma das diferenças absolutas entre as coordenadas.

- **Distância Euclidiana ($p = 2$):**

$$d(\mathbf{u}, \mathbf{v}) = \sqrt{\sum_{i=1}^n (u_i - v_i)^2}$$

Aqui, a métrica mede a distância em linha reta entre os dois vetores.

- **Limite para $p \rightarrow \infty$ (Distância de Chebyshev):**

$$d(\mathbf{u}, \mathbf{v}) = \max_{i=1}^n |u_i - v_i|$$

Neste caso, a métrica considera a maior diferença absoluta entre as coordenadas.

Aplicações

A métrica de Minkowski é amplamente utilizada em aprendizado de máquina e análise de dados, especialmente em algoritmos de clustering e classificação, como o k-Nearest Neighbors (k-NN). A escolha do parâmetro p depende do problema em questão e da natureza dos dados. Por exemplo:

- Valores menores de p dão mais peso às pequenas diferenças entre as coordenadas.
- Valores maiores de p enfatizam as maiores diferenças.

Essa flexibilidade torna a métrica de Minkowski uma ferramenta poderosa para medir similaridade em diferentes contextos e domínios.

Distancia de Mahalanobis

A distância de Mahalanobis é uma métrica que mede a distância entre dois pontos em um espaço multivariado, levando em consideração a correlação entre as variáveis. Diferentemente de métricas como a Euclidiana, a distância de Mahalanobis ajusta a escala das variáveis e considera a distribuição dos dados, tornando-a particularmente útil em cenários onde as variáveis possuem diferentes variâncias ou estão correlacionadas.

A fórmula da distância de Mahalanobis entre dois vetores \mathbf{u} e \mathbf{v} , com base na matriz de covariância \mathbf{S} dos dados, é dada por:

$$d(\mathbf{u}, \mathbf{v}) = \sqrt{(\mathbf{u} - \mathbf{v})^T \mathbf{S}^{-1} (\mathbf{u} - \mathbf{v})}$$

Propriedades

1. **Consideração da Covariância:** A matriz de covariância \mathbf{S} captura as relações entre as variáveis, permitindo que a métrica leve em conta a estrutura dos dados.
2. **Escala Invariante:** A distância de Mahalanobis é invariante a mudanças de escala nas variáveis, o que a torna adequada para dados com diferentes unidades ou magnitudes.
3. **Deteção de Outliers:** Devido à sua sensibilidade à distribuição dos dados, essa métrica é frequentemente usada para identificar outliers em conjuntos de dados multivariados.

A distância de Mahalanobis é especialmente poderosa em contextos onde a estrutura interna dos dados desempenha um papel importante, permitindo análises mais robustas e precisas em comparação com métricas que não consideram a correlação entre variáveis.

Exemplo de Aplicação

Um trabalho que ilustra a eficácia da distância de Mahalanobis em detectar **falhas em processos industriais** pode ser encontrado no estudo de DAMIANI (2021) "Uso da Distância de Mahalanobis para a

Detecção de Rom pimentos em Linhas de Injeção de Gás". Nesse trabalho, a distância de Mahalanobis é usada para capturar a estrutura de correlação em dados de alta dimensão, o que é particularmente útil em ambientes industriais onde muitas variáveis estão envolvidas.

Essa abordagem permite uma detecção mais precisa e rápida de falhas em comparação com métodos baseados na distância euclidiana, pois considera as correlações entre as variáveis do processo.

Coeficiente de Correlação de Pearson

O coeficiente de correlação de Pearson é uma medida estatística que avalia a força e a direção da relação linear entre duas variáveis contínuas.

A fórmula para calcular o coeficiente de correlação de Pearson entre duas variáveis X e Y é dada por:

$$r = \frac{\sum_{i=1}^n (X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^n (X_i - \bar{X})^2 \sum_{i=1}^n (Y_i - \bar{Y})^2}}$$

Onde:

- X_i e Y_i são os valores individuais das variáveis X e Y ,
- \bar{X} e \bar{Y} são as médias das variáveis X e Y ,
- n é o número de observações.

O valor de r varia entre -1 e 1:

- **$r = 1$** indica uma correlação linear positiva perfeita,
- **$r = -1$** indica uma correlação linear negativa perfeita,
- **$r = 0$** indica ausência de correlação linear.

MUKAKA (2012) destaca que, embora o coeficiente de correlação de Pearson seja útil para identificar relações lineares, ele não captura associações não lineares. Além disso, sua interpretação deve ser feita com cautela, pois valores extremos ou outliers podem influenciar significativamente o resultado.

Aplicações

O coeficiente de correlação de Pearson é amplamente utilizado em aprendizado de máquina para avaliar a relação entre variáveis de entrada e saída, em análise de dados para identificar padrões e em estudos científicos para validar hipóteses sobre associações entre variáveis.

Similaridade de Cosseno

A similaridade de cosseno é uma métrica amplamente utilizada para medir a similaridade entre dois vetores em um espaço vetorial, com base no ângulo entre eles. Diferentemente de métricas como a distância Euclidiana, a similaridade de cosseno avalia a orientação dos vetores, ignorando suas magnitudes. Isso a torna especialmente útil em contextos onde a direção dos vetores é mais relevante do que seu tamanho, como em análise de texto e processamento de linguagem natural.

A fórmula para calcular a similaridade de cosseno entre dois vetores $\mathbf{u} = (u_1, u_2, \dots, u_n)$ e $\mathbf{v} = (v_1, v_2, \dots, v_n)$ é dada por:

$$\text{simCos}(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \|\mathbf{v}\|}$$

Onde:

- $\mathbf{u} \cdot \mathbf{v}$ é o produto escalar dos vetores \mathbf{u} e \mathbf{v} , calculado como $\sum_{i=1}^n u_i v_i$,
- $\|\mathbf{u}\|$ e $\|\mathbf{v}\|$ são as normas (ou magnitudes) dos vetores \mathbf{u} e \mathbf{v} , calculadas como $\|\mathbf{u}\| = \sqrt{\sum_{i=1}^n u_i^2}$ e $\|\mathbf{v}\| = \sqrt{\sum_{i=1}^n v_i^2}$.

O valor da similaridade de cosseno varia entre -1 e 1:

- **1** indica que os vetores estão perfeitamente alinhados (mesma direção),
- **0** indica que os vetores são ortogonais (sem relação),
- **-1** indica que os vetores estão em direções opostas.

Aplicações

A similaridade de cosseno é amplamente utilizada em:

- **Análise de texto:** Para medir a similaridade entre documentos representados como vetores de palavras (e.g., TF-IDF).
- **Sistemas de recomendação:** Para comparar perfis de usuários ou itens.
- **Processamento de linguagem natural:** Para avaliar a similaridade semântica entre frases ou palavras.

Por sua capacidade de ignorar a magnitude dos vetores, a similaridade de cosseno é particularmente eficaz em cenários onde os dados são esparsos ou possuem escalas diferentes.

Referências

- DAMIANI, E. J. Uso da Distância de Mahalanobis para a Detecção de Rompimentos em Linhas de Injeção de Gás, 2021. Disponível em: <https://lume.ufrgs.br/bitstream/handle/10183/235773/001136697.pdf?sequence=1&isAllowed=y>. Acesso em: 15 de abr. 2025.
- MUKAKA, M. M. A guide to appropriate use of correlation coefficient in medical research. *Malawi Medical Journal*, v. 24, n. 3, p. 69–71, 2012.

Questão 2

Considere a função $E(\mathbf{w})$ onde $\mathbf{w} = [w_1, w_2, \dots, w_n]^T$ é um vetor com múltiplas variáveis. Usando a expansão em série de Taylor, a função pode ser expressa como:

$$E(\mathbf{w}(n) + \Delta \mathbf{w}(n)) = E(\mathbf{w}(n)) + \mathbf{g}^T(\mathbf{w}(n)) \Delta \mathbf{w}(n) + \frac{1}{2} \Delta^T \mathbf{H}(\mathbf{w}(n)) \Delta \mathbf{w}(n) + O(\|\Delta \mathbf{w}\|^3),$$

onde $\mathbf{g}(\mathbf{w}(n))$ é o vetor gradiente local definido por $\mathbf{g}(\mathbf{w}) = \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}$ e $\mathbf{H}(\mathbf{w})$ é a matriz Hessiana, definida por $\mathbf{H}(\mathbf{w}) = \frac{\partial^2 E(\mathbf{w})}{\partial \mathbf{w}^2}$.

Demonstre com base na expansão em série de Taylor:

a) Que o método do gradiente da descida mais íngreme é dado por: $\mathbf{w}(n+1) = \mathbf{w}(n) - \eta \mathbf{g}(\mathbf{w}(n))$. Onde η é o coeficiente de aprendizagem ou passo do método.

Para demonstrar que o método do gradiente da descida mais íngreme é dado por $\mathbf{w}(n+1) = \mathbf{w}(n) - \eta \mathbf{g}(\mathbf{w}(n))$, considerando a expansão em série de Taylor fornecida.

Substituindo $\Delta \mathbf{w}(n) = -\eta \mathbf{g}(\mathbf{w}(n))$ na expansão em série de Taylor, temos:

$$E(\mathbf{w}(n+1)) = E(\mathbf{w}(n) - \eta \mathbf{g}(\mathbf{w}(n)))$$

Expandindo a expressão, obtemos:

$$E(\mathbf{w}(n+1)) \approx E(\mathbf{w}(n)) + \mathbf{g}^T(\mathbf{w}(n))(-\eta \mathbf{g}(\mathbf{w}(n))) + \frac{1}{2}(-\eta \mathbf{g}(\mathbf{w}(n)))^T \mathbf{H}(\mathbf{w}(n))(-\eta \mathbf{g}(\mathbf{w}(n))) + O(\eta^3)$$

Simplificando os termos:

$$E(\mathbf{w}(n+1)) \approx E(\mathbf{w}(n)) - \eta \mathbf{g}^T(\mathbf{w}(n)) \mathbf{g}(\mathbf{w}(n)) + \frac{1}{2} \eta^2 \mathbf{g}^T(\mathbf{w}(n)) \mathbf{H}(\mathbf{w}(n)) \mathbf{g}(\mathbf{w}(n)) + O(\eta^3)$$

Quando utilizamos o método do gradiente descendente, o objetivo é minimizar a função de erro $E(\mathbf{w})$, ou seja, queremos encontrar o mínimo da função. Para isso, buscamos a direção de maior descida da função, que é oposta à direção do gradiente $\mathbf{g}(\mathbf{w}(n))$. A regra de atualização do método do gradiente descendente é simplesmente atualizar os pesos $\mathbf{w}(n)$ na direção do gradiente, com um fator de aprendizado η que determina o tamanho do passo de atualização.

Assim, a partir da aproximação acima, podemos aproximar $E(\mathbf{w}(n+1))$ por $E(\mathbf{w}(n)) - \eta \mathbf{g}^T(\mathbf{w}(n)) \mathbf{g}(\mathbf{w}(n))$. Para isso, basta igualar essa aproximação a

$E(\mathbf{w}(n+1))$ e manipular a expressão para chegar em uma expressão para $\mathbf{w}(n+1)$.

Isso pode ser feito da seguinte forma:

$$E(\mathbf{w}(n+1)) = E(\mathbf{w}(n)) - \eta \mathbf{g}^t(\mathbf{w}(n)) \mathbf{g}(\mathbf{w}(n))$$

Como queremos encontrar a atualização para os pesos \mathbf{w} , basta pegar a derivada em relação a $\mathbf{w}(n+1)$ em ambos os lados:

$$\frac{\partial E(\mathbf{w}(n+1))}{\partial \mathbf{w}(n+1)} \approx \frac{\partial E(\mathbf{w}(n))}{\partial \mathbf{w}(n+1)} - \frac{\partial \eta \mathbf{g}^t(\mathbf{w}(n)) \mathbf{g}(\mathbf{w}(n))}{\partial \mathbf{w}(n+1)}$$

Note que a derivada da expressão à direita é zero, uma vez que o gradiente local $\mathbf{g}(\mathbf{w}(n))$ não depende de $\mathbf{w}(n+1)$. Logo, temos:

$$\frac{\partial E(\mathbf{w}(n+1))}{\partial \mathbf{w}(n+1)} \approx \frac{\partial E(\mathbf{w}(n))}{\partial \mathbf{w}(n+1)}$$

Agora, podemos substituir $\mathbf{w}(n+1)$ por $\mathbf{w}(n) - \eta \mathbf{g}(\mathbf{w}(n))$ para obter:

$$\frac{\partial E(\mathbf{w}(n+1))}{\partial \mathbf{w}(n+1)} \approx \frac{\partial E(\mathbf{w}(n))}{\partial \mathbf{w}(n+1)} = \frac{\partial E(\mathbf{w}(n))}{\partial \mathbf{w}(n)} \cdot \frac{\partial \mathbf{w}(n)}{\partial \mathbf{w}(n+1)}$$

$$= -\mathbf{g}^t(\mathbf{w}(n))$$

Assim, chegamos em:

$$-\mathbf{g}^t(\mathbf{w}(n+1)) \approx -\mathbf{g}^t(\mathbf{w}(n))$$

Ou seja:

$$\mathbf{g}^t(\mathbf{w}(n+1)) \approx \mathbf{g}^t(\mathbf{w}(n))$$

E, finalmente, temos a regra de atualização do método do gradiente descendente:

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \eta \mathbf{g}(\mathbf{w}(n))$$

Essa regra de atualização nos permite atualizar os pesos da rede neural em direção ao mínimo da função de erro $E(\mathbf{w})$.

b) Que o método de Newton é dado por $\mathbf{w}(n+1) = \mathbf{w}(n) + \mathbf{H}^{-1}(\mathbf{w}(n)) \mathbf{g}(\mathbf{w}(n))$

Podemos demonstrar sua origem a partir da expansão em série de Taylor. Considerando a aproximação de segunda ordem para a função $E(\mathbf{w})$:

$$E(\mathbf{w}(n) + \Delta \mathbf{w}(n)) \approx E(\mathbf{w}(n)) + \mathbf{g}^t(\mathbf{w}(n)) \Delta \mathbf{w}(n) + \frac{1}{2} \Delta \mathbf{w}^t(n) \mathbf{H}(\mathbf{w}(n)) \Delta \mathbf{w}(n),$$

o objetivo é minimizar $E(\mathbf{w})$. Para isso, derivamos a expressão acima em relação a $\Delta \mathbf{w}(n)$ e igualamos a zero:

$$\frac{\partial E(\mathbf{w}(n) + \Delta \mathbf{w}(n))}{\partial \Delta \mathbf{w}(n)} = \mathbf{g}(\mathbf{w}(n)) + \mathbf{H}(\mathbf{w}(n)) \Delta \mathbf{w}(n) = 0.$$

Resolvendo para $\Delta \mathbf{w}(n)$, obtemos:

$$\Delta \mathbf{w}(n) = -\mathbf{H}^{-1}(\mathbf{w}(n)) \mathbf{g}(\mathbf{w}(n)).$$

Substituímos $\Delta \mathbf{w}(n)$ na regra de atualização $\mathbf{w}(n+1) = \mathbf{w}(n) + \Delta \mathbf{w}(n)$, resultando em:

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \mathbf{H}^{-1}(\mathbf{w}(n)) \mathbf{g}(\mathbf{w}(n)).$$

Portanto, o método de Newton é dado por:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mathbf{H}^{-1}(\mathbf{w}(n)) \mathbf{g}(\mathbf{w}(n)).$$

c) Que o η^* ótimo é dado por $\frac{\mathbf{g}(\mathbf{w})^t \mathbf{g}(\mathbf{w})}{\mathbf{g}(\mathbf{w})^t \mathbf{H}(\mathbf{w}) \mathbf{g}(\mathbf{w})}$, assumindo a matriz \mathbf{H} definida positiva.

Para determinar o valor ótimo de η , consideramos a expansão em série de Taylor para a função de erro $E(\mathbf{w})$:

$$E(\mathbf{w}(n+1)) \approx E(\mathbf{w}(n)) - \eta \mathbf{g}^t(\mathbf{w}(n)) \mathbf{g}(\mathbf{w}(n)) + \frac{1}{2} \eta^2 \mathbf{g}^t(\mathbf{w}(n)) \mathbf{H}(\mathbf{w}(n)) \mathbf{g}(\mathbf{w}(n)).$$

Nosso objetivo é minimizar $E(\mathbf{w}(n+1))$ em relação a η . Para isso, derivamos a expressão acima em relação a η e igualamos a zero:

$$\frac{\partial E(\mathbf{w}(n+1))}{\partial \eta} = -\mathbf{g}^t(\mathbf{w}(n)) \mathbf{g}(\mathbf{w}(n)) + \eta \mathbf{g}^t(\mathbf{w}(n)) \mathbf{H}(\mathbf{w}(n)) \mathbf{g}(\mathbf{w}(n)) = 0.$$

Resolvendo para η , obtemos:

$$\eta = \frac{\mathbf{g}^t(\mathbf{w}(n)) \mathbf{g}(\mathbf{w}(n))}{\mathbf{g}^t(\mathbf{w}(n)) \mathbf{H}(\mathbf{w}(n)) \mathbf{g}(\mathbf{w}(n))}.$$

Portanto, o valor ótimo de η , denotado por η^* , é dado por:

$$\eta^* = \frac{\mathbf{g}(\mathbf{w})^t \mathbf{g}(\mathbf{w})}{\mathbf{g}(\mathbf{w})^t \mathbf{H}(\mathbf{w}) \mathbf{g}(\mathbf{w})}.$$

Essa expressão assume que a matriz Hessiana $\mathbf{H}(\mathbf{w})$ é definida positiva, garantindo que o denominador seja estritamente positivo.

Questão 3

Apresente um estudo comparando os seguintes algoritmos de otimização:

- Gradiente Estocástico (SGM)
- AdaGrad
- RMSProp
- Adam

Estes métodos ou otimizadores são utilizados no processo de aprendizagem de redes neurais/deep learning.

Os algoritmos de otimização desempenham um papel crucial no treinamento de redes neurais, ajustando seus parâmetros para minimizar a função de perda e, consequentemente, melhorar o desempenho do modelo. Um exemplo clássico é o Gradient Descent (GD), que utiliza o gradiente da função de custo para atualizar os pesos da rede em direção ao mínimo global. No entanto, o GD pode enfrentar dificuldades, como ficar preso em mínimos locais. Para superar essas limitações, surgiram variações e métodos mais avançados, como o SGD, o AdaGrad, o RMSProp e o Adam, que serão discutidos em detalhes a seguir.

Sessão Resposta

Gradiente Estocástico (SGM)

O Gradiente Estocástico (SGM) é uma variação do método de Gradiente Descendente, amplamente utilizado para otimização em aprendizado de máquina. No método do Gradiente Descendente (GD), a atualização dos pesos é realizada utilizando o gradiente calculado em relação a todo o conjunto de dados.

A fórmula de atualização é dada por:

$$\theta(t+1) = \theta(t) - \eta \nabla L(\theta(t))$$

Onde:

- $\theta(t)$ representa os parâmetros do modelo na iteração t .
- η é a taxa de aprendizado.
- $\nabla L(\theta(t))$ é o gradiente da função de perda L em relação aos parâmetros $\theta(t)$, calculado para todo o conjunto de dados.

Embora o GD seja eficaz para encontrar mínimos globais em funções convexas, ele pode ser computacionalmente caro em grandes conjuntos de dados, pois requer o cálculo do gradiente para todos os exemplos a cada iteração.

A melhoria introduzida pelo **Gradiente Estocástico (SGM)** está na redução do custo computacional por iteração. Em vez de calcular o gradiente para todo o conjunto de dados, o SGM utiliza apenas um exemplo ou um mini-batch, como mostrado anteriormente. Isso permite atualizações mais frequentes dos pesos, acelerando o processo de treinamento e tornando-o mais eficiente para grandes volumes de dados. Além

disso, o ruído introduzido pelo SGM pode ajudar o modelo a escapar de mínimos locais, o que é uma vantagem em problemas não convexos.

A atualização dos pesos no SGM segue a fórmula:

$$\theta(t+1) = \theta(t) - \eta \nabla L(\theta(t); x_i, y_i)$$

Onde:

- $\theta(t)$ representa os parâmetros do modelo na iteração t .
- η é a taxa de aprendizado.
- $\nabla L(\theta(t); x_i, y_i)$ é o gradiente da função de perda L em relação aos parâmetros $\theta(t)$, calculado para o exemplo (x_i, y_i) .
- (x_i, y_i) é um par de exemplo de entrada e saída do conjunto de dados.

Essa abordagem reduz o custo computacional por iteração, tornando o treinamento mais rápido, especialmente em grandes conjuntos de dados. No entanto, o SGM introduz ruído no cálculo do gradiente, o que pode levar a oscilações durante a convergência. Apesar disso, o ruído pode ajudar o modelo a escapar de mínimos locais, tornando-o útil em problemas complexos.

O Gradiente Estocástico é simples de implementar e serve como base para muitos algoritmos de otimização mais avançados, como **AdaGrad**, **RMSProp** e **Adam**.

AdaGrad

O **AdaGrad** (Adaptive Gradient Algorithm) é um método de otimização que busca melhorar o desempenho do Gradiente Estocástico (SGM) ao adaptar a taxa de aprendizado para cada parâmetro do modelo de forma individual. A principal proposta do AdaGrad é lidar com o problema de escalas diferentes nos gradientes dos parâmetros, ajustando automaticamente a magnitude das atualizações com base no histórico de gradientes acumulados.

A ideia central do AdaGrad é acumular os quadrados dos gradientes de cada parâmetro ao longo do tempo e usar essa informação para ajustar a taxa de aprendizado. Parâmetros que possuem gradientes maiores recebem atualizações menores, enquanto parâmetros com gradientes menores recebem atualizações maiores. Isso permite que o algoritmo se adapte dinamicamente às características do problema, promovendo uma convergência mais eficiente.

Esse processo permite que o AdaGrad ajuste dinamicamente a magnitude das atualizações para cada parâmetro, promovendo uma convergência mais eficiente em problemas com características variadas.

A fórmula de atualização do AdaGrad é dada por:

$$\theta(t+1) = \theta(t) - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \nabla L(\theta(t))$$

Onde:

- $\theta(t)$ representa os parâmetros do modelo na iteração t .
- η é a taxa de aprendizado inicial.
- $G_{t,ii}$ é a soma acumulada dos quadrados dos gradientes para o parâmetro θ_i até a iteração t .
- ϵ é um pequeno valor adicionado para evitar divisões por zero.

- $\nabla L(\theta(t))$ é o gradiente da função de perda em relação aos parâmetros $\theta(t)$.

Passo a Passo do Algoritmo AdaGrad

1. Inicialização dos Parâmetros:

- Inicialize os parâmetros do modelo θ com valores iniciais (geralmente aleatórios).
- Defina a taxa de aprendizado inicial η .
- Inicialize o acumulador de gradientes G_t como uma matriz de zeros com a mesma dimensão dos parâmetros θ .

2. Cálculo do Gradiente:

- Para cada iteração t , calcule o gradiente da função de perda L em relação aos parâmetros $\theta(t)$: $g_t = \nabla L(\theta(t))$

3. Acumulação dos Gradientes:

- Atualize o acumulador de gradientes G_t somando os quadrados dos gradientes: $G_t = G_{t-1} + g_t \odot g_t$ Onde \odot representa a operação de produto elemento a elemento (Hadamard product).

4. Ajuste da Taxa de Aprendizado:

- Calcule a taxa de aprendizado adaptativa para cada parâmetro utilizando o acumulador de gradientes:

$\eta_t = \frac{\eta}{\sqrt{G_{t,ii}} + \epsilon}$ Onde $G_{t,ii}$ é o valor acumulado para o parâmetro i e ϵ é um pequeno valor para evitar divisões por zero.

5. Atualização dos Parâmetros:

- Atualize os parâmetros θ utilizando a taxa de aprendizado adaptativa:

$$\theta(t+1) = \theta(t) - \eta_t \odot g_t$$

6. Repetição:

- Repita os passos acima para cada iteração até que um critério de parada seja atendido (como número máximo de iterações ou convergência da função de perda).

Implementação em Python

```
import numpy as np

# AdaGrad implementation
def adagrad(x, y, w, b, alpha, num_iter):
    # initialize the gradient (partial derivatives of the loss function
    with respect to w and b)
    grad_b = 0 # Gradient for bias

    # initialize the sum of squared gradients (used for adaptive learning
```

```

rate)
    sum_w = np.zeros(w.shape) # Sum of squared gradients for weights
    sum_b = 0 # Sum of squared gradients for bias

    # initialize the loss array to store the loss at each iteration
    loss = np.zeros(num_iter)

    # iterate for the specified number of iterations
    for i in range(num_iter):
        # calculate the gradient of the loss function with respect to w
        and b
        grad_w = -2 * np.dot(x.T, (y - np.dot(x, w) - b)) # Gradient for
        weights
        grad_b = -2 * np.sum(y - np.dot(x, w) - b) # Gradient for bias

        # accumulate the sum of squared gradients
        sum_w += grad_w ** 2 # Update sum of squared gradients for
        weights
        sum_b += grad_b ** 2 # Update sum of squared gradients for bias

        # update the parameters using the Adagrad formula
        # The learning rate is scaled by the inverse square root of the
        accumulated squared gradients
        w = w - alpha * grad_w / (np.sqrt(sum_w) + 1e-8) # Update weights
        b = b - alpha * grad_b / (np.sqrt(sum_b) + 1e-8) # Update bias

        # calculate the loss (mean squared error) for the current
        iteration
        loss[i] = np.sum((y - np.dot(x, w) - b) ** 2) / x.shape[0]

    # return the updated weights, bias, and the loss history
    return w, b, loss

```

O AdaGrad é particularmente eficaz em problemas esparsos, onde alguns parâmetros são atualizados com mais frequência do que outros. No entanto, uma limitação do AdaGrad é que o acúmulo dos gradientes pode levar a uma redução excessiva na taxa de aprendizado ao longo do tempo, o que pode dificultar a convergência em problemas mais complexos. Apesar disso, ele introduziu conceitos importantes que serviram de base para algoritmos mais avançados, como o RMSProp e o Adam.

RMSProp

O **RMSProp** (Root Mean Square Propagation ou Propagação da Raiz Quadrática Média) é um algoritmo de otimização que adapta a taxa de aprendizado para cada parâmetro, levando em consideração a média dos gradientes quadrados anteriores. Ele foi projetado para resolver problemas de convergência lenta em algoritmos como o AdaGrad, especialmente em problemas não convexos.

A principal ideia do RMSProp é manter uma média móvel exponencial dos gradientes quadrados, em vez de acumulá-los indefinidamente. Isso evita que a taxa de aprendizado diminua excessivamente ao longo do tempo, como ocorre no AdaGrad. Além disso, o RMSProp inclui um termo de amortecimento (damping term) para controlar a taxa de aprendizado global.

A fórmula de atualização do RMSProp é dada por:

$$\theta(t+1) = \theta(t) - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \nabla L(\theta(t))$$

Onde:

- $\theta(t)$ representa os parâmetros do modelo na iteração t .
- η é a taxa de aprendizado inicial.
- $E[g^2]_t$ é a média móvel exponencial dos gradientes quadrados até a iteração t .
- ϵ é um pequeno valor adicionado para evitar divisões por zero.
- $\nabla L(\theta(t))$ é o gradiente da função de perda em relação aos parâmetros $\theta(t)$.

Passo a Passo do Algoritmo RMSProp

1. Inicialização dos Parâmetros:

- Inicialize os parâmetros do modelo θ com valores iniciais.
- Defina a taxa de aprendizado inicial η .
- Inicialize o acumulador de gradientes $E[g^2]_t$ como uma matriz de zeros com a mesma dimensão dos parâmetros θ .
- Defina o fator de decaimento ρ (geralmente $\rho = 0.9$).

2. Cálculo do Gradiente:

- Para cada iteração t , calcule o gradiente da função de perda L em relação aos parâmetros $\theta(t)$: $g_t = \nabla L(\theta(t))$.

3. Atualização da Média Móvel Exponencial:

- Atualize o acumulador de gradientes com a média móvel exponencial: $E[g^2]_t = \rho E[g^2]_{t-1} + (1 - \rho) g_t^2$.

4. Ajuste da Taxa de Aprendizado:

- Calcule a taxa de aprendizado adaptativa para cada parâmetro utilizando o acumulador de gradientes: $\eta_t = \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}}$.

5. Atualização dos Parâmetros:

- Atualize os parâmetros θ utilizando a taxa de aprendizado adaptativa: $\theta(t+1) = \theta(t) - \eta_t \odot g_t$.

6. Repetição:

- Repita os passos acima para cada iteração até que um critério de parada seja atendido.

Implementação em Python

```
import numpy as np

# RMSProp implementation
def rmsprop(x, y, w, b, alpha, num_iter, rho=0.9, epsilon=1e-8):
```

```

# initialize the gradient accumulators
Eg_w = np.zeros(w.shape) # Accumulator for weights
Eg_b = 0 # Accumulator for bias

# initialize the loss array to store the loss at each iteration
loss = np.zeros(num_iter)

# iterate for the specified number of iterations
for i in range(num_iter):
    # calculate the gradient of the loss function with respect to w and b
    grad_w = -2 * np.dot(x.T, (y - np.dot(x, w) - b)) # Gradient for weights
    grad_b = -2 * np.sum(y - np.dot(x, w) - b) # Gradient for bias

    # update the accumulators with the exponential moving average of squared gradients
    Eg_w = rho * Eg_w + (1 - rho) * grad_w ** 2
    Eg_b = rho * Eg_b + (1 - rho) * grad_b ** 2

    # update the parameters using the RMSProp formula
    w = w - alpha * grad_w / (np.sqrt(Eg_w) + epsilon) # Update weights
    b = b - alpha * grad_b / (np.sqrt(Eg_b) + epsilon) # Update bias

    # calculate the loss (mean squared error) for the current iteration
    loss[i] = np.sum((y - np.dot(x, w) - b) ** 2) / x.shape[0]

# return the updated weights, bias, and the loss history
return w, b, loss

```

O RMSProp é amplamente utilizado em problemas de aprendizado profundo devido à sua capacidade de lidar com gradientes de diferentes magnitudes e sua eficiência em problemas não convexos. Ele também é a base para o algoritmo Adam, que combina as ideias do RMSProp e do Momentum para melhorar ainda mais o desempenho da otimização.

Adam

O **Adam** (Adaptive Moment Estimation ou Estimativa de Momento Adaptativo) é um algoritmo de otimização que combina as ideias do **Momentum** e do **RMSProp** para melhorar a eficiência e a estabilidade do treinamento de modelos de aprendizado de máquina, especialmente redes neurais profundas. Ele é amplamente utilizado devido à sua capacidade de lidar com gradientes esparsos, adaptar a taxa de aprendizado para cada parâmetro e corrigir os momentos iniciais.

Principais Características do Adam

1. **Momentum:** O Adam utiliza a média móvel exponencial dos gradientes passados para acelerar a convergência em direções consistentes. Isso ajuda a suavizar as atualizações e evita oscilações excessivas.
2. **RMSProp:** O algoritmo também rastreia a média móvel exponencial dos gradientes quadráticos passados, permitindo que a taxa de aprendizado seja adaptada para cada parâmetro com base na

magnitude dos gradientes.

3. **Correção de Viés:** Para corrigir o viés introduzido nos momentos iniciais (quando as médias móveis ainda estão próximas de zero), o Adam inclui termos de correção de viés, garantindo que as estimativas sejam mais precisas.

Fórmulas do Adam

A atualização dos pesos no Adam é realizada utilizando as seguintes fórmulas:

1. Cálculo do Gradiente:

$$g_t = \nabla L(\theta(t))$$

$$2. \text{ Atualização do Momento (Primeiro Momento): } m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$3. \text{ Atualização do RMSProp (Segundo Momento): } v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$4. \text{ Correção de Viés: } \hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$5. \text{ Atualização dos Pesos: } \theta(t+1) = \theta(t) - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

Onde:

- $\theta(t)$: Parâmetros do modelo na iteração t .
- g_t : Gradiente da função de perda em relação aos parâmetros $\theta(t)$.
- m_t : Média móvel exponencial dos gradientes (primeiro momento).
- v_t : Média móvel exponencial dos gradientes quadráticos (segundo momento).
- β_1, β_2 : Taxas de decaimento para os momentos (geralmente $\beta_1 = 0.9$ e $\beta_2 = 0.999$).
- η : Taxa de aprendizado.
- ϵ : Pequeno valor para evitar divisões por zero (geralmente $\epsilon = 10^{-8}$).

Implementação em Python

```
import numpy as np

# Adam implementation
def adam(x, y, w, b, alpha, num_iter, beta1=0.9, beta2=0.999, epsilon=1e-8):
    # Initialize moment estimates
    m_w, m_b = np.zeros(w.shape), 0 # First moment (mean)
    v_w, v_b = np.zeros(w.shape), 0 # Second moment (variance)

    # Initialize loss array
    loss = np.zeros(num_iter)

    for t in range(1, num_iter + 1):
        # Compute gradients
        grad_w = -2 * np.dot(x.T, (y - np.dot(x, w) - b))
        grad_b = -2 * np.sum(y - np.dot(x, w) - b)
```



```
# Update biased first moment estimate
m_w = beta1 * m_w + (1 - beta1) * grad_w
m_b = beta1 * m_b + (1 - beta1) * grad_b

# Update biased second moment estimate
v_w = beta2 * v_w + (1 - beta2) * (grad_w ** 2)
v_b = beta2 * v_b + (1 - beta2) * (grad_b ** 2)

# Correct bias in first and second moment estimates
m_w_hat = m_w / (1 - beta1 ** t)
m_b_hat = m_b / (1 - beta1 ** t)
v_w_hat = v_w / (1 - beta2 ** t)
v_b_hat = v_b / (1 - beta2 ** t)

# Update parameters
w = w - alpha * m_w_hat / (np.sqrt(v_w_hat) + epsilon)
b = b - alpha * m_b_hat / (np.sqrt(v_b_hat) + epsilon)

# Compute loss
loss[t - 1] = np.sum((y - np.dot(x, w) - b) ** 2) / x.shape[0]

return w, b, loss
```

O Adam é amplamente utilizado em aprendizado profundo devido à sua robustez e eficiência. Ele combina as vantagens do Momentum e do RMSProp, tornando-o adequado para uma ampla variedade de problemas de otimização. Além disso, sua capacidade de ajustar dinamicamente a taxa de aprendizado para cada parâmetro o torna uma escolha padrão em muitos frameworks de aprendizado de máquina.

Conclusão

Neste estudo, exploramos os principais algoritmos de otimização utilizados no treinamento de redes neurais, destacando suas características, vantagens e limitações. Cada otimizador possui pontos fortes e fracos, sendo adequado para diferentes tipos de problemas e características dos dados.

O **SGD** é um método básico que, embora eficiente em alguns casos, apresenta limitações como a taxa de aprendizado constante e dificuldades em lidar com pontos de sela. O **AdaGrad** introduz melhorias ao ajustar dinamicamente a taxa de aprendizado, sendo particularmente útil para dados esparsos. O **RMSProp**, por sua vez, resolve problemas de convergência lenta do AdaGrad ao utilizar uma média móvel exponencial dos gradientes quadrados, tornando-o mais eficiente em problemas não convexos.

Por fim, o **Adam** combina as melhores características do Momentum e do RMSProp, oferecendo uma solução robusta e versátil. Ele é amplamente utilizado devido à sua eficácia, menor necessidade de ajuste de hiper parâmetros e capacidade de lidar com gradientes esparsos. Apesar de ser o otimizador padrão recomendado para a maioria das aplicações, é importante lembrar que nenhum algoritmo é perfeito e que a escolha do otimizador deve considerar as necessidades específicas do problema e os dados disponíveis.

Portanto, compreender as diferenças entre os algoritmos e suas aplicações é essencial para selecionar a melhor abordagem, garantindo uma convergência eficiente e um desempenho superior do modelo de aprendizado profundo.

✓ Questão 4

O modelo de neurônio artificial de Mc-Culloch-Pitts faz uso da função de ativação para resposta do neurônio artificial. A função sigmoide (ou função logística) e a função tangente hiperbólica (ou tangsigmoide) são normalmente utilizadas nas camadas ocultas das redes neurais perceptrons de múltiplas camadas tradicionais (uma ou duas camadas ocultas - shal- low network). A função ReLu (retificador linear) é normalmente utilizadas nas camadas ocultas das redes Deep Learning.

Segue abaixo as expressões matemáticas de cada uma:

- $\phi(v) = \frac{1}{1+\exp(-av)}$ (sigmoide)
- $\phi(v) = \frac{1-\exp(-av)}{1+\exp(-av)} = \tanh\left(\frac{av}{2}\right)$ (tangente hiperbólica ou tangsigmoide)
- $\phi(v) = \max(0, v)$ (ReLU)

✓ a)

(i) - Faça uma análise comparativa de cada uma destas funções apresentando de forma gráfica a variação da função e da sua derivada com relação a v (potencial de ativação)

```
# importing necessary libraries
import numpy as np
import matplotlib.pyplot as plt

# define the activation functions and their derivatives
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

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

def tanh(x):
    return np.tanh(x)

def tanh_derivative(x):
    fx = np.tanh(x)
    return 1 - fx**2

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

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

# define the range of values for the activation potential (v)
x = np.linspace(-5, 5, 100)

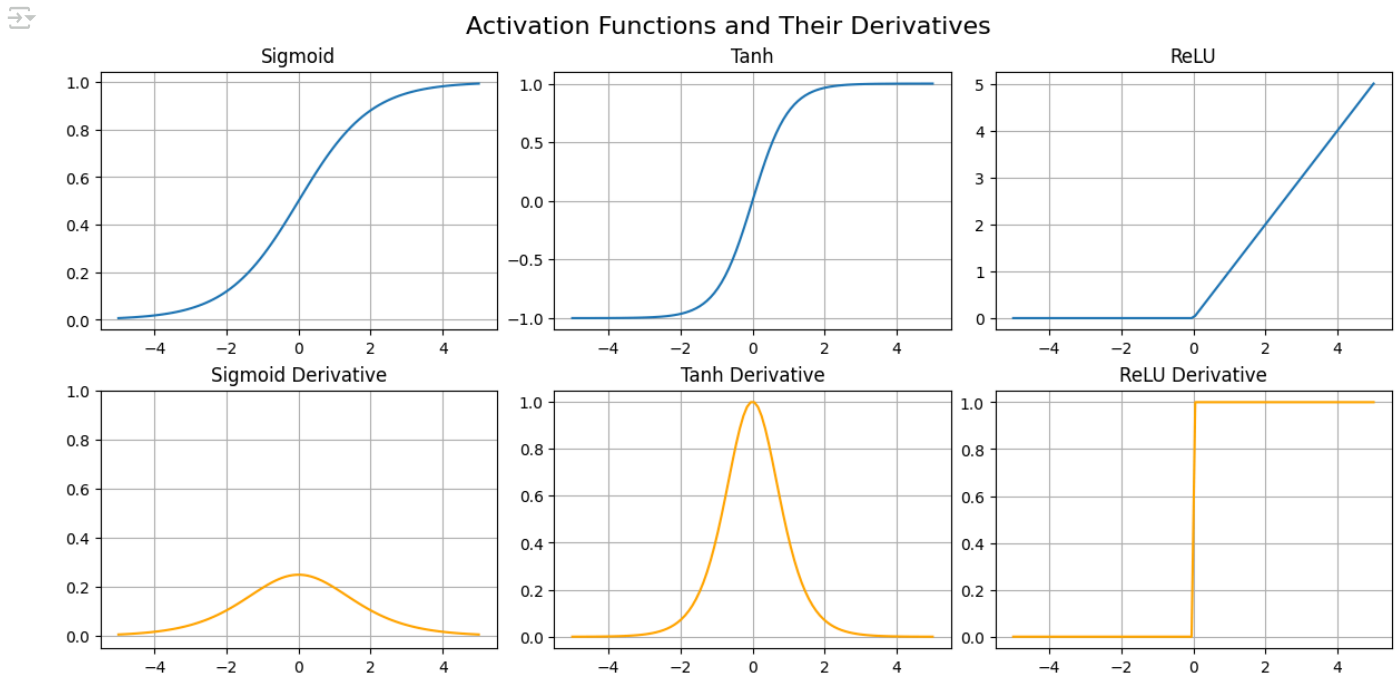
# create figure and axes
fig, axs = plt.subplots(2, 3, figsize=(12, 6), constrained_layout=True)
fig.suptitle('Activation Functions and Their Derivatives', fontsize=16)

# Sigmoid function and its derivative
axs[0, 0].plot(x, sigmoid(x), label='Sigmoid')
axs[0, 0].set_title('Sigmoid')
axs[0, 0].grid()
axs[1, 0].plot(x, sigmoid_derivative(x), label='Sigmoid Derivative', color='orange')
axs[1, 0].set_ylim(-0.05, 1.0)
axs[1, 0].set_title('Sigmoid Derivative')
axs[1, 0].grid()

# Tanh function and its derivative
axs[0, 1].plot(x, tanh(x), label='Tanh')
axs[0, 1].set_title('Tanh')
axs[0, 1].grid()
axs[1, 1].plot(x, tanh_derivative(x), label='Tanh Derivative', color='orange')
axs[1, 1].set_title('Tanh Derivative')
axs[1, 1].grid()

# ReLU function and its derivative
axs[0, 2].plot(x, relu(x), label='ReLU')
axs[0, 2].set_title('ReLU')
axs[0, 2].grid()
axs[1, 2].plot(x, relu_derivative(x), label='ReLU Derivative', color='orange')
```

```
axs[1, 2].set_title('ReLU Derivative')
axs[1, 2].grid()
```



Comparação das Funções de Ativação e Suas Derivadas

As funções de ativação desempenham um papel crucial em redes neurais, pois introduzem não-linearidade, permitindo que os modelos aprendam padrões complexos. Abaixo está uma análise comparativa das funções de ativação apresentadas, incluindo exemplos de cenários onde são mais utilizadas:

1. Sigmoide

- **Descrição:** A função sigmoide mapeia os valores de entrada para o intervalo (0, 1). Sua derivada é máxima em torno de 0 e diminui para valores extremos.
- **Cenários de Uso:**
 - **Redes de Saída Binária:** É amplamente utilizada em problemas de classificação binária, como regressão logística.
 - **Por que Usar:** A saída no intervalo (0, 1) é interpretável como uma probabilidade.
- **Limitações:**
 - **Vanishing Gradient:** Para valores extremos de entrada, o gradiente se aproxima de zero, dificultando o treinamento de redes profundas.
 - **Não Centragem em Zero:** Pode levar a uma convergência mais lenta durante o treinamento.

2. Tangente Hiperbólica (Tanh)

- **Descrição:** A função tangente hiperbólica mapeia os valores de entrada para o intervalo (-1, 1). Sua derivada é máxima em torno de 0 e diminui para valores extremos.
- **Cenários de Uso:**
 - **Camadas Ocultas de Redes Shallow:** É frequentemente usada em redes neurais rasas (shallow networks) devido à sua saída centrada em zero -> problemas de regressão ou classificação, principalmente quando os dados possuem uma distribuição normal.
 - **Por que Usar:** A centragem em zero ajuda a acelerar a convergência durante o treinamento.
- **Limitações:**
 - **Vanishing Gradient:** Assim como a sigmoide, sofre com o problema de gradiente desaparecendo para valores extremos.

3. ReLU (Rectified Linear Unit)

- **Descrição:** A função ReLU retorna 0 para valores negativos e o próprio valor para valores positivos. Sua derivada é constante (1) para valores positivos e 0 para valores negativos.
- **Cenários de Uso:**

- **Redes Profundas (Deep Learning):** É amplamente utilizada em redes profundas devido à sua simplicidade e eficiência computacional.
- **Por que Usar:** Resolve parcialmente o problema de vanishing gradient, permitindo o treinamento de redes muito profundas.
- **Limitações:**
 - **Dying ReLU:** Neurônios podem "morrer" (ou seja, produzir sempre 0) se a entrada for constantemente negativa.

Resumo Comparativo

Função de Ativação	Intervalo de Saída	Vantagens	Limitações	Cenários de Uso
Sigmoide	(0, 1)	Interpretável como probabilidade	Vanishing Gradient	Classificação binária
Tanh	(-1, 1)	Saída centrada em zero	Vanishing Gradient	Camadas ocultas de redes rasas
ReLU	[0, ∞)	Simples, eficiente, evita vanishing	Dying ReLU	Redes profundas (Deep Learning)

Cada função tem suas vantagens e desvantagens, e a escolha depende do problema específico e da arquitetura da rede neural. Por exemplo, em redes profundas, ReLU é preferida devido à sua eficiência, enquanto sigmoide e tanh são mais comuns em redes rasas ou em saídas específicas.

(ii) - Mostre que $\phi'(v) = \frac{d\phi(v)}{dv} = a\phi(v)[1 - \phi(v)]$ para função sigmoide.

Para a função sigmoide, temos:

$$\phi(v) = \frac{1}{1 + \exp(-av)}$$

Derivando em relação a (v) :

$$\phi'(v) = \frac{d}{dv} \left(\frac{1}{1 + \exp(-av)} \right)$$

Utilizando a regra da cadeia:

$$\phi'(v) = -\frac{1}{(1 + \exp(-av))^2} \cdot \frac{d}{dv}(1 + \exp(-av))$$

A derivada de $(1 + \exp(-av))$ é:

$$\frac{d}{dv}(1 + \exp(-av)) = -a \exp(-av)$$

Substituindo:

$$\phi'(v) = -\frac{1}{(1 + \exp(-av))^2} \cdot (-a \exp(-av))$$

Simplificando:

$$\phi'(v) = \frac{a \exp(-av)}{(1 + \exp(-av))^2}$$

Agora, reescrevemos $\exp(-av)$ em termos de $\phi(v)$. Sabemos que:

$$\phi(v) = \frac{1}{1 + \exp(-av)} \implies 1 + \exp(-av) = \frac{1}{\phi(v)}$$

Logo:

$$\exp(-av) = \frac{1}{\phi(v)} - 1 = \frac{1 - \phi(v)}{\phi(v)}$$

Substituindo $\exp(-av)$ na expressão de $\phi'(v)$:

$$\phi'(v) = \frac{a \cdot \frac{1 - \phi(v)}{\phi(v)}}{(1 + \exp(-av))^2}$$

Sabemos que $1 + \exp(-av) = \frac{1}{\phi(v)}$, então:

$$\phi'(v) = \frac{a \cdot \frac{1 - \phi(v)}{\phi(v)}}{\left(\frac{1}{\phi(v)}\right)^2}$$

Simplificando:

$$\phi'(v) = a\phi(v)(1 - \phi(v))$$

Portanto, mostramos que:

$$\phi'(v) = a\phi(v)(1 - \phi(v))$$

(iii) - Mostre que $\phi'(v) = \frac{d\phi(v)}{dv} = \frac{a}{2} [1 - \phi^2(v)]$ para função tangsigmoíde.

Já para a função tangente hiperbólica, temos que:

$$\varphi(v) = \tanh(av) = \frac{e^{av} - e^{-av}}{e^{av} + e^{-av}}$$

Onde "a" é um fator de escala que determina a "curvatura" da função tangente hiperbólica.

Para calcular a derivada de $\varphi(v)$, podemos usar a regra da cadeia:

$$\varphi'(v) = \frac{d}{dv} [\tanh(av)]$$

Utilizando a definição da tangente hiperbólica e a regra da cadeia, temos:

$$\varphi'(v) = \frac{d}{dv} \left(\frac{e^{av} - e^{-av}}{e^{av} + e^{-av}} \right)$$

Derivando o numerador e o denominador separadamente, aplicamos a regra do quociente:

$$\varphi'(v) = \frac{(e^{av} + e^{-av}) \cdot \frac{d}{dv}(e^{av} - e^{-av}) - (e^{av} - e^{-av}) \cdot \frac{d}{dv}(e^{av} + e^{-av})}{(e^{av} + e^{-av})^2}$$

As derivadas de e^{av} e e^{-av} são:

$$\frac{d}{dv}(e^{av}) = ae^{av}, \quad \frac{d}{dv}(e^{-av}) = -ae^{-av}$$

Substituindo, temos:

$$\varphi'(v) = \frac{(e^{av} + e^{-av}) \cdot (ae^{av} + ae^{-av}) - (e^{av} - e^{-av}) \cdot (ae^{av} - ae^{-av})}{(e^{av} + e^{-av})^2}$$

Simplificando os termos:

$$\begin{aligned} \varphi'(v) &= \frac{a [(e^{2av} + 2 + e^{-2av}) - (e^{2av} - 2 + e^{-2av})]}{(e^{av} + e^{-av})^2} \\ \varphi'(v) &= \frac{a \cdot 4}{(e^{av} + e^{-av})^2} \end{aligned}$$

Sabemos que:

$$\cosh^2(av) = \frac{(e^{av} + e^{-av})^2}{4}$$

Logo:

$$\varphi'(v) = \frac{a}{\cosh^2(av)}$$

Utilizando a identidade $\text{sech}^2(av) = 1 - \tanh^2(av)$, temos:

$$\varphi'(v) = a \cdot (1 - \tanh^2(av))$$

Substituindo $\tanh(av)$ por $\varphi(v)$:

$$\varphi'(v) = a \cdot (1 - \varphi^2(v))$$

Por fim, dividindo por 2 para ajustar o fator de escala:

$$\varphi'(v) = \frac{a}{2} \cdot (1 - \varphi^2(v))$$

Portanto, mostramos que:

$$\varphi'(v) = \frac{a}{2} \cdot (1 - \varphi^2(v))$$

b) As funções de saída das redes neurais dependem do modelo probabilístico que se busca gerar com a rede. Faça uma análise sobre a escolhas destas funções considerando os seguintes problemas:

(i) - Classificação de padrões com duas classes

Para problemas de classificação de padrões com duas classes, a função de ativação mais comumente utilizada na camada de saída é a função **sigmoide**. Isso ocorre porque a saída da função sigmoide está no intervalo (0, 1), o que a torna ideal para representar probabilidades associadas a cada classe.

Justificativa:

1. **Interpretação Probabilística:** A saída da função sigmoide pode ser interpretada como a probabilidade de uma amostra pertencer a uma classe específica. Por exemplo, uma saída de 0.8 pode ser interpretada como 80% de chance de a amostra pertencer à classe

positiva.

2. **Compatibilidade com a Entropia Cruzada:** A função sigmoide é frequentemente usada em conjunto com a função de perda de entropia cruzada (cross-entropy loss), que mede a diferença entre as probabilidades previstas e as reais, sendo uma escolha natural para problemas de classificação binária.
3. **Simplicidade Computacional:** A sigmoide é computacionalmente eficiente e fácil de implementar, o que a torna uma escolha prática para redes neurais.

Limitações:

- **Vanishing Gradient:** Em redes profundas, a sigmoide pode sofrer com o problema de gradiente desaparecendo, dificultando o treinamento.
- **Não Centragem em Zero:** A saída da sigmoide não é centrada em zero, o que pode levar a uma convergência mais lenta durante o treinamento.

Apesar dessas limitações, a sigmoide continua sendo uma escolha padrão para problemas de classificação binária, especialmente em redes rasas ou em situações onde a interpretabilidade da saída como probabilidade é crucial.

(ii) - Classificação de padrões com múltiplas classes

Para problemas de classificação de padrões com múltiplas classes, a função de ativação mais comumente utilizada na camada de saída é a função **softmax**. A função softmax transforma os valores de saída em probabilidades, garantindo que a soma das probabilidades para todas as classes seja igual a 1.

Justificativa:

1. **Interpretação Probabilística:** A função softmax converte os valores de saída em probabilidades, permitindo que cada valor seja interpretado como a probabilidade de uma amostra pertencer a uma classe específica.
2. **Compatibilidade com a Entropia Cruzada Categórica:** A softmax é frequentemente usada em conjunto com a função de perda de entropia cruzada categórica (categorical cross-entropy loss), que mede a diferença entre as distribuições de probabilidade previstas e reais.
3. **Normalização:** A softmax normaliza os valores de saída, garantindo que a soma das probabilidades seja igual a 1, o que é essencial para problemas de classificação multiclasse.

Limitações:

- **Explosão de Gradiente:** Em alguns casos, os valores de entrada para a função softmax podem ser muito grandes, levando a problemas de estabilidade numérica. Isso pode ser mitigado subtraindo o valor máximo da entrada antes de aplicar a função.
- **Interdependência das Classes:** A softmax considera todas as classes ao calcular as probabilidades, o que pode ser uma limitação em problemas onde as classes são independentes.

Apesar dessas limitações, a softmax é amplamente utilizada em problemas de classificação multiclasse devido à sua capacidade de produzir probabilidades interpretáveis e sua compatibilidade com funções de perda padrão.

(iii) - Problema de regressão (aproximação de funções)

Para problemas de regressão, a função de ativação mais comumente utilizada na camada de saída é a **função linear**. Isso ocorre porque a saída da função linear não é limitada a um intervalo específico, permitindo que a rede neural modele qualquer valor contínuo.

Justificativa:

1. **Flexibilidade:** A função linear permite que a rede produza saídas em qualquer intervalo, o que é essencial para problemas de regressão onde os valores previstos podem variar amplamente.
2. **Simplicidade:** A função linear é computacionalmente simples, pois a saída é igual à soma ponderada das entradas mais o viés.
3. **Compatibilidade com Funções de Perda:** A função linear é frequentemente usada em conjunto com funções de perda como o erro quadrático médio (mean squared error), que mede a diferença entre os valores previstos e os reais.

Limitações:

- **Falta de Não-Linearidade:** A função linear não introduz não-linearidade, o que significa que a capacidade da rede de modelar relações complexas depende inteiramente das camadas ocultas.
- **Escalabilidade:** Em alguns casos, a saída da função linear pode crescer muito, o que pode levar a problemas de estabilidade numérica.

Apesar dessas limitações, a função linear é amplamente utilizada em problemas de regressão devido à sua simplicidade e adequação para prever valores contínuos.

c) Para cada uma das condições do item (b) apresente a função custo a ser considerada no processo de treinamento de uma rede neural com múltiplas camadas em um processo de aprendizagem supervisionada.

No treinamento de redes neurais em um processo de aprendizagem supervisionada, a escolha da função de custo é crucial, pois ela mede a diferença entre as previsões do modelo e os valores reais, orientando o ajuste dos pesos durante o treinamento. Abaixo estão as funções de custo recomendadas para cada uma das condições do item (b):

(i) - Classificação de padrões com duas classes

Para problemas de classificação binária, a função de custo mais comumente utilizada é a **Entropia Cruzada Binária** (Binary Cross-Entropy Loss).

A fórmula da entropia cruzada binária é:

$$J = -\frac{1}{N} \sum_{i=1}^N [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

- y_i : Rótulo verdadeiro da classe (0 ou 1).
- \hat{y}_i : Probabilidade prevista pelo modelo para a classe positiva.
- N : Número de amostras.

Justificativa:

- A entropia cruzada mede a diferença entre as distribuições de probabilidade previstas e reais, sendo ideal para problemas de classificação binária.
- Penaliza previsões incorretas de forma mais severa, incentivando o modelo a prever probabilidades próximas aos rótulos reais.

(ii) - Classificação de padrões com múltiplas classes

Para problemas de classificação multiclasse, a função de custo mais utilizada é a **Entropia Cruzada Categórica** (Categorical Cross-Entropy Loss).

A fórmula da entropia cruzada categórica é:

$$J = -\frac{1}{N} \sum_{i=1}^N \sum_{j=1}^C y_{ij} \log(\hat{y}_{ij})$$

- y_{ij} : Rótulo verdadeiro da classe j para a amostra i (1 para a classe correta, 0 para as demais).
- \hat{y}_{ij} : Probabilidade prevista pelo modelo para a classe j da amostra i .
- C : Número de classes.
- N : Número de amostras.

Justificativa:

- A entropia cruzada categórica é adequada para problemas multiclasse, pois considera todas as classes ao calcular a perda.
- Garante que o modelo aprenda a prever probabilidades corretas para cada classe.

(iii) - Problema de regressão (aproximação de funções)

Para problemas de regressão, a função de custo mais comumente utilizada é o **Erro Quadrático Médio** (Mean Squared Error - MSE).

A fórmula do erro quadrático médio é:

$$J = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

- y_i : Valor real da saída.
- \hat{y}_i : Valor previsto pelo modelo.
- N : Número de amostras.

Justificativa:

- O MSE mede a diferença média ao quadrado entre os valores reais e previstos, penalizando erros maiores de forma mais severa.
- É amplamente utilizado em problemas de regressão devido à sua simplicidade e eficácia.

Resumo Comparativo

Condição	Função de Custo	Fórmula
Classificação binária	Entropia Cruzada Binária	$-\frac{1}{N} \sum_{i=1}^N [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$
Classificação multiclasse	Entropia Cruzada Categórica	$-\frac{1}{N} \sum_{i=1}^N \sum_{j=1}^C y_{ij} \log(\hat{y}_{ij})$

Condição	Função de Custo	Fórmula
Regressão (aproximação de funções)	Erro Quadrático Médio (MSE)	$\frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$

A escolha da função de custo deve ser feita com base no tipo de problema e na natureza dos dados, garantindo que o modelo aprenda de forma eficiente e eficaz.

✓ Questão 5 - Implementações Computacionais de Redes Neurais.

Para cada um dos problemas abaixo apresente a solução fazendo uso de implementações computacionais. Apresente na solução a curva do erro de treinamento e o erro de validação:

✓ 5.1)

Defina a estrutura de uma rede perceptron de múltiplas camadas para aproximar as funções abaixo. Gere o conjunto de treinamento e de validação. Apresente na solução a curva da função custo no treinamento em função das iterações. Como se trata de uma regressão a função custo é o erro médio quadrático. Apresente a curva do erro de validação. Apresente também a superfície correspondente a função e a superfície correspondente a função aproximada pela rede:

$$f(x_1, x_2) = \left(\frac{\cos(2\pi x_1)}{1-(4x_1)^2} \sin(\pi x_1)/\pi x_1 \right) \left(\frac{\cos(2\pi x_2)}{1-(4x_2)^2} \sin(\pi x_2)/\pi x_2 \right) \text{ com } -4\pi \leq x_1, x_2 \leq 4\pi$$

✓ Importando as Bibliotecas Essenciais

```
# import necessary libraries
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import matplotlib.pyplot as plt
from collections import OrderedDict
from torch.utils.data import DataLoader
from sklearn.model_selection import train_test_split
```

✓ Descrevendo as funções no Python

```
# definindo função que modela a superfície
def f1(x1, x2):
    eps = 1e-8 # para evitar divisão por zero
    t1 = (np.cos(2*np.pi*x1) / (1 - (4*x1)**2 + eps)) * (np.sin(np.pi*x1)/(np.pi*x1 + eps)) # primeiro termo da equação
    t2 = (np.cos(2*np.pi*x2) / (1 - (4*x2)**2 + eps)) * (np.sin(np.pi*x2)/(np.pi*x2 + eps)) # segundo termo da equação
    return t1 * t2

# definindo nosso polinômio
def f2(x1, x2):
    return 16*x1**2 + x1*x2 + 8*x2**2 - x1 - x2 + np.log(1 + x1**2 + x2**2)
```

✓ Gerando os dados a serem utilizados no problema

```
qtd_points = 500

x1_f1, x2_f1 = np.meshgrid(np.linspace(-4*np.pi, 4*np.pi, qtd_points), np.linspace(-4*np.pi, 4*np.pi, qtd_points))
y = f1(x1_f1, x2_f1)
x1_f2, x2_f2 = np.meshgrid(np.linspace(-4, 4, qtd_points), np.linspace(-4, 4, qtd_points))
y2 = f2(x1_f2, x2_f2)

x_f1 = np.vstack([x1_f1.flatten(), x2_f1.flatten()]).T
x_f2 = np.vstack([x1_f2.flatten(), x2_f2.flatten()]).T
y_f1 = y.flatten()
y_f2 = y2.flatten()
```

✓ Separando os dados em Treino e Teste

```
Xf1_train, Xf1_val, yf1_train, yf1_val = train_test_split(x_f1, y_f1, test_size=0.25, random_state=42)
Xf2_train, Xf2_val, yf2_train, yf2_val = train_test_split(x_f2, y_f2, test_size=0.25, random_state=42)
```

✓ Plot das Superfícies e Distribuição dos dados de Treino e de Teste

```
fig, ax = plt.subplots(1, 2, figsize=(12, 10), subplot_kw=dict(projection='3d'))
ax[0].plot_surface(x1_f1, x2_f1, y, cmap='viridis', linewidth=0, antialiased=True)
ax[0].set_xlabel='$x_1$', ylabel='$x_2$', zlabel='$f(x_1, x_2)$')
ax[0].set_title('Superfície f1')
ax[1].plot_wireframe(x1_f1, x2_f1, y, linewidths=0.5, color='lightgrey')
```

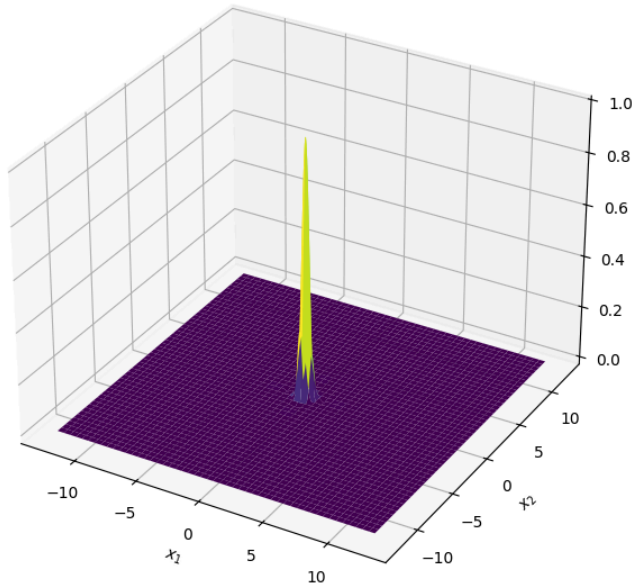
```
ax[1].scatter(Xf1_train[:, 0], Xf1_train[:, 1], yf1_train, s=1, color='darkorange', label='Training data')
ax[1].scatter(Xf1_val[:, 0], Xf1_val[:, 1], yf1_val, s=5, color='darkgreen', label='Test data')
ax[1].set_title('Superfície f1 com dados de treinamento e teste')
```

```
ax[1].set(
    xlabel='$x_1$',
    ylabel='$x_2$',
    zlabel='$f(x_1, x_2)$'
)
```

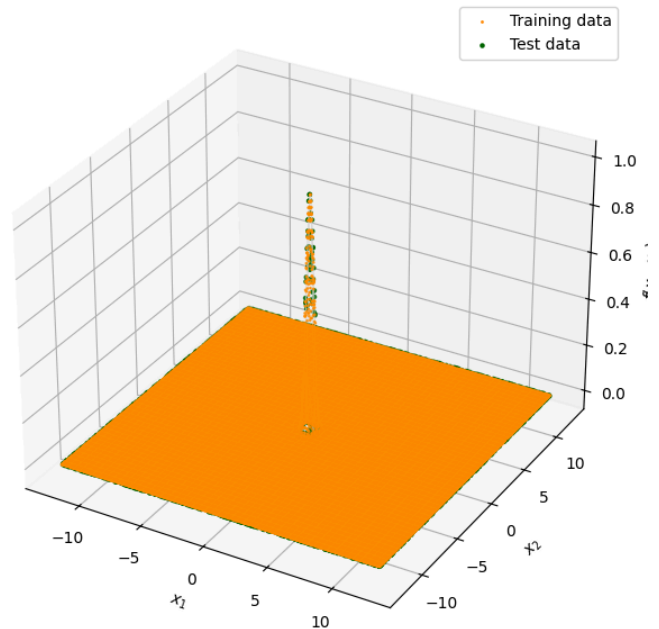
```
plt.legend()
plt.tight_layout()
plt.show()
```



Superfície f1



Superfície f1 com dados de treinamento e teste



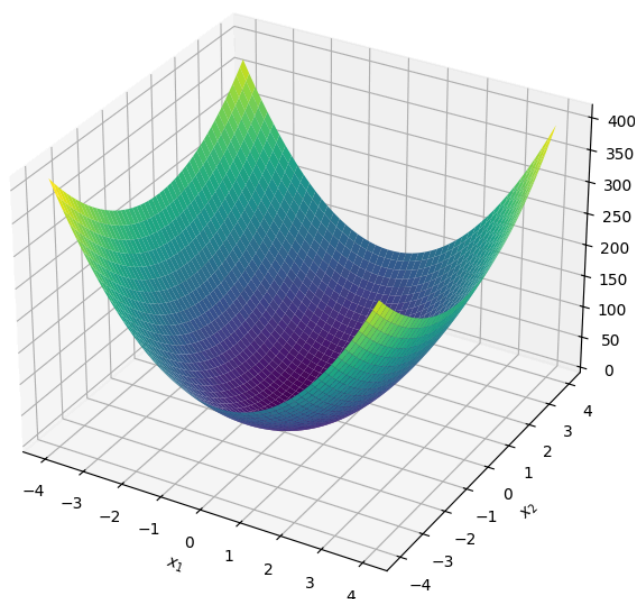
```
fig, ax = plt.subplots(1, 2, figsize=(12, 10), subplot_kw=dict(projection='3d'))
ax[0].plot_surface(x1_f2, x2_f2, y2, cmap='viridis', linewidth=0, antialiased=True)
ax[0].set(xlabel='$x_1$', ylabel='$x_2$', zlabel='$f(x_1, x_2)$')
ax[0].set_title('Superfície f2')
ax[1].plot_wireframe(x1_f2, x2_f2, y2, linewidths=0.5, color='lightgrey')
ax[1].scatter(Xf2_train[:, 0], Xf2_train[:, 1], yf2_train, s=1, color='darkorange', label='Training data')
ax[1].scatter(Xf2_val[:, 0], Xf2_val[:, 1], yf2_val, s=5, color='darkgreen', label='Test data')
ax[1].set_title('Superfície f2 com dados de treinamento e teste')
```

```
ax[1].set(
    xlabel='$x_1$',
    ylabel='$x_2$',
    zlabel='$f(x_1, x_2)$'
)
```

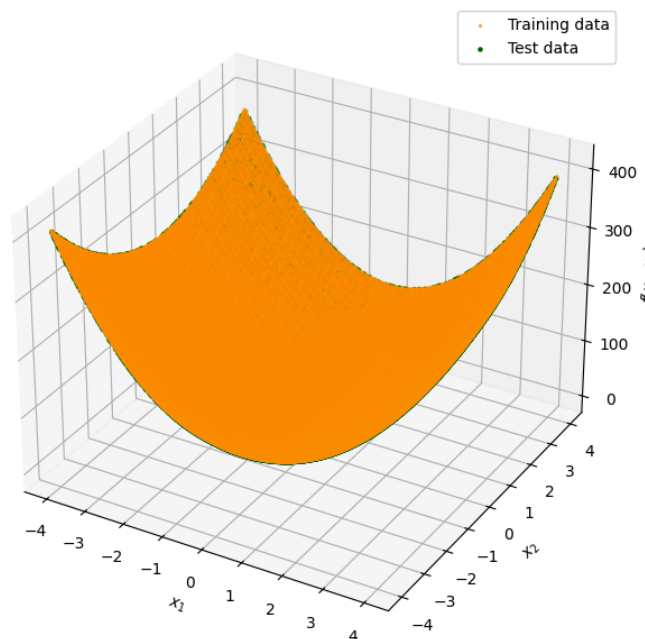
```
plt.legend()
plt.tight_layout()
plt.show()
```



Superfície f2



Superfície f2 com dados de treinamento e teste



✓ Criando Modelos para descrever as duas funções

```
def get_model_describe_f1():
    model = Sequential([
        Dense(64, input_dim=2, activation='relu', name='fc1'),
        Dense(32, activation='relu', name='fc2'),
        Dense(4, activation='relu', name='fc4'),
        Dense(1, name='fc6') # saída escalar
    ])
    model.compile(optimizer='adam', loss='mse')
    return model
```

```
def get_model_describe_f2():
    model = Sequential([
        Dense(64, input_dim=2, activation='relu', name='fc1'),
        Dense(32, activation='relu', name='fc2'),
        Dense(4, activation='relu', name='fc4'),
        Dense(1, name='fc6') # saída escalar
    ])
    model.compile(optimizer='adam', loss='mse')
    return model
```

```
model_describe_f1 = get_model_describe_f1()
model_describe_f2 = get_model_describe_f2()
```

✓ Treinando os Modelos e Analisando seus Desempenhos

```
hist_f1 = model_describe_f1.fit(Xf1_train, yf1_train, validation_data=(Xf1_val, yf1_val),
                                epochs=20, batch_size=32, verbose=True)
```

```
Epoch 1/20
5860/5860 — 4s 620us/step - loss: 0.0042 - val_loss: 2.0960e-05
Epoch 2/20
5860/5860 — 3s 594us/step - loss: 1.4240e-05 - val_loss: 2.2942e-05
Epoch 3/20
5860/5860 — 4s 622us/step - loss: 1.2562e-05 - val_loss: 1.2426e-05
Epoch 4/20
5860/5860 — 4s 627us/step - loss: 1.1318e-05 - val_loss: 5.6545e-06
Epoch 5/20
5860/5860 — 3s 593us/step - loss: 9.3186e-06 - val_loss: 3.9610e-06
Epoch 6/20
5860/5860 — 4s 595us/step - loss: 6.1043e-06 - val_loss: 5.2906e-06
Epoch 7/20
5860/5860 — 4s 598us/step - loss: 5.3924e-06 - val_loss: 7.8815e-06
```

```

Epoch 8/20
5860/5860 ————— 4s 597us/step - loss: 6.3483e-06 - val_loss: 1.3014e-05
Epoch 9/20
5860/5860 ————— 4s 597us/step - loss: 4.9211e-06 - val_loss: 7.5399e-06
Epoch 10/20
5860/5860 ————— 4s 620us/step - loss: 3.9738e-06 - val_loss: 2.6912e-06
Epoch 11/20
5860/5860 ————— 4s 605us/step - loss: 3.6600e-06 - val_loss: 4.3279e-06
Epoch 12/20
5860/5860 ————— 4s 621us/step - loss: 3.8602e-06 - val_loss: 3.3044e-06
Epoch 13/20
5860/5860 ————— 4s 598us/step - loss: 3.4233e-06 - val_loss: 2.5043e-06
Epoch 14/20
5860/5860 ————— 4s 677us/step - loss: 3.0854e-06 - val_loss: 2.7093e-06
Epoch 15/20
5860/5860 ————— 4s 651us/step - loss: 2.7758e-06 - val_loss: 3.8425e-06
Epoch 16/20
5860/5860 ————— 4s 677us/step - loss: 2.7330e-06 - val_loss: 1.4540e-06
Epoch 17/20
5860/5860 ————— 4s 603us/step - loss: 2.5728e-06 - val_loss: 4.1482e-06
Epoch 18/20
5860/5860 ————— 4s 619us/step - loss: 3.0143e-06 - val_loss: 1.1984e-06
Epoch 19/20
5860/5860 ————— 4s 605us/step - loss: 2.3622e-06 - val_loss: 1.2064e-06
Epoch 20/20
5860/5860 ————— 4s 602us/step - loss: 2.3244e-06 - val_loss: 1.4722e-06

```

```

hist_f2 = model_describe_f2.fit(Xf2_train, yf2_train, validation_data=(Xf2_val, yf2_val),
                                epochs=20, batch_size=32, verbose=True)

```

```

Epoch 1/20
5860/5860 ————— 4s 565us/step - loss: 3264.0376 - val_loss: 2.9568
Epoch 2/20
5860/5860 ————— 3s 566us/step - loss: 1.9955 - val_loss: 0.7029
Epoch 3/20
5860/5860 ————— 3s 562us/step - loss: 0.6973 - val_loss: 0.4130
Epoch 4/20
5860/5860 ————— 3s 559us/step - loss: 0.4451 - val_loss: 0.4648
Epoch 5/20
5860/5860 ————— 3s 568us/step - loss: 0.3640 - val_loss: 0.5958
Epoch 6/20
5860/5860 ————— 3s 562us/step - loss: 0.3091 - val_loss: 0.1891
Epoch 7/20
5860/5860 ————— 3s 561us/step - loss: 0.2630 - val_loss: 0.3109
Epoch 8/20
5860/5860 ————— 3s 574us/step - loss: 0.2940 - val_loss: 0.3151
Epoch 9/20
5860/5860 ————— 3s 575us/step - loss: 0.2392 - val_loss: 0.1257
Epoch 10/20
5860/5860 ————— 3s 570us/step - loss: 0.2254 - val_loss: 0.1872
Epoch 11/20
5860/5860 ————— 3s 562us/step - loss: 0.2433 - val_loss: 0.1591
Epoch 12/20
5860/5860 ————— 4s 599us/step - loss: 0.2051 - val_loss: 0.1509
Epoch 13/20
5860/5860 ————— 3s 566us/step - loss: 0.2088 - val_loss: 0.2416
Epoch 14/20
5860/5860 ————— 3s 562us/step - loss: 0.1987 - val_loss: 0.1408
Epoch 15/20
5860/5860 ————— 3s 589us/step - loss: 0.1945 - val_loss: 0.1792
Epoch 16/20
5860/5860 ————— 3s 565us/step - loss: 0.1919 - val_loss: 0.1453
Epoch 17/20
5860/5860 ————— 3s 575us/step - loss: 0.1919 - val_loss: 0.1201
Epoch 18/20
5860/5860 ————— 3s 564us/step - loss: 0.1919 - val_loss: 0.1087
Epoch 19/20
5860/5860 ————— 3s 567us/step - loss: 0.1793 - val_loss: 0.1273
Epoch 20/20
5860/5860 ————— 3s 572us/step - loss: 0.1901 - val_loss: 0.4486

```

```
plt.figure(figsize=(12,5))
```

```

plt.subplot(1,2,1)
plt.plot(hist_f1.history['loss'], label='Treinamento')
plt.plot(hist_f1.history['val_loss'], label='Validação')
plt.title('Function 1 - MSE x Epochs')
plt.xlabel('Epochs')
plt.ylabel('MSE')
plt.legend()

```

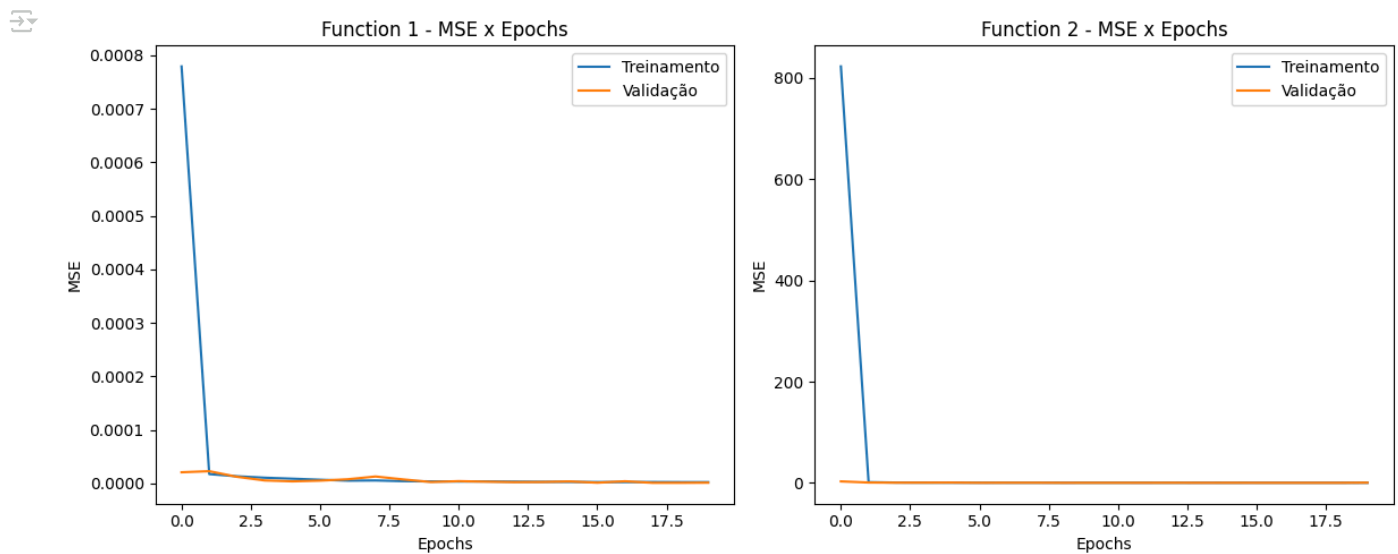
```

plt.subplot(1,2,2)
plt.plot(hist_f2.history['loss'], label='Treinamento')
plt.plot(hist_f2.history['val_loss'], label='Validação')
plt.title('Function 2 - MSE x Epochs')
plt.xlabel('Epochs')

```

```
plt.ylabel('MSE')
plt.legend()
```

```
plt.tight_layout()
plt.show()
```



✓ Aplicando Modelo aos dados de Teste e gerando Predict Points

```
z1_pred = model_describe_f1.predict(Xf1_val).reshape(yf1_val.shape)
```

```
1954/1954 ————— 1s 270us/step
```

```
z2_pred = model_describe_f2.predict(Xf2_val).reshape(yf2_val.shape)
```

```
1954/1954 ————— 1s 275us/step
```

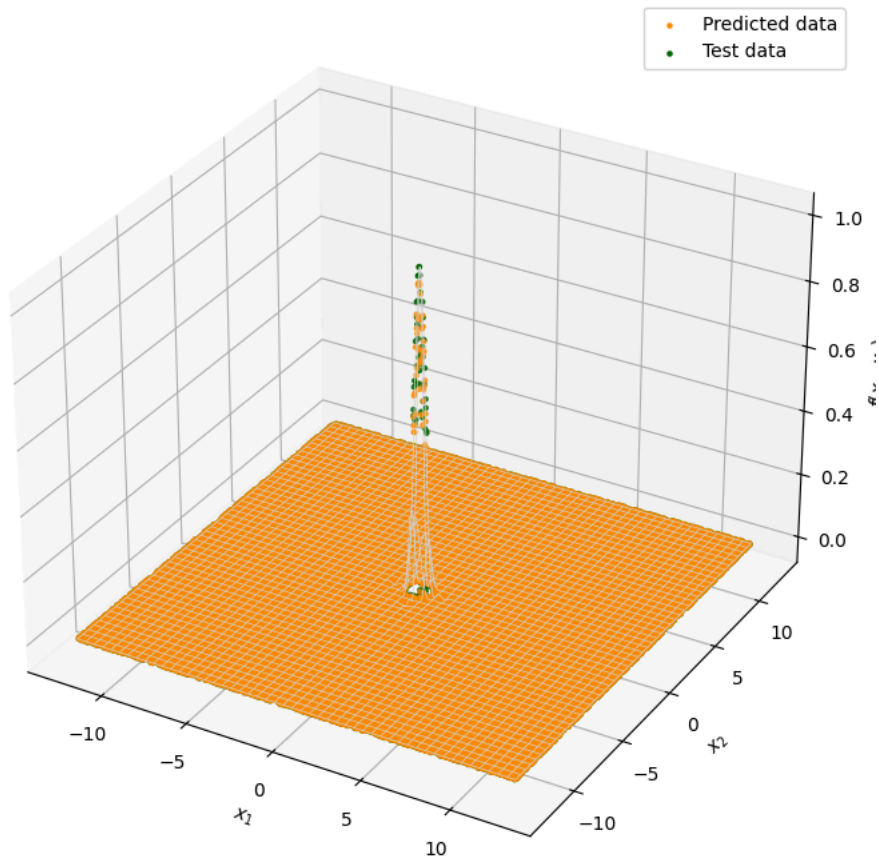
✓ Plots - Comparação Predicted Data x Test Data

```
fig, ax = plt.subplots(figsize=(10, 7), subplot_kw=dict(projection='3d'))

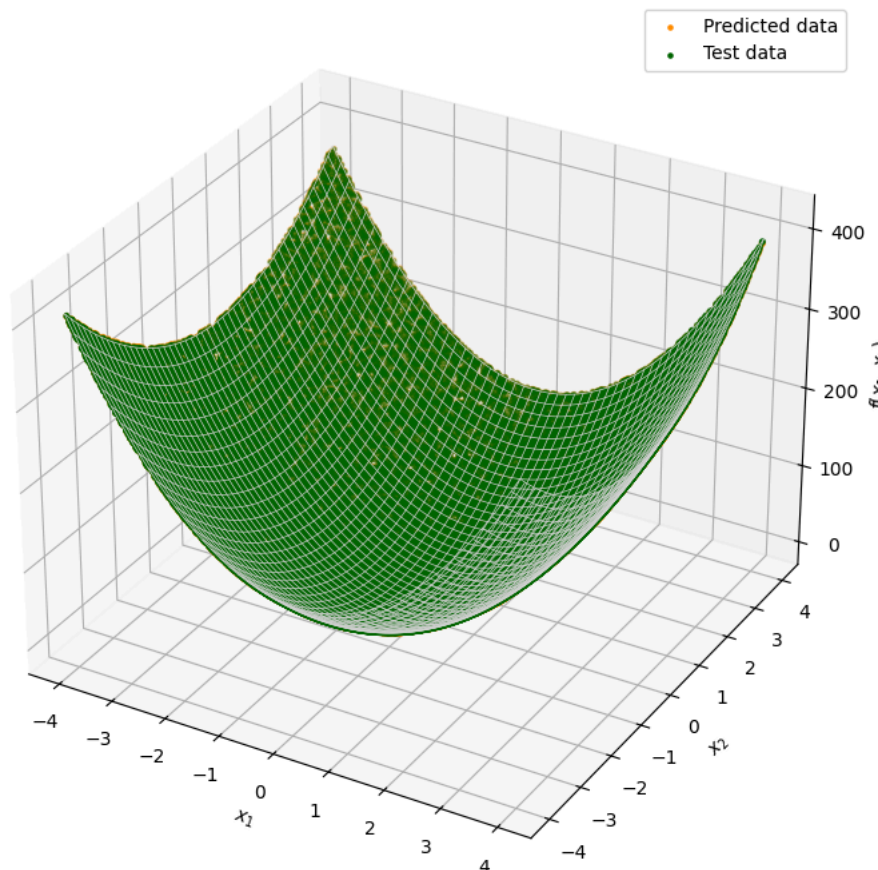
ax.plot_wireframe(x1_f1, x2_f1, y, linewidths=0.5, color='lightgrey')
ax.scatter(Xf1_val[:,0], Xf1_val[:,1], z1_pred, s=5, color='darkorange', label='Predicted data')
ax.scatter(Xf1_val[:,0], Xf1_val[:,1], yf1_val, s=5, color='darkgreen', label='Test data')

ax.set(
    xlabel='$x_1$',
    ylabel='$x_2$',
    zlabel='$f(x_1, x_2)$'
)

plt.legend()
plt.tight_layout()
plt.show()
```



```
fig, ax = plt.subplots(figsize=(10, 7), subplot_kw=dict(projection='3d'))
ax.plot_wireframe(x1_f2, x2_f2, y2, linewidths=0.5, color='lightgrey')
ax.scatter(Xf2_val[:,0], Xf2_val[:,1], z2_pred, s=5, color='darkorange', label='Predicted data')
ax.scatter(Xf2_val[:,0], Xf2_val[:,1], yf2_val, s=5, color='darkgreen', label='Test data')
ax.set(
    xlabel='$x_1$',
    ylabel='$x_2$',
    zlabel='$f(x_1, x_2)$'
)
plt.legend()
plt.tight_layout()
plt.show()
```



Questão 5.2 - Classificação de Espirais

Considere o problema das espirais. Sendo a espiral 1 uma classe e a espiral 2 outra classe. Gere as curvas das espirais usando as seguintes equações:

Para espiral 1:

$$x = \frac{\theta}{4} \cos(\theta), \quad y = \frac{\theta}{4} \sin(\theta), \quad \theta \geq 0$$

Para espiral 2:

$$x = \left(\frac{\theta}{4} + 0.8\right) \cos(\theta), \quad y = \left(\frac{\theta}{4} + 0.8\right) \sin(\theta), \quad \theta \geq 0$$

Solucione este problema utilizando uma rede perceptron de múltiplas camadas. Gere a partir das equações os dados para treinamento e teste. Determine a matriz de confusão.

Questão 5.3 - Classificação de Padrões Bidimensionais

Considere o problema de classificação de padrões bidimensionais constituído neste caso de 5 padrões. A distribuição dos padrões tem como base um quadrado centrado na origem interceptando os eixos nos pontos +1 e -1 de cada eixo. Os pontos +1 e -1 de cada eixo são os centros de quatro semicírculos que se interceptam no interior do quadrado, originando quatro classes. As regiões de não interseção formam a quinta classe. Após gerar aleatoriamente dados que venham formar estas distribuições de dados, selecione um conjunto de treinamento e um conjunto de validação. Defina uma arquitetura da rede perceptron a ser usada para solução do problema. Treine a rede perceptron para classificar os padrões associados a cada uma das classes. Verifique o desempenho do classificador usando o conjunto de validação, calculando: a matriz de confusão, a acurácia, o recall, a precisão e o F1-score.

✓ Questão 5.4 - Predição de uma Série Temporal

Considere o problema de predição de uma série temporal definida como $x(n) = v(n) + \beta v(n-1)v(n-2)$, com média zero e variância dada por $\sigma_x^2 = \sigma_v^2 + \beta^2 \sigma_v^2$ onde $v(n)$ é um ruído branco gaussiano, como variância unitária e $\beta = 0.5$. Utilizando uma rede perceptrons de múltiplas camadas (feedforward) estime $\hat{x}(n+1) = f(x(n), x(n-1), x(n-2), x(n-3))$ usando como entrada o valor presente e os três últimos valores da série, isto é, no conjunto de treinamento utilize uma janela deslizante com n as três amostras anteriores. Isso corresponde as entradas da rede neural. Avalie o desempenho mostrando a curva da série temporal, a curva de predição e a curva do erro de predição $e(n+1) = x(n+1) - \hat{x}(n+1)$


```
# importing libraries
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.preprocessing.sequence import TimeseriesGenerator
from tensorflow.keras.layers import Dense, Activation, BatchNormalization
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

▼ Descrevendo as Funções do problema para o Python e Gerando os Dados

```
# parameters based on the problem 5.4
n_seed = 42
beta = 0.5
deviation = 1
n_samples = 10000

# generating white noise using numpy.random.Generator with a fixed seed
rng = np.random.default_rng(seed=n_seed)
v = rng.normal(beta, deviation, n_samples)
x = v + beta * np.roll(v, -1) * np.roll(v, -2)

look_back = 4

X_data = []
y_data = []

for i in range(look_back, len(x) - 1):
    X_data.append(x[i - look_back:i])
    y_data.append(x[i + 1])

X_data = np.array(X_data)
y_data = np.array(y_data)
```

▼ Separando os dados em Treino e Teste

```
split_index = int(0.8 * len(X_data)) # 80% dos dados para treino
X_train, X_test = X_data[:split_index], X_data[split_index:]
y_train, y_test = y_data[:split_index], y_data[split_index:]

print(f"Formato de X_train: {X_train.shape}")
print(f"Formato de y_train: {y_train.shape}")
print(f"Formato de X_test: {X_test.shape}")
print(f"Formato de y_test: {y_test.shape}")

↗ Formato de X_train: (7996, 4)
Formato de y_train: (7996,)
Formato de X_test: (1999, 4)
Formato de y_test: (1999,)

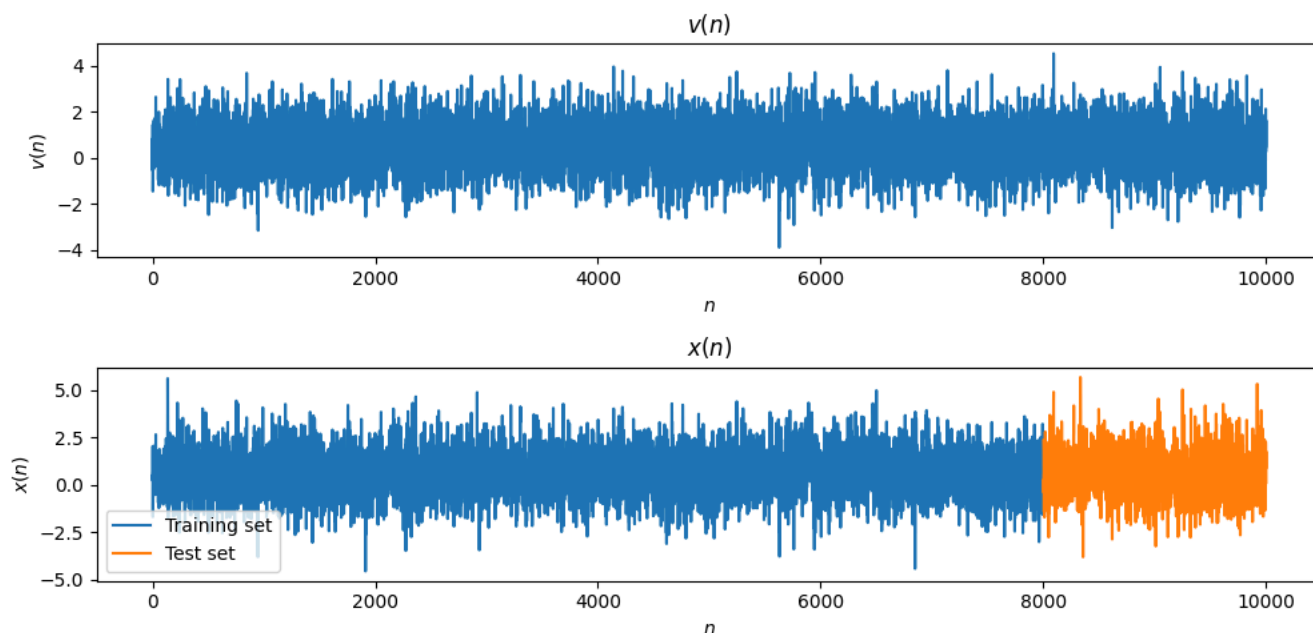
# plotting the data
fig, axes = plt.subplots(nrows=2, figsize=(10, 5))
split_x_index = int(0.8 * len(x))

axes[0].plot(v)
axes[0].set(
    title='$v(n)$',
    xlabel='$n$',
    ylabel='$v(n)$'
)

axes[1].plot(x[:split_x_index], label='Training set')
axes[1].plot(np.arange(n_samples-0.2*len(x), n_samples), x[split_x_index:], label='Test set')

axes[1].legend()
axes[1].set(
    title='$x(n)$',
    xlabel='$n$',
    ylabel='$x(n)$'
)

plt.tight_layout()
plt.show()
```



Colocando os dados em escala

```
# Normalizar os dados (boa prática para redes neurais)
scaler_x = StandardScaler()
X_train_scaled = scaler_x.fit_transform(X_train)
X_test_scaled = scaler_x.transform(X_test)

scaler_y = StandardScaler()
y_train_scaled = scaler_y.fit_transform(y_train.reshape(-1, 1))
y_test_scaled = scaler_y.transform(y_test.reshape(-1, 1))

print(f"Formato de X_train_scaled: {X_train_scaled.shape}")
print(f"Formato de y_train_scaled: {y_train_scaled.shape}")
```

```
Formato de X_train_scaled: (7996, 4)
Formato de y_train_scaled: (7996, 1)
```

Definindo nosso Modelo

```
# defining the model
def get_model_define_time_serie():
    model = Sequential()
    model.add(Dense(64, input_shape=(4, )))
    model.add(Activation('relu'))
    model.add(BatchNormalization())
    model.add(Dense(32))
    model.add(Activation('relu'))
    model.add(BatchNormalization())
    model.add(Dense(4))
    model.add(Activation('relu'))
    model.add(BatchNormalization())
    model.add(Dense(1))

    model.compile(
        loss='mse',
        optimizer='adam',
        metrics=['mse']
    )

    return model

model = get_model_define_time_serie()

/Users/efrainmpp/Documents/Mestrado/Neural-Network-PPGEEC2321/.venv/lib/python3.9/site-packages/keras/src/layers/core/de
super().__init__(activity_regularizer=activity_regularizer, **kwargs)

# checking the model summary
model.summary()
```


 Model: "sequential_21"

Layer (type)	Output Shape	Param #
dense_91 (Dense)	(None, 64)	320
activation_70 (Activation)	(None, 64)	0
batch_normalization_48 (BatchNormalization)	(None, 64)	256
dense_92 (Dense)	(None, 32)	2,080
activation_71 (Activation)	(None, 32)	0
batch_normalization_49 (BatchNormalization)	(None, 32)	128
dense_93 (Dense)	(None, 4)	132
activation_72 (Activation)	(None, 4)	0
batch_normalization_50 (BatchNormalization)	(None, 4)	16
dense_94 (Dense)	(None, 1)	5

Total params: 2,937 (11.47 KB)
 Trainable params: 2,737 (10.69 KB)
 Non-trainable params: 200 (800.00 B)

✓ Treinando o Modelo

```
history = model.fit(
    X_train, y_train,
    validation_data=(X_test, y_test),
    epochs=100,
    batch_size=32,
    verbose=True,
    shuffle=False,
    callbacks=[EarlyStopping(monitor='val_loss', patience=10),
               ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=5, min_lr=1e-6)]
)
```

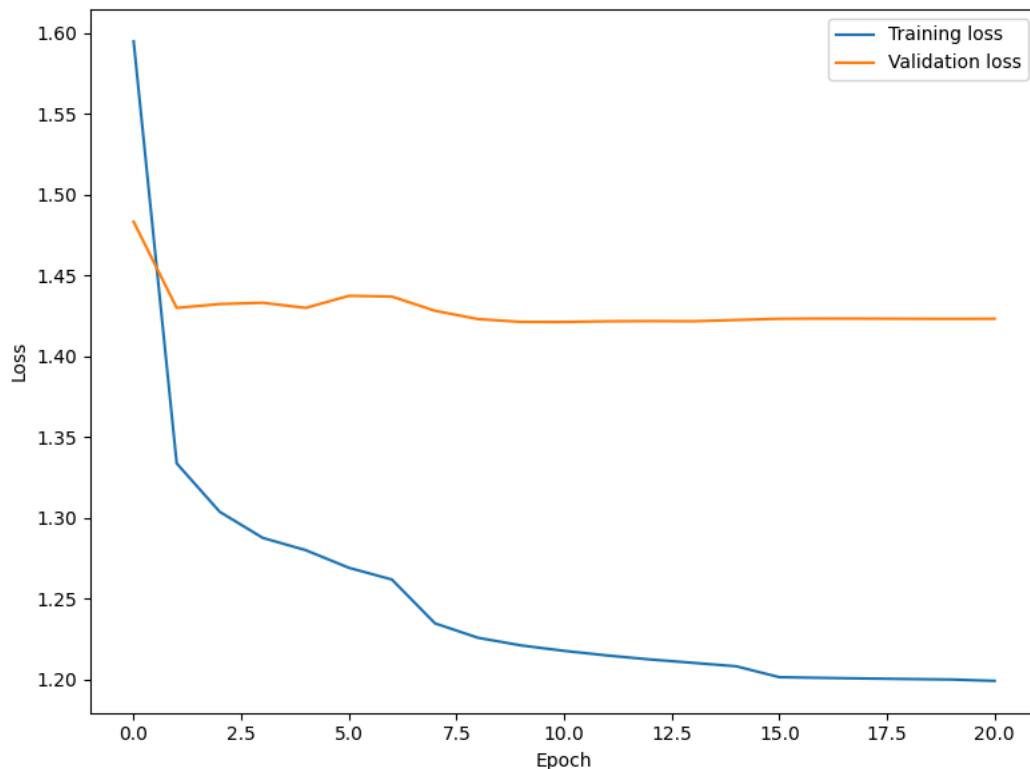
 Epoch 1/100
 250/250 ————— 1s 1ms/step - loss: 1.7942 - mse: 1.7942 - val_loss: 1.4833 - val_mse: 1.4833 - learning_ra
 Epoch 2/100
 250/250 ————— 0s 886us/step - loss: 1.3206 - mse: 1.3206 - val_loss: 1.4300 - val_mse: 1.4300 - learning_
 Epoch 3/100
 250/250 ————— 0s 871us/step - loss: 1.2851 - mse: 1.2851 - val_loss: 1.4324 - val_mse: 1.4324 - learning_
 Epoch 4/100
 250/250 ————— 0s 864us/step - loss: 1.2680 - mse: 1.2680 - val_loss: 1.4332 - val_mse: 1.4332 - learning_
 Epoch 5/100
 250/250 ————— 0s 862us/step - loss: 1.2608 - mse: 1.2608 - val_loss: 1.4300 - val_mse: 1.4300 - learning_
 Epoch 6/100
 250/250 ————— 0s 862us/step - loss: 1.2504 - mse: 1.2504 - val_loss: 1.4375 - val_mse: 1.4375 - learning_
 Epoch 7/100
 250/250 ————— 0s 851us/step - loss: 1.2437 - mse: 1.2437 - val_loss: 1.4370 - val_mse: 1.4370 - learning_
 Epoch 8/100
 250/250 ————— 0s 865us/step - loss: 1.2312 - mse: 1.2312 - val_loss: 1.4283 - val_mse: 1.4283 - learning_
 Epoch 9/100
 250/250 ————— 0s 856us/step - loss: 1.2187 - mse: 1.2187 - val_loss: 1.4231 - val_mse: 1.4231 - learning_
 Epoch 10/100
 250/250 ————— 0s 854us/step - loss: 1.2126 - mse: 1.2126 - val_loss: 1.4214 - val_mse: 1.4214 - learning_
 Epoch 11/100
 250/250 ————— 0s 900us/step - loss: 1.2083 - mse: 1.2083 - val_loss: 1.4213 - val_mse: 1.4213 - learning_
 Epoch 12/100
 250/250 ————— 0s 861us/step - loss: 1.2050 - mse: 1.2050 - val_loss: 1.4217 - val_mse: 1.4217 - learning_
 Epoch 13/100
 250/250 ————— 0s 859us/step - loss: 1.2025 - mse: 1.2025 - val_loss: 1.4219 - val_mse: 1.4219 - learning_
 Epoch 14/100
 250/250 ————— 0s 849us/step - loss: 1.2005 - mse: 1.2005 - val_loss: 1.4218 - val_mse: 1.4218 - learning_
 Epoch 15/100
 250/250 ————— 0s 862us/step - loss: 1.1983 - mse: 1.1983 - val_loss: 1.4225 - val_mse: 1.4225 - learning_
 Epoch 16/100
 250/250 ————— 0s 862us/step - loss: 1.1947 - mse: 1.1947 - val_loss: 1.4233 - val_mse: 1.4233 - learning_
 Epoch 17/100
 250/250 ————— 0s 896us/step - loss: 1.1939 - mse: 1.1939 - val_loss: 1.4235 - val_mse: 1.4235 - learning_
 Epoch 18/100
 250/250 ————— 0s 870us/step - loss: 1.1933 - mse: 1.1933 - val_loss: 1.4234 - val_mse: 1.4234 - learning_
 Epoch 19/100
 250/250 ————— 0s 1ms/step - loss: 1.1927 - mse: 1.1927 - val_loss: 1.4233 - val_mse: 1.4233 - learning_ra
 Epoch 20/100
 250/250 ————— 0s 874us/step - loss: 1.1922 - mse: 1.1922 - val_loss: 1.4232 - val_mse: 1.4232 - learning_
 Epoch 21/100
 250/250 ————— 0s 877us/step - loss: 1.1915 - mse: 1.1915 - val_loss: 1.4233 - val_mse: 1.4233 - learning_

```
# plotting error curves
fig, ax = plt.subplots(figsize=(8, 6))

ax.plot(history.history['loss'], label='Training loss')
ax.plot(history.history['val_loss'], label='Validation loss')

ax.legend()
ax.set(
    ylabel='Loss',
    xlabel='Epoch'
)

plt.legend()
plt.tight_layout()
plt.show()
```



Validando nosso Modelo

```
# Fazer previsões no conjunto de teste (escalonado)
y_pred_scaled = model.predict(X_test_scaled)

# Desnormalizar as previsões e os valores reais
y_pred = scaler_y.inverse_transform(y_pred_scaled)
y_test_original = scaler_y.inverse_transform(y_test_scaled)

# Calcular o erro de predição
erro = y_test_original.flatten() - y_pred.flatten()
```



63/63 — 0s 547us/step

```
# Visualizar os resultados
plt.figure(figsize=(12, 8))

# Curva da Série Temporal Real (no conjunto de teste)
plt.subplot(2, 1, 1)
plt.plot(range(len(y_test_original)), y_test_original, label='Série Temporal Real (x(n))')
plt.plot(range(len(y_pred)), y_pred, label='Predição  $\hat{x}(n+1)$ ', color='orange')
plt.title('Série Temporal Real (Conjunto de Teste) x Serie Temporal Predita')
plt.xlabel('Amostra (n)')
plt.ylabel('x(n)')
plt.legend()

# Curva do Erro de Predição
plt.subplot(2, 1, 2)
plt.plot(range(len(erro)), erro, label='Erro de Predição', color='red')
plt.title('Erro de Predição')
plt.xlabel('Tempo (n)')
```

```
plt.ylabel('$e(n+1) = x(n+1) - \hat{x}(n+1)$')
plt.axhline(y=0, color='black', linestyle='--')
plt.legend()

plt.tight_layout()
plt.show()

# Calcular e imprimir o Erro Quadrático Médio no conjunto de teste (original)
mse_original = np.mean(errors?)
```