Universidade Federal de Uberlândia
Instituto de Matemática e Estatística

Licenciatura em Matemática

Redes Neurais Artificiais: Uma Abordagem Didática e Aplicações em Problemas de Matemática Aplicada

Thamires Sartori Pereira

Uberlândia-MG
2026

Thamires Sartori Pereira

Redes Neurais Artificiais: Uma Abordagem Didática e Aplicações em Problemas de Matemática Aplicada

Trabalho de conclusão de curso apresentado à Coordenação do Curso de Licenciatura em Matemática como requisito parcial para obtenção do grau de Licenciado em Matemática.

Orientador: Santos Alberto Enriquez Remigio

Uberlândia-MG

2026

Universidade Federal de Uberlândia
Instituto de Matemática e Estatística

Coordenação do Curso de Licenciatura em Matemática

A banca examinadora, conforme abaixo assinado, certifica a adequação deste trabalho de conclusão de curso para obtenção do grau de Licenciado em Matemática.

Uberlândia, ____ de _____ de 20____

BANCA EXAMINADORA

_____

Santos Alberto Enriquez Remigio

_____

Nome completo do segundo membro da banca de defesa

_____

Nome completo do terceiro membro da banca de defesa

Uberlândia-MG

2026

Agradecimentos

Agradeço, primeiramente, aos meus pais, Ligia e Wagner, por me apoiarem de infinitas maneiras ao longo de toda a minha trajetória acadêmica. Agradeço por me aconselharem, por me darem coragem e força para continuar, por acreditarem em mim e na minha capacidade, e por sempre rezarem pela minha saúde e pelo meu sucesso. Agradeço, também, por me oferecerem um lar sempre que necessário, sendo meu porto seguro em todos os momentos, mesmo de longe.

Aos meus familiares, pelo apoio, incentivo e compreensão durante os momentos de maior dificuldade. Em especial, agradeço à minha avó e à minha tia, Esmeraldina e Vanessa, por sempre acreditarem em mim e me incentivarem a continuar, mesmo quando o caminho parecia difícil.

À Pedro Henrique, pelo apoio, incentivo e companheirismo ao longo dessa trajetória. Agradeço por ter acreditado em meu potencial e por ter me aconselhado, em um momento importante da minha vida, a ingressar na Universidade Federal de Uberlândia. Sou grata por todo o apoio oferecido durante esses anos, pelas conversas, conselhos e pela presença nos momentos bons e difíceis, que contribuíram para que eu pudesse seguir em frente com mais segurança e confiança.

À Universidade Federal de Uberlândia, pela oportunidade de formação e por proporcionar os conhecimentos e experiências que contribuíram para minha formação acadêmica e profissional. Ao PET-Matemática, pelo fomento, pelo apoio e pela promoção de uma educação de qualidade, além das experiências e aprendizados que contribuíram para a minha formação.

Ao meu orientador, Santos, pela orientação, disponibilidade, paciência e pelas contribuições fundamentais para o desenvolvimento deste trabalho. Agradeço por ser mais do que um professor, por me aconselhar não apenas nos estudos, mas também em minhas decisões, sempre com um carinho que muitas vezes se aproximou de um cuidado paterno.

Aos meus amigos e colegas, que estiveram presentes ao longo dessa caminhada, compartilhando momentos de aprendizado, dificuldades e conquistas e, principalmente, tornando essa trajetória mais leve. Em especial, agradeço à Maria Eduarda, amiga de outra vida, com quem compartilhei momentos de tristeza, reflexão, amor e felicidade, sempre dividindo o carinho e a dedicação pela educação e pela vida. Obrigada por ter estado comigo até aqui e por continuar ao meu lado nas próximas trajetórias.

Aos professores que fizeram parte da minha formação, por todo o conhecimento transmitido e por contribuírem, direta ou indiretamente, para a minha construção acadêmica e profissional.

Resumo

Este trabalho apresenta um estudo introdutório sobre redes neurais artificiais, com ênfase nos fundamentos matemáticos que sustentam sua estrutura e seu processo de aprendizagem. O objetivo central consistiu em compreender, sob uma perspectiva matemática, como esses modelos operam, destacando especialmente o papel da combinação linear e do gradiente na construção e no treinamento de redes neurais. Para isso, foram retomados conceitos de álgebra linear, cálculo diferencial e geometria analítica, estabelecendo-se uma base teórica adequada para a interpretação do neurônio artificial, das funções de ativação, da função de custo e do algoritmo de retropropagação. Inicialmente, discutiu-se a modelagem do neurônio artificial a partir da combinação linear entre entradas, pesos e viés, evidenciando sua relação com retas, hiperplanos e fronteiras de decisão. Em seguida, foram apresentadas algumas das principais funções de ativação utilizadas em redes neurais, ressaltando-se a importância da não linearidade para a capacidade de representação do modelo. Posteriormente, abordaram-se a função de custo, o gradiente e a regra da cadeia, mostrando como esses elementos se articulam no processo de retropropagação do erro e na atualização dos parâmetros por meio do método do gradiente descendente. Como forma de ilustrar concretamente os conceitos estudados, foram desenvolvidas duas aplicações computacionais em Python. A primeira consistiu em um problema de ajuste linear entre temperaturas em graus Celsius e Fahrenheit, no qual a rede aprendeu satisfatoriamente a relação esperada entre entrada e saída. A segunda envolveu um problema de classificação não linear no plano, em que a rede deveria distinguir pontos localizados no interior e no exterior de uma circunferência. Nesse caso, observou-se a importância da camada oculta e das funções de ativação não lineares para a aprendizagem de uma fronteira de decisão mais compatível com a geometria do problema. Em ambas as aplicações, a formulação matricial mostrou-se particularmente útil para organizar os cálculos e favorecer a implementação computacional. Assim, conclui-se que o estudo de redes neurais pode constituir uma oportunidade relevante de articulação entre a Matemática e temas contemporâneos da computação, contribuindo para uma compreensão mais fundamentada desses modelos e abrindo possibilidades para aprofundamentos futuros em arquiteturas mais gerais e em novas aplicações.

Palavras-chave: Redes Neurais Artificiais; Combinação Linear; Gradiente; Gradiente Descente; Retropropagação; Matemática Aplicada. .

Abstract

This work presents an introductory study on artificial neural networks, with emphasis on the mathematical foundations that support their structure and learning process. The main objective was to understand how these models operate from a mathematical perspective, especially highlighting the role of linear combination and gradient in the construction and training of neural networks. To this end, concepts from linear algebra, differential calculus, and analytic geometry were revisited, establishing an appropriate theoretical basis for the interpretation of the artificial neuron, activation functions, the cost function, and the backpropagation algorithm. Initially, the modeling of the artificial neuron was discussed based on the linear combination of inputs, weights, and bias, emphasizing its relationship with lines, hyperplanes, and decision boundaries. Next, some of the main activation functions used in neural networks were presented, highlighting the importance of nonlinearity for the representational capacity of the model. Subsequently, the cost function, gradient, and chain rule were addressed, showing how these elements are articulated in the backpropagation process and in the updating of parameters through the gradient descent method. As a way of concretely illustrating the concepts studied, two computational applications were developed in Python. The first consisted of a linear fitting problem involving temperatures in degrees Celsius and Fahrenheit, in which the network successfully learned the expected relationship between input and output. The second involved a nonlinear classification problem in the plane, in which the network had to distinguish points located inside and outside a circle. In this case, the importance of the hidden layer and nonlinear activation functions for learning a decision boundary more compatible with the geometry of the problem was observed. In both applications, the matrix formulation proved particularly useful for organizing the calculations and supporting the computational implementation. Thus, it is concluded that the study of neural networks may constitute a relevant opportunity to connect Mathematics with contemporary topics in computing, contributing to a more grounded understanding of these models and opening possibilities for future developments in more general architectures and new applications.

Keywords: Artificial Neural Networks; Linear Combination; Gradient; Gradient Descent; Backpropagation; Applied Mathematics..

Sumário

Lista de Figuras

1. Introdução

1.1 Contextualização das redes neurais artificiais

Uma rede neural artificial pode ser entendida, de modo geral, como um modelo construído para aprender a classificar um conjunto de dados em duas ou mais categorias, de modo a produzir uma saída que auxilie na tomada de decisão. A partir de exemplos apresentados durante o treinamento, a rede busca identificar regularidades nos dados e estabelecer critérios que lhe permitam associar novas entradas a classes ou respostas específicas. Assim, ainda que suas aplicações sejam variadas, uma de suas ideias centrais consiste em receber informações, processá-las e, com base nesse processamento, decidir entre diferentes possibilidades.

Para tornar essa ideia mais concreta, suponha que se queira decidir se vai chover ou não a partir de informações como temperatura e umidade do ar. Se utilarmos os registros de vários dias anteriores, indicando os valores dessas variáveis e se, de fato, choveu ou não em cada caso, então a rede neural pode ser treinada com esses exemplos. A partir desse conjunto de dados, ela busca identificar padrões que relacionem as entradas observadas ao resultado esperado, aprendendo a associar novas informações a uma possível decisão. Desse modo, ao receber os dados de um novo dia, a rede pode produzir uma saída indicando a previsão de chuva ou de ausência de chuva com base no que aprendeu durante o treinamento.

Exemplos como esse ajudam a compreender por que, nas últimas décadas, as redes neurais artificiais passaram a ocupar um lugar de destaque no desenvolvimento de tecnologias voltadas à realização de tarefas como classificação, reconhecimento de padrões, previsão, processamento de linguagem e apoio à tomada de decisão. Esse crescimento está relacionado, entre outros fatores, ao aumento da capacidade computacional disponível, à ampliação do volume de dados e ao aperfeiçoamento dos métodos de treinamento, elementos que contribuíram para tornar esses modelos cada vez mais presentes em diferentes setores da sociedade.

De modo geral, as redes neurais artificiais foram concebidas a partir de uma inspiração no funcionamento do sistema nervoso biológico, especialmente na maneira como neurônios se conectam e transmitem sinais. Evidentemente, não se trata de uma reprodução fiel da estrutura cerebral, mas de uma abstração que procura representar, de forma simplificada, a recepção de estímulos, a transmissão de informações e a produção de respostas. Assim, os chamados neurônios artificiais são organizados em camadas e interligados de modo a formar

uma estrutura capaz de receber dados de entrada, processá-los e produzir uma saída.

Essa inspiração biológica contribuiu para a formulação de modelos computacionais voltados à aprendizagem a partir de exemplos, um *regime supervisionado*. Com isso, redes neurais passaram a ser empregadas em uma ampla variedade de contextos, e a diversidade dessas aplicações mostra que tais modelos se consolidaram como ferramentas relevantes em áreas como Computação, Engenharia, Economia, Biologia, Saúde e Matemática aplicada.

Diante desse cenário, torna-se importante reconhecer que as redes neurais artificiais não são apenas um tema de interesse tecnológico, mas também um campo de estudo que conversa com diferentes áreas do conhecimento. Seu desenvolvimento histórico, inspirado em redes neurais biológicas e ampliado pelo avanço dos recursos computacionais, ajuda a explicar por que esses modelos se tornaram tão significativos no mundo contemporâneo e por que continuam despertando interesse em pesquisas e aplicações cada vez mais diversificadas.

1.2 JUSTIFICATIVA E OBJETIVOS DO ESTUDO

Embora as redes neurais artificiais sejam frequentemente apresentadas como ferramentas da Computação e da inteligência artificial, seu funcionamento está apoiado em uma estrutura fortemente matemática. Apesar disso, nem sempre a matemática recebe a devida atenção, já que muitos contatos iniciais com o tema privilegiam o uso prático dos modelos, seus resultados e suas aplicações, sem explicitar com clareza os processos matemáticos que tornam possível seu funcionamento. Como consequência, muitas vezes se conhece o que a rede é capaz de fazer, mas não se compreende como ela opera internamente.

É justamente nessa lacuna que se insere este trabalho. A proposta não é enfatizar apenas a implementação computacional das redes neurais, mas evidenciar que sua estrutura depende de ideias matemáticas fundamentais, sem as quais não é possível compreender de maneira consistente o processamento das informações e o mecanismo de aprendizagem. Nesse sentido, o estudo volta-se para dois elementos centrais: a combinação linear e o gradiente.

A combinação linear aparece na própria organização interna dos neurônios artificiais, pois é por meio dela que as entradas são ponderadas e reunidas para compor o valor processado em cada unidade. Já o gradiente desempenha papel decisivo no treinamento, uma vez que orienta o ajuste dos parâmetros da rede na busca pela redução do erro. Esses dois conceitos, portanto, constituem partes essenciais da lógica matemática que sustenta o funcionamento das redes neurais artificiais.

Desse modo, o enfoque adotado neste TCC consiste na análise desses fundamentos, buscando tornar mais visível a matemática presente em um tema que frequentemente é tratado de maneira predominantemente técnica ou instrumental — com a utilização de funções que já entregam todo o processo da rede neural em seus algoritmos. A intenção é mostrar que compreender redes neurais artificiais exige mais do que observar seus usos na Computação: exige examinar a estrutura matemática que lhes dá suporte e que permite explicar como tais modelos recebem dados, os transformam e aprendem com eles.

1.3 Organização do trabalho

Este trabalho foi desenvolvido a partir de estudo bibliográfico sobre redes neurais artificiais, tendo como base materiais de naturezas distintas, como slides, artigos e livros. Essa diversidade de fontes permitiu construir uma compreensão gradual do tema, desde seus aspectos mais introdutórios até seus fundamentos matemáticos mais diretamente relacionados ao funcionamento e ao treinamento desses modelos. Além disso, ao longo do desenvolvimento do estudo, foram elaborados códigos em Python com a finalidade de apoiar a visualização de conceitos, testar exemplos e contribuir para uma compreensão mais concreta de algumas das ideias discutidas no texto.

A organização do trabalho foi pensada de modo a conduzir o leitor desde uma apresentação geral das redes neurais artificiais até a análise de elementos matemáticos específicos que sustentam sua estrutura e seu processo de aprendizagem. Nesse sentido, a introdução tem a função de situar o tema no cenário contemporâneo, apresentando as redes neurais como modelos amplamente utilizados em diferentes áreas, bem como destacar sua inspiração nas redes neurais biológicas. Ainda nesse capítulo introdutório, explicita-se a justificativa do estudo, evidenciando a importância de compreender a base matemática desses modelos.

Nos capítulos seguintes, são desenvolvidos os conceitos fundamentais necessários para a compreensão de uma rede neural artificial. Inicialmente, discute-se a estrutura do neurônio artificial e os elementos que compõem seu funcionamento, de modo a construir a base conceitual para as etapas posteriores. Em seguida, dedica-se atenção especial à combinação linear, entendida como parte essencial do processamento realizado pelos neurônios, uma vez que é por meio dela que as entradas, os pesos e o viés se articulam na produção do valor que será transformado pela função de ativação.

Na continuidade, o trabalho volta-se ao estudo do gradiente, destacando seu papel no processo de treinamento da rede neural. Nessa etapa, busca-se mostrar como a noção de gradiente está relacionada ao ajuste dos parâmetros do modelo e à redução do erro, permitindo compreender de forma mais rigorosa como a rede aprende a partir dos dados apresentados. Assim, combinação linear e gradiente constituem o núcleo matemático desta monografia, pois são tratados como conceitos centrais para a interpretação da lógica interna das redes neurais artificiais.

Além da discussão teórica, o trabalho também procura aproximar esses fundamentos de situações aplicáveis à matemática aplicada. Para isso, ao longo do texto são considerados dois problemas, com o objetivo mobilizar os conceitos aprendidos em exemplos específicos. Essa articulação entre fundamentação teórica e aplicação busca mostrar que a matemática envolvida nas redes neurais não se limita a uma formulação abstrata, mas se manifesta de modo efetivo em contextos nos quais esses modelos são empregados.

Por fim, o trabalho se encerra com as considerações finais, nas quais são retomados os principais aspectos desenvolvidos ao longo dos capítulos, destacando-se a relevância de uma leitura matematicamente orientada das redes neurais artificiais. Desse modo, procura-se sintetizar as contribuições do estudo e reforçar a importância de compreender esses modelos não apenas

como ferramentas computacionais, mas também como construções fundamentadas em ideias matemáticas essenciais.

2. Preliminares Matemáticos

Para iniciar o estudo sobre Rede Neurais, é preciso estabelecer previamente alguns conceitos fundamentais de Matemática e Estatística. Em particular, a compreensão de modelos neurais, bem como de seus procedimentos de ajuste e análise, exige familiaridade com noções de estatística básica, álgebra linear e cálculo diferencial, pois esses conteúdos aparecem de forma recorrente na formulação de funções, no tratamento de dados e na descrição de algoritmos de otimização.

Assim, este capítulo reúne os principais preliminares matemáticos que serão utilizados ao longo do texto, com o objetivo de uniformizar a notação adotada e garantir que os resultados apresentados nos capítulos seguintes sejam compreendidos de maneira clara e consistente. Ao final deste capítulo, espera-se que o leitor disponha do arcabouço necessário para acompanhar, com segurança, as definições e discussões que serão aprofundadas na sequência do trabalho, consolidando os conhecimentos.

2.1 Álgebra Linear

2.1.1 Matrizes em $\mathbb{R}^{m\times n}$

Definição 1. *Uma matriz de ordem $m \times n$ é um arranjo de $m \cdot n$ números, dispostos em m linhas e n colunas. Toda matriz tem associada um nome que pode ser representado com uma letra maiúscula, por exemplo: A. Para representar os elementos de A escrevemos a_{ij}, onde $a_{ij} \in \mathbb{R}$, $i = 1, \ldots, m$ e $j = 1, \ldots, n$, ou*

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

Existe o caso particular da matriz com somente uma coluna, que é denominada ***matriz coluna****, ou* ***vetor coluna****, e uma matriz com somente uma linha é denominada* ***matriz linha****, ou* ***vetor linha****.*

A entrada na linha i e na coluna j de uma matriz A também é geralmente denotada por $[A]_{ij}$. Assim, podemos simplificar a notação da matriz acima, como

$$[A]_{ij} = a_{ij}.$$

Quando uma matriz contém o mesmo número de linhas e colunas, dizemos que é uma ***matriz quadrada****.*

Definição 2. *Se A for uma matriz $m \times n$ qualquer, então a* ***transposta de*** *A, denotada por A^T, é definida como a matriz $n \times m$ que resulta da troca das linhas com as colunas de A; ou seja, a primeira coluna de A^T é a primeira linha de A, a segunda coluna de A^T é a segunda linha de A, e assim por diante.*

Usando notação matricial, a definição acima pode ser matematicamente escrita como segue:

$$[A^T]_{ij} = [A]_{ji}.$$

Exemplo 2.1.1. *Sendo*

$$A = \begin{bmatrix} 1 & 4 & -2 \\ 0 & 5 & 3 \end{bmatrix}, \quad \textit{então} \quad A^T = \begin{bmatrix} 1 & 0 \\ 4 & 5 \\ -2 & 3 \end{bmatrix}.$$

Definição 3. *Uma matriz quadrada é dita simétrica se $A = A^T$.*

Ou seja, uma matriz quadrada $A = [a_{ij}]$ é simétrica se, e só se, $[A]_{ij} = [A]_{ji}$ para quaisquer valores de i e j.

Definição 4. *A* matriz identidade *de ordem n, denotada por I_n, é a matriz quadrada cujos elementos são dados por*

$$[I_n]_{ij} = \begin{cases} 1, & \textit{se } i = j, \\ 0, & \textit{se } i \neq j. \end{cases}$$

Ou seja, trata-se da matriz que possui 1 em todas as posições da diagonal principal e 0 em todas as outras entradas.

A seguir falaremos de algumas operações envolvendo matrizes que são bastante utilizadas.

Definição 5. *Se A e B são matrizes do mesmo tipo $m \times n$, então a* ***soma*** *$A + B$ é a matriz $m \times n$ obtida somando as entradas de B às entradas correspondentes de A, e a* ***diferença*** *$A - B$ é a matriz $m \times n$ obtida subtraindo as entradas de B das entradas correspondentes de A.*

Em notação matricial, se A e B têm o mesmo tipo, as operações indicadas acima são dadas por

$$A + B = [A]_{ij} + [B]_{ij} = a_{ij} + b_{ij}, \qquad A - B = [A]_{ij} - [B]_{ij} = a_{ij} - b_{ij}.$$

Exemplo 2.1.2. *Sendo*

$$A = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix} \quad e \quad B = \begin{bmatrix} 3 & 2 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix},$$

temos:

$$A+B=\begin{bmatrix}2+3 & 1+2 & 0+0\\ 1+1 & 2+0 & 1+1\\ 1+0 & 1+1 & 2+0\end{bmatrix}=\begin{bmatrix}5 & 3 & 0\\ 2 & 2 & 2\\ 1 & 2 & 2\end{bmatrix},$$

$$A-B=\begin{bmatrix}2-3 & 1-2 & 0-0\\ 1-1 & 2-0 & 1-1\\ 1-0 & 1-1 & 2-0\end{bmatrix}=\begin{bmatrix}-1 & -1 & 0\\ 0 & 2 & 0\\ 1 & 0 & 2\end{bmatrix}.$$

Observe que as operações de soma e subtração não estão definidas para matrizes de tipos distintos.

Definição 6 (Multiplicação de Matriz por um escalar)**.** *Se A for uma matriz e c um escalar, então o* ***produto*** *cA é a matriz obtida multiplicando cada entrada da matriz A por c. Dizemos que a matriz cA é um* ***múltiplo escalar*** *de A. Em notação matricial, $[cA]_{ij} = c\,a_{ij}$.*

Definição 7 (Multiplicação de Matrizes)**.** *Sejam A uma matriz de ordem $m \times n$ e B uma matriz de ordem $n \times p$. O* ***produto*** *AB é definido apenas quando o número de colunas de A é igual ao número de linhas de B. A matriz resultante $C = AB$ terá ordem $m \times p$, e cada elemento c_{ij} é calculado como:*

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

Ou seja, o elemento c_{ij} é o produto escalar da i-ésima linha de A pela j-ésima coluna de B.

Exemplo 2.1.3. *Sejam*

$$A=\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{bmatrix} \quad e \quad B=\begin{bmatrix}1 & 2\\ 0 & 1\\ 1 & 0\end{bmatrix}.$$

O produto AB é possível, pois A é 2×3 e B é 3×2, resultando em uma matriz 2×2.

$$AB=\begin{bmatrix}1\cdot 1+2\cdot 0+3\cdot 1 & 1\cdot 2+2\cdot 1+3\cdot 0\\ 4\cdot 1+5\cdot 0+6\cdot 1 & 4\cdot 2+5\cdot 1+6\cdot 0\end{bmatrix}=\begin{bmatrix}4 & 4\\ 10 & 13\end{bmatrix}.$$

Portanto,

$$AB=\begin{bmatrix}4 & 4\\ 10 & 13\end{bmatrix}.$$

A matriz identidade funciona como o elemento neutro da multiplicação de matrizes: para qualquer matriz A de ordem compatível, vale

$$I_n A = A \qquad \text{e} \qquad AI_n = A.$$

Definição 8. *Seja A e B matrizes $m \times n$. O produto de Hadamard de A e B é definido por $[A \odot B]_{ij} = [A]_{ij}[B]_{ij}$ para todo $1 \leq i \leq m$,$1 \leq j \leq n$.*

Exemplo 2.1.4. *Considere as matrizes*

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad e \quad B = \begin{bmatrix} 5 & -1 \\ 2 & 0 \end{bmatrix}.$$

Pela definição do produto de Hadamard, multiplicamos os elementos correspondentes de A e B. Assim,

$$A \odot B = \begin{bmatrix} 1 \cdot 5 & 2 \cdot (-1) \\ 3 \cdot 2 & 4 \cdot 0 \end{bmatrix} = \begin{bmatrix} 5 & -2 \\ 6 & 0 \end{bmatrix}.$$

Esse exemplo mostra que, no produto de Hadamard, a multiplicação é feita entrada por entrada, diferentemente do produto usual de matrizes.

Teorema 1. *Seja A e B matrizes $m \times n$, então $[A \odot B]_{ij} = [B \odot A]_{ij}$.*

Teorema 2. ***O produto de Hadamard é linear.*** *Se $\alpha \in C$, e A, B e C são matrizes $m \times m$, então $C \odot (A + B) = C \circ A + C \circ B$. Além disso, $\alpha(A \circ B) = (\alpha A) \odot B = A \odot (\alpha B)$.*

Definição 9 (Determinante de uma matriz)**.** *Seja A uma matriz quadrada de ordem n. O* determinante *de A é um número real, denotado por* $\det(A)$ *ou simplesmente* $\det A$*, associado à matriz e que fornece informações importantes sobre suas propriedades algébricas.*

No caso de uma matriz 2×2,

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix},$$

o determinante é definido por

$$\det(A) = ad - bc.$$

Exemplo 2.1.5. *Considere a matriz*

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}.$$

Então,

$$\det(A) = (1)(4) - (2)(3) = 4 - 6 = -2.$$

Para matrizes de ordem maior, o determinante pode ser calculado por diferentes métodos, como a expansão por cofatores ou a redução por operações elementares de linha. Esses procedimentos serão explorados mais adiante, quando tratarmos de matrizes, sistemas lineares e transformações lineares de forma mais aprofundada.

A notação de posto de uma matriz A aparece de diversas maneiras na literatura brasileira, como $posto(A)$, $\rho(A)$ ou $rank(A)$, mas nesta obra denotaremos pela última nomenclatura.

Definição 10 (Posto de uma Matriz)**.** *O $rank(A)$ indica a quantidade máxima de linhas ou colunas linearmente independentes da matriz. Ele está diretamente relacionado à ideia de dimensão do espaço vetorial gerado pelas linhas ou colunas da matriz.*

O posto pode ser calculado por meio de: Redução de Gauss ou Cálculo do maior menor com determinante $\neq 0$.

A redução de Gauss consiste em aplicar sucessivas *operações elementares nas linhas* de uma matriz para transformá-la em uma forma mais simples, chamada **forma escalonada**. Essas operações são:

1. Trocar a posição de duas linhas;
2. Multiplicar uma linha por um número real não nulo;
3. Somar a uma linha um múltiplo de outra linha.

Ao final do processo, o número de **linhas não nulas** da matriz escalonada é igual ao **posto** da matriz original.

Exemplo 2.1.6 (Cálculo do posto de uma matriz pelo método de Gauss)**.** *Considere a matriz*

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 1 & 1 & 1 \end{bmatrix}.$$

Vamos aplicar o método de eliminação de Gauss para encontrar o seu posto.

1. Subtraímos 2 *vezes a primeira linha da segunda:*

$$L_2 \leftarrow L_2 - 2L_1 \Rightarrow \begin{bmatrix} 1 & 2 & 3 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}.$$

2. Subtraímos a primeira linha da terceira:

$$L_3 \leftarrow L_3 - L_1 \Rightarrow \begin{bmatrix} 1 & 2 & 3 \\ 0 & 0 & 0 \\ 0 & -1 & -2 \end{bmatrix}.$$

3. Para facilitar a análise, podemos multiplicar a terceira linha por -1*:*

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 0 & 0 \\ 0 & 1 & 2 \end{bmatrix}.$$

Percebemos que existem ***duas linhas não nulas****. Logo, o posto da matriz é:*

$$rank(A) = 2.$$

Interpretação: o posto 2 indica que há duas linhas (ou colunas) linearmente independentes, isto é, as demais podem ser obtidas como combinações lineares dessas duas.

Definição 11 (Matriz Inversa)**.** *Seja A uma matriz quadrada de ordem n. Dizemos que A é* ***invertível*** *(ou* ***não singular****) se existir uma matriz B da mesma ordem tal que:*

$$A \cdot B = B \cdot A = I,$$

onde I é a matriz identidade. Essa matriz B é chamada de ***matriz inversa*** *de A e é denotada por A^{-1}.*

Uma matriz é invertível se, e somente se, seu determinante é diferente de zero, isto é, $\det(A) \neq 0$.

Exemplo 2.1.7. *Considere a matriz*

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}.$$

O determinante de A é:

$$\det(A) = 1 \cdot 4 - 2 \cdot 3 = -2.$$

Como $\det(A) \neq 0$*, A é invertível.*

Exemplo 2.1.8. *Vamos calcular a inversa da matriz*

$$A = \begin{bmatrix} 2 & 1 \\ 5 & 3 \end{bmatrix}$$

utilizando o ***método de Gauss-Jordan****. Segundo este método, o objetivo é transformar a parte esquerda em I, e no processo a parte direita se tornará A^{-1}.*

$$\left[\begin{array}{cc|cc} 2 & 1 & 1 & 0 \\ 5 & 3 & 0 & 1 \end{array}\right]$$

1. *Dividimos a primeira linha por 2:*

$$L_1 \leftarrow \tfrac{1}{2}L_1 \Rightarrow \left[\begin{array}{cc|cc} 1 & \frac{1}{2} & \frac{1}{2} & 0 \\ 5 & 3 & 0 & 1 \end{array}\right]$$

2. *Subtraímos 5 vezes a primeira linha da segunda:*

$$L_2 \leftarrow L_2 - 5L_1 \Rightarrow \left[\begin{array}{cc|cc} 1 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & -\frac{5}{2} & 1 \end{array}\right]$$

3. *Multiplicamos a segunda linha por 2:*

$$L_2 \leftarrow 2L_2 \Rightarrow \left[\begin{array}{cc|cc} 1 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 1 & -5 & 2 \end{array}\right]$$

4. *Subtraímos metade da segunda linha da primeira:*

$$L_1 \leftarrow L_1 - \tfrac{1}{2}L_2 \Rightarrow \left[\begin{array}{cc|cc} 1 & 0 & 3 & -1 \\ 0 & 1 & -5 & 2 \end{array}\right]$$

Portanto, a matriz inversa é:

$$A^{-1} = \begin{bmatrix} 3 & -1 \\ -5 & 2 \end{bmatrix}.$$

Definição 12 (Forma Matricial de um Sistema Linear)**.** *Considere um sistema linear composto por m equações e n incógnitas. A forma geral deste sistema é dada por*

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1, \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2, \\ \quad \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m, \end{cases}$$

onde a_{ij} são os coeficientes do sistema, x_j são as incógnitas e b_i os termos independentes.

Podemos representar este mesmo sistema de equação em forma estendida *(ou aumentada) pela matriz*

$$\left[A \mid \vec{b}\right] = \left[\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} & b_m \end{array}\right].$$

Com isso, podemos representar o sistema de maneira compacta utilizando a notação matricial:

$$A\,\vec{x} = \vec{b},$$

onde:

- *A é a* ***matriz dos coeficientes****, composta pelos coeficientes das incógnitas do sistema;*
- *$\vec{x}$ é o* ***vetor das incógnitas****;*
- *$\vec{b}$ é o* ***vetor dos termos independentes****.*

Essa forma é especialmente útil pois permite aplicar métodos matriciais (como eliminação de Gauss, cálculo de inversa ou decomposições) para resolver o sistema de forma mais geral e organizada.

Exemplo 2.1.9. *Considere o sistema linear:*

$$\begin{cases} 2x_1 + x_2 = 5, \\ x_1 + 3x_2 = 8. \end{cases}$$

Podemos escrever esse sistema na forma matricial:

$$A\,\vec{x} = \vec{b}, \quad \textit{onde} \quad A = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}, \quad \vec{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad \vec{b} = \begin{bmatrix} 5 \\ 8 \end{bmatrix}.$$

Assim,

$$\begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 8 \end{bmatrix}.$$

Essa representação permite utilizar propriedades de matrizes e vetores para resolver o sistema. Por exemplo, se A é uma matriz quadrada e invertível, podemos obter a solução única:

$$\vec{x} = A^{-1}\vec{b}.$$

Definição 13. *Matriz Definida Positiva*

Uma matriz quadrada $A \in \mathbb{R}^{n\times n}$ é dita ***definida positiva*** *se for simétrica e satisfizer:*

$$\vec{x}^T A\vec{x} > 0, \quad \textit{para todo } \vec{x} \in \mathbb{R}^n,\ \vec{x} \neq \vec{0}.$$

De forma análoga, dizemos que A é ***semidefinida positiva*** *se $\vec{x}^T A\vec{x} \geq 0$ para todo $\vec{x} \in \mathbb{R}^n$.*

2.1.2 VETORES EM $\mathbb{R}^n$

Como vimos anteriormente, vetores são casos particulares de matrizes e, neste trabalho, denotaremos vetores com letras minúsculas com uma setinha em cima, como em $\vec{u}, \vec{v}, \vec{w}$. Além disso, adotaremos a convenção de que todos os vetores são representados como vetores coluna. Por essa razão, em diversos momentos será necessário utilizar a transposição, escrevendo $\vec{v}^T$ quando precisarmos trabalhar com vetores linha.

A seguir falaremos de propriedades e operações comumente utilizadas com vetores.

Definição 14 (Combinação Linear)**.** *Sejam $\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n$ vetores de um espaço vetorial V. Dizemos que um vetor $\vec{u} \in V$ é uma* ***combinação linear*** *desses vetores se existirem números reais $a_1, a_2, \ldots, a_n$ tais que:*

$$\vec{u} = a_1\vec{v}_1 + a_2\vec{v}_2 + \cdots + a_n\vec{v}_n.$$

Os coeficientes $a_1, a_2, \ldots, a_n$ são chamados de coeficientes da combinação linear. Quando um vetor pode ser escrito como combinação linear de outros, dizemos que ele depende linearmente desses vetores.

Definição 15 (Vetores linearmente independentes)**.** *Em um espaço vetorial, dizemos que um conjunto de vetores é* ***linearmente independente*** *quando o único modo de escrever o vetor nulo como uma combinação linear desses vetores é tomando todos os coeficientes iguais a zero. Ou seja, os vetores $\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n$ são linearmente independentes se a equação*

$$\alpha_1\vec{v}_1 + \alpha_2\vec{v}_2 + \cdots + \alpha_n\vec{v}_n = \vec{0}$$

implica que $\alpha_1 = \alpha_2 = \cdots = \alpha_n = 0$.

Exemplo 2.1.10. *Considere os vetores* $\vec{u} = (1,0)^T$ *e* $\vec{v} = (0,1)^T$. *A equação* $\alpha\vec{u} + \beta\vec{v} = (0,0)$ *resulta em* $(\alpha, \beta) = (0,0)$, *logo esses vetores são linearmente independentes. Já os vetores* $\vec{u} = (1,2)$ *e* $\vec{v} = (2,4)$ *são linearmente dependentes, pois* $\vec{v} = 2\vec{u}$.

Exemplo 2.1.11. *Considere os vetores* $\vec{u} = (1,0,0)^T$, $\vec{v} = (0,1,0)^T$ *e* $\vec{w} = (0,0,1)^T$. *A única combinação linear que resulta no vetor nulo é*

$$\alpha\vec{u} + \beta\vec{v} + \gamma\vec{w} = (0,0,0) \Rightarrow (\alpha, \beta, \gamma) = (0,0,0) \Rightarrow \alpha = \beta = \gamma = 0,$$

portanto são linearmente independentes. Por outro lado, os vetores $\vec{u} = (1,1,1)^T$, $\vec{v} = (2,3,4)^T$ *e* $\vec{w} = (3,4,5)^T$ *são linearmente dependentes, pois um deles pode ser escrito como combinação linear dos outros.*

Definição 16 (Norma Euclidiana)**.** *A norma euclidiana (ou comprimento) de um vetor* $\vec{u} = (u_1, u_2, \ldots, u_n) \in \mathbb{R}^n$ *é dada por:*

$$\|\vec{u}\| = \sqrt{u_1^2 + u_2^2 + \cdots + u_n^2}.$$

A norma euclidiana representa a distância do vetor $\vec{u}$ *até a origem no espaço* $\mathbb{R}^n$.

Definição 17 (Vetor Unitário)**.** *Um vetor* $\vec{u} \in \mathbb{R}^n$ *é chamado de unitário quando possui norma igual a 1, isto é:*

$$\|\vec{u}\| = 1.$$

Dado um vetor não nulo $\vec{v}$, *podemos obter o vetor unitário* $\vec{u}$ *com a mesma direção e sentido de* $\vec{v}$ *dividindo-o pela sua norma:*

$$\vec{u} = \frac{\vec{v}}{\|\vec{v}\|}.$$

Definição 18 (Produto Escalar)**.** *Sejam* $\vec{u} = (u_1, u_2, \ldots, u_n)^T$ *e* $\vec{v} = (v_1, v_2, \ldots, v_n)^T$ *dois vetores em* $\mathbb{R}^n$. *O* ***produto escalar*** *entre* $\vec{u}$ *e* $\vec{v}$ *é definido por:*

$$\vec{u}^T \cdot \vec{v} = \sum_{i=1}^{n} u_i v_i = u_1 v_1 + u_2 v_2 + \cdots + u_n v_n.$$

Pode-se mostrar que o produto escalar admite a seguinte expressão equivalente:

$$\vec{u} \cdot \vec{v} = \|\vec{u}\| \, \|\vec{v}\| \cos(\theta),$$

onde θ *é o ângulo formado entre os vetores* $\vec{u}$ *e* $\vec{v}$. *Assim, o produto escalar mede o alinhamento entre dois vetores.*

Definição 19 (Transformação Linear). *Sejam V e W espaços vetoriais sobre o mesmo corpo $\mathbb{R}$. Uma função $T : V \to W$ é chamada de* ***transformação linear*** *se, para todos os vetores $\vec{u}, \vec{v} \in V$ e para todo escalar $\alpha \in \mathbb{R}$, valem as propriedades:*

$$(i)\ T(\vec{u}+\vec{v}) = T(\vec{u}) + T(\vec{v}), \quad (ii)\ T(\alpha\vec{u}) = \alpha T(\vec{u}).$$

Essas condições garantem que T preserva as operações de adição e multiplicação por escalar.

Exemplo 2.1.12. *Considere a transformação linear $T : \mathbb{R}^2 \to \mathbb{R}^2$ definida por*

$$T(\vec{x}) = A\vec{x}, \quad onde \quad A = \begin{bmatrix} 2 & 1 \\ 0 & 3 \end{bmatrix}.$$

Para o vetor

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix},$$

temos:

$$T(\vec{x}) = \begin{bmatrix} 2 & 1 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2x_1 + x_2 \\ 3x_2 \end{bmatrix}.$$

Por exemplo, se

$$\vec{x} = \begin{bmatrix} 1 \\ 2 \end{bmatrix},$$

então

$$T(\vec{x}) = \begin{bmatrix} 2(1)+2 \\ 3(2) \end{bmatrix} = \begin{bmatrix} 4 \\ 6 \end{bmatrix}.$$

Portanto, T transforma o vetor $\vec{x} = (1, 2)$ em $T(\vec{x}) = (4, 6)$, preservando as propriedades lineares de soma e multiplicação escalar.

Definição 20 (Transformação afim). *Uma transformação afim é uma aplicação da forma*

$$f(x) = Wx + b,$$

onde $W \in \mathbb{R}^{m\times n}$ e $b \in \mathbb{R}^m$.

Definição 21 (Espaço Vetorial). *Um espaço vetorial sobre $\mathbb{R}$ é um conjunto V não vazio, cujos elementos são chamados de vetores, munido de duas operações:*

- ***Adição vetorial*** *$+ : V \times V \to V$, que associa a cada par $(\vec{u}, \vec{v})$ um vetor $\vec{u} + \vec{v} \in V$;*
- ***Multiplicação por escalar*** *$\cdot : \mathbb{R} \times V \to V$, que associa a cada $c \in \mathbb{R}$ e $\vec{v} \in V$ um vetor $c\vec{v} \in V$.*

Para quaisquer $\vec{u}, \vec{v}, \vec{w} \in V$ e $a, b \in \mathbb{R}$, essas operações devem satisfazer:

1. *$\vec{u} + \vec{v} = \vec{v} + \vec{u}$ (comutatividade)*

2. $(\vec{u}+\vec{v})+\vec{w}=\vec{u}+(\vec{v}+\vec{w})$ *(associatividade)*

3. *Existe* $\vec{0}\in V$ *tal que* $\vec{v}+\vec{0}=\vec{v}$ *(elemento neutro da adição)*

4. *Para cada* $\vec{v}$ *existe* $-\vec{v}\in V$ *tal que* $\vec{v}+(-\vec{v})=\vec{0}$ *(inverso aditivo)*

5. $a(b\vec{v})=(ab)\vec{v}$

6. $1\cdot\vec{v}=\vec{v}$ *(elemento neutro da multiplicação)*

7. $a(\vec{u}+\vec{v})=a\vec{u}+a\vec{v}$

8. $(a+b)\vec{v}=a\vec{v}+b\vec{v}$

Exemplo 2.1.13. *O conjunto* $\mathbb{R}^2=\{(x_1,x_2):x_1,x_2\in\mathbb{R}\}$ *é um espaço vetorial. A adição é dada por*

$$(x_1,x_2)+(y_1,y_2)=(x_1+y_1,\,x_2+y_2),$$

e a multiplicação por escalar por

$$c(x_1,x_2)=(cx_1,\,cx_2).$$

Essas operações satisfazem todos os axiomas de um espaço vetorial.

Definição 22. *Uma* ***base*** *de um espaço vetorial* V *é um conjunto ordenado de vetores*

$$\mathcal{B}=\{\vec{v}_1,\vec{v}_2,\ldots,\vec{v}_n\}$$

tal que:

1. *Os vetores de* $\mathcal{B}$ *são* ***linearmente independentes****;*

2. *Os vetores de* $\mathcal{B}$ ***geram todo o espaço*** V*, isto é, qualquer vetor* $\vec{u}\in V$ *pode ser escrito de maneira única como uma combinação linear:*

$$\vec{u}=a_1\vec{v}_1+a_2\vec{v}_2+\cdots+a_n\vec{v}_n,$$

com $a_1,\ldots,a_n\in\mathbb{R}$.

Exemplo 2.1.14. *No espaço* $\mathbb{R}^2$*, a base mais comum é*

$$\{(1,0),(0,1)\},$$

pois qualquer vetor (x_1,x_2) *pode ser escrito como*

$$(x_1,x_2)=x_1(1,0)+x_2(0,1).$$

Definição 23 (Hiperplano). *Considere o espaço $\mathbb{R}^n$ e um vetor não nulo $\vec{\alpha} \in \mathbb{R}^n$.*

Chamamos de hiperplano linear *o conjunto*

$$H = \{\vec{v} \in \mathbb{R}^n : \ \vec{\alpha}^T \vec{v} = 0\}.$$

Mais geralmente, para $a \in \mathbb{R}$, chamamos de hiperplano afim *o conjunto*

$$J = \{\vec{v} \in \mathbb{R}^n : \ \vec{\alpha}^T \vec{v} = a\},$$

isto é, uma translação de um hiperplano linear.

A noção de hiperplano é fundamental no estudo de Redes Neurais Artificiais, pois ele descreve, do ponto de vista geométrico, ela cria divisões na minha dimensão de estudo. Em $\mathbb{R}^n$, um hiperplano (linear ou afim) possui dimensão $n-1$ e pode ser interpretado como uma superfície que separa o espaço em dois semiespaços.

2.2 Fundamentos de Cálculo e Análise

Nesta seção, apresentamos algumas definições e propriedades fundamentais de Cálculo e Análise que servirão de base para as discussões desenvolvidas ao longo do trabalho. Em particular, as noções introduzidas a seguir foram elaboradas com base, principalmente, no livro de Stewart [7], que constitui uma referência importante para o estudo de limites, continuidade, derivadas e suas interpretações geométricas e analíticas.

Definição 24. *Uma função $f : \mathbb{R} \to \mathbb{R}$ é uma regra que associa a cada número real x um único número real $f(x)$. Em outras palavras, para cada entrada $x \in \mathbb{R}$ existe uma única saída $f(x) \in \mathbb{R}$.*

Exemplo 2.2.1. *Considere a função $f : \mathbb{R} \to \mathbb{R}$ dada por*

$$f(x) = x^2.$$

Cada número real x é associado ao seu quadrado. Por exemplo: $f(2) = 4$, $f(-3) = 9$.

Definição 25 (Função Afim). *Uma função afim é uma aplicação $f : \mathbb{R} \to \mathbb{R}$ da forma*

$$f(x) = ax + b,$$

*onde $a, b \in \mathbb{R}$. O número a é chamado de **coeficiente angular** da função, pois indica a taxa de variação de f, enquanto b é chamado de **coeficiente linear**, representando o valor de f no ponto $x = 0$.*

Exemplo 2.2.2. *Considere a função afim*

$$f(x) = 2x - 3.$$

O coeficiente angular é $a = 2$, indicando que a cada aumento de 1 unidade em x, a função cresce 2 unidades. O coeficiente linear é $b = -3$, logo o gráfico cruza o eixo vertical no ponto $(0, -3)$.

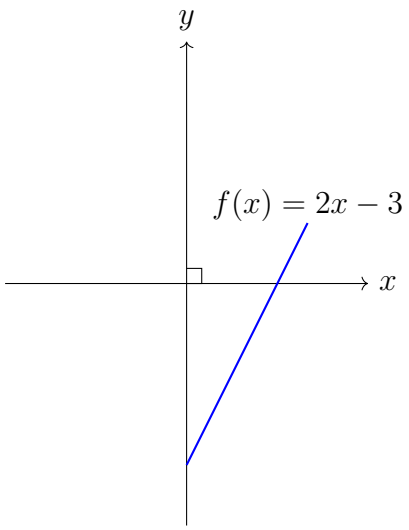


Figura 2.1: Gráfico de uma função afim $f(x) = 2x - 3$.
Fonte: Autoria própria.

Definição 26. *Uma função $f : \mathbb{R}^n \to \mathbb{R}$ atribui a cada vetor $\vec{x} = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$ um número real $f(\vec{x})$. Esse tipo de função é chamada de função escalar de várias variáveis.*

Exemplo 2.2.3. *Considere $f : \mathbb{R}^2 \to \mathbb{R}$ definida por*

$$f(x, y) = x^2 + y^2.$$

A cada ponto (x, y) do plano, associamos o valor $x^2 + y^2$.

Definição 27 (Função Vetorial)**.** *Uma função vetorial é uma regra ou aplicação*

$$\vec{f} : \mathbb{R} \to \mathbb{R}^n,$$

que associa a cada número real $t \in \mathbb{R}$ um vetor em $\mathbb{R}^n$. Em outras palavras,

$$\vec{f}(t) = \begin{pmatrix} f_1(t) \\ f_2(t) \\ \vdots \\ f_n(t) \end{pmatrix},$$

onde cada componente $f_i : \mathbb{R} \to \mathbb{R}$ é uma função real de variável real.

Exemplo 2.2.4. *Considere a função vetorial $\vec{f} : \mathbb{R} \to \mathbb{R}^2$ definida por*

$$\vec{f}(t) = \begin{pmatrix} t^2 \\ 2t + 1 \end{pmatrix}.$$

Para cada valor de t, a função retorna um vetor do plano. Por exemplo,

$$\vec{f}(1) = \begin{pmatrix} 1 \\ 3 \end{pmatrix}.$$

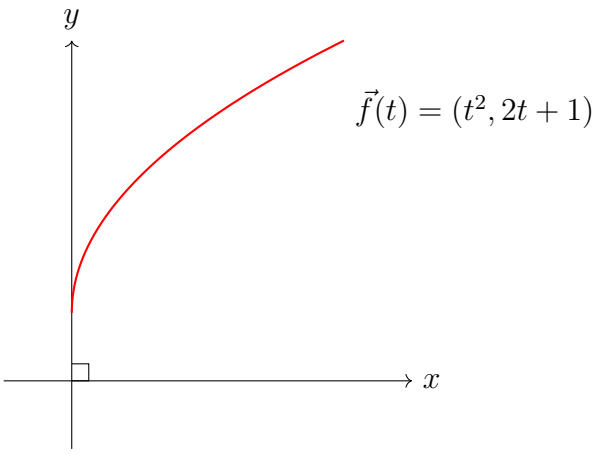


Figura 2.2: Gráfico da função vetorial $\vec{f}(t)$.
Fonte: Autoria própria.

Definição 28. *Sejam $g : A \subset \mathbb{R}^n \to B \subset \mathbb{R}^m$ e $f : B \subset \mathbb{R}^m \to C \subset \mathbb{R}^k$. Chamamos de* função composta *a aplicação em que primeiro aplicamos g ao ponto de entrada, e depois aplicamos f ao resultado.*

Assim, a função composta é dada por

$$h(x) = f(g(x)).$$

Exemplo 2.2.5. *Considere as funções*

$$g(x) = x + 1 \qquad e \qquad f(y) = y^2.$$

A função composta, aplicando primeiro g e depois f, é

$$f(g(x)) = (x+1)^2.$$

Se trocarmos a ordem, obtemos uma função diferente:

$$g(f(x)) = x^2 + 1.$$

Portanto, a ordem da composição importa.

Definição 29 (Limite de Funções)**.** *Seja $f : \mathbb{R} \to \mathbb{R}$ uma função real. Dizemos que o* ***limite*** *de $f(x)$ quando x tende a $a \in \mathbb{R}$ é o número real L se, para todo $\varepsilon > 0$, existir $\delta > 0$ tal que*

$$0 < |x - a| < \delta \quad \Rightarrow \quad |f(x) - L| < \varepsilon.$$

Neste caso, escrevemos

$$\lim_{x \to a} f(x) = L.$$

Seja $f : \mathbb{R}^n \to \mathbb{R}$ uma função. Dizemos que o limite de $f(\vec{x})$ quando $\vec{x}$ tende a $\vec{a} \in \mathbb{R}^n$ é o número real L se, para todo $\varepsilon > 0$, existir $\delta > 0$ tal que

$$0 < \|\vec{x} - \vec{a}\| < \delta \quad \Rightarrow \quad |f(\vec{x}) - L| < \varepsilon.$$

Nesse caso, escrevemos

$$\lim_{\vec{x} \to \vec{a}} f(\vec{x}) = L.$$

Essa definição generaliza a noção de limite para funções de várias variáveis.

Definição 30 (Continuidade)**.** *Uma função $f : \mathbb{R} \to \mathbb{R}$ é dita **contínua** em $a \in \mathbb{R}$ se*

$$\lim_{x \to a} f(x) = f(a).$$

Isto significa que o valor da função coincide com o limite quando a variável se aproxima de a.

Uma função $f : \mathbb{R}^n \to \mathbb{R}$ é contínua em $\vec{a} \in \mathbb{R}^n$ se

$$\lim_{\vec{x} \to \vec{a}} f(\vec{x}) = f(\vec{a}).$$

Assim como no caso unidimensional, a continuidade indica que não há "saltos" no valor da função ao redor do ponto $\vec{a}$.

2.2.1 DERIVADAS, GRADIENTES, REGRA DA CADEIA

Nesta seção discutimos a noção de derivada de uma função real de variável real, bem como a ideia de função diferenciável, fundamental no estudo do cálculo diferencial. A seguir, apresentamos a definição de função derivável para funções de R em R. Posteriormente, será apresentada uma formulação equivalente desse conceito.

Definição 31. *Seja $f : \mathbb{R} \to \mathbb{R}$. Dizemos que f é **derivável** em um ponto $a \in \mathbb{R}$ se existe o limite*

$$f'(a) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h}.$$

*Esse valor, quando existe, é chamado de **derivada de f em** a.*

A quantidade

$$\frac{f(a+h) - f(a)}{h}$$

representa a *taxa média de variação* da função em um pequeno intervalo ao redor de a. A derivada é o limite dessa taxa quando o intervalo se torna infinitamente pequeno.

INTERPRETAÇÃO GEOMÉTRICA

A derivada fornece a inclinação da **reta tangente** ao gráfico de f no ponto $(a, f(a))$.

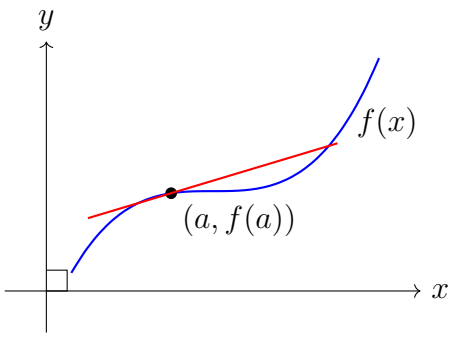


Figura 2.3: Representação geométrica de uma curva e de sua reta tangente em um ponto.
Fonte: Autoria própria.

Exemplo 2.2.6. *Considere a função* $f(x) = x^2$. *Então*

$$f'(a) = \lim_{h\to 0} \frac{(a+h)^2 - a^2}{h} = \lim_{h\to 0} \frac{2ah + h^2}{h} = \lim_{h\to 0}(2a + h) = 2a.$$

Assim, a derivada de f *no ponto* a *é* $f'(a) = 2a$.

A derivada em um ponto fornece informação local. Para estender essa noção, introduzimos o conceito de função diferenciável.

Definição 32 (Derivadas Parciais)**.** *Seja* $f : U \subset \mathbb{R}^n \to \mathbb{R}$ *uma função de várias variáveis, onde um ponto é escrito como* $\vec{x} = (x_1, x_2, \ldots, x_n)$. *A* ***derivada parcial*** *de* f *em relação à variável* x_i *é definida como o limite:*

$$\frac{\partial f}{\partial x_i}(\vec{x}) = \lim_{h\to 0} \frac{f(x_1, \ldots, x_i + h, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n)}{h},$$

quando esse limite existir.

Intuitivamente, mantemos todas as outras variáveis constantes e derivamos f *somente na direção da coordenada* x_i.

Exemplo 2.2.7. *Considere a função*

$$f(x, y) = x^2 y + 3y.$$

A derivada parcial de f *em relação a* x *é obtida tratando* y *como constante:*

$$\frac{\partial f}{\partial x}(x, y) = 2xy.$$

A derivada parcial em relação a y é obtida tratando x como constante:

$$\frac{\partial f}{\partial y}(x, y) = x^2 + 3.$$

Exemplo 2.2.8. *Para uma função de três variáveis,*

$$f(x, y, z) = xe^{yz},$$

temos:

$$\frac{\partial f}{\partial x} = e^{yz}, \qquad \frac{\partial f}{\partial y} = xze^{yz}, \qquad \frac{\partial f}{\partial z} = xye^{yz}.$$

Observe que, em cada derivada, apenas a variável correspondente sofre diferenciação; as demais são tratadas como constantes.

Definição 33 (Gradiente)**.** *Seja $f : U \subset \mathbb{R}^n \to \mathbb{R}$ uma função diferenciável. O **gradiente** de f no ponto $\vec{x} \in U$ é o vetor formado pelas derivadas parciais de f em relação a cada coordenada:*

$$\nabla f(\vec{x}) = \left(\frac{\partial f}{\partial x_1}(\vec{x}), \frac{\partial f}{\partial x_2}(\vec{x}), \ldots, \frac{\partial f}{\partial x_n}(\vec{x}) \right)^T.$$

Exemplo 2.2.9. *Considere a função $f(x, y) = x^2 + 3y^2$. Suas derivadas parciais são:*

$$\frac{\partial f}{\partial x} = 2x, \qquad \frac{\partial f}{\partial y} = 6y.$$

Logo, o gradiente é:

$$\nabla f(x, y) = (2x,\, 6y).$$

No ponto $(1, 1)$, temos:

$$\nabla f(1, 1) = (2,\, 6),$$

que aponta na direção de maior crescimento de f nesse ponto.

Definição 34. *Uma função $f : \mathbb{R} \to \mathbb{R}$ é dita **diferenciável** em $a \in \mathbb{R}$ se existe um número real L tal que*

$$\lim_{h \to 0} \frac{f(a + h) - f(a) - Lh}{h} = 0.$$

Nesse caso, dizemos que f é diferenciável em a e o valor L coincide com $f'(a)$.

A definição anterior pode ser estendida para funções de várias variáveis. Considere agora uma função

$$f : \mathbb{R}^n \to \mathbb{R}.$$

*Dizemos que f é **diferenciável** em um ponto $\vec{a} \in \mathbb{R}^n$ se existe um vetor*

$$\vec{L} \in \mathbb{R}^n$$

tal que

$$\lim_{\vec{h}\to\vec{0}} \frac{f(\vec{a}+\vec{h}) - f(\vec{a}) - \vec{L}\cdot\vec{h}}{\|\vec{h}\|} = 0,$$

onde $\vec{L}\cdot\vec{h}$ *denota o produto interno usual em* $\mathbb{R}^n$.

Nesse caso, dizemos que f *é diferenciável em* $\vec{a}$*, e o vetor* $\vec{L}$ *coincide com o* ***gradiente*** *de* f *no ponto* $\vec{a}$*, isto é,*

$$\vec{L} = \nabla f(\vec{a}).$$

Essa definição indica que, muito perto de $\vec{a}$*, a função pode ser aproximada linearmente por*

$$f(\vec{x}) \approx f(\vec{a}) + \nabla f(\vec{a}) \cdot (\vec{x} - \vec{a}),$$

o que generaliza a expressão unidimensional que, muito perto de a*, a função se comporta como a reta*

$$f(a) + f'(a)(x - a),$$

ou seja, a função pode ser aproximada linearmente.

Assim, no contexto do nosso trabalho a noção de diferenciabilidade em $\mathbb{R}^n$ será essencial para descrever aproximações lineares em vários parâmetros simultaneamente.

Além disso, a noção de **incremento** é fundamental no estudo do Cálculo Diferencial, pois descreve a variação efetiva de uma função quando sua variável de entrada sofre uma pequena alteração. Essa ideia será essencial para compreender derivadas direcionais, diferenciais e aproximações lineares.

Seja $x \in \mathbb{R}$. Quando a variável passa do valor x para o valor $x + \Delta x$, definimos

$$\Delta x$$

como o *incremento de* x. Trata-se simplesmente da diferença entre o novo valor e o valor inicial:

$$\Delta x = (x + \Delta x) - x.$$

Para uma função $f : \mathbb{R} \to \mathbb{R}$, o incremento correspondente à variação Δx é dado por

$$\Delta f = f(x + \Delta x) - f(x),$$

o qual representa a variação real da função em resposta à alteração na variável de entrada.

Exemplo 2.2.10. *Considere a função* $f(x) = x^2$*. Dado um incremento* Δx*, temos*

$$\Delta f = (x + \Delta x)^2 - x^2 = 2x\,\Delta x + (\Delta x)^2.$$

Para valores pequenos de Δx*, o termo* $(\Delta x)^2$ *se torna desprezível, de modo que*

$$\Delta f \approx 2x\,\Delta x,$$

o que corresponde à aproximação indicada pela derivada de f, isto é, $f'(x) = 2x$.

Para uma função $f : \mathbb{R}^n \to \mathbb{R}$, o incremento da variável é um vetor

$$\Delta\vec{x} = (\Delta x_1, \Delta x_2, \ldots, \Delta x_n).$$

O incremento da função é definido então por

$$\Delta f = f(\vec{x} + \Delta\vec{x}) - f(\vec{x}).$$

Se f é diferenciável no ponto $\vec{x}$, então incrementos suficientemente pequenos produzem a aproximação linear

$$\Delta f \approx \nabla f(\vec{x}) \cdot \Delta\vec{x},$$

mostrando que a variação da função pode ser aproximada pelo produto interno entre o gradiente e o incremento na variável.

Definição 35 (Derivada Direcional)**.** *Seja $f : U \subset \mathbb{R}^n \to \mathbb{R}$ diferenciável e seja $\vec{u}$ um vetor **unitário** ($\|\vec{u}\| = 1$). A **derivada direcional** de f em $\vec{x}$ na direção de $\vec{u}$ é dada por:*

$$D_{\vec{u}} f(\vec{x}) = \lim_{h \to 0} \frac{f(\vec{x} + h\vec{u}) - f(\vec{x})}{h},$$

quando o limite existir.

Se f é diferenciável em $\vec{x}$, então:

$$D_{\vec{u}} f(\vec{x}) = \nabla f(\vec{x})^T \vec{u},$$

isto é, a derivada direcional é o produto interno entre o gradiente e o vetor unitário.

Teorema 3. *Considere uma função $f : \mathbb{R}^n \to \mathbb{R}$, um ponto $\vec{a} \in \mathbb{R}^n$ e um vetor unitário $\vec{u}$. A derivada direcional de f na direção de $\vec{u}$ é*

$$D_{\vec{u}} f(\vec{a}) = \nabla f(\vec{a}) \cdot \vec{u}.$$

Usando a definição de produto interno, temos

$$\nabla f(\vec{a}) \cdot \vec{u} = \|\nabla f(\vec{a})\| \, \|\vec{u}\| \cos\theta = \|\nabla f(\vec{a})\| \cos\theta,$$

pois $\|\vec{u}\| = 1$. Assim, a derivada direcional é dada pelo produto da norma do gradiente com o cosseno do ângulo entre $\nabla f(\vec{a})$ e $\vec{u}$.

Desse teorema seque que, dado um ponto a, a direção de crescimento máximo da função é a direção do vetor gradiente, isto é, $u = \nabla f(a)$.

Definição 36 (Ponto crítico)**.** *Seja $f : U \subset \mathbb{R}^n \to \mathbb{R}$ diferenciável em U. Dizemos que $a \in U$ é um **ponto crítico** de f se*

$$\nabla f(a) = \vec{0},$$

ou se alguma das derivadas parciais de f não existir em a. Em outras palavras, em um ponto crítico a "taxa de variação"nas direções coordenadas é zero (ou não está definida).

Definição 37 (Máximo local). *Um ponto $a \in U$ é um **máximo local** de f se existe uma vizinhança V de a tal que*

$$f(a) \geq f(x) \quad \text{para todo } x \in V.$$

*Se a desigualdade for estrita para todo $x \in V \setminus \{a\}$, chamamos de **máximo local estrito**.*

Definição 38 (Mínimo local). *Um ponto $a \in U$ é um **mínimo local** de f se existe uma vizinhança V de a tal que*

$$f(a) \leq f(x) \quad \text{para todo } x \in V.$$

*Se a desigualdade for estrita para todo $x \in V \setminus \{a\}$, chamamos de **mínimo local estrito**.*

Teorema 4 (Regra da Cadeia). *Se g for derivável em x e f for derivável em $g(x)$, então a função composta $F = f \circ g$ definida por $F(x) = f(g(x))$ é derivável em x e F' é dada pelo produto*

$$F'(x) = f'(g(x)) \cdot g'(x).$$

Na notação de Leibniz, se $y = f(u)$ e $u = g(x)$ forem funções deriváveis, então

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}.$$

Definição 39 (Conjunto de Nível). *Seja $f : U \subset \mathbb{R}^n \to \mathbb{R}$ uma função escalar. Para um valor real fixo c, o **conjunto de nível** (ou **conjunto de contorno**) de f relativo a c é o subconjunto de U definido por*

$$L_c(f) \;=\; \{\, \vec{x} \in U : f(\vec{x}) = c \,\}.$$

Em particular, para $n = 2$ cada conjunto de nível é, em geral, uma curva no plano (chamada também de curva de nível*); para $n = 3$ os conjuntos de nível são superfícies.*

Exemplo 2.2.11. *Considere $f : \mathbb{R}^2 \to \mathbb{R}$ dada por*

$$f(x, y) = x^2 + y^2.$$

Os conjuntos de nível são

$$L_c(f) = \{(x, y) \in \mathbb{R}^2 : x^2 + y^2 = c\},$$

ou seja, circunferências centradas na origem quando $c > 0$. Para $c = 0$ temos o ponto $(0, 0)$; para $c < 0$ não há pontos reais no conjunto de nível.

Observações:

- Curvas de nível são muito úteis para visualizar a topografia de uma função de duas variáveis: regiões com curvas muito próximas indicam variação rápida.

- O gradiente ∇f é perpendicular às curvas de nível (em pontos onde $\nabla f \neq 0$), e aponta na direção de maior crescimento de f.

2.2.2 GRADIENTE DESCENDENTE

O método do *gradiente descendente* é um procedimento iterativo utilizado para encontrar mínimos (locais) de funções reais. Sua fundamentação é inteiramente baseada em conceitos de cálculo diferencial e pode ser apresentada de forma puramente matemática.

Inicialmente, consideremos uma função de uma variável real

$$f : \mathbb{R} \to \mathbb{R}.$$

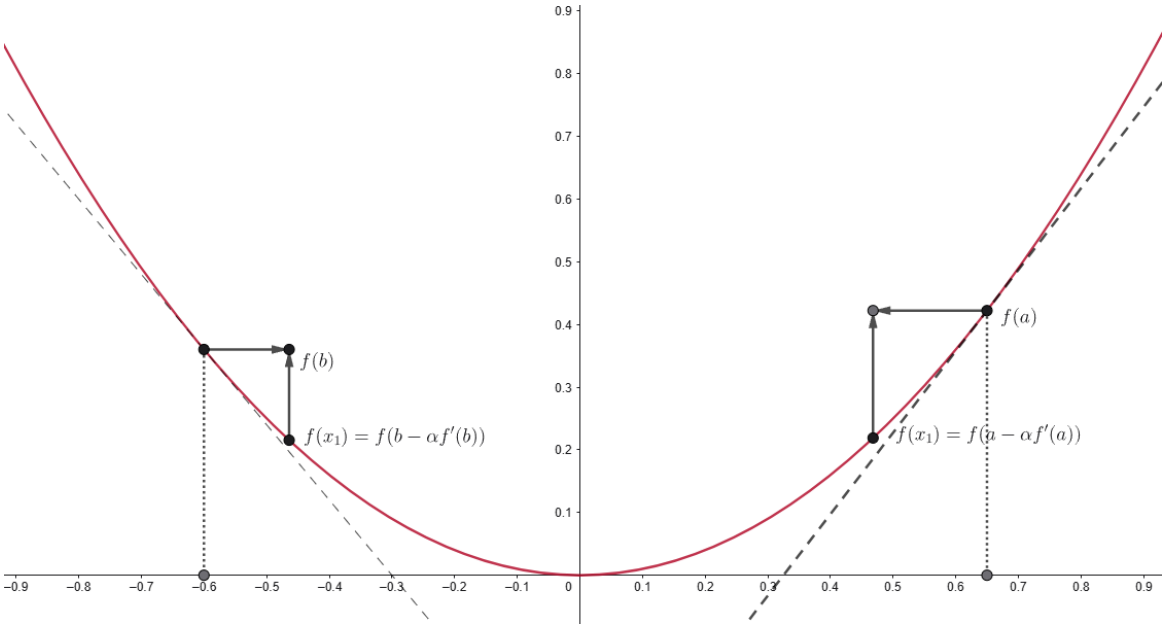


Figura 2.4: Representação do gradiente descendente
Fonte: Autoria própria.

Para tornar essa ideia mais concreta, a Figura 2.4 ilustra o comportamento do método em uma função quadrática. Nela, observam-se dois pontos do gráfico, indicados por a e b, nos quais foram traçadas as retas tangentes correspondentes. Essas tangentes fornecem uma aproximação linear local da função e ajudam a visualizar como a derivada orienta o próximo passo do método.

Na Figura 2.4, o ponto a, à direita do vértice da parábola, apresenta derivada positiva, isto é, $f'(a) > 0$. Nesse caso, a reta tangente é crescente e, para diminuir o valor da função, o método desloca o ponto para a esquerda, produzindo uma nova aproximação $x_1 = a - \alpha f'(a)$. Já no ponto b, à esquerda do vértice, a derivada é negativa, ou seja, $f'(b) < 0$. Assim, o termo $-\alpha f'(b)$ passa a representar um deslocamento para a direita. Em ambos os casos, o procedimento conduz o ponto em direção à região de mínimo da função.

Seja $x_0 \in \mathbb{R}$ um ponto do domínio de f. A derivada $f'(x_0)$ representa a inclinação da reta tangente ao gráfico de f no ponto $(x_0, f(x_0))$ e fornece informação sobre o comportamento

local da função. Se $f'(x_0) > 0$, a função é crescente em uma vizinhança de x_0; se $f'(x_0) < 0$, a função é decrescente nessa vizinhança. Portanto, para diminuir o valor da função, devemos nos deslocar na direção oposta ao sinal da derivada. Isso motiva a seguinte regra de atualização:

$$x_{k+1} = x_k - \alpha f'(x_k),$$

•

onde $\alpha \in (0, 1)$ é um parâmetro definido, denominado *passo* ou *taxa de aprendizagem*. Seu valor determina a intensidade da atualização realizada a cada iteração: quanto maior α, maior o deslocamento; quanto menor α, mais gradual é a atualização.

A interpretação geométrica fornecida pela figura é particularmente importante: em cada etapa, o método utiliza apenas uma informação local — a inclinação da reta tangente no ponto atual — para decidir como atualizar a variável. Desse modo, mesmo sem conhecer diretamente o ponto de mínimo, o algoritmo constrói uma sequência de aproximações que tende a se mover para regiões em que os valores de f são menores.

O processo consiste em repetir essa atualização a partir de um ponto inicial x_0, gerando uma sequência $\{x_k\}_{k\in\mathbb{N}}$ que, sob hipóteses adequadas sobre f e sobre a escolha de α, converge para um ponto crítico da função.

No caso de funções de duas variáveis,

$$f : \mathbb{R}^2 \to \mathbb{R}, \quad f(x, y),$$

a derivada é substituída pelo *gradiente*, definido como

$$\nabla f(x, y) = \begin{pmatrix} \frac{\partial f}{\partial x}(x, y) \\ \frac{\partial f}{\partial y}(x, y) \end{pmatrix}.$$

O gradiente indica a direção de maior crescimento da função. Assim, para buscar um mínimo, o deslocamento deve ocorrer na direção oposta ao gradiente.

A regra de atualização passa a ser

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} x_k \\ y_k \end{pmatrix} - \alpha \nabla f(x_k, y_k).$$

De forma mais geral, seja

$$f : \mathbb{R}^n \to \mathbb{R}$$

uma função diferenciável. O gradiente de f é o vetor

$$\nabla f(x_1, \ldots, x_n) = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix},$$

e o método do gradiente descendente é dado pela iteração

$$\vec{x}_{k+1} = \vec{x}_k - \alpha \nabla f(\vec{x}_k),$$

onde $\vec{x}_k \in \mathbb{R}^n$ representa o ponto atual.

Esse método baseia-se na ideia de que, em uma vizinhança suficientemente pequena, uma função diferenciável pode ser bem aproximada por uma função afim, e que o gradiente fornece a informação direcional necessária para reduzir o valor da função de forma sistemática. O gradiente descendente utiliza, portanto, a direção oposta ao gradiente para conduzir o processo iterativo em direção a um mínimo.

3. REDES NEURAIS ARTIFICIAIS: ESTUDO TEÓRICO

3.1 FUNDAMENTOS BIOLÓGICOS DAS REDES NEURAIS

O estudo das redes neurais artificiais tem como ponto de partida a compreensão do funcionamento do sistema nervoso biológico, em especial do neurônio, que é a unidade fundamental de processamento de informação no cérebro humano. Baseado em Bear, Connors e Paradiso [1], o sistema nervoso é composto por bilhões de neurônios interconectados, formando uma rede altamente complexa, capaz de processar informações, aprender com experiências e adaptar seu comportamento ao longo do tempo.

Um neurônio biológico é constituído basicamente por três partes principais: os dendritos, o corpo celular (ou soma) e o axônio. Os dendritos são responsáveis por receber sinais provenientes de outros neurônios; o corpo celular realiza a integração desses sinais; e o axônio, uma estrutura encontrada apenas em neurônios, é extremamente especifializada em conduzir impulsos nervosos até outras células, por meio das sinapses.

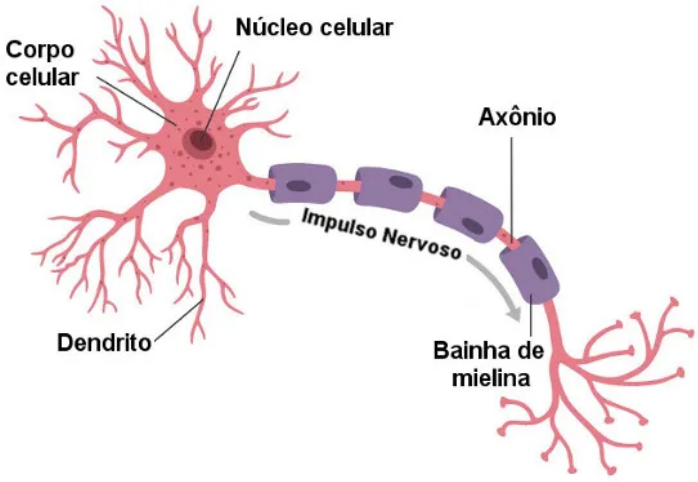


Figura 3.1: Representação de um neurônio

Fonte: Fonte: SANTOS, Vanessa Sardinha dos. *Neurônio: o que é, tipos, função, estrutura.* Mundo Educação, [s.d.]. Disponível em: https://mundoeducacao.uol.com.br/biologia/neuronios.htm. Acesso em: 12 mar. 2026.

As sinapses são conexões especializadas entre neurônios e desempenham papel central no processamento de informações no sistema nervoso. Em geral, estabelecem-se entre o terminal

axonal de um neurônio e o dendrito, o corpo celular ou, em alguns casos, o axônio de outro neurônio. Na maioria das sinapses, a informação é transmitida inicialmente sob a forma de impulsos elétricos que se propagam ao longo do axônio do neurônio pré-sináptico. Ao atingir o terminal axonal, esse sinal elétrico é convertido em um sinal químico capaz de atravessar a fenda sináptica. Na membrana do neurônio pós-sináptico, o sinal químico é novamente convertido em um sinal elétrico, permitindo a continuidade da transmissão da informação.

O sinal químico envolvido nesse processo é denominado *neurotransmissor*, sendo armazenado em vesículas sinápticas localizadas no terminal axonal e liberado mediante a chegada do potencial de ação. Diferentes tipos de neurônios utilizam diferentes neurotransmissores, o que contribui para a diversidade de respostas observadas no sistema nervoso.

Essa transformação da informação — de elétrica para química e novamente para elétrica — é fundamental para as capacidades computacionais do cérebro. Alterações nos mecanismos de transmissão sináptica estão diretamente relacionadas a processos como memória e aprendizagem.

Dependendo do tipo de sinapse, essa influência pode ser excitatória ou inibitória, contribuindo para o aumento ou a diminuição do potencial elétrico do neurônio pós-sináptico. Ou seja, os neurônios não apenas passam informação, eles também controlam se o próximo neurônio vai ativar ou não, por meio desse mecanismo de excitação e inibição. O funcionamento do neurônio está associado ao chamado potencial de ação, que ocorre quando o potencial elétrico da membrana celular atinge um determinado limiar. Caso esse limiar seja alcançado, o neurônio dispara um impulso elétrico que se propaga ao longo do axônio. Caso contrário, nenhum sinal é transmitido.

Outro conceito fundamental do sistema nervoso biológico é a plasticidade sináptica. Esse fenômeno refere-se à capacidade das sinapses de se modificarem ao longo do tempo, fortalecendo-se ou enfraquecendo-se em função da atividade neural. A plasticidade é amplamente reconhecida como a base biológica do aprendizado e da memória, permitindo que o cérebro se adapte a novos estímulos e experiências.

Apesar de sua enorme complexidade, o sistema nervoso biológico apresenta princípios gerais que podem ser estudados e abstraídos. A partir da observação do comportamento coletivo dos neurônios torna-se possível desenvolver modelos simplificados que preservam características essenciais, como integração de sinais, transmissão de informação e capacidade de adaptação.

Esses princípios biológicos fornecem, portanto, a motivação conceitual para o desenvolvimento de modelos matemáticos e computacionais inspirados no funcionamento do cérebro humano, os quais serão explorados nas seções seguintes deste trabalho.

3.2 Surgimento e Evolução das Redes Neurais Artificiais

O surgimento das Redes Neurais Artificiais está diretamente relacionado às tentativas iniciais de compreender e modelar, de forma matemática, o funcionamento do sistema nervoso.

Diferentemente de abordagens puramente biológicas, os primeiros modelos buscaram abstrair o comportamento dos neurônios de modo lógico e formal, permitindo sua análise e aplicação em problemas matemáticos e computacionais.

Um marco fundamental nesse processo foi o trabalho de Warren McCulloch e Walter Pitts, publicado em 1943. Nesse estudo, os autores propuseram um modelo matemático simplificado de neurônio, no qual a atividade neuronal era descrita por meio de operações lógicas. Nesse modelo, o neurônio artificial recebe múltiplos sinais de entrada, associados a valores binários, e produz uma saída também binária, dependendo do atendimento de um determinado limiar. No modelo proposto por McCulloch e Pitts, o limiar corresponde a um valor mínimo que a soma das entradas deve atingir para que o neurônio produza uma saída ativa. Assim, o neurônio dispara ou não dispara, de forma análoga a uma função lógica.

3.2.1 NEURÔNIO DE MCCULLOCH E PITTS

A principal contribuição de McCulloch e Pitts foi demonstrar que redes formadas por esses neurônios artificiais seriam capazes de representar qualquer função lógica finita. Esse resultado estabeleceu uma conexão direta entre a neurofisiologia e a lógica matemática, indicando que processos cognitivos poderiam, ao menos em parte, ser interpretados como sistemas formais de manipulação de símbolos. Embora extremamente simplificado, o modelo lançou as bases conceituais para o desenvolvimento posterior das redes neurais artificiais.

O chamado *neurônio binário* é um dos modelos mais simples de neurônio artificial. Nesse modelo, a saída do neurônio pode assumir apenas dois estados possíveis, usualmente representados pelos valores 0 ou 1.

Cada neurônio possui um limiar fixo, denotado por θ, e recebe sinais provenientes de suas conexões de entrada, associadas a pesos considerados idênticos nesse modelo simplificado. O funcionamento do neurônio binário baseia-se na soma desses valores de entrada: se a soma for maior ou igual ao limiar θ, o neurônio é ativado e produz saída igual a 1; caso contrário, o neurônio permanece inativo, produzindo saída igual a 0.

Esse modelo estabelece uma abstração inicial do comportamento de ativação neuronal e serve como base para o desenvolvimento de modelos mais elaborados. No próximo capítulo, serão apresentados de forma detalhada os conceitos de entradas, saídas, pesos sinápticos e funções associadas ao funcionamento das redes neurais artificiais.

3.2.2 O PERCEPTRON DE ROSENBLATT

Na década seguinte, usando o modelo de neurônio de McCulloch-Pitss, Frank Rosenblatt deu um passo além ao introduzir o perceptron, considerado o primeiro modelo de rede neural artificial capaz de aprender a partir de dados. Proposto em 1958, o perceptron baseia-se na ideia de um neurônio artificial que combina linearmente suas entradas, ponderadas por pesos ajustáveis, e produz uma saída determinada por uma função de ativação do tipo limiar. Diferentemente do modelo de McCulloch e Pitts, os pesos do perceptron não são fixos, mas

podem ser modificados por meio de um algoritmo de aprendizagem.

A Figura 3.2 apresenta a organização conceitual descrita no *Perceptron Operator's Manual* (Rosenblatt, 1958, *apud* [4]). No Mark I Perceptron, uma imagem televisiva digitalizada em uma grade de 20×20 pixels era utilizada como entrada. Em seguida, essa entrada era encaminhada por um conjunto dito "aleatório" de conexões para um conjunto de unidades de associação, que por sua vez alimentavam unidades de resposta. Essa configuração possui semelhanças com abordagens atuais para aprendizado em imagens, pois estabelece uma etapa intermediária de combinação e transformação de sinais antes da decisão final, lembrando arquiteturas que empregam camadas intermediárias e que, em alguns contextos, se aproximam do que é conhecido na literatura como *extreme learning machine*.

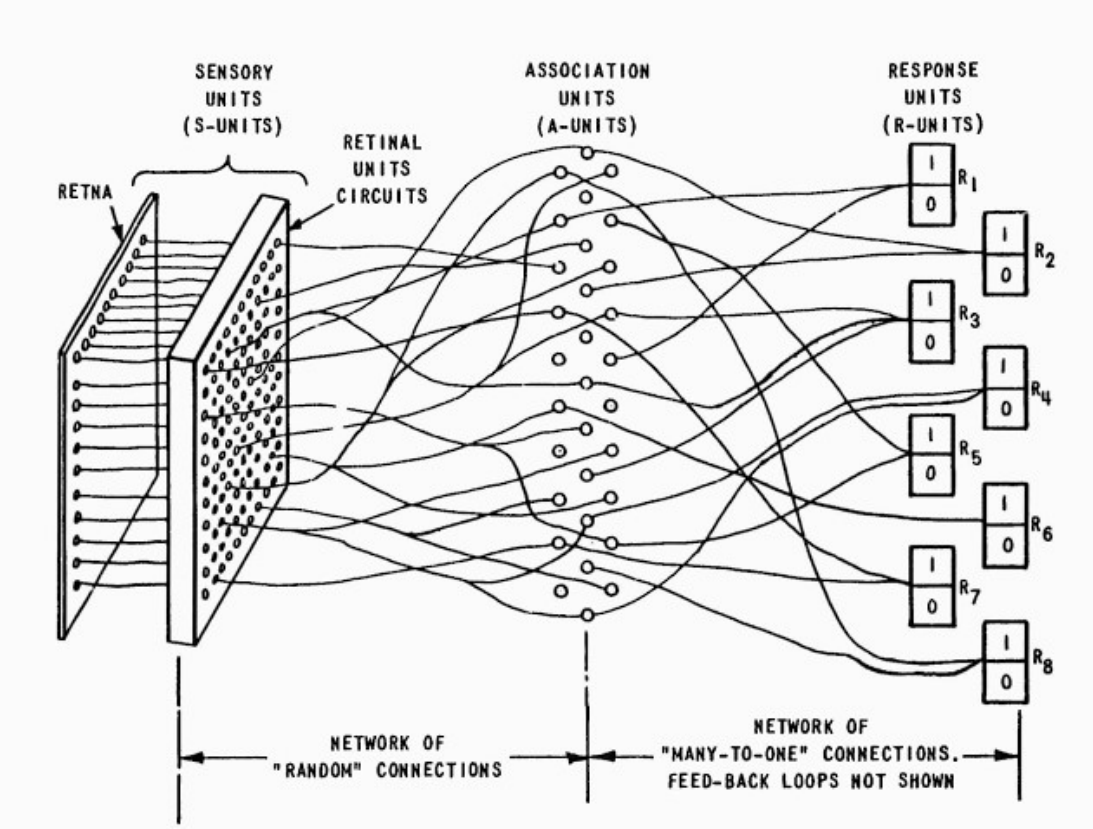


Figura 3.2: Organização conceitual do Mark I Perceptron (Figura 2 do manual de operação).
Fonte: HAY, John C.; MURRAY, Albert E. *Mark I Perceptron Operators' Manual.* Buffalo, New York: Cornell Aeronautical Laboratory, 15 fev. 1960. Disponível em: https://apps.dtic.mil/sti/tr/pdf/AD0236965.pdf. Acesso em: 12 mar. 2026.

O aspecto inovador do perceptron reside justamente em seu mecanismo de aprendizado supervisionado. A partir da comparação entre a saída produzida pelo modelo e a saída desejada, os pesos sinápticos são atualizados de forma iterativa, permitindo que o sistema aprenda a classificar padrões. Esse processo tornou o perceptron uma ferramenta promissora para tarefas como reconhecimento de padrões e classificação de dados, despertando grande interesse científico na época. Uma das primeiras aplicações em visão computacional, reconhecendo letras do alfeto ou figuras geométricas tais como triângulos.

Apesar de suas limitações, como a incapacidade de resolver problemas não linearmente

separáveis, o perceptron representou um avanço decisivo ao introduzir formalmente o conceito de aprendizagem em modelos neurais artificiais. Dessa forma, os trabalhos de McCulloch e Pitts e de Rosenblatt constituem os pilares iniciais da área de Redes Neurais Artificiais, estabelecendo tanto a base teórica quanto os primeiros mecanismos de aprendizado que influenciariam profundamente o desenvolvimento posterior do campo.

Mesmo com os estudos de McCulloch, Pitts e Rosenblatt, as redes neurais só passaram a ser estudadas e aplicadas de forma mais consistente a partir da década de 1980, ainda processadas em computadores clássicos. Um marco importante ocorreu em 2006, com o uso das unidades de processamento gráfico (GPUs) para realizar operações complexas em redes neurais [6]. Esse avanço permitiu o desenvolvimento de novos algoritmos e o surgimento do conceito de redes neurais profundas, mais conhecidas como *deep learning*, em 2012, possibilitando um volume de cálculos antes impensável e revolucionando a inteligência artificial.

3.3 Estrutura de um Neurônio Artificial

Um neurônio, no contexto de Redes Neurais Artificiais, é uma unidade de processamento de informação fundamental para o funcionamento do modelo. A Figura 3.3 apresenta um esquema desse neurônio artificial, que servirá de base para a definição de camadas e para a descrição dos principais processos em uma rede neural. Segundo Haykin [3], o neurônio é composto por três partes fundamentais:

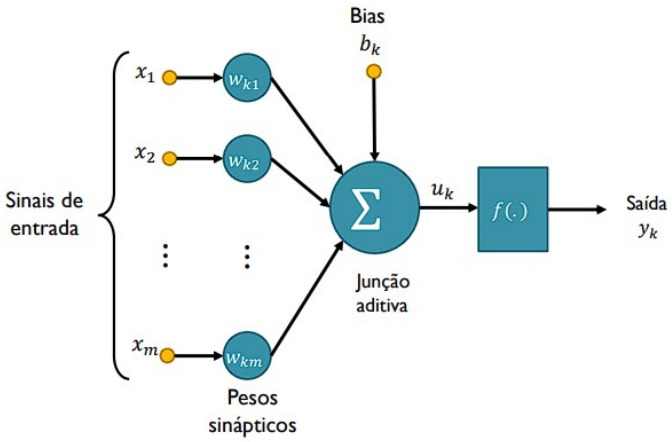


Figura 3.3: Representação de um neurônio artificial
Fonte: BREVE, Fabricio. *Redes neurais artificiais.*. [Slides]. 14 abr. 2022. Material de aula.

1. Um conjunto de **sinapses**, cada uma caracterizada por um *peso* próprio. O *dado de entrada* x_j que chega pela sinapse j, conectada ao neurônio k, é multiplicado pelo peso sináptico w_{kj}. É importante notar a convenção dos índices: o primeiro índice refere-se ao neurônio em questão (neste caso, k) e o segundo à sinapse (neste caso, j). Diferentemente de uma sinapse biológica, o peso sináptico em modelos artificiais pode assumir valores reais positivos ou negativos.

No neurônio biológico, as sinapses correspondem aos pontos de conexão entre neurônios, por meio dos quais sinais químicos são transmitidos. A intensidade dessa conexão pode variar, dependendo da eficácia sináptica. No modelo artificial, essa intensidade é representada matematicamente pelo peso sináptico w_{kj}. Assim, cada peso quantifica a influência que uma entrada x_j exerce sobre o neurônio k. Dessa forma, as sinapses biológicas encontram sua contrapartida matemática nos pesos das redes neurais artificiais.

2. **Soma ponderada**, responsável por combinar os sinais de entrada ponderados pelos respectivos pesos sinápticos. Esse bloco realiza, em termos matemáticos, uma *combinação linear* das entradas, também chamado de *combinador linear*.

   No neurônio biológico, os sinais provenientes das sinapses produzem pequenos potenciais elétricos nos dendritos, que se propagam até o corpo celular, onde são integrados. Esse processo de acumulação é denominado integração sináptica. Caso o potencial resultante ultrapasse um valor crítico, ocorre o disparo do potencial de ação. No modelo artificial, a soma ponderada representa matematicamente essa integração dos sinais recebidos. Cada peso w_{kj} modela a intensidade da conexão sináptica, enquanto as entradas x_j representam os estímulos recebidos.

3. **Uma função de ativação**, denotada por $\varphi(\cdot)$, cuja finalidade é restringir (ou controlar) a amplitude da saída do neurônio. Em muitos modelos, essa função é chamada também de *função restritiva*, pois limita o intervalo de valores possíveis para a saída. Nos modelos mais simples, como o neurônio binário McCulloch-Pitss, utiliza-se um intervalo normalizado, como $[0, 1]$ ou, alternativamente, $[-1, 1]$.

   A função de ativação, por sua vez, desempenha papel análogo ao mecanismo biológico de disparo: ela determina a saída do neurônio a partir do valor acumulado u_k, podendo modelar tanto um comportamento binário (como no modelo de McCulloch e Pitts) quanto respostas graduais, dependendo da função escolhida.

Em termos matemáticos, o neurônio k pode ser descrito pelo par de equações

$$u_k = \sum_{j=1}^{m} w_{kj}\, x_j, \tag{3.1}$$

e

$$y_k = \varphi(u_k + b_k), \tag{3.2}$$

em que $x_1, x_2, \ldots, x_m$ são os sinais de entrada; $w_{k1}, w_{k2}, \ldots, w_{km}$ são os pesos sinápticos do neurônio k; u_k é a saída do combinador linear; b_k é o viés; $\varphi(\cdot)$ é a função de ativação; e y_k é a saída do neurônio.

Até este ponto, descrevemos o neurônio artificial a partir de uma *combinação linear* dos sinais de entrada. Essa escrita é particularmente útil para explicitar o papel de cada sinapse e compreender o significado dos índices envolvidos. Entretanto, para fins de notação e para facilitar generalizações, é conveniente reescrever a mesma expressão em linguagem vetorial (ou

matricial). Para isso, organizamos as entradas em um vetor coluna $\vec{x} \in \mathbb{R}^m$ e os pesos do neurônio k em um vetor coluna $\vec{w}_k \in \mathbb{R}^m$, de modo que a soma ponderada possa ser expressa por meio de um produto interno:

$$u_k = \vec{w}_k^T \vec{x}.$$

Nessa forma, a operação realizada pelo combinador linear é interpretada como a projeção de $\vec{x}$ na direção de $\vec{w}_k$, o que fornece uma interpretação geométrica imediata. Essa padronização é essencial e muito utilizada nas redes neurais usualmente aplicadas, quando a discussão envolve a organização de vários neurônios em camadas e a representação global dos pesos de uma rede por matrizes.

3.3.1 VIÉS OU *BIAS*

No modelo de neurônio artificial, além da soma ponderada das entradas, introduz-se um termo adicional denominado *viés* (ou *bias*), usualmente representado por b (ou b_k para o neurônio k). Esse termo é somado ao resultado da combinação linear e atua como um deslocamento na entrada da função de ativação. O modelo matemático do perceptron pode ser escrito como

$$u = \vec{w}^T \vec{x} + b,$$

e a saída como

$$y = \varphi(u) = \varphi(\vec{w}^T \vec{x} + b),$$

em que $\varphi(\cdot)$ é a função de ativação.

A finalidade do viés pode ser compreendida geometricamente quando interpretamos a equação do perceptron como uma fronteira de decisão. Sem viés, a condição que determina a separação entre classes (isto é, a fronteira) assume a forma

$$\vec{w}^T \vec{x} = 0,$$

o que define um hiperplano que necessariamente passa pela origem. Nesse caso, a capacidade do modelo de separar conjuntos de dados fica restrita, pois a posição do hiperplano é "fixada" no sentido de que ele não pode ser transladado. A Figura 3.4 destaca essa limitação: o hiperplano (reta, no caso bidimensional) precisa atravessar a origem para cumprir a condição de separação.

Ao introduzir o viés, a fronteira de decisão passa a ser dada por

$$\vec{w}^T \vec{x} + b = 0,$$

isto é, pelo conjunto de pontos em que o classificador fica "indiferente" entre as classes, pois é exatamente nessa condição que a regra de decisão muda de um rótulo para o outro. Como consequência, essa equação define um hiperplano afim, que pode ser entendido como uma translação de um hiperplano linear. Em termos geométricos, o parâmetro b desloca o hiperplano no espaço, permitindo que a separação ocorra mesmo quando a origem não é um ponto "natural"

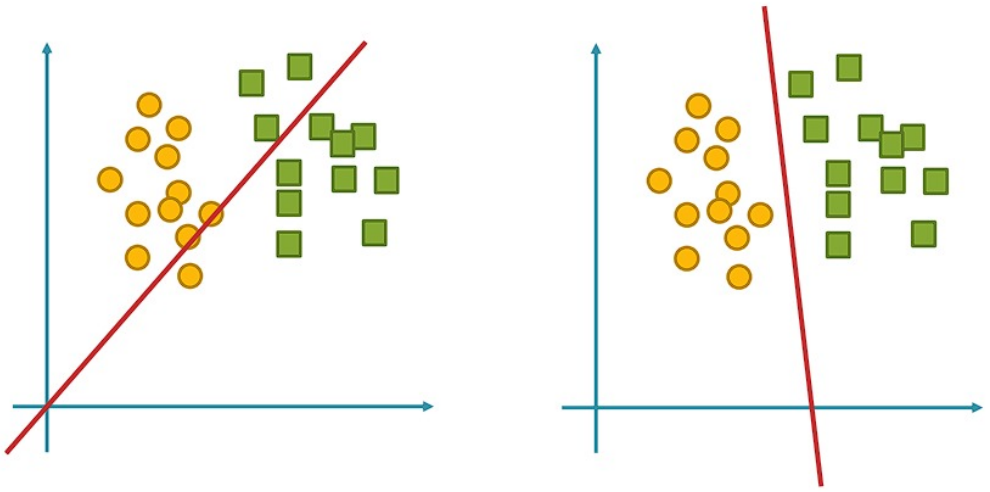


Figura 3.4: Efeito do viés (*bias*) no deslocamento do hiperplano de decisão: à esquerda, fronteira passando pela origem ($\vec{w}^T\vec{x} = 0$); à direita, fronteira transladada ($\vec{w}^T\vec{x} + b = 0$).

Fonte: SOARES, Alexsandro Santos. *Treinamento de Redes Neurais Artificiais*. Universidade Federal de Uberlândia, Faculdade de Computação, [s.d.]. Slides de aula.

para a divisão entre as classes. Assim, o viés aumenta a flexibilidade do classificador, tornando possível ajustar a fronteira de decisão para melhor adequação aos dados.

Em síntese, o viés desempenha o papel de *termo de ajuste* na função discriminante do neurônio artificial: ele controla o deslocamento da fronteira de decisão e, consequentemente, amplia o conjunto de padrões que podem ser representados por modelos lineares, como o perceptron. Em particular, a classe atribuída a um ponto $\vec{x}$ depende do sinal de $\vec{w}^T\vec{x} + b$, e a transição entre decisões distintas ocorre exatamente sobre a fronteira definida pela igualdade.

3.3.2 Função de ativação

Uma vez definida a combinação linear das entradas e incorporado o viés, resta caracterizar o elemento responsável por transformar a *entrada* do neurônio em sua *saída*: a *função de ativação*. No neurônio artificial, após o cálculo

$$u = \vec{w}^T\vec{x} + b,$$

a saída é obtida pela aplicação de uma função $\varphi : \mathbb{R} \to \mathbb{R}$, isto é,

$$y = \varphi(u).$$

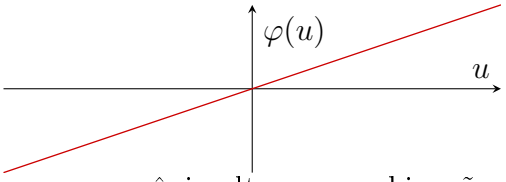
A principal finalidade da função de ativação é introduzir *não linearidade* no modelo [3]. Sem essa não linearidade, a composição de neurônios permaneceria equivalente a uma transformação afim, limitando significativamente a capacidade de representação da rede. Além disso, a escolha de $\varphi(\cdot)$ controla propriedades importantes como o intervalo possível da saída, a suavidade do modelo e o comportamento do treinamento por métodos iterativos. Se for necessário um resultado não-linear, pode ser usado uma função de ativação que busca a não linearidade, mas

como veremos no restante da seção existem funções de ativação com resultados lineares.

A seguir, destacam-se algumas funções de ativação amplamente empregadas na literatura e em aplicações.

FUNÇÃO LINEAR Relaciona a entrada diretamente com sua saída.

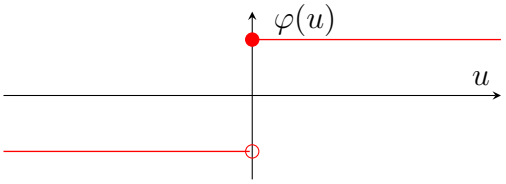
$$\varphi(u) = u.$$



Usamos a função linear quando não queremos que o neurônio altere a combinação afim $\vec{w}^T\vec{x}+b$. Além disso, em muitos algoritmos de treinamento, assume-se que a função de ativação seja diferenciável, de modo a permitir o cálculo das derivadas necessárias na atualização dos parâmetros.

FUNÇÃO DEGRAU (LIMIAR) É a função clássica usada no perceptron, apropriada para decisões binárias. A saída é 1 se a entrada é maior ou igual a um certo limiar θ.

$$\varphi(u) = \begin{cases} 1, & \text{se } u \geq \theta, \\ -1, & \text{se } u < \theta. \end{cases}$$



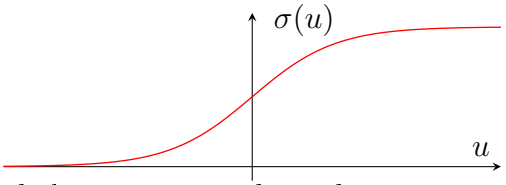
Se a função for Binária, utilizamos $\{0, 1\}$; se a função for bipolar, como o usado no trabalho de McCulloc-Pitss, utilizamos $\{-1, 1\}$.

Embora seja simples e interpretável, não é diferenciável em $u = 0$, o que dificulta sua utilização em procedimentos de otimização baseados em derivadas.

FUNÇÃO SIGMOIDE Define uma transição suave entre 0 e 1:

$$\varphi(u) = \frac{1}{1+e^{-u}}.$$

Na literatura, essa função é denotada por σ, assim $\varphi(u) = \sigma(u)$.



Uma função estritamente crescente que oferece balanceamento adequado entre comportamento linear e não-linear. Por muitos anos foi a função de ativação mais popular.

Derivando $\sigma(u)$, obtém-se a forma conhecida

$$\sigma'(u) = \sigma(u)\big(1 - \sigma(u)\big).$$

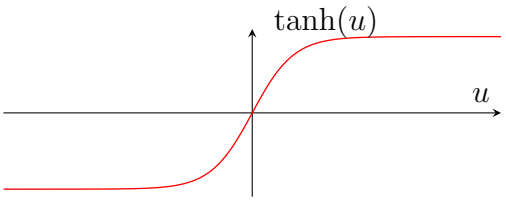
Essa expressão evidencia o fenômeno de **saturação**: quando u assume valores grandes em módulo, a saída $\sigma(u)$ se aproxima de um dos extremos do intervalo $(0, 1)$. De fato, se $u \gg 0$, então $\sigma(u) \approx 1$ e, consequentemente, $\sigma'(u) \approx 1 \cdot (1 - 1) = 0$; analogamente, se $u \ll 0$, então $\sigma(u) \approx 0$ e $\sigma'(u) \approx 0 \cdot (1 - 0) = 0$. Ou seja, nas regiões em que a sigmoide se aproxima de 0

ou 1, sua derivada torna-se muito pequena, fazendo com que o gradiente que chega às camadas anteriores é praticamente nulo. Esse comportamento reduz a magnitude das atualizações dos parâmetros e pode tornar o treinamento lento, especialmente em redes profundas.

Note que a derivada da função sigmoide, assim como a das próximas funções apresentadas, pode ser escrita em função da própria função. Essa observação é muito importante, pois torna os cálculos computacionais muito mais rápidos quando for necessário obter o gradiente de alguma função.

Tangente hiperbólica Semelhante à sigmoide, porém centrada em zero:

$$\varphi(u) = \frac{e^u - e^{-u}}{e^u + e^{-u}}.$$



Na literatura, essa função é denotada por tanh, assim $\varphi(u) = \tanh(u)$.

Seus valores pertencem a $(-1, 1)$, o que muitas vezes favorece o treinamento em comparação à sigmoide, ainda que também possa saturar.

Derivando tanh(u), temos

$$\frac{d}{du}\tanh(u) = 1 - \tanh^2(u).$$

Assim, quando $|u|$ é grande, tem-se $\tanh(u) \approx \pm 1$ e, portanto, $1 - \tanh^2(u) \approx 0$, caracterizando a saturação e produzindo gradientes muito pequenos nas camadas anteriores.

Função radial (RBF) Diferentemente das funções de ativação que dependem diretamente do sinal u, uma *função radial* (ou *radial basis function* – RBF) depende da *distância* entre a entrada e um centro. Em sua forma mais comum, considera-se um centro $\vec{c} \in \mathbb{R}^n$ e define-se

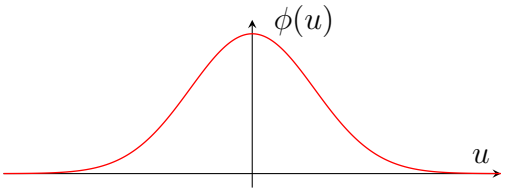
$$r = \|\vec{x} - \vec{c}\|, \qquad \varphi(\vec{x}) = \phi(r).$$

Uma escolha amplamente utilizada é a RBF Gaussiana,

$$\phi(r) = \exp\left(-\frac{r^2}{2\sigma^2}\right),$$

em que $\sigma > 0$ controla a largura da região de influência do centro. Essa função atinge valor máximo em $r = 0$ (isto é, quando $\vec{x} = \vec{c}$) e decai à medida que a distância aumenta, produzindo uma ativação *local* no espaço de entrada.

Para fins de visualização, é comum representar o comportamento em função de uma variável escalar u, interpretando u como uma distância (isto é, $u = r$). Fixando $\sigma = 1$, obtemos



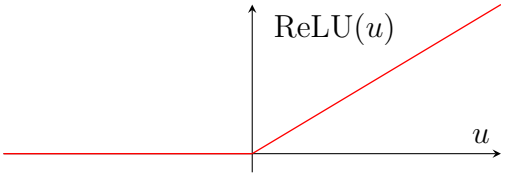
$$\phi(u) = \exp\left(-\frac{u^2}{2}\right).$$

A função radial é usada quando queremos que a ativação dependa de **proximidade** no espaço de entrada, isto é, quando padrões parecidos com um centro devem ativar o neurônio.

RELU (*RECTIFIED LINEAR UNIT*) Uma das funções mais utilizada com a ascensão das redes de aprendizado profundo (*deep learning*).

$$\varphi(u) = \max\{0, u\}.$$

Na literatura, essa função é denotada por ReLU, assim $\varphi(u) = \text{ReLU}(u)$.



A ReLU preserva valores positivos e zera valores negativos, funcionando como uma função linear por partes; la é computacionalmente eficiente, converge rapidamente, além de ser não-linear - possibilitando a retropropagação. No entanto, sua desvangem é que neurônios podem "morrer", ou seja, na faixa menor que o zero o gradiente é zero, portanto o neurônio deixa de contribuir para a rede e não há aprendizado.

Exemplo 3.3.1. *Considere um único neurônio com ativação ReLU,*

$$u = \text{ReLU}(u) = \max\{0, u\}, \qquad u = wx + b.$$

Suponha que, em certo momento do treinamento, os parâmetros fiquem

$$w = 1, \quad b = -5,$$

e que os valores de entrada do problema satisfaçam $x \in [0, 4]$. *Então, para qualquer entrada nesse intervalo,*

$$u = x - 5 \leq 4 - 5 = -1 < 0,$$

logo

$$y = \text{ReLU}(u) = 0.$$

Como a derivada da ReLU é

$$\text{ReLU}'(u) = \begin{cases} 1, & u > 0, \\ 0, & u \leq 0, \end{cases}$$

temos $\text{ReLU}'(u) = 0$ *para todos os exemplos do conjunto. Portanto, o gradiente associado a esse neurônio zera, pois ele sempre aparece multiplicado por* $\text{ReLU}'(u)$. *Assim,* w *e* b *não se*

alteram, o neurônio permanece com $u \leq 0$ e continua produzindo saída nula para qualquer entrada do problema. Esse é o sentido de dizer que o neurônio "morre": ele deixa de contribuir para a rede e não recupera a ativação durante o treinamento.

As funções mostradas e ilustradas mostram apenas a forma geral de tais funções, estas formas podem ser modificadas, por exemplo, multiplicando por uma constante. Assim como novos funções de ativação vem sendo propostas para tentar contornar os problemas da ReLU.

Em síntese, não existe uma única função de ativação "correta" para todos os contextos. A escolha de $\varphi(\cdot)$ é, em certa medida, arbitrária no sentido de que pode ser feita entre diferentes alternativas matematicamente válidas; contudo, na prática, essa escolha deve ser orientada pelo comportamento desejado para a saída do neurônio e pelas propriedades do problema em estudo. Funções com saída limitada (como a sigmoide e a tangente hiperbólica) podem ser úteis quando se deseja restringir o intervalo de valores e obter transições suaves, enquanto funções lineares por partes (como a ReLU) são frequentemente adotadas quando se busca simplicidade e boa resposta para valores positivos. Portanto, a definição da função de ativação deve levar em conta o tipo de tarefa (classificação ou regressão), a escala da saída esperada e a conveniência analítica e computacional do modelo, de modo que a arquitetura produza resultados coerentes com os objetivos estabelecidos.

3.3.3 Dados de Entrada

Até agora vimos que nosso objetivo é classificar dados de duas ou mais maneiras fazendo uso da combinção linear, do viés e da função de ativação. Para isso, é necess ário um conjunto de dados com valores de entrada e valores de saída esperados para que a máquina, ou rede neural, seja capaz de classificar, aprendendo a partir de dados exemplos que já possuimos. Esse processo é chamado de regime **supervisionado**, no qual cada vetor $\vec{x}_i$ vem acompanhado de um rótulo t_i (saída desejada), formando pares $(\vec{x}_i, t_i)$. Essa é a configuração adotada nas seções seguintes, pois é exatamente nela que se formula a regra de aprendizagem do perceptron e se descreve como os pesos e o viés são atualizados ao longo do treinamento.

A notação

$$\mathcal{D} = \{(\vec{x}_i, t_i)\}_{i=1}^{N}, \qquad \vec{x}_i \in \mathbb{R}^m,$$

indica um **conjunto de dados (dataset) supervisionado** composto por N exemplos.

Em detalhes:

- $\mathcal{D}$ é o conjunto (ou coleção) de treinamento.
- $\{(\vec{x}_i, t_i)\}_{i=1}^{N}$ significa que temos pares ordenados $(\vec{x}_i, t_i)$ para $i = 1, 2, \ldots, N$.
- $\vec{x}_i$ é o **vetor de entrada** do i-ésimo exemplo, e por $\vec{x}_i \in \mathbb{R}^m$ entende-se que cada entrada

possui m componentes reais, isto é,

$$\vec{x}_i = \begin{bmatrix} x_{i1} \\ x_{i2} \\ \vdots \\ x_{im} \end{bmatrix} \in \mathbb{R}^m,$$

onde m é o número de atributos/variáveis medidos em cada exemplo.

- t_i é o **saída desejada** associado ao i-ésimo exemplo.

Assim, $\mathcal{D}$ organiza os dados como **entradas** (vetores em $\mathbb{R}^m$) acompanhadas de suas **saídas corretas**, que são exatamente as informações usadas para treinar o perceptron.

Este trabalho está focado em redes supervisionadas mas para fins de contextualização, vale mencionar que também existem abordagens **não supervisionadas**, nas quais não há rótulos e a aprendizagem busca identificar estruturas nos dados; contudo, tais métodos não serão desenvolvidos neste trabalho.

3.3.4 ALGORITMO PERCEPTRON

Explicado as estruturas de nosso neurônio, precisamos concluir todo o processo que ocorre na rede neural. Nesta seção formalizamos o *algoritmo do perceptron* (regra de aprendizagem do perceptron), um procedimento iterativo clássico para ajustar os parâmetros de um classificador linear. A ideia central é simples: a fronteira de decisão é um hiperplano (possivelmente afim) e, a cada erro de classificação, atualizamos os pesos de modo a deslocar essa fronteira na direção que corrige o erro.

Considere um conjunto de treinamento supervisionado

$$\mathcal{D} = \{(\vec{x}_i, t_i)\}_{i=1}^N, \qquad \vec{x}_i \in \mathbb{R}^m,$$

em que t_i representa o rótulo desejado. Na formulação mais comum, como vimos no trabalho de McCulloch-Pitts, toma-se $t_i \in \{-1, 1\}$. Para um vetor de pesos $\vec{w} \in \mathbb{R}^m$ e viés $b \in \mathbb{R}$, definimos a função discriminante

$$u(\vec{x}) = \vec{w}^T \vec{x} + b.$$

A partir de $u(\vec{x})$, o perceptron produz uma saída y por meio de uma função de ativação, e compara essa saída com o rótulo desejado t. A ideia central do treinamento é simples: ajustar os parâmetros apenas quando houver discrepância entre o que o modelo previu e o que era esperado, isto é, quando o exemplo estiver incorretamente classificado.

Assim, uma forma clássica da regra de atualização do perceptron é:

$$\vec{w}_{k+1} = \vec{w}_k + \alpha(t - y)\,\vec{x}, \qquad b_{k+1} = b_k + \alpha(t - y),$$

em que $\alpha > 0$ é a taxa de aprendizagem. Intuitivamente, quando o modelo erra ($t \neq y$), os

pesos e o viés são modificados na direção que aumenta a compatibilidade entre a entrada $\vec{x}$ e a classe correta, deslocando a fronteira de decisão para reduzir o erro naquele exemplo.

Mais a frente no trabalho falaremos de funções de custo que podem ser implementadas na rede neural para analisar a discrepância em relação aos nossos valores de t. Porém, neste momento, trabalharemos com a noção básica de erro: $E = (t_i - y_i)$, para a introdução do algoritmo.

Descrição algorítmica

Uma implementação conceitual do treinamento pode ser descrita como segue.

Algoritmo 1 Algoritmo de Aprendizagem do Perceptron (Regime Supervisionado)

Entrada: Pesos iniciais $\vec{w}$; viés inicial b; conjunto de treinamento $\{(\vec{x}_i, t_i)\}_{i=1}^{m}$; taxa de aprendizagem α; número máximo de épocas N.
Saída: Pesos finais $\vec{w}$, viés final b, número de épocas EP.

$EP = 0$
enquanto $EP < N$ **faça**
  $erro\_epoca = 0$
  para $i = 1$ **até** m **faça**
    $u_i = \vec{w} \cdot \vec{x}_i + b$
    Compute a saída: $y_i = f(u_i)$
    Compute o erro: $E_i = t_i - y_i$
    se $E_i \neq 0$ **então**
      Atualize:

$$\vec{w} \leftarrow \vec{w} + \alpha\, E_i\, \vec{x}_i, \quad b \leftarrow b + \alpha\, E_i$$

      $erro\_epoca = erro\_epoca + 1$
    end se
  end para
  $EP = EP + 1$
  se $erro\_epoca = 0$ **então**
    interrompa (convergência: não houve erros na época)
  end se
end enquanto
SAÍDA ($\vec{w}$, b, EP)
PARE

3.3.5 Teorema de Convergência do Perceptron

Teorema 5 (Convergência do Perceptron)**.** *Considere um conjunto de dados de treinamento* $\{(\vec{x}_i, t_i)\}_{i=1}^{m}$*, com* $t_i \in \{-1, +1\}$*. Suponha que os dados sejam* ***linearmente separáveis****, isto é, exista* $(\vec{w}^*, b^*)$ *tal que*

$$t_i\big(\vec{w}^* \cdot \vec{x}_i + b^*\big) > 0, \quad \textit{para todo } i = 1, \ldots, m.$$

Então, para qualquer inicialização $(\vec{w}_0, b_0)$ *e qualquer taxa de aprendizagem* $\alpha > 0$*, o algoritmo de aprendizagem do perceptron realiza apenas um número* ***finito*** *de atualizações e, após*

um número finito de passos, encontra parâmetros $(\vec{w}, b)$ *que classificam corretamente todos os exemplos do conjunto de treinamento [2, 5].*

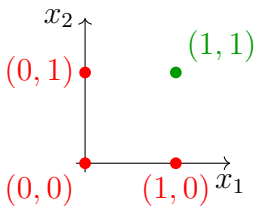
Nos exemplos a seguir, consideramos vetores de entrada da forma

$$x = (x_1, x_2)^T \in \{0,1\}^2.$$

Os quatro pontos que utilizaremos são

$$(0,0),\ (0,1),\ (1,0),\ (1,1).$$

Exemplo 3.3.2 (Problema AND). *Considere entradas binárias* $\vec{x} = (x_1, x_2) \in \{0,1\}^2$ *e a tabela verdade do operador lógico AND:*



x_1	x_2	AND
0	0	0
0	1	0
1	0	0
1	1	1

Figura 3.5: Representação geométrica e tabela verdade do operador lógico AND: a classe 0 (vermelho) e a classe 1 (verde).

No gráfico acima, os pontos em vermelho indicam saída 0 *e o ponto em verde indica saída* 1. *Queremos encontrar um hiperplano, que neste caso é uma reta por se tratar de* $\mathbb{R}^2$, *que separe o ponto de coordenadas* $(1,1)$ *dos demais.*

Adotando $t = 1$ *para saída* 1 *e* $t = -1$ *para saída* 0, *podemos obter os parâmetros de separação por meio do processo iterativo do perceptron. Partindo de pesos e viés iniciais (por exemplo, nulos) e aplicando sucessivamente a regra de atualização*

$$\vec{w} \leftarrow \vec{w} + \alpha(t-y)\vec{x}, \qquad b \leftarrow b + \alpha(t-y),$$

sempre que um exemplo for classificado incorretamente, os parâmetros vão sendo ajustados até que todos os pontos sejam corretamente separados. Para o problema AND, esse procedimento conduz a uma solução possível dada por

$$\vec{w} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \qquad b = -\frac{3}{2}.$$

A fronteira de decisão correspondente é

$$\vec{w}^T\vec{x} + b = x_1 + x_2 - \frac{3}{2} = 0.$$

Verificando os quatro pontos, isto é, calculados o valor de $x_1 + x_2 - \frac{3}{2}$ *para cada ponto da tabela,*

obtemos:

$$(0,0) \mapsto -\frac{3}{2} < 0, \qquad (0,1) \mapsto -\frac{1}{2} < 0, \qquad (1,0) \mapsto -\frac{1}{2} < 0, \qquad (1,1) \mapsto \frac{1}{2} > 0.$$

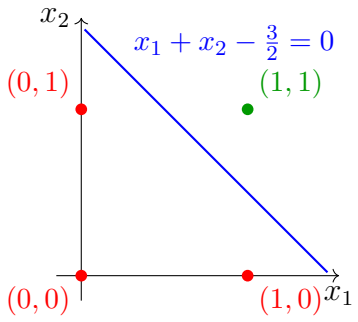


Figura 3.6: Pontos do problema AND e a fronteira de decisão $x_1 + x_2 - \frac{3}{2} = 0$.
Fonte: Autoria própria.

Logo, o AND é linearmente separável: existe um hiperplano (neste caso, uma reta) que separa a classe $\{(1,1)\}$ *das demais entradas.*

O perceptron é um modelo *linear* no espaço de entrada, pois a decisão depende do sinal de $\vec{w}^T\vec{x} + b$. Logo, seu desempenho está diretamente ligado à hipótese de **separabilidade linear** dos dados: se existir algum hiperplano que separe corretamente as classes, então o algoritmo é capaz de encontrar um classificador que zera os erros no conjunto de treinamento.

Definição 40 (Separabilidade linear)**.** *Dado um conjunto de exemplos rotulados* $\mathcal{D} = \{(\vec{x}_i, t_i)\}_{i=1}^N$*, com* $\vec{x}_i \in \mathbb{R}^m$ *e* $t_i \in \{-1, 1\}$*, dizemos que* $\mathcal{D}$ *é* ***linearmente separável*** *se existem* $\vec{w} \in \mathbb{R}^m$ *e* $b \in \mathbb{R}$ *tais que*

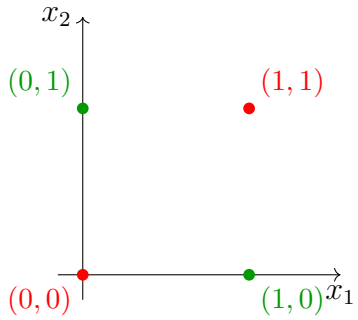
$$t_i\big(\vec{w}^T\vec{x}_i + b\big) > 0, \qquad i = 1, \ldots, N.$$

Nesse caso, o hiperplano $\vec{w}^T\vec{x} + b = 0$ *separa corretamente as duas classes.*

Nem sempre é possível encontrar uma fronteira de decisão a partir do processo perceptron de camada única, como veremos o exemplo a seguir:

Exemplo 3.3.3 (Problema XOR)**.** *Considere entradas binárias* $\vec{x} = (x_1, x_2) \in \{0,1\}^2$ *e a tabela verdade do operador lógico XOR.*

No gráfico acima, os pontos em vermelho indicam saída 0 *e os pontos em verde indicam saída* 1*. Diferentemente do caso AND, agora queremos verificar se existe um hiperplano, que neste caso é uma reta por se tratar de* $\mathbb{R}^2$*, capaz de separar simultaneamente os pontos* $(0,1)$ *e* $(1,0)$ *dos pontos* $(0,0)$ *e* $(1,1)$*. Como essas classes ocupam vértices opostos do quadrado* $\{0,1\}^2$*, não é possível traçar uma única reta que faça essa separação, o que mostra que o XOR não é linearmente separável. Portanto, o XOR* ***não*** *é linearmente separável e um perceptron de camada única não consegue classificá-lo corretamente em todos os casos. Essa limitação motiva a introdução de camadas* ***ocultas*** *(ou intermediárias), que permitem composições não lineares capazes de representar fronteiras de decisão mais complexas.*



x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Figura 3.7: Representação geométrica e tabela verdade do operador lógico XOR: a classe 0 (vermelho) e a classe 1 (verde).

Fonte: Autoria própria.

3.4 Arquitetura de Redes Neurais Artificias

Até aqui, nosso estudo esteve centrado no *perceptron de camada única*, isto é, o modelo recebe recebe um vetor de entrada $\vec{x}$, realiza uma combinação linear $\vec{w}^T\vec{x}+b$ e aplica uma função de ativação para produzir a decisão, apenas esse processo descrito ocorre iterativamente. Neste modelo simples, o processo de aprendizagem consiste em ajustar os pesos $\vec{w}$ e o viés b a partir de exemplos rotulados, buscando uma regra de decisão capaz de separar corretamente os dados sempre que essa separação puder ser descrita por um hiperplano. Assim, quando o conjunto é linearmente separável, o perceptron de camada única pode estabelecer uma fronteira de decisão compatível com a tarefa.

Entretanto, como observado no exemplo do operador XOR, existem configurações de pontos para as quais não há uma única reta que realize a separação desejada; logo, um perceptron de camada única não consegue classificar corretamente todos os casos. Essa limitação mostra que para representar fronteiras de decisão mais complexas, é necessário ir além de uma única transformação afim seguida da função de ativação.

Dessa forma, iniciamos agora a discussão sobre **arquiteturas** de redes neurais, a maneira como neurônios podem ser organizados em **camadas** e como a informação flui entre elas. Em particular, veremos como a introdução de **camadas ocultas** permite composições não lineares e amplia com ela o poder de representação do modelo, oferecendo um caminho natural para superar limitações como a do XOR.

3.4.1 Camada de entrada e camada de saída

Em uma rede neural a **camada de entrada** (camada 0) é responsável por receber os dados do problema. Seus "neurônios" não realizam processamento no mesmo sentido das camadas seguintes; eles apenas representam as variáveis de entrada, isto é, os valores $x_1, \ldots, x_{n_0}$ que compõem o vetor $\vec{x}$. Por isso, costuma-se dizer que a camada de entrada funciona como uma interface entre o conjunto de dados e a rede: ela organiza e disponibiliza as informações iniciais para que sejam transformadas pelas camadas posteriores.

Já a **camada de saída** (camada L) produz a resposta final do modelo. Ela recebe

as ativações da última camada e calcula seus próprios potenciais $u_i^{[L]}$ e ativações $y_i^{[L]}$, que são interpretadas como a predição da rede.

3.4.2 Camadas ocultas

Chamamos de **camadas ocultas** (*hidden layers*) as camadas de neurônios que ficam *entre* a camada de entrada e a camada de saída. Elas recebem como entrada as ativações produzidas pela camada anterior e produzem novas ativações, que são repassadas adiante na rede. Diferentemente da camada de entrada, cujos valores são diretamente observáveis nos dados, e da camada de saída, que fornece a predição final do modelo, as camadas ocultas não são observadas externamente: seus valores internos (potenciais $u_i^{[k]}$ e ativações $y_i^{[k]}$) são valores computacionais, que não aparecem durante o resultado obtido.

Um perceptron de camada única realiza apenas uma transformação afim seguida de uma função de ativação, o que limita a fronteira de decisão a um hiperplano. Ao empilhar camadas ocultas, a rede passa a compor transformações sucessivas, permitindo modelar relações não lineares e, consequentemente, fronteiras de decisão mais complexas.

3.4.3 Camadas intermediárias

No contexto de arquiteturas com múltiplas camadas, é comum utilizar também o termo **camadas intermediárias** para se referir às camadas que não são nem a entrada nem a saída. Em muitos textos, *camada intermediária* e *camada oculta* aparecem como sinônimos; neste trabalho, adotaremos a seguinte distinção conceitual: toda camada oculta é uma camada intermediária, mas o termo *intermediária* da ideia de estar "no meio" do caminho, enquanto o termo *oculta* nos lembra que não observamos relação direta dessas variáveis com os dados e aos rótulos.

Assim, quando tratarmos da arquitetura, usaremos "intermediária" para destacar o papel estrutural dessas camadas nas transformações; quando discutirmos o processamento interno e as quantidades calculadas ($u_i^{[k]}$ e $y_i^{[k]}$), usaremos "oculta" para ressaltar que essas camadas operam como níveis internos de representação. Essa diferenciação ajuda a organizar a leitura e será útil quando discutirmos como a **profundidade** (número de camadas) e a **largura** (número de neurônios por camada) influenciam o tipo de padrão que a rede consegue aprender.

Essa separação de papéis é importante para a lógica do fluxo de informação: a camada de entrada fornece os dados, as camadas intermediárias ou ocultas constroem representações progressivamente mais adequadas, e a camada de saída sintetiza essas representações em uma decisão.

3.4.4 Notação arquitetural e notação de camadas

Para descrever redes neurais com clareza, adotamos uma notação que explicita a quantidade de neurônios em cada camada. Por exemplo, a arquitetura **2–3–1** indica uma rede com **2** variáveis na camada de entrada, **3** neurônios em uma camada oculta e **1** neurônio na camada

de saída. De modo geral, uma rede n_0–n_1–$\cdots$–n_L possui n_0 entradas e n_k neurônios na camada k, para $k = 1, \ldots, L$, sendo L o índice da última camada (saída).

Além dessa notação arquitetural, utilizaremos a seguinte notação de variáveis em camadas:

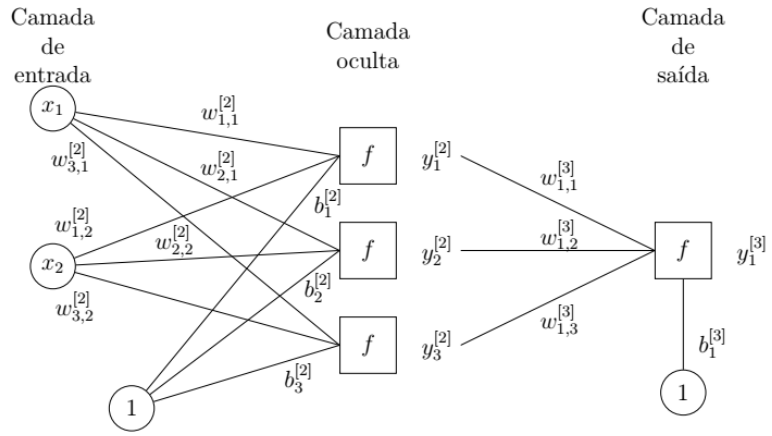


Figura 3.8: Exemplo de notação arquitetural: rede 2–3–1.

- x_i: a i-ésima entrada na camada de entrada, com $i = 1, \ldots, n_0$.
- $u_i^{[k]}$: o *potencial de ativação* (ou entrada líquida) do i-ésimo neurônio da camada k, isto é, a soma ponderada das ativações da camada anterior com o viés:

$$u_i^{[k]} = \sum_{j=1}^{n_{k-1}} w_{i,j}^{[k]}\, y_j^{[k-1]} + b_i^{[k]}.$$

  Observação: na camada de entrada ($k = 0$), não há aplicação de função de ativação; assim, a saída dessa camada coincide com o vetor de entrada. Em particular, temos

$$y^{[0]} = \vec{x} \quad (\text{isto é, } y_i^{[0]} = x_i \text{ para todo } i = 1, \ldots, n_0).$$

- $y_i^{[k]}$: a ativação (saída) do i-ésimo neurônio da camada k, obtida pela aplicação da função de ativação f:

$$y_i^{[k]} = f\left(u_i^{[k]}\right).$$

- $w_{i,j}^{[k]}$: o peso da conexão entre o j-ésimo neurônio da camada $k-1$ (camada anterior) e o i-ésimo neurônio da camada k (camada atual).
- $b_i^{[k]}$: o viés (*bias*) associado ao i-ésimo neurônio da camada k, responsável por deslocar a fronteira de decisão.

Essa convenção permite representar, de forma padronizada, tanto a estrutura da rede (quantidade de neurônios por camada) quanto as grandezas envolvidas no fluxo de informação e no cálculo das ativações.

3.5 Retropopagação do erro

Até aqui, descrevemos como um neurônio produz uma saída a partir de uma combinação linear, do viés e de uma função de ativação, bem como a regra de aprendizagem do perceptron no regime supervisionado. No entanto, ao introduzirmos arquiteturas com múltiplas camadas, o ajuste dos parâmetros deixa de depender apenas de um erro local na saída e passa a exigir um mecanismo para *distribuir* a informação do erro por toda a rede, de modo que cada peso e cada viés sejam atualizados de forma coerente com o objetivo do treinamento. Esse mecanismo é conhecido como **retropropagação** (*backpropagation*).

A retropropagação se apoia em dois elementos centrais: (i) uma medida quantitativa de quão distante a predição da rede está do valor desejado e (ii) um procedimento para calcular como essa medida varia em relação aos parâmetros da rede. O primeiro elemento é dado por uma **função de perda** (ou função de erro), e o segundo é obtido aplicando-se a regra da cadeia para computar derivadas de forma eficiente ao longo das camadas.

Usando a descida do gradiente, conceito que vimos no capítulo de Fundamentos, podemos iterativamente nos mover para mais próximo de um valor mínimo do nosso erro, usando pequenos passos na direção informada pelo gradiente. Assim, retropropagação calcula os gradientes e a descida do gradiente utiliza esses gradientes para utilizar os parâmetros.

3.5.1 Função de perda e Função de custo

Em um problema supervisionado, cada vetor de entrada $\vec{x}_i$ vem acompanhado de um rótulo t_i. Ao apresentar $\vec{x}_i$ à rede, obtém-se uma saída y_i. A **função de perda** quantifica, por meio de um número real, o quão distante a predição está do valor desejado, isto é, ela mede o erro cometido em *um único exemplo*. Denotaremos essa medida por

$$\mathcal{L}(y_i, t_i).$$

Para enfatizar o valor numérico obtido após aplicar essa função, também utilizaremos a notação

$$E = \mathcal{L}(y_i, t_i),$$

isto é, E representa o *valor da perda* (ou custo do exemplo) resultante da aplicação de $\mathcal{L}$ à saída produzida e ao rótulo correspondente.

Assim, quando $\mathcal{L}(y_i, t_i)$ (ou, de forma equivalente, E) assume valores pequenos, entende-se que y_i está próximo de t_i; por outro lado, valores maiores indicam maior discrepância e sinalizam que os parâmetros da rede devem ser ajustados em busca de melhores resultados.

Como o treinamento é realizado sobre um conjunto de dados

$$\mathcal{D} = \{(\vec{x}_i, t_i)\}_{i=1}^{N},$$

é natural agregar as perdas de cada processo para obter uma medida que generalize o desempe-

nho do modelo. Essa agregação define a **função de custo** $J(\theta)$, que depende dos parâmetros da rede θ (pesos e vieses). Uma escolha padrão é a média das perdas:

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(y_i, t_i).$$

Dessa forma, enquanto $\mathcal{L}$ mede o erro local de um processo, $J(\theta)$ mede o erro global no conjunto de treinamento. Assim, treinar uma rede neural consiste em **minimizar** $J(\theta)$, buscando parâmetros que tornem a perda média tão pequena quanto possível.

A seguir, apresentamos funções de perda usuais e as correspondentes funções de custo obtidas pela sua agregação no conjunto de dados.

PERDA QUADRÁTICA (*SQUARED ERROR*) Uma das escolhas mais simples é penalizar o erro por meio do quadrado da diferença entre a saída prevista e o valor desejado:

$$\mathcal{L}_Q(y, t) = (t - y)^2.$$

Por elevar o erro ao quadrado, essa perda atribui penalização maior a discrepâncias grandes e menor a discrepâncias pequenas. Essa característica faz com que erros mais distantes do alvo tenham maior influência no processo de treinamento.

ERRO QUADRÁTICO MÉDIO (MSE – *MEAN SQUARED ERROR*) Quando consideramos um conjunto de treinamento $\mathcal{D} = \{(\vec{x}_i, t_i)\}_{i=1}^{N}$, a forma mais comum de agregar a perda quadrática é pela média:

$$J(\theta) = \text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(t_i - y_i)^2.$$

O MSE é amplamente empregado em problemas de regressão, pois mede a discrepância média entre os valores previstos e os valores reais, com forte penalização para grandes erros. Além disso, a perda quadrática é diferenciável, o que favorece o uso de métodos baseados em gradiente na retropropagação.

PERDA ABSOLUTA (*ABSOLUTE ERROR*) Outra alternativa é penalizar o erro pelo valor absoluto da diferença:

$$\mathcal{L}_A(y, t) = |t - y|.$$

Nesse caso, o crescimento da penalização é linear: um erro duas vezes maior gera uma perda duas vezes maior. Em comparação com a perda quadrática, a perda absoluta tende a ser menos sensível a valores extremos, pois não amplifica grandes erros pelo quadrado.

Erro Absoluto Médio (MAE – *Mean Absolute Error*) Assim como no caso do MSE, podemos considerar a média das perdas absolutas ao longo do conjunto de treinamento:

$$J(\theta) = \text{MAE} = \frac{1}{N}\sum_{i=1}^{N} |t_i - y_i|.$$

O MAE é bastante utilizado em regressão quando se deseja uma medida de erro mais robusta a *outliers*. Por outro lado, a presença do valor absoluto faz com que a função não seja diferenciável em $t_i - y_i = 0$, o que exige cuidados na implementação (por exemplo, uso de subgradientes) quando o treinamento é feito por métodos baseados em derivadas.

Em síntese, a escolha entre MSE e MAE depende do comportamento que se deseja para a penalização do erro: o MSE enfatiza discrepâncias grandes e favorece soluções que evitam grandes erros, enquanto o MAE penaliza de forma linear e tende a ser mais robusto quando há observações extremas no conjunto de dados.

3.5.2 Modelagem da propagação do erro

Uma vez definida a função de custo, o treinamento de uma rede neural consiste em ajustar pesos e vieses de modo a reduzir esse custo. Para isso, precisamos compreender **como o erro observado na saída se relaciona com cada parâmetro** distribuído ao longo das camadas. A retropropagação formaliza exatamente essa relação: ela descreve, de maneira sistemática, como a informação do erro calculado na camada de saída retorna para as camadas anteriores, permitindo obter as derivadas necessárias para atualizar os parâmetros.

Como as redes neurais são construídas por *composições sucessivas* (combinação linear seguida de função de ativação, repetida ao longo das camadas), o instrumento matemático central é a **regra da cadeia**, qual permite decompor a variação de uma quantidade final (o custo) em uma sequência de derivadas. É justamente essa decomposição que torna possível calcular gradientes camada a camada.

3.5.3 Retropopagação do erro em uma rede 2-2-2

A seguir, calcularemos como o erro total E varia quando alteramos um peso específico da camada de saída, isto é, obter derivadas do tipo $\dfrac{\partial E}{\partial w_{i,j}^{[L]}}$. O ponto-chave é que um peso não afeta E diretamente: ele altera primeiro o potencial do neurônio de saída, o que modifica sua ativação; essa ativação influencia o erro local e, por fim, a perda total. Por isso, o gradiente surge naturalmente como um produto de derivadas encadeadas.

Para isso, faremos uso do diagrama a seguir, que explicita o *fluxo de informação* no passo direto e o *caminho de dependência* utilizado no cálculo das derivadas. Nele, a camada de entrada fornece x_1 e x_2 (além do termo constante 1 — conhecido como viés), que alimentam os somadores da camada oculta, produzindo os potenciais $u_1^{[1]}$ e $u_2^{[1]}$ e, após a aplicação de f_1, as ativações $y_1^{[1]}$ e $y_2^{[1]}$. Em seguida, essas ativações são combinadas na camada de saída para formar $u_1^{[2]}$ e $u_2^{[2]}$ e, após a ativação f_2, gerar as saídas $y_1^{[2]}$ e $y_2^{[2]}$. Por fim, cada saída é

comparada com seu respectivo alvo (t_1 e t_2), produzindo erros locais que são agregados pela função de perda $\mathcal{L}$, resultando no erro total do exemplo, denotado por E.

Essa representação será útil porque permite visualizar, de forma direta, por quais etapas um parâmetro (por exemplo, um peso $w_{i,j}^{[L]}$) influencia o valor final de E, justificando a aplicação da regra da cadeia ao longo do caminho correspondente.

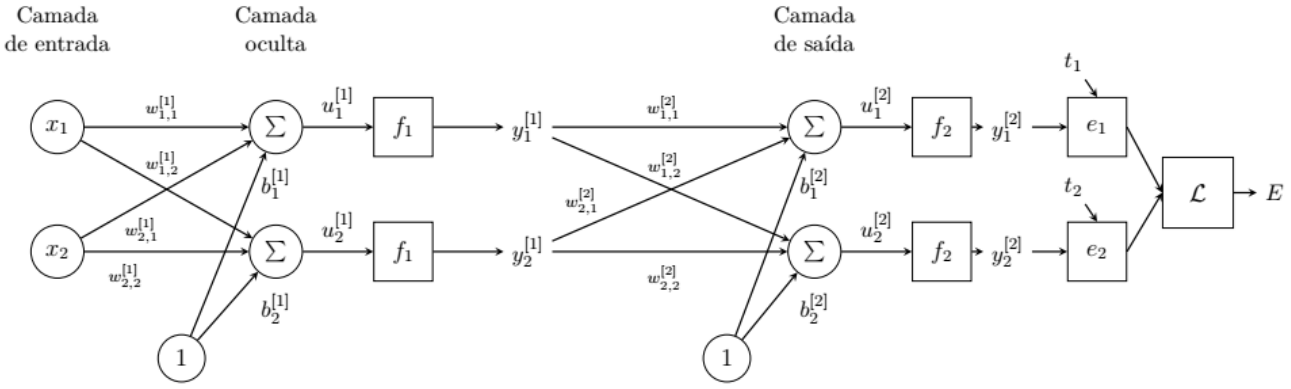


Figura 3.9: Diagrama para modelagem da retropropagação do erro.
Fonte: Autoria própria.

CÁLCULO DO GRADIENTE NA CAMADA DE SAÍDA Para ilustrar, considere o peso $w_{1,1}^{[2]}$, que conecta o neurônio $y_1^{[1]}$ (camada oculta) ao neurônio de saída 1. O caminho de dependência é

$$w_{1,1}^{[2]} \longrightarrow u_1^{[2]} \longrightarrow y_1^{[2]} \longrightarrow e_1 \longrightarrow E,$$

de modo que a regra da cadeia fornece

$$\frac{\partial E}{\partial w_{1,1}^{[2]}} = \frac{\partial E}{\partial e_1} \cdot \frac{\partial e_1}{\partial y_1^{[2]}} \cdot \frac{\partial y_1^{[2]}}{\partial u_1^{[2]}} \cdot \frac{\partial u_1^{[2]}}{\partial w_{1,1}^{[2]}}.$$

Nesse ponto, é importante notar que os fatores intermediários dependem das definições escolhidas para o erro e para a função de custo. Por exemplo, *no caso particular* em que adotamos o erro local como

$$e_1 = y_1^{[2]} - t_1$$

e a função de custo no exemplo como

$$E = \frac{1}{2}e_1^2 + \frac{1}{2}e_2^2,$$

temos

$$\frac{\partial E}{\partial e_1} = e_1 \qquad \text{e} \qquad \frac{\partial e_1}{\partial y_1^{[2]}} = 1.$$

Além disso, como $y_1^{[2]} = f_2(u_1^{[2]})$, segue que

$$\frac{\partial y_1^{[2]}}{\partial u_1^{[2]}} = f_2'\left(u_1^{[2]}\right).$$

Por fim, como

$$u_1^{[2]} = w_{1,1}^{[2]} y_1^{[1]} + w_{1,2}^{[2]} y_2^{[1]} + b_1^{[2]},$$

obtemos

$$\frac{\partial u_1^{[2]}}{\partial w_{1,1}^{[2]}} = y_1^{[1]}.$$

Substituindo esses resultados na cadeia, concluímos que

$$\frac{\partial E}{\partial w_{1,1}^{[2]}} = e_1 \, f_2'\Big(u_1^{[2]}\Big) \; y_1^{[1]}.$$

Vale destacar que, se a definição de E (ou de e_1) for modificada, então os termos $\frac{\partial E}{\partial e_1}$ e $\frac{\partial e_1}{\partial y_1^{[2]}}$ mudam de forma correspondente, embora a estrutura do cálculo pela regra da cadeia permaneça a mesma.

É conveniente reconhecer o termo $e_1 \, f_2'(u_1^{[2]})$ como o *delta* do neurônio de saída 1:

$$\delta_1^{[2]} = \frac{\partial E}{\partial u_1^{[2]}} = e_1 \, f_2'(u_1^{[2]}).$$

De maneira mais geral, sem fixar uma forma específica para o erro, o delta na camada de saída pode ser escrito como

$$\delta_i^{[L]} = \left(\frac{\partial E}{\partial e_i} \, \frac{\partial e_i}{\partial y_i^{[L]}} \right) \, f'\Big(u_i^{[L]}\Big).$$

Assim, obtemos a forma compacta do gradiente:

$$\frac{\partial E}{\partial w_{1,1}^{[2]}} = \delta_1^{[2]} \, y_1^{[1]}.$$

O mesmo raciocínio vale para $w_{1,2}^{[2]}$, pois o único termo que muda é $\dfrac{\partial u_1^{[2]}}{\partial w_{1,2}^{[2]}} = y_2^{[1]}$, levando a

$$\frac{\partial E}{\partial w_{1,2}^{[2]}} = \delta_1^{[2]} \, y_2^{[1]}.$$

De forma análoga, para o neurônio de saída 2 obtém-se

$$\frac{\partial E}{\partial w_{2,1}^{[2]}} = \delta_2^{[2]} \, y_1^{[1]}, \qquad \frac{\partial E}{\partial w_{2,2}^{[2]}} = \delta_2^{[2]} \, y_2^{[1]},$$

onde $\delta_2^{[2]} = \dfrac{\partial E}{\partial u_2^{[2]}} = e_2 \, f_2'(u_2^{[2]})$ no caso quadrático.

O PAPEL DO VIÉS NO GRADIENTE O viés participa do mesmo encadeamento, com a diferença de que ele entra em $u_i^{[2]}$ somando diretamente. Em particular,

$$u_i^{[2]} = \cdots + b_i^{[2]} \quad \Rightarrow \quad \frac{\partial u_i^{[2]}}{\partial b_i^{[2]}} = 1.$$

Assim, pela regra da cadeia,

$$\frac{\partial E}{\partial b_i^{[2]}} = \frac{\partial E}{\partial u_i^{[2]}} \cdot \frac{\partial u_i^{[2]}}{\partial b_i^{[2]}} = \delta_i^{[2]}.$$

Portanto, enquanto o gradiente de cada peso é o produto entre o delta do neurônio de destino e a ativação de entrada, o gradiente do viés coincide com o próprio delta, refletindo que o viés atua como um deslocamento direto do potencial.

CÁLCULO DO GRADIENTE NA CAMADA INTERMEDIÁRIA Após obtermos os deltas na camada de saída, o cálculo dos gradientes na camada intermediária (camada oculta) passa a ficar *organizado* e, sobretudo, *reaproveita* exatamente as quantidades já computadas na saída. A ideia é que um peso da camada intermediária não influencia o erro total E por um único caminho: ele altera o potencial de um neurônio oculto, muda a ativação desse neurônio e, a partir daí, essa alteração se propaga para **todos** os neurônios da camada de saída que recebem essa ativação. Assim, o gradiente na camada intermediária aparece como a soma das contribuições associadas a cada neurônio de saíd.

A Figura a seguir retoma a arquitetura da rede neural já apresentada anteriormente, agora com o acréscimo de cores para destacar caminhos específicos de propagação. Enquanto a versão anterior mostrava apenas a estrutura geral da rede, esta nova representação permite visualizar com mais clareza como determinadas conexões, ativações intermediárias e contribuições para o erro final podem ser acompanhadas ao longo das camadas. A linha vermelha destaca o encadeamento principal de propagação da informação até o erro final, enquanto a linha azul mostra uma influência adicional que atua como complemento necessário nesse processo, pois evidencia que a ativação intermediária não contribui apenas por um único caminho.

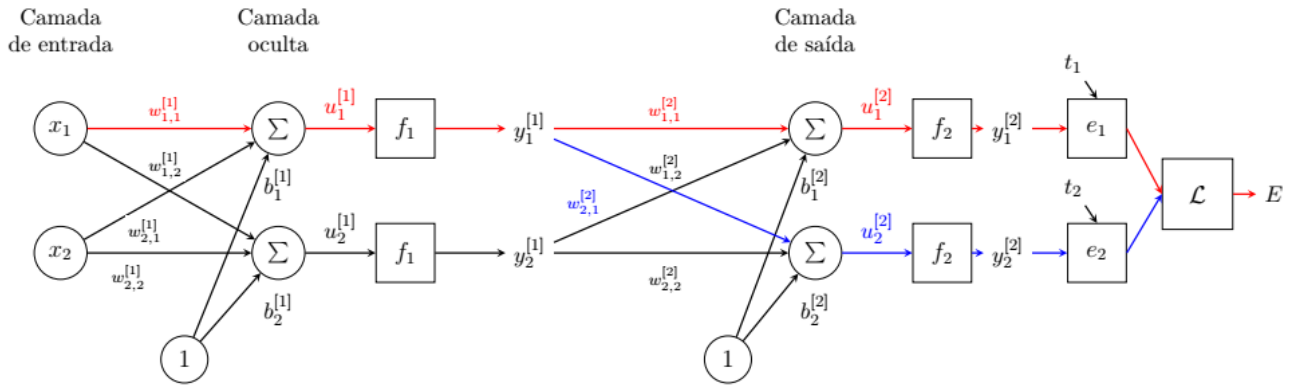


Figura 3.10: Diagrama para modelagem da retropropagação do erro, destacando o fluxo direto da propagação dos sinais até a saída e a etapa de cálculo do erro pela função de custo.
Fonte: Autoria própria.

Para fixar ideias na arquitetura da Figura 3.10, considere o peso $w_{1,1}^{[1]}$, que conecta a entrada x_1 ao primeiro neurônio da camada oculta. Esse peso afeta primeiramente $u_1^{[1]}$ e, em seguida, $y_1^{[1]}$; a partir daí, $y_1^{[1]}$ alimenta os dois neurônios da camada de saída, influenciando tanto $u_1^{[2]}$ quanto $u_2^{[2]}$. Portanto, existem dois caminhos do tipo

$$w_{1,1}^{[1]} \to u_1^{[1]} \to y_1^{[1]} \to u_1^{[2]} \to y_1^{[2]} \to e_1 \to E$$

e

$$w_{1,1}^{[1]} \to u_1^{[1]} \to y_1^{[1]} \to u_2^{[2]} \to y_2^{[2]} \to e_2 \to E$$

Como $w_{1,1}^{[1]}$ atua diretamente em $u_1^{[1]}$, temos o encadeamento inicial

$$w_{1,1}^{[1]} \longrightarrow u_1^{[1]} \longrightarrow y_1^{[1]}.$$

A partir de $y_1^{[1]}$, a influência se bifurca, pois $y_1^{[1]}$ alimenta os dois neurônios da camada de saída, afetando tanto $u_1^{[2]}$ quanto $u_2^{[2]}$ e, consequentemente, o custo final E. Assim, escrevemos separadamente:

$$\frac{\partial E}{\partial w_{1,1}^{[1]}} = \frac{\partial E}{\partial e_1}\frac{\partial e_1}{\partial y_1^{[2]}}\frac{\partial y_1^{[2]}}{\partial u_1^{[2]}}\frac{\partial u_1^{[2]}}{\partial y_1^{[1]}}\frac{\partial y_1^{[1]}}{\partial u_1^{[1]}}\frac{\partial u_1^{[1]}}{\partial w_{1,1}^{[1]}} + \frac{\partial E}{\partial e_2}\frac{\partial e_2}{\partial y_2^{[2]}}\frac{\partial y_2^{[2]}}{\partial u_2^{[2]}}\frac{\partial u_2^{[2]}}{\partial y_1^{[1]}}\frac{\partial y_1^{[1]}}{\partial u_1^{[1]}}\frac{\partial u_1^{[1]}}{\partial w_{1,1}^{[1]}}.$$

Nessa expressão, aparecem de forma explícita três tipos de derivadas com papéis distintos: o gradiente de E em relação ao erro individual e_i; o gradiente de e_i em relação à saída $y_i^{[2]}$; e o gradiente de E em relação ao potencial de ativação $u_i^{[2]}$. Como

$$y_i^{[2]} = f\big(u_i^{[2]}\big),$$

podemos escrever

$$\frac{\partial E}{\partial u_i^{[2]}} = \frac{\partial E}{\partial e_i}\frac{\partial e_i}{\partial y_i^{[2]}}\frac{\partial y_i^{[2]}}{\partial u_i^{[2]}}, \qquad i = 1, 2.$$

Substituindo essa identificação na expressão anterior, obtemos

$$\frac{\partial E}{\partial w_{1,1}^{[1]}} = \frac{\partial E}{\partial u_1^{[2]}}\frac{\partial u_1^{[2]}}{\partial y_1^{[1]}}\frac{\partial y_1^{[1]}}{\partial u_1^{[1]}}\frac{\partial u_1^{[1]}}{\partial w_{1,1}^{[1]}} + \frac{\partial E}{\partial u_2^{[2]}}\frac{\partial u_2^{[2]}}{\partial y_1^{[1]}}\frac{\partial y_1^{[1]}}{\partial u_1^{[1]}}\frac{\partial u_1^{[1]}}{\partial w_{1,1}^{[1]}}.$$

Observe que os dois termos possuem a mesma parte final, pois ambos passam pelo neurônio oculto $y_1^{[1]}$. Colocando esse fator comum em evidência, segue que

$$\frac{\partial E}{\partial w_{1,1}^{[1]}} = \left(\frac{\partial E}{\partial u_1^{[2]}}\frac{\partial u_1^{[2]}}{\partial y_1^{[1]}} + \frac{\partial E}{\partial u_2^{[2]}}\frac{\partial u_2^{[2]}}{\partial y_1^{[1]}}\right)\frac{\partial y_1^{[1]}}{\partial u_1^{[1]}}\frac{\partial u_1^{[1]}}{\partial w_{1,1}^{[1]}}.$$

Essa forma evidencia a ideia central da retropropagação: o erro produzido na camada de saída é transmitido para trás, acumulando as contribuições de todos os caminhos que passam

pelo neurônio oculto considerado. Usando a definição de $\delta_1^{[2]}$e $\delta_2^{[2]}$, isto é,

$$\delta_1^{[2]} = \frac{\partial E}{\partial u_1^{[2]}} \qquad \text{e} \qquad \delta_2^{[2]} = \frac{\partial E}{\partial u_2^{[2]}},$$

obtemos a seguinte expressão para a derivada parcial de E em relação a $w_{1,1}^{[1]}$ pode ser reescrita como

$$\frac{\partial E}{\partial w_{1,1}^{[1]}} = \left(\delta_1^{[2]} \frac{\partial u_1^{[2]}}{\partial y_1^{[1]}} + \delta_2^{[2]} \frac{\partial u_2^{[2]}}{\partial y_1^{[1]}} \right) \frac{\partial y_1^{[1]}}{\partial u_1^{[1]}} \frac{\partial u_1^{[1]}}{\partial w_{1,1}^{[1]}}.$$

Além disso, pela própria definição dos potenciais na saída,

$$u_1^{[2]} = w_{1,1}^{[2]} y_1^{[1]} + w_{1,2}^{[2]} y_2^{[1]} + b_1^{[2]} \quad \Rightarrow \quad \frac{\partial u_1^{[2]}}{\partial y_1^{[1]}} = w_{1,1}^{[2]},$$

$$u_2^{[2]} = w_{2,1}^{[2]} y_1^{[1]} + w_{2,2}^{[2]} y_2^{[1]} + b_2^{[2]} \quad \Rightarrow \quad \frac{\partial u_2^{[2]}}{\partial y_1^{[1]}} = w_{2,1}^{[2]}.$$

Como $y_1^{[1]} = f_1(u_1^{[1]})$, temos $\dfrac{\partial y_1^{[1]}}{\partial u_1^{[1]}} = f_1'(u_1^{[1]})$; e, como $u_1^{[1]} = w_{1,1}^{[1]} x_1 + w_{1,2}^{[1]} x_2 + b_1^{[1]}$, segue que $\dfrac{\partial u_1^{[1]}}{\partial w_{1,1}^{[1]}} = x_1$. Substituindo, obtemos a forma compacta

$$\frac{\partial E}{\partial w_{1,1}^{[1]}} = \left(\delta_1^{[2]} \, w_{1,1}^{[2]} + \delta_2^{[2]} \, w_{2,1}^{[2]} \right) f_1'(u_1^{[1]}) \, x_1.$$

Isso motiva definir o *delta* da camada intermediária como

$$\delta_1^{[1]} = \frac{\partial E}{\partial u_1^{[1]}} = \left(\delta_1^{[2]} \, w_{1,1}^{[2]} + \delta_2^{[2]} \, w_{2,1}^{[2]} \right) f_1'(u_1^{[1]}),$$

de modo que o gradiente assume o mesmo padrão visto na camada de saída:

$$\frac{\partial E}{\partial w_{1,1}^{[1]}} = \delta_1^{[1]} \, x_1 \qquad \text{(isto é, } \delta_1^{[1]} \, y_1^{[0]}\text{)}.$$

De forma análoga, para os demais pesos que chegam ao neurônio oculto 1,

$$\frac{\partial E}{\partial w_{1,2}^{[1]}} = \delta_1^{[1]} \, x_2, \qquad \frac{\partial E}{\partial b_1^{[1]}} = \delta_1^{[1]},$$

e, para o neurônio oculto 2,

$$\delta_2^{[1]} = \left(\delta_1^{[2]} \, w_{1,2}^{[2]} + \delta_2^{[2]} \, w_{2,2}^{[2]} \right) f_1'(u_2^{[1]}),$$

o que implica

$$\frac{\partial E}{\partial w_{2,1}^{[1]}} = \delta_2^{[1]} \, x_1, \qquad \frac{\partial E}{\partial w_{2,2}^{[1]}} = \delta_2^{[1]} \, x_2, \qquad \frac{\partial E}{\partial b_2^{[1]}} = \delta_2^{[1]}.$$

Em síntese, o cálculo na camada intermediária faz uso direto dos deltas da camada de saída: a cada neurônio oculto associa-se um delta que combina as contribuições provenientes de cada neurônio de saída, ponderadas pelos pesos que os conectam, e multiplicadas pela derivada da ativação local. Uma vez obtido $\delta^{[1]}$, os gradientes voltam a assumir o formato padrão "delta do neurônio de destino" vezes "entrada que chega ao peso". Assim, a regra da cadeia não apenas produz as fórmulas, mas também esclarece a interpretação: cada parâmetro é atualizado proporcionalmente ao quanto contribuiu para o erro observado na saída.

3.5.4 Forma matricial da propagação e da retropropagação

As expressões obtidas na subseção anterior foram escritas neurônio a neurônio, o que é importante para explicitar o papel da regra da cadeia no cálculo dos gradientes. No entanto, quando passamos à implementação computacional, é mais conveniente reunir essas mesmas informações em vetores e matrizes. Essa reorganização não altera o significado das quantidades envolvidas; ela apenas torna a escrita mais compacta e os cálculos mais adequados ao tratamento numérico.

No caso da rede 2-2-2, podemos agrupar as entradas da camada inicial no vetor

$$y^{[0]} = x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}.$$

De modo análogo, os potenciais de ativação e as saídas da camada oculta podem ser escritos como

$$u^{[1]} = \begin{bmatrix} u_1^{[1]} \\ u_2^{[1]} \end{bmatrix}, \qquad y^{[1]} = \begin{bmatrix} y_1^{[1]} \\ y_2^{[1]} \end{bmatrix},$$

enquanto, na camada de saída, escrevemos

$$u^{[2]} = \begin{bmatrix} u_1^{[2]} \\ u_2^{[2]} \end{bmatrix}, \qquad y^{[2]} = \begin{bmatrix} y_1^{[2]} \\ y_2^{[2]} \end{bmatrix}.$$

Os pesos da camada oculta ficam reunidos na matriz

$$W^{[1]} = \begin{bmatrix} w_{1,1}^{[1]} & w_{1,2}^{[1]} \\ w_{2,1}^{[1]} & w_{2,2}^{[1]} \end{bmatrix}, \qquad b^{[1]} = \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \end{bmatrix},$$

e os pesos da camada de saída na matriz

$$W^{[2]} = \begin{bmatrix} w_{1,1}^{[2]} & w_{1,2}^{[2]} \\ w_{2,1}^{[2]} & w_{2,2}^{[2]} \end{bmatrix}, \qquad b^{[2]} = \begin{bmatrix} b_1^{[2]} \\ b_2^{[2]} \end{bmatrix}.$$

Com essa notação, a propagação direta pode ser escrita de forma compacta por

$$u^{[1]} = W^{[1]} y^{[0]} + b^{[1]}, \qquad y^{[1]} = f_1\big(u^{[1]}\big),$$

$$u^{[2]} = W^{[2]} y^{[1]} + b^{[2]}, \qquad y^{[2]} = f_2\big(u^{[2]}\big)\,.$$

Essas equações são exatamente as mesmas obtidas anteriormente, mas agora reunidas em blocos. Por exemplo, a primeira igualdade corresponde simultaneamente às duas expressões

$$u_1^{[1]} = w_{1,1}^{[1]} x_1 + w_{1,2}^{[1]} x_2 + b_1^{[1]}, \qquad u_2^{[1]} = w_{2,1}^{[1]} x_1 + w_{2,2}^{[1]} x_2 + b_2^{[1]}.$$

Assim, a escrita matricial apenas sintetiza as relações já deduzidas na forma escalar.

O mesmo pode ser feito com a retropropagação. Reunindo os deltas da camada de saída no vetor

$$\delta^{[2]} = \begin{bmatrix} \delta_1^{[2]} \\ \delta_2^{[2]} \end{bmatrix},$$

temos, pela definição já obtida na subseção anterior,

$$\delta^{[2]} = \frac{\partial E}{\partial y^{[2]}} \odot f_2'\big(u^{[2]}\big)\,,$$

em que $\odot$ denota o produto de Hadamard, estudado no capítulo de Fundamentos Matemáticos. Em seguida, os deltas da camada oculta podem ser reunidos em

$$\delta^{[1]} = \begin{bmatrix} \delta_1^{[1]} \\ \delta_2^{[1]} \end{bmatrix},$$

e escritos matricialmente como

$$\delta^{[1]} = \big(W^{[2]}\big)^T \delta^{[2]} \odot f_1'\big(u^{[1]}\big)\,.$$

Essa fórmula resume, em uma única expressão, o fato já observado no cálculo componente a componente: cada delta da camada oculta é obtido pela soma das contribuições vindas dos neurônios da camada seguinte, ponderadas pelos pesos correspondentes, e multiplicada pela derivada da ativação local.

Uma vez calculados os vetores de deltas, os gradientes dos pesos e vieses também assumem forma matricial simples:

$$\frac{\partial E}{\partial W^{[2]}} = \delta^{[2]} \big(y^{[1]}\big)^T, \qquad \frac{\partial E}{\partial b^{[2]}} = \delta^{[2]},$$

$$\frac{\partial E}{\partial W^{[1]}} = \delta^{[1]} \big(y^{[0]}\big)^T, \qquad \frac{\partial E}{\partial b^{[1]}} = \delta^{[1]}.$$

Observe que essa escrita preserva o mesmo padrão encontrado anteriormente:

$$\frac{\partial E}{\partial w_{i,j}^{[k]}} = \delta_i^{[k]}\, y_j^{[k-1]}, \qquad \frac{\partial E}{\partial b_i^{[k]}} = \delta_i^{[k]}.$$

A diferença é que agora todos os gradientes de uma mesma camada são calculados de uma só vez por meio de produtos matriciais.

Essa forma de organização é especialmente útil na implementação computacional, pois linguagens e bibliotecas numéricas trabalham de maneira muito eficiente com vetores e matrizes. Desse modo, a formulação matricial não apenas simplifica a notação, mas também evidencia com mais clareza a estrutura algébrica da propagação direta e da retropropagação, preparando naturalmente a passagem para o código.

3.5.5 Formulação geral da retropopagação em redes multicamadas

Após a modelagem explícita realizada na seção anterior (para a arquitetura da Figura 3.9), podemos agora registrar as expressões na forma geral para uma rede com L camadas, preservando a mesma notação.

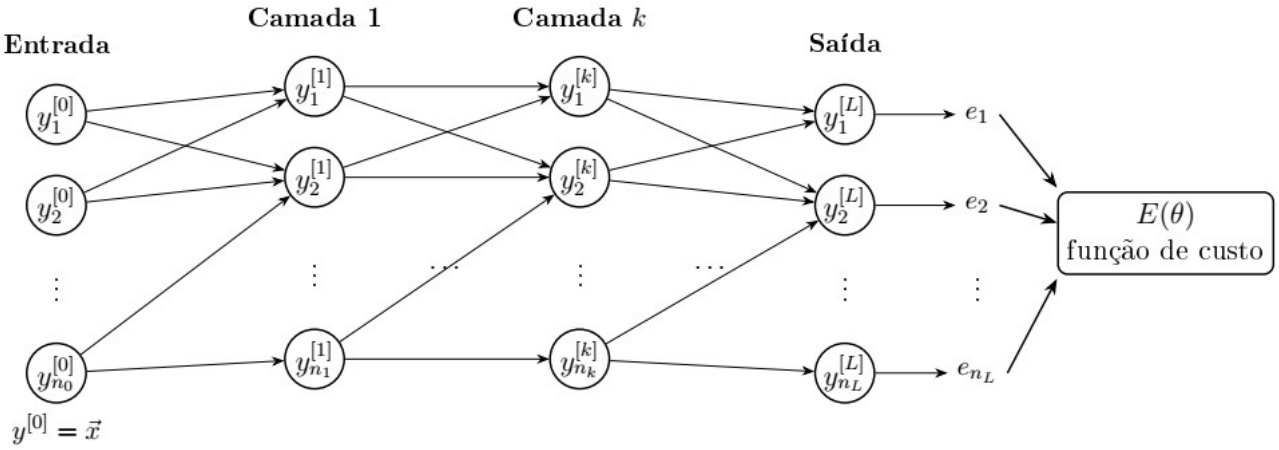


Figura 3.11: Esquema geral de uma rede neural multicamadas, ilustrando a passagem das ativações entre as camadas e a formulação geral usada para descrever o potencial de cada neurônio.

Fonte: Autoria própria.

A Figura 3.11 apresenta, de forma esquemática, uma rede neural multicamadas em sua formulação geral. Nessa representação, estão sendo evidenciados apenas os valores y, isto é, as saídas produzidas em cada camada e que passam a servir como entradas para os neurônios da camada seguinte. Assim, a figura não tem como objetivo detalhar todos os pesos, vieses ou combinações lineares envolvidas, mas sim destacar o fluxo das ativações ao longo da rede. Essa escolha é conveniente neste ponto da exposição, pois reforça a interpretação de que cada neurônio recebe como entrada os valores $y_j^{[k-1]}$ provenientes da camada anterior, o que será fundamental para a escrita das expressões gerais da propagação direta e, posteriormente, da retropropagação.

Considere uma rede com camadas indexadas por $k = 1, \ldots, L$, sendo L a camada de saída. Para cada camada k, com n_k neurônios, definimos o potencial (entrada líquida) do neurônio i por

$$u_i^{[k]} = \sum_{j=1}^{n_{k-1}} w_{i,j}^{[k]} \, y_j^{[k-1]} + b_i^{[k]},$$

e a ativação correspondente por

$$y_i^{[k]} = f\left(u_i^{[k]}\right).$$

Na camada de entrada, adotamos $y^{[0]} = \vec{x}$. Assim, o passo direto (*forward*) consiste em calcular sucessivamente $(u^{[1]}, y^{[1]}), (u^{[2]}, y^{[2]}), \ldots, (u^{[L]}, y^{[L]})$.

Para um único exemplo supervisionado $(\vec{x}, t)$, a rede produz uma saída $y^{[L]}$ ao final do *forward*. A perda do exemplo é escrita como $\mathcal{L}(y^{[L]}, t)$ e, portanto, a erro pode ser entendido como função dos parâmetros θ:

$$E(\theta) = \mathcal{L}\big(y^{[L]}(\theta), t\big).$$

Nosso objetivo é calcular como J varia quando alteramos cada peso e cada viés. Para organizar esse cálculo, introduzimos em cada neurônio o **sinal de erro** (ou *delta*)

$$\delta_i^{[k]} = \frac{\partial E}{\partial u_i^{[k]}},$$

isto é, a variação do custo em relação ao potencial $u_i^{[k]}$ do neurônio i na camada k.

DELTA NA CAMADA DE SAÍDA Na camada de saída ($k = L$), a regra da cadeia fornece

$$\delta_i^{[L]} = \frac{\partial E}{\partial y_i^{[L]}} \cdot \frac{\partial y_i^{[L]}}{\partial u_i^{[L]}} = \frac{\partial \mathcal{L}\big(y^{[L]}, t\big)}{\partial y_i^{[L]}} \cdot f'\left(u_i^{[L]}\right).$$

Essa expressão evidencia que o erro na saída combina a sensibilidade da perda em relação à predição com a sensibilidade do neurônio em relação ao seu potencial.

DELTA EM CAMADAS INTERMEDIÁRIAS Para uma camada intermediária $k < L$, o potencial $u_i^{[k]}$ influencia o custo apenas por meio dos neurônios da camada seguinte. Assim, a regra da cadeia produz a recorrência

$$\delta_i^{[k]} = \frac{\partial E}{\partial u_i^{[k]}} = \left(\sum_{\ell=1}^{n_{k+1}} \frac{\partial E}{\partial u_\ell^{[k+1]}} \frac{\partial u_\ell^{[k+1]}}{\partial y_i^{[k]}}\right) \frac{\partial y_i^{[k]}}{\partial u_i^{[k]}}.$$

Como $u_\ell^{[k+1]} = \sum_{j=1}^{n_k} w_{\ell,j}^{[k+1]} y_j^{[k]} + b_\ell^{[k+1]}$, segue que $\frac{\partial u_\ell^{[k+1]}}{\partial y_i^{[k]}} = w_{\ell,i}^{[k+1]}$ e, como $y_i^{[k]} = f(u_i^{[k]})$, temos $\frac{\partial y_i^{[k]}}{\partial u_i^{[k]}} = f'(u_i^{[k]})$. Portanto,

$$\delta_i^{[k]} = f'\left(u_i^{[k]}\right) \sum_{\ell=1}^{n_{k+1}} w_{\ell,i}^{[k+1]}\, \delta_\ell^{[k+1]}.$$

Esta expressão é a versão geral do que foi observado na modelagem: o delta em uma camada surge como a combinação dos deltas da camada seguinte, ponderados pelos pesos que conectam as camadas, multiplicada pela derivada local da ativação.

GRADIENTES DOS PESOS E VIESES Uma vez obtidos os deltas, as derivadas do custo em relação aos parâmetros seguem diretamente da forma de $u_i^{[k]}$. Para o peso $w_{i,j}^{[k]}$ (conexão entre

o neurônio j da camada $k-1$ e o neurônio i da camada k),

$$\frac{\partial E}{\partial w_{i,j}^{[k]}} = \frac{\partial E}{\partial u_i^{[k]}} \cdot \frac{\partial u_i^{[k]}}{\partial w_{i,j}^{[k]}} = \delta_i^{[k]}\, y_j^{[k-1]}.$$

Para o viés,

$$\frac{\partial E}{\partial b_i^{[k]}} = \frac{\partial E}{\partial u_i^{[k]}} \cdot \frac{\partial u_i^{[k]}}{\partial b_i^{[k]}} = \delta_i^{[k]},$$

pois $\dfrac{\partial u_i^{[k]}}{\partial b_i^{[k]}} = 1$. Assim, o gradiente de cada peso é o produto entre o delta do neurônio de destino e a ativação que chega pela camada anterior, enquanto o gradiente do viés coincide com o próprio delta.

Em síntese, a retropropagação pode ser vista como um procedimento em três etapas: (i) calcular $\delta^{[L]}$ na saída a partir da perda e de $f'(u)$; (ii) propagar os deltas para trás pela recorrência acima; e (iii) obter os gradientes de pesos e vieses pelas expressões $\frac{\partial E}{\partial w_{i,j}^{[k]}} = \delta_i^{[k]} y_j^{[k-1]}$ e $\frac{\partial E}{\partial b_i^{[k]}} = \delta_i^{[k]}$.

3.5.6 Algoritmo: retropropagação e atualização por descida do gradiente

Com os gradientes obtidos pela retropropagação, o ajuste dos parâmetros é realizado por um método de otimização. A escolha mais clássica é a descida do gradiente, na qual cada peso e cada viés é atualizado na direção oposta ao gradiente da função de custo. Nesta etapa, utilizamos a notação J em vez de E porque já não estamos apenas tratando do erro associado a um exemplo isolado, mas sim de um processo iterativo de treinamento. Como a otimização busca minimizar, ao longo das iterações, a função custo em relação aos parâmetros da rede, as atualizações passam a ser descritas em termos das derivadas de J. Denotando por $\alpha > 0$ a taxa de aprendizagem, as atualizações gerais para cada camada k podem ser escritas como

$$w_{i,j}^{[k]} \leftarrow w_{i,j}^{[k]} - \alpha\, \frac{\partial J}{\partial w_{i,j}^{[k]}}, \qquad b_i^{[k]} \leftarrow b_i^{[k]} - \alpha\, \frac{\partial J}{\partial b_i^{[k]}}.$$

Como vimos, pela retropropagação temos

$$\frac{\partial J}{\partial w_{i,j}^{[k]}} = \delta_i^{[k]} y_j^{[k-1]}, \qquad \frac{\partial J}{\partial b_i^{[k]}} = \delta_i^{[k]},$$

de modo que a atualização pode ser escrita diretamente em termos dos deltas:

$$w_{i,j}^{[k]} \leftarrow w_{i,j}^{[k]} - \alpha\, \delta_i^{[k]} y_j^{[k-1]}, \qquad b_i^{[k]} \leftarrow b_i^{[k]} - \alpha\, \delta_i^{[k]}.$$

Esse ciclo (passo direto $\rightarrow$ retropropagação $\rightarrow$ atualização) é repetido ao longo das épocas até que um critério de parada seja satisfeito, como alcançar um número máximo de iterações ou

observar um custo mínimo desejado.

Algoritmo 2 Treinamento por Retropropagação e Gradiente Descendente (Regime Supervisionado)

Entrada: Conjunto $\mathcal{D} = \{(\vec{x}_i, t_i)\}_{i=1}^{N}$; taxa de aprendizagem α; número de épocas N_{ep}; parâmetros iniciais $\{w_{i,j}^{[k]}, b_i^{[k]}\}$.

Saída: Parâmetros ajustados $\{w_{i,j}^{[k]}, b_i^{[k]}\}$.

1: **para** $ep = 1$ **até** N_{ep} **faça**
2: **para cada** exemplo $(\vec{x}, t)$ em $\mathcal{D}$ **faça**
3: **(Forward)** Calcule $y^{[0]} = \vec{x}$ e, para $k = 1, \ldots, L$, calcule $u^{[k]}$ e $y^{[k]}$.
4: **(Custo)** Calcule $J = \mathcal{L}(y^{[L]}, t)$.
5: **(Backward)** Calcule $\delta^{[L]}$ na saída e propague $\delta^{[k]}$ para $k = L-1, \ldots, 1$.
6: **(Atualização)** Para cada camada $k = 1, \ldots, L$:

$$w_{i,j}^{[k]} \leftarrow w_{i,j}^{[k]} - \alpha\, \delta_i^{[k]} y_j^{[k-1]}, \qquad b_i^{[k]} \leftarrow b_i^{[k]} - \alpha\, \delta_i^{[k]}.$$

7: **end para**
8: **end para**

4. Aplicações Ilustrativas de Redes Neurais Artificiais

Neste capítulo são apresentadas três aplicações com o objetivo de ilustrar, em situações concretas, alguns dos conceitos discutidos ao longo deste trabalho. Após o estudo da estrutura das redes neurais artificiais e da análise de elementos matemáticos centrais, como a combinação linear e o gradiente, busca-se agora observar como essas ideias podem ser empregadas na construção e no treinamento de modelos voltados à resolução de problemas específicos.

Para isso, são considerados dois problemas distintos, implementados por meio de códigos em Python. O primeiro problema consiste no ajuste de uma relação linear entre dados, permitindo visualizar de forma mais simples o papel dos parâmetros da rede e seu processo de atualização. O segundo problema envolve uma tarefa de classificação não linear, na qual a rede precisa aprender uma região de decisão a partir de exemplos fornecidos. Já o terceiro problema apresenta uma aplicação em que a rede é utilizada para aproximar uma solução associada a uma equação diferencial.

A escolha desses dois casos busca contemplar diferentes possibilidades de uso das redes neurais artificiais, mostrando que seu emprego não se restringe a um único tipo de tarefa. Ao contrário, tais modelos podem ser aplicados tanto em problemas mais elementares, que ajudam a explicitar seu funcionamento interno, quanto em situações mais elaboradas, nas quais a rede precisa capturar relações não lineares ou aproximar comportamentos descritos matematicamente.

Além disso, a utilização de códigos em Python neste capítulo tem a função de aproximar a formulação teórica da experimentação computacional. Por meio dessas implementações, torna-se possível observar de maneira mais concreta como as entradas são processadas, como os parâmetros são ajustados ao longo do treinamento e como a rede produz saídas que podem ser interpretadas de acordo com os problemas propostos. Nesse contexto, ao longo deste capítulo a implementação dos modelos será discutida com o auxílio de trechos de códigos desenvolvidos em Python, apresentados apenas na medida necessária para esclarecer as etapas principais da construção e do treinamento das redes neurais consideradas.

A opção por apresentar apenas partes do código tem como objetivo preservar a fluidez da exposição e destacar os elementos mais relevantes para a compreensão dos métodos utilizados. Uma breve introdução à linguagem Python, bem como os códigos completos empregados na resolução dos problemas apresentados neste capítulo, encontram-se reunidos no Apêndice A,

permitindo ao leitor interessado examinar com maior detalhe os aspectos computacionais das implementações realizadas. Assim, nas seções seguintes são apresentados, respectivamente, os dois problemas considerados, bem como as etapas principais de sua implementação e treinamento utilizando redes neurais artificiais.

4.1 AJUSTE LINEAR COM REDES NEURAIS ARTIFICIAIS

4.1.1 APRESENTAÇÃO DO PROBLEMA

O primeiro problema considerado neste capítulo consiste em utilizar uma rede neural artificial para aprender a relação entre valores em graus Celsius e seus correspondentes em graus Fahrenheit. A escolha dessa aplicação se justifica por seu caráter didático. Como a relação entre Celsius e Fahrenheit já é conhecida e pode ser descrita por uma função afim, torna-se mais fácil interpretar o papel desempenhado pelos parâmetros da rede ao longo do treinamento. Nesse caso, a rede busca ajustar um peso w e um viés b de modo que a saída produzida pelo modelo, dada por $\hat{y} = wC + b$, se aproxime dos valores esperados em Fahrenheit. Assim, o problema oferece um exemplo direto de como a combinação linear aparece no funcionamento de uma rede neural.

Além disso, essa aplicação permite acompanhar, de forma concreta, o processo de aprendizagem da rede neural. No código, após a inicialização aleatória dos parâmetros w e b, o modelo calcula a saída prevista para cada entrada, compara essa previsão com os valores reais e, a partir dessa diferença, determina o erro correspondente. Em seguida, são obtidas as quantidades dw e db, que representam as variações da função de custo em relação ao peso w e ao viés b, indicando como esses parâmetros devem ser corrigidos para reduzir o erro. Com base nessas quantidades, os parâmetros são atualizados segundo uma taxa de aprendizagem previamente fixada, em um procedimento iterativo repetido ao longo de 2000 épocas. Desse modo, essa aplicação não apenas ilustra o ajuste linear realizado pela rede, mas também evidencia o papel do gradiente no refinamento progressivo dos parâmetros durante o treinamento.

Neste exemplo, adotou-se o total de 2000 épocas como um limite de iterações do treinamento, por se tratar de um valor experimental suficiente para evidenciar a redução progressiva do erro e o ajuste dos parâmetros w e b. Assim, essa escolha não decorre de uma exigência teórica, mas da necessidade de tornar visível, de forma clara, o processo de aprendizagem da rede ao longo das atualizações sucessivas.

Ao final da implementação, o código exibe os valores aproximados encontrados para w e b, além de construir um gráfico que compara os dados reais com a reta ajustada pela rede neural.

4.1.2 IMPLEMENTAÇÃO COMPUTACIONAL

A implementação computacional foi realizada em Python, com o auxílio das bibliotecas `numpy` e `matplotlib`. O código foi construído de modo a reproduzir, em um caso simples, as

etapas fundamentais do treinamento de uma rede neural artificial.

Inicialmente, são definidos os dados de entrada e saída do problema. No código, o vetor `C` representa os valores de temperatura em graus Celsius, distribuídos uniformemente no intervalo de 0 a 100, enquanto o vetor `F` contém os valores correspondentes em graus Fahrenheit, obtidos pela relação

$$F = 1{,}8C + 32.$$

Desse modo, o conjunto de dados utilizado é formado por pares (x_i, t_i), em que x_i corresponde à temperatura em Celsius e t_i ao valor esperado em Fahrenheit. Essa etapa é importante porque define o problema de aprendizagem que será apresentado à rede.

```
C = np.linspace(0, 100, 50).reshape(-1, 1)
F = 1.8 * C + 32
```

Em seguida, são inicializados aleatoriamente os parâmetros w e b, que correspondem, respectivamente, ao peso e ao viés do modelo. Além disso, são fixados a taxa de aprendizagem α, o número de épocas N_{ep} e a quantidade de exemplos n. Esses elementos determinam o comportamento do treinamento: a taxa de aprendizagem controla o tamanho dos ajustes realizados em cada atualização, enquanto o número de épocas indica quantas vezes o conjunto de dados será percorrido ao longo do processo.

```
w = np.random.randn(1, 1)
b = np.random.randn(1, 1)
alpha = 1e-4
N_ep = 2000
n = len(C)
```

Um ponto que merece observação é a forma como esses parâmetros são inicializados. Em algumas implementações de redes neurais, os pesos e os vieses são inicialmente tomados como zero; em outras, adota-se a inicialização aleatória, como ocorre neste exemplo. A escolha dessa etapa depende do tipo de modelo e da estratégia de treinamento empregada. No caso considerado, opta-se por valores aleatórios para w e b, de modo que o processo de aprendizagem possa começar a partir de um estado inicial não trivial, permitindo que os parâmetros possam começar com valores iniciais que já os diferenciam de uns aos outros, o que pode favorecer o processo de aprendizado.

O laço de treinamento constitui a parte central da implementação, pois é nele que a rede ajusta seus parâmetros a partir dos exemplos do conjunto de dados. Em cada época, o algoritmo percorre todos os pares (x_i, t_i) e, para cada exemplo, executa sucessivamente as etapas de propagação direta, cálculo do erro, retropropagação e atualização dos parâmetros. No código, essa dinâmica aparece na estrutura de repetição

```
for ep in range(N_ep):
    soma_erro_quadratico = 0.0
```

```
    for i in range(n):
        x = C[i].reshape(1, 1)
        t = F[i].reshape(1, 1)
```

Inicialmente, para cada exemplo, são selecionados o valor de entrada x e a saída desejada t. Em seguida, realiza-se a etapa de propagação direta (*forward*), na qual se calcula a combinação linear entre entrada, peso e viés:

$$u = xw + b.$$

Como, neste exemplo, foi adotada uma função de ativação linear, a saída da rede coincide com o próprio potencial de ativação, isto é,

$$y = u.$$

No programa, essa etapa aparece explicitamente no trecho

```
u = x @ w + b
y = u
```

Nesse caso, a saída da rede é obtida diretamente pela combinação linear entre a entrada, o peso e o viés, sem a introdução de não linearidades adicionais. Após o cálculo da saída, determina-se o erro cometido pela rede naquele exemplo, comparando o valor previsto y com o valor esperado t. No código, isso é feito por meio da instrução

```
erro = y - t
```

Essa quantidade representa o desvio associado a um único exemplo do conjunto de treinamento. Entretanto, para acompanhar o comportamento global do processo de aprendizagem ao longo das épocas, o código acumula os quadrados desses erros individuais. Isso aparece na instrução `soma_erro_quadratico += float(erro**2)`. Como a variável `erro` está armazenada como um arranjo 1×1 do `NumPy`, o comando `float` é utilizado para converter o resultado de `erro**2` em um valor escalar, permitindo sua soma direta ao acumulador. Ao final de cada época, essa soma é dividida pelo número de exemplos, produzindo o erro quadrático médio dado por

$$EQM = \frac{1}{n}\sum_{i=1}^{n}(y_i - t_i)^2.$$

No programa, esse valor é obtido e armazenado por meio das instruções

```
eqm = soma_erro_quadratico / n
historico_eqm.append(eqm)
```

O armazenamento desses valores em `historico_eqm` é relevante porque permite analisar a evolução do treinamento ao longo das épocas, verificando se o erro médio diminui de forma progressiva.

Na etapa de retropropagação, o código calcula a quantidade `delta` a partir da expressão

```
delta = 2 * erro
```

Essa escolha decorre da derivada da função de custo quadrática em relação à saída da rede. De fato, considerando

$$J = (y - t)^2,$$

tem-se

$$\frac{dJ}{dy} = 2(y - t).$$

Logo, como $erro = y - t$, segue que

$$delta = 2\,erro.$$

Além disso, como a função de ativação adotada é linear, sua derivada é igual a 1. Assim, não surge nenhum fator adicional nessa etapa, e o valor de δ já fornece diretamente a direção de ajuste necessária para os parâmetros.

Na sequência, o algoritmo atualiza o peso e o viés segundo a regra do gradiente descendente. No código, isso aparece em

```
w = w - alpha * (x.T @ delta)
b = b - alpha * delta
```

Em termos matemáticos, essas atualizações podem ser escritas como

$$w \leftarrow w - \alpha\frac{\partial J}{\partial w}, \qquad b \leftarrow b - \alpha\frac{\partial J}{\partial b},$$

ou, de forma mais explícita neste caso,

$$w \leftarrow w - \alpha(x^T\delta), \qquad b \leftarrow b - \alpha\delta.$$

Lembremos que $\delta = \frac{\partial J}{\partial u}$ e que $u = xw + b$. Assim, pela regra da cadeia,

$$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial u}\frac{\partial u}{\partial w} = \delta\, x^T = x^T\delta,$$

enquanto

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial u}\frac{\partial u}{\partial b} = \delta \cdot 1 = \delta.$$

Esse é um dos pontos mais importantes da implementação, pois corresponde ao momento em que a rede efetivamente aprende, isto é, ajusta seus parâmetros com o objetivo de reduzir gradualmente o erro cometido nas previsões.

Concluídas todas as épocas, o código utiliza os parâmetros finais para calcular a saída da rede em todo o conjunto de entradas e, com isso, comparar os valores previstos com os dados esperados. Além disso, são exibidos os valores aproximados encontrados para w, b e para o erro quadrático médio final, bem como dois gráficos: um deles compara os dados reais com a reta

ajustada pela rede, enquanto o outro apresenta a evolução do erro quadrático médio ao longo do treinamento.

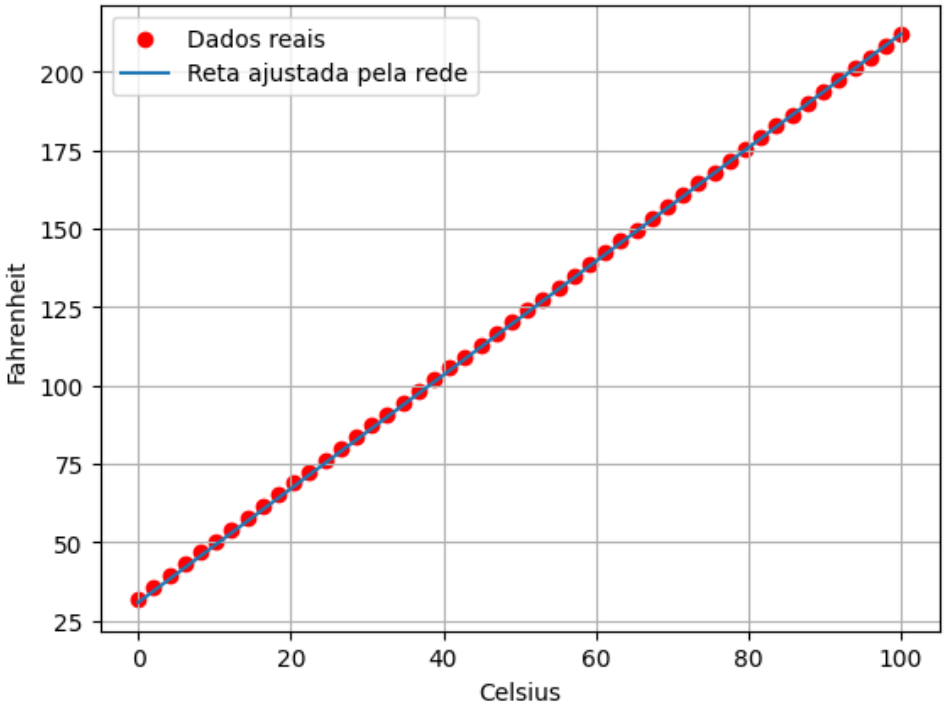


Figura 4.1: Resultado do ajuste linear obtido pela rede neural.
Fonte: Autoria própria.

A Figura 4.1 apresenta o resultado do ajuste produzido pela rede neural após o término do treinamento. Nela, os pontos em vermelho representam os dados originais do conjunto de treinamento, enquanto a reta traçada corresponde à função aprendida pela rede. Como se trata de um problema linear, observa-se que a reta ajustada acompanha adequadamente o comportamento dos dados, indicando que os valores finais de w e b foram capazes de modelar a relação entre as temperaturas em Celsius e Fahrenheit. Em uma das execuções realizadas, por exemplo, foram obtidos os valores aproximados $w \approx 1{,}8126$ e $b \approx 30{,}7315$. Esses resultados mostram que a rede aprendeu coeficientes bastante próximos daqueles esperados teoricamente para a conversão entre Celsius e Fahrenheit, cuja relação é dada por $F = 1{,}8\,C + 32$. Assim, mesmo não reproduzindo exatamente os valores teóricos, o modelo foi capaz de encontrar uma aproximação bastante satisfatória, o que se reflete tanto nos parâmetros finais quanto no bom ajuste visual apresentado no gráfico.

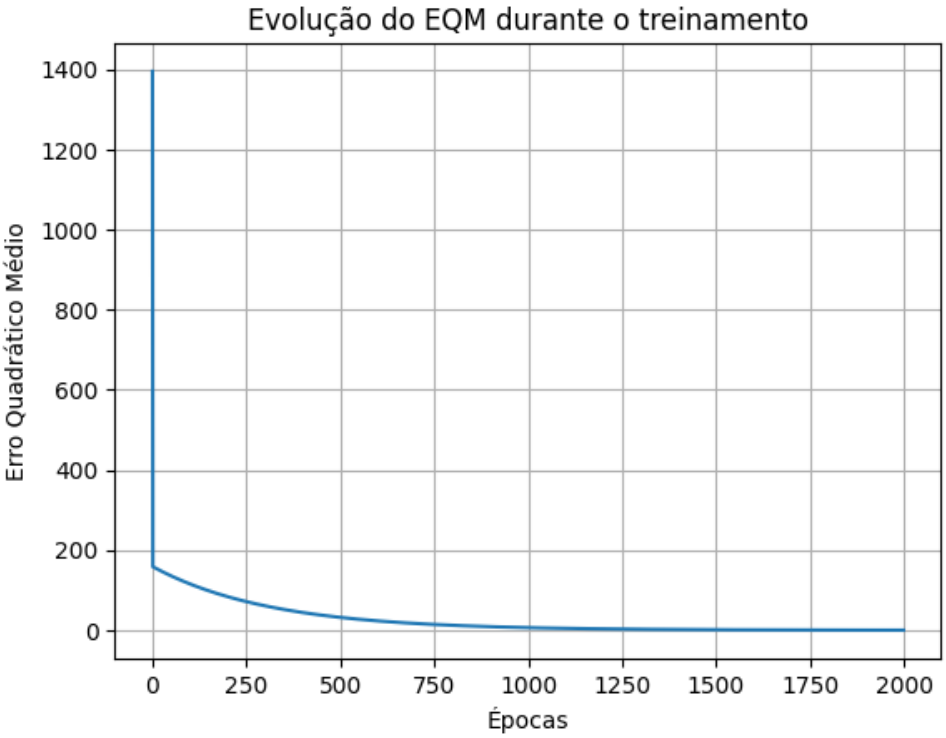


Figura 4.2: Evolução do erro quadrático médio ao longo do treinamento.
Fonte: Autoria própria.

Por sua vez, a Figura 4.2 mostra a evolução do erro quadrático médio ao longo das épocas. Esse gráfico permite analisar o comportamento da função de custo durante o treinamento, evidenciando a redução progressiva do erro à medida que os parâmetros são atualizados pelo gradiente descendente. Em particular, a tendência decrescente da curva indica que a rede está aprendendo de forma adequada, aproximando suas saídas dos valores esperados a cada iteração.

4.1.3 Influência do número de épocas no desempenho do treinamento

Uma vez apresentada a implementação do problema e analisado o comportamento da rede ao longo do treinamento, torna-se interessante investigar como a variação do número de épocas interfere no desempenho do modelo. Essa análise é relevante porque o número de épocas determina quantas vezes o algoritmo percorre o conjunto de treinamento realizando atualizações nos parâmetros w e b. Em termos práticos, um número reduzido de épocas pode não ser suficiente para que a rede aproxime adequadamente os dados, ao passo que um número maior tende a favorecer o refinamento progressivo do ajuste.

Com o objetivo de observar esse comportamento de forma mais clara, foram realizadas diferentes execuções do mesmo problema, mantendo-se fixos os demais parâmetros do treinamento e variando-se apenas a quantidade de épocas. Neste caso, consideraram-se os valores 500, 750, 1000, 1250 e 1500 épocas. Em cada experimento, registrou-se o ajuste final produzido pela rede e a evolução do erro quadrático médio ao longo do processo. Desse modo, torna-se possível comparar como o aprendizado se desenvolve quando o treinamento é interrompido em

diferentes estágios.

Quando o número de épocas é menor, a rede ainda se encontra em uma etapa intermediária de aprendizagem. Nessa situação, os parâmetros w e b passaram por menos atualizações, de modo que a reta ajustada pode permanecer um pouco mais distante da relação teórica entre Celsius e Fahrenheit. Consequentemente, o erro quadrático médio tende a assumir valores mais elevados, refletindo o fato de que o modelo ainda não incorporou de maneira completa o padrão presente nos dados.

À medida que o número de épocas aumenta, observa-se uma melhora progressiva no ajuste obtido. Isso ocorre porque cada nova época fornece oportunidades adicionais para corrigir os parâmetros segundo a direção indicada pelo gradiente. Assim, o peso e o viés aproximam-se gradualmente dos valores ideais, e a reta aprendida pela rede passa a representar com maior fidelidade a relação afim subjacente ao problema.

As Figuras 4.3, 4.4, 4.5, 4.6 e 4.7 ilustram o comportamento do ajuste linear obtido pela rede para diferentes quantidades de épocas. Em cada uma delas, os pontos representam os dados reais, enquanto a reta indica a função aproximada aprendida pelo modelo ao final do treinamento. A comparação entre essas figuras permite perceber que, conforme o número de épocas cresce, a reta ajustada tende a se alinhar cada vez mais aos dados observados.

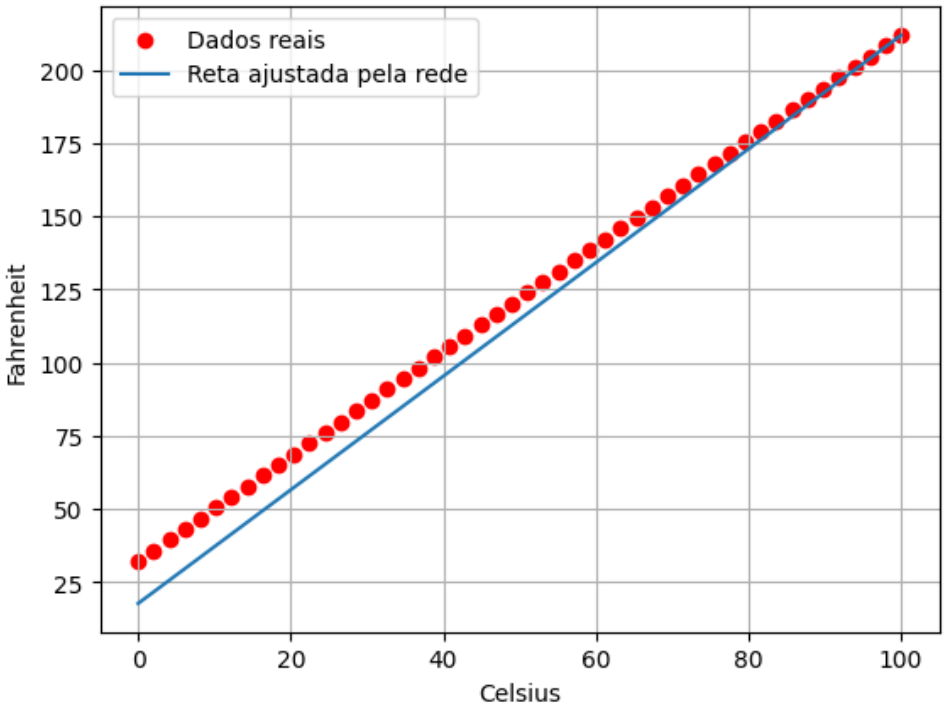


Figura 4.3: Ajuste linear produzido pela rede após 500 épocas de treinamento.
Fonte: Autoria própria.

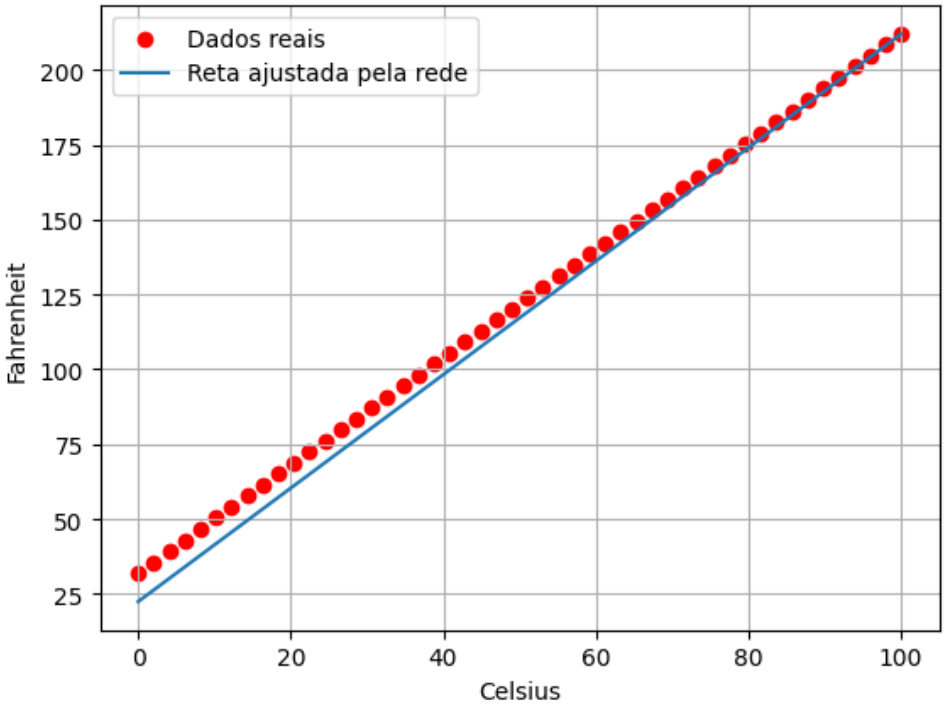


Figura 4.4: Ajuste linear produzido pela rede após 750 épocas de treinamento.
Fonte: Autoria própria.

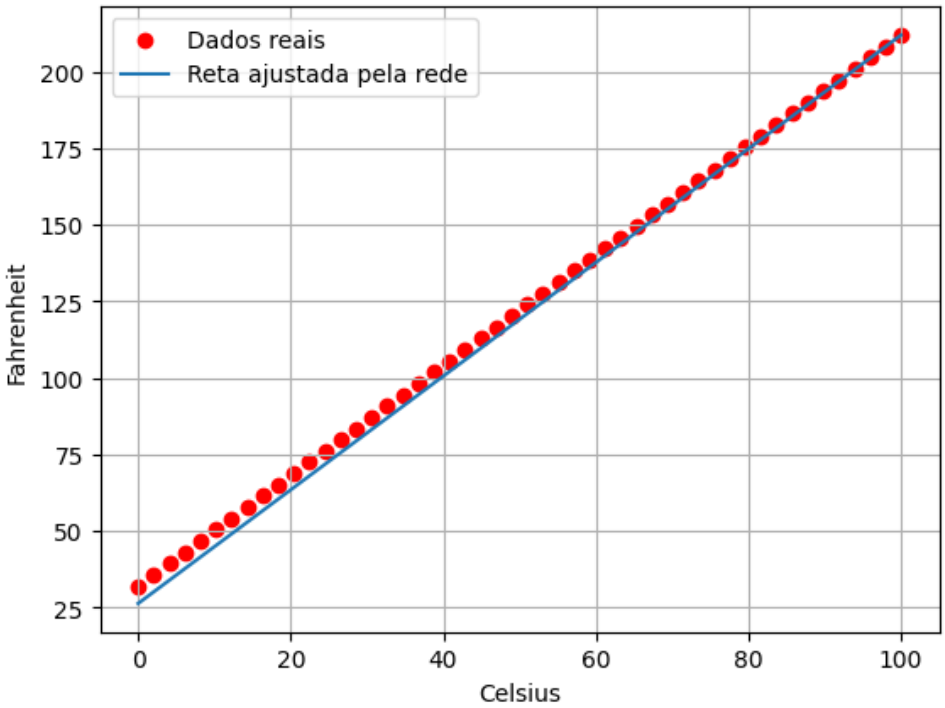


Figura 4.5: Ajuste linear produzido pela rede após 1000 épocas de treinamento.
Fonte: Autoria própria.

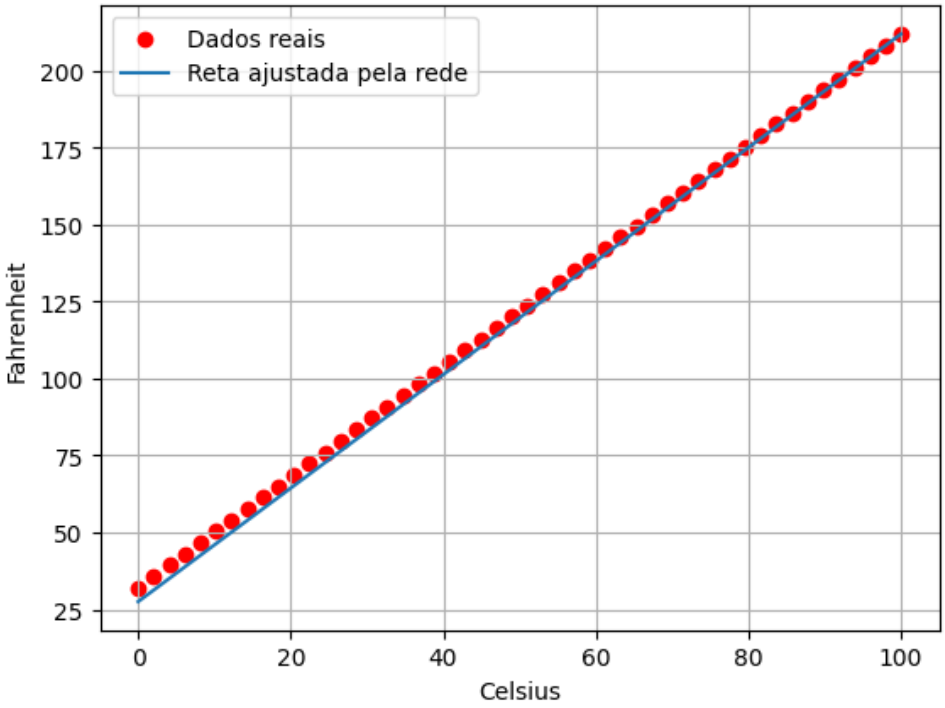


Figura 4.6: Ajuste linear produzido pela rede após 1250 épocas de treinamento.
Fonte: Autoria própria.

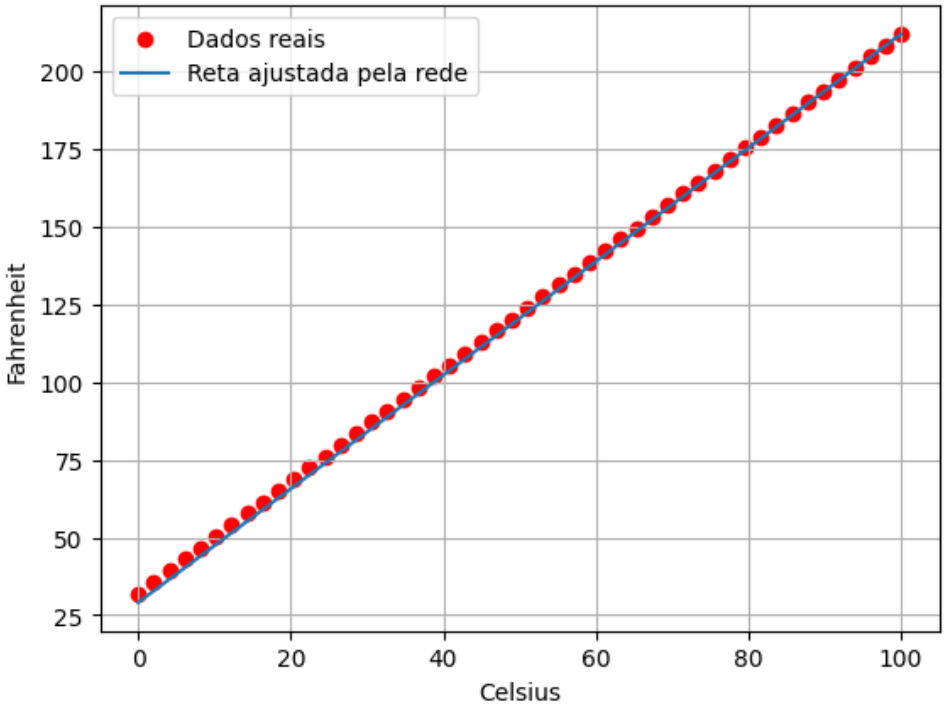


Figura 4.7: Ajuste linear produzido pela rede após 1500 épocas de treinamento.
Fonte: Autoria própria.

Em paralelo, o erro quadrático médio tende a diminuir, evidenciando a convergência do treinamento. A Figura 4.8 apresenta ao erro quadrático médio após 500 épocas de treinamento. Podemos comparar esse gráfico com a Figura 4.2, visualizando e comparando o comportamento

da função de custo ao longo do treinamento. Em todos os casos, espera-se uma tendência decrescente da curva, embora essa redução seja mais limitada quando o processo é interrompido após um número menor de épocas.

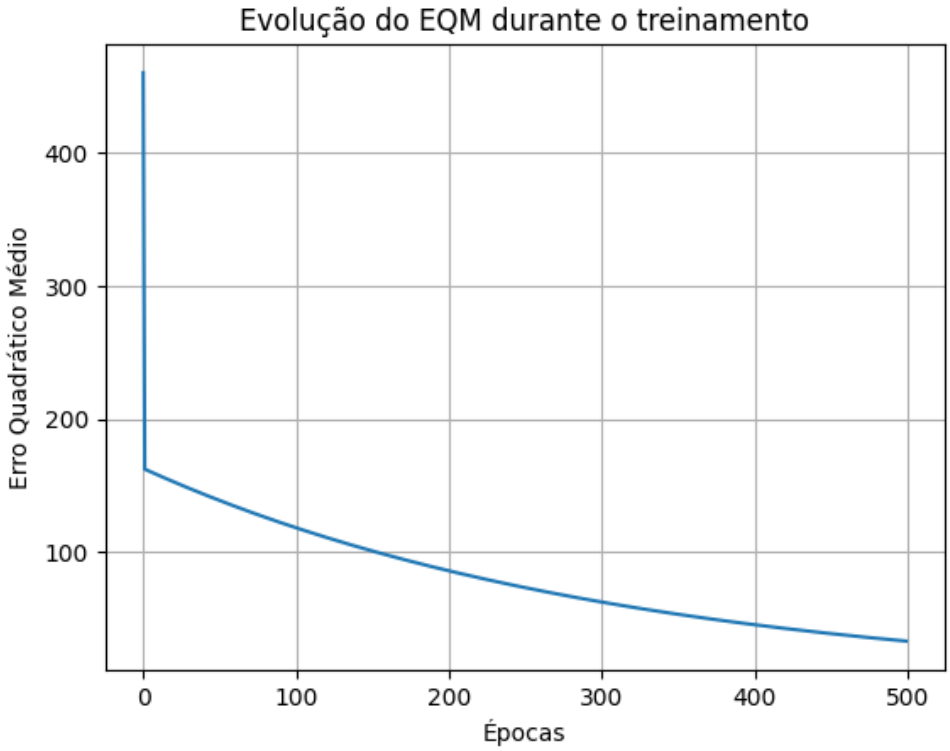


Figura 4.8: Evolução do erro quadrático médio ao longo de 500 épocas.
Fonte: Autoria própria.

4.2 Classificação não linear com redes neurais artificiais

4.2.1 Apresentação do problema

O segundo problema considerado neste capítulo consiste em utilizar uma rede neural artificial para realizar uma tarefa de classificação em duas variáveis. Diferentemente do exemplo anterior, em que o objetivo era ajustar uma relação linear entre entrada e saída, aqui busca-se identificar se um ponto do plano pertence ou não ao interior de uma região circular. Mais precisamente, para cada ponto (x_1, x_2), deseja-se associar à classe 1 os pontos que satisfazem a condição

$$x_1^2 + x_2^2 < 1,$$

enquanto os pontos que não satisfazem essa desigualdade são associados à classe 0.

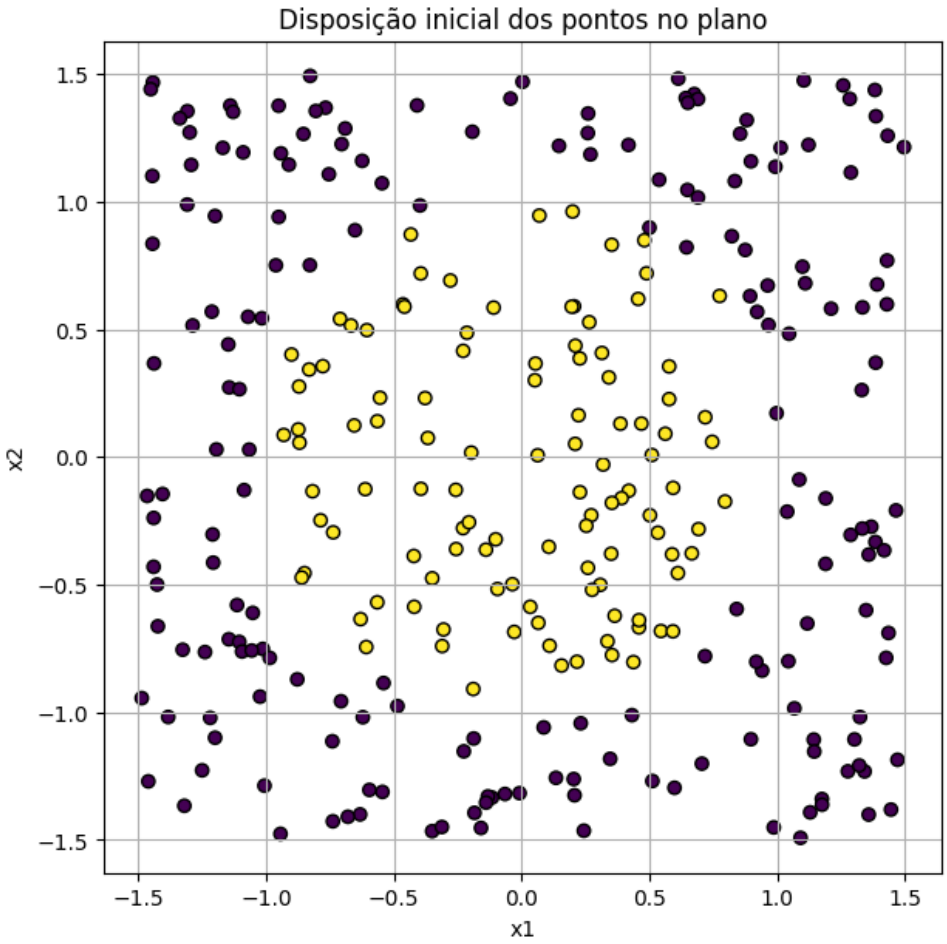


Figura 4.9: Disposição inicial dos pontos do conjunto de dados no plano, com destaque para a separação entre os pontos internos e externos ao círculo unitário.

A Figura 4.9 apresenta a disposição inicial dos pontos do conjunto de dados no plano, antes do processo de aprendizagem da rede. Nessa representação, os pontos já aparecem identificados de acordo com sua classe, o que permite visualizar de forma imediata a estrutura do problema: os elementos de uma classe concentram-se no interior do círculo unitário, enquanto os da outra ocupam a região externa. Essa visualização inicial é importante porque antecipa a natureza geométrica da tarefa que será proposta à rede neural.

A escolha desse problema possui interesse didático porque a fronteira de decisão envolvida não é linear. De fato, a equação

$$x_1^2 + x_2^2 = 1$$

descreve uma circunferência, de modo que a separação entre as classes não pode ser realizada adequadamente por um modelo baseado apenas em uma única combinação linear. Assim, esse exemplo evidencia a necessidade de uma arquitetura com camada oculta e função de ativação não linear, capaz de aprender regiões de classificação mais complexas.

Além disso, esse problema permite visualizar de maneira concreta o processo de aprendizagem da rede em sua formulação matricial. Ao longo do treinamento, os pesos e vieses são

ajustados para que a saída produzida pelo modelo se aproxime do valor desejado para cada ponto do conjunto de dados. Ao final, será possível comparar a disposição inicial dos pontos com a região de classificação aprendida pela rede, bem como acompanhar a evolução do erro quadrático médio ao longo das épocas, o que permite analisar tanto a qualidade visual da classificação quanto o comportamento do treinamento.

4.2.2 Implementação computacional

A implementação computacional deste problema também foi realizada em Python, com o auxílio das bibliotecas `numpy` e `matplotlib`. O código foi construído para reproduzir, de maneira simples, as etapas fundamentais do treinamento supervisionado de uma rede neural artificial aplicada a um problema de classificação não linear no plano.

Inicialmente, são gerados os dados do problema. O conjunto contém $N = 300$ pontos distribuídos aleatoriamente no quadrado $[-1{,}5, 1{,}5] \times [-1{,}5, 1{,}5]$. Cada coluna da matriz `X` representa um ponto do plano, com duas coordenadas de entrada. Em seguida, define-se o vetor de saídas desejadas `T`, atribuindo-se valor 1 aos pontos localizados no interior do círculo unitário e valor 0 aos pontos situados fora dele. No código, essa etapa aparece em

```
np.random.seed(0)
N = 300
X = np.random.uniform(-1.5, 1.5, (2, N))

T = (X[0]**2 + X[1]**2 < 1).astype(float).reshape(1, N)
```

Desse modo, o conjunto de dados utilizado é formado por pares $(\vec{x}_i, t_i)$, em que $\vec{x}_i = (x_{1,i}, x_{2,i})$ corresponde ao ponto de entrada e $t_i \in \{0, 1\}$ indica a classe à qual ele pertence. A tarefa da rede é, portanto, aprender a distinguir os pontos internos e externos ao círculo de raio 1.

Para esse problema, escolhemos por uma arquitetura 2-4-1, isto é, uma rede com 2 neurônios de entrada, 4 neurônios na camada oculta e 1 neurônio na camada de saída. Essa escolha foi feita porque a fronteira de decisão do problema não é linear: ela é dada por uma curva circular. Uma arquitetura muito pequena, como 2-2-1 ou 2-2-2, pode ser útil para fins didáticos, mas tende a ter menor flexibilidade para aproximar adequadamente essa fronteira. Ao introduzir 4 neurônios na camada oculta, a rede passa a dispor de maior capacidade de representação, mantendo ainda uma estrutura suficientemente simples para ser analisada e implementada.

Em seguida, são definidas as funções de ativação utilizadas na rede. Na camada oculta, emprega-se a tangente hiperbólica,

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}},$$

cuja derivada é dada por

$$\tanh'(x) = 1 - \tanh^2(x).$$

Já na camada de saída, utiliza-se a função sigmoide,

$$\sigma(x) = \frac{1}{1+e^{-x}},$$

cuja derivada pode ser escrita como

$$\sigma'(x) = \sigma(x)\big(1-\sigma(x)\big).$$

No programa, essas funções são implementadas por meio de

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

def tanh_prime(x):
    return 1 - np.tanh(x)**2

def sigmoide(x):
    return 1 / (1 + np.exp(-x))

def sigmoide_prime(x):
    s = sigmoide(x)
    return s * (1 - s)
```

A tangente hiperbólica é utilizada na camada oculta porque introduz a não linearidade necessária para que a rede possa modelar uma fronteira curva. Já a sigmoide é apropriada na saída porque produz valores no intervalo $(0, 1)$, o que combina naturalmente com a interpretação do problema como uma classificação binária.

Na sequência, são inicializados aleatoriamente os parâmetros da rede. Como a arquitetura escolhida é 2-4-1, os pesos e vieses são organizados de acordo com as dimensões das camadas. No código, essa etapa aparece em

```
W1 = np.random.randn(4, 2)
b1 = np.random.randn(4, 1)

W2 = np.random.randn(1, 4)
b2 = np.random.randn(1, 1)

alpha = 0.05
N_ep = 5000
n = N
```

Essa organização pode ser entendida de maneira bastante direta. Como a camada de entrada possui 2 variáveis e a camada oculta possui 4 neurônios, a matriz de pesos da primeira

camada deve transformar vetores de $\mathbb{R}^2$ em vetores de $\mathbb{R}^4$. Por isso,

$$W^{[1]} \in \mathbb{R}^{4\times 2}.$$

Mais explicitamente,

$$W^{[1]} = \begin{bmatrix} w_{1,1}^{[1]} & w_{1,2}^{[1]} \\ w_{2,1}^{[1]} & w_{2,2}^{[1]} \\ w_{3,1}^{[1]} & w_{3,2}^{[1]} \\ w_{4,1}^{[1]} & w_{4,2}^{[1]} \end{bmatrix}, \qquad b^{[1]} = \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} \in \mathbb{R}^{4\times 1}.$$

Isso significa que cada uma das 4 linhas de $W^{[1]}$ reúne os pesos que alimentam um neurônio da camada oculta a partir das 2 entradas. O vetor $b^{[1]}$, por sua vez, contém um viés para cada neurônio oculto.

De forma análoga, como a camada oculta possui 4 neurônios e a camada de saída possui apenas 1, a matriz de pesos da segunda camada deve transformar vetores de $\mathbb{R}^4$ em elementos de $\mathbb{R}$. Assim,

$$W^{[2]} \in \mathbb{R}^{1\times 4}, \qquad b^{[2]} \in \mathbb{R}^{1\times 1}.$$

Escrevendo explicitamente,

$$W^{[2]} = \begin{bmatrix} w_{1,1}^{[2]} & w_{1,2}^{[2]} & w_{1,3}^{[2]} & w_{1,4}^{[2]} \end{bmatrix}, \qquad b^{[2]} = \begin{bmatrix} b_1^{[2]} \end{bmatrix}.$$

Em outras palavras, o único neurônio de saída recebe 4 valores vindos da camada oculta, cada um ponderado por um peso, além de um viés escalar. Essa leitura por dimensões é importante porque mostra como a forma matricial traduz, diretamente, a estrutura da rede.

Assim como no primeiro problema, a inicialização aleatória faz com que os parâmetros comecem com valores distintos. Isso evita que todos os neurônios da camada oculta se comportem de forma idêntica desde o início do treinamento, o que prejudicaria a aprendizagem.

O laço de treinamento constitui a parte central da implementação, pois é nele que a rede ajusta seus parâmetros com base nos exemplos do conjunto de dados. Em cada época, o algoritmo percorre todos os pares $(\vec{x}_i, t_i)$ e, para cada exemplo, executa as etapas de propagação direta, cálculo do erro, retropropagação e atualização dos parâmetros. No código, essa dinâmica aparece na estrutura

```
for ep in range(N_ep):
    soma_erro_quadratico = 0.0

    for i in range(n):
        y0 = X[:, i].reshape(2, 1)
        t = T[:, i].reshape(1, 1)
```

Para cada exemplo, a entrada é organizada no vetor coluna

$$y^{[0]} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \in \mathbb{R}^{2\times 1},$$

enquanto o alvo t é um escalar armazenado como arranjo 1×1.

Na etapa de propagação direta, calcula-se inicialmente a combinação linear da camada oculta:

$$u^{[1]} = W^{[1]} y^{[0]} + b^{[1]}.$$

Como $W^{[1]}$ é 4×2, $y^{[0]}$ é 2×1 e $b^{[1]}$ é 4×1, o resultado é um vetor coluna $u^{[1]} \in \mathbb{R}^{4\times 1}$. Em seguida, aplica-se a tangente hiperbólica componente a componente, obtendo-se

$$y^{[1]} = \tanh\big(u^{[1]}\big) \in \mathbb{R}^{4\times 1}.$$

Na sequência, calcula-se a entrada da camada de saída:

$$u^{[2]} = W^{[2]} y^{[1]} + b^{[2]}.$$

Como $W^{[2]}$ é 1×4, $y^{[1]}$ é 4×1 e $b^{[2]}$ é 1×1, obtemos $u^{[2]} \in \mathbb{R}^{1\times 1}$. Aplicando a sigmoide, resulta

$$y^{[2]} = \sigma\big(u^{[2]}\big) \in \mathbb{R}^{1\times 1}.$$

No programa, essas etapas aparecem em

```
u1 = W1 @ y0 + b1
y1 = tanh(u1)

u2 = W2 @ y1 + b2
y2 = sigmoide(u2)
```

Assim, a saída final da rede é obtida por duas transformações lineares separadas por uma não linearidade. É essa composição que permite à rede aprender uma região de classificação mais complexa do que uma simples reta.

Após o cálculo da saída, determina-se o erro cometido pela rede naquele exemplo por meio da diferença entre a saída prevista e o valor esperado:

$$erro = y^{[2]} - t.$$

No código, isso é feito por meio da instrução

```
erro = y2 - t
```

Para medir quantitativamente esse erro, adotou-se a função custo quadrática. Para um

único exemplo, ela pode ser escrita como

$$E = \|y^{[2]} - t\|^2.$$

Como a saída da rede neste problema é escalar, isso equivale simplesmente a

$$E = (y^{[2]} - t)^2.$$

A escolha desse erro foi feita por duas razões principais. Em primeiro lugar, ele é simples e está em perfeita consonância com a dedução teórica feita anteriormente para a retropropagação, pois suas derivadas aparecem de maneira imediata. Em segundo lugar, ele permite acompanhar de forma clara o decréscimo do erro médio ao longo das épocas.

No código, os quadrados dos erros individuais são acumulados ao longo da época por meio da instrução

```
soma_erro_quadratico += float(np.sum(erro**2))
```

Como `erro` é armazenado como um arranjo 1×1 do `NumPy`, usa-se `float` para converter o resultado em um escalar numérico. Ao final de cada época, essa soma é dividida pelo número de exemplos, produzindo o erro quadrático médio:

$$EQM = \frac{1}{n}\sum_{i=1}^{n}(y_i - t_i)^2.$$

No programa, isso aparece em

```
eqm = soma_erro_quadratico / n
historico_eqm.append(eqm)
```

O armazenamento desses valores em `historico_eqm` é importante porque permite observar a evolução do treinamento e verificar se o ajuste dos parâmetros está, de fato, reduzindo o erro médio.

Na etapa de retropropagação, calcula-se inicialmente o delta da camada de saída. Como a saída usa a função sigmoide e o custo adotado é quadrático, temos

$$\delta^{[2]} = 2\,(y^{[2]} - t) \odot \sigma'(u^{[2]}).$$

Como a saída é escalar, o produto de Hadamard coincide aqui com o produto usual. No código, essa etapa aparece como

```
delta2 = 2 * erro * sigmoide_prime(u2)
```

Em seguida, esse erro é propagado para a camada oculta. Utilizando a regra da cadeia, obtém-se

$$\delta^{[1]} = \left((W^{[2]})^T \delta^{[2]}\right) \odot \tanh'(u^{[1]}).$$

Nesse caso, $(W^{[2]})^T$ possui dimensão 4×1, $\delta^{[2]}$ tem dimensão 1×1, e, portanto, $(W^{[2]})^T\delta^{[2]}$ resulta em um vetor 4×1, compatível com $\tanh'(u^{[1]})$. No programa, isso aparece como

```
delta1 = (W2.T @ delta2) * tanh_prime(u1)
```

Uma vez determinados os deltas, calculam-se os gradientes dos pesos e dos vieses. Para a camada de saída, temos

$$\frac{\partial E}{\partial W^{[2]}} = \delta^{[2]}(y^{[1]})^T, \qquad \frac{\partial E}{\partial b^{[2]}} = \delta^{[2]}.$$

Como $\delta^{[2]}$ é 1×1 e $(y^{[1]})^T$ é 1×4, o gradiente $\frac{\partial E}{\partial W^{[2]}}$ possui dimensão 1×4, exatamente a mesma de $W^{[2]}$.

Para a primeira camada, temos

$$\frac{\partial E}{\partial W^{[1]}} = \delta^{[1]}(y^{[0]})^T, \qquad \frac{\partial E}{\partial b^{[1]}} = \delta^{[1]}.$$

Como $\delta^{[1]}$ é 4×1 e $(y^{[0]})^T$ é 1×2, segue que $\frac{\partial E}{\partial W^{[1]}}$ tem dimensão 4×2, a mesma de $W^{[1]}$.

No código, isso aparece em

```
dW2 = delta2 @ y1.T
db2 = delta2

dW1 = delta1 @ y0.T
db1 = delta1
```

Essas expressões mostram de forma bastante clara uma das vantagens da formulação matricial: todos os gradientes de uma camada são obtidos de uma só vez, sem necessidade de escrever separadamente a derivada de cada peso.

Por fim, a atualização dos parâmetros é feita pelo método do gradiente descendente:

$$W^{[2]} \leftarrow W^{[2]} - \alpha\frac{\partial E}{\partial W^{[2]}}, \qquad b^{[2]} \leftarrow b^{[2]} - \alpha\frac{\partial E}{\partial b^{[2]}},$$

$$W^{[1]} \leftarrow W^{[1]} - \alpha\frac{\partial E}{\partial W^{[1]}}, \qquad b^{[1]} \leftarrow b^{[1]} - \alpha\frac{\partial E}{\partial b^{[1]}}.$$

No programa, isso aparece em

```
W2 = W2 - alpha * dW2
b2 = b2 - alpha * db2

W1 = W1 - alpha * dW1
b1 = b1 - alpha * db1
```

Esse é o momento em que a rede efetivamente aprende, isto é, modifica seus parâmetros de modo a reduzir progressivamente o erro cometido nas classificações.

Concluídas todas as épocas, o código utiliza os parâmetros finais para calcular a saída da rede em uma malha de pontos do plano. Essa malha é construída a partir de dois vetores igualmente espaçados, permitindo avaliar a rede em uma região contínua do domínio. No código, essa etapa aparece em

```
xx, yy = np.meshgrid(
    np.linspace(-1.5, 1.5, 200),
    np.linspace(-1.5, 1.5, 200)
)

grid = np.vstack([xx.ravel(), yy.ravel()])

u1_grid = W1 @ grid + b1
y1_grid = tanh(u1_grid)

u2_grid = W2 @ y1_grid + b2
y2_grid = sigmoide(u2_grid)

pred = y2_grid.reshape(xx.shape)
```

Com isso, obtém-se uma aproximação da região de classificação aprendida pela rede em todo o plano considerado. Para isso, constrói-se inicialmente uma malha regular de pontos no plano, isto é, um conjunto denso e organizado de pares (x_1, x_2), que não participa do treinamento, mas serve para avaliar a rede em toda a região observada. Esses pontos são então reorganizados em uma matriz cujas colunas representam entradas da rede, permitindo calcular, de uma só vez, as saídas correspondentes a todos os pontos da malha. Como a saída está no intervalo $(0, 1)$, pode-se interpretar valores maiores que 0,5 como pertencentes à classe dos pontos internos ao círculo. Essa comparação com 0,5, entretanto, não constitui uma nova função de ativação e não interfere no processo de aprendizagem da rede; trata-se apenas de um critério de decisão utilizado ao final, com a finalidade de transformar a saída contínua produzida pela sigmoide em uma classificação binária e, assim, tornar possível a visualização da fronteira de decisão aprendida.

Por fim, o programa produz dois gráficos. O primeiro mostra a região classificada pela rede, juntamente com os pontos do conjunto de treinamento:

```
plt.contourf(xx, yy, pred > 0.5, alpha=0.3)
plt.scatter(X[0], X[1], c=T.ravel(), edgecolors="k")
```

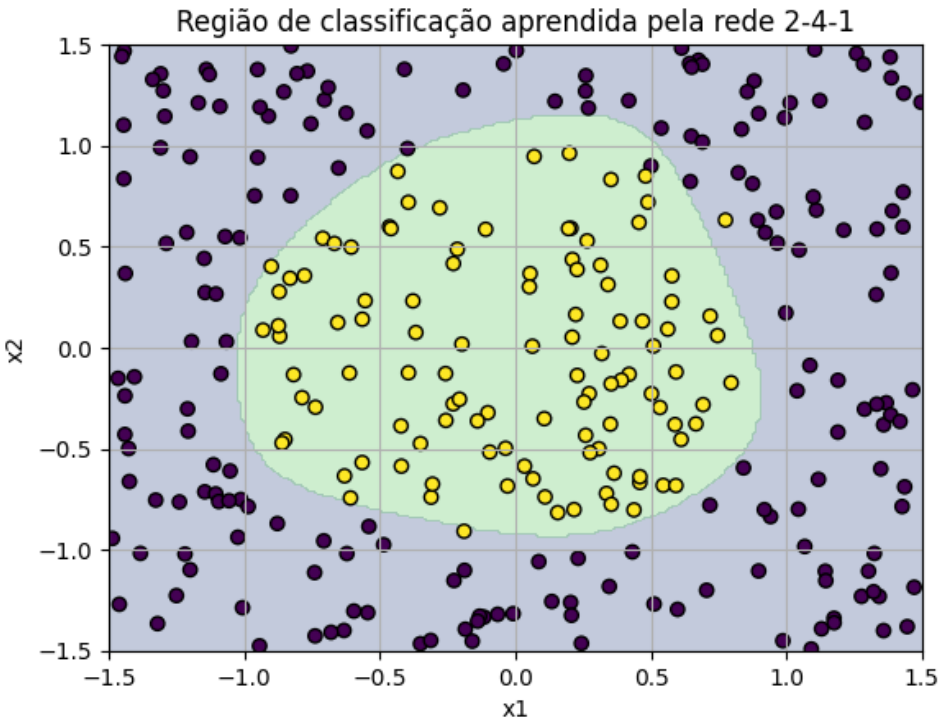


Figura 4.10: Região de classificação aprendida pela rede 2-4-1 para o problema de separação entre pontos internos e externos ao círculo unitário.
Fonte: Autoria própria.

Na Figura 4.10, a região preenchida representa a classe prevista pela rede, enquanto os pontos coloridos indicam os dados originais do problema. Já o segundo gráfico mostra a evolução do erro quadrático médio ao longo das épocas, permitindo acompanhar o comportamento da função custo durante o treinamento.

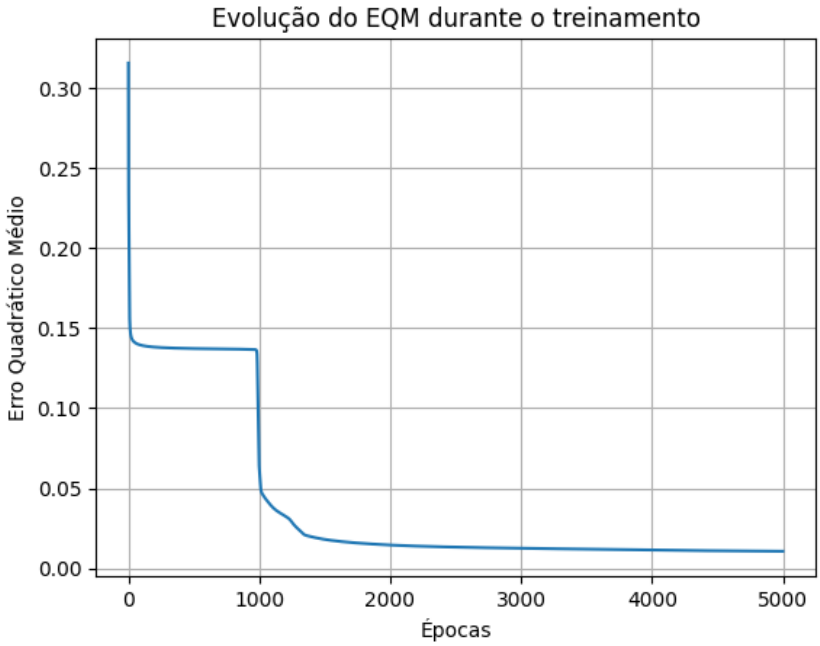


Figura 4.11: Evolução do erro quadrático médio ao longo do treinamento da rede 2-4-1 em 5000 épocas.

Fonte: Autoria própria.

Em conjunto, essas etapas mostram com clareza como uma rede 2-4-1 pode aprender uma tarefa de classificação não linear, ao mesmo tempo em que evidenciam a utilidade da formulação matricial para a implementação computacional.

4.2.3 Influência do número de épocas no desempenho do treinamento

Assim como no problema anterior, também é pertinente investigar, neste segundo caso, como a variação do número de épocas interfere no desempenho da rede neural. Essa análise torna-se ainda mais interessante aqui, pois se trata de uma tarefa de classificação não linear, cuja fronteira de decisão é mais complexa do que uma simples reta. Desse modo, o aumento do número de épocas não apenas contribui para a redução do erro, mas também permite observar como a região de classificação aprendida pela rede vai, progressivamente, se ajustando à geometria circular do problema.

Com o objetivo de tornar essa evolução mais visível, podem ser consideradas diferentes execuções do mesmo experimento, mantendo-se fixos os demais parâmetros da rede e variando-se apenas a quantidade de épocas. Para este problema, uma escolha particularmente interessante consiste em analisar os casos de 100, 500, 1000, 3000 e 5000 épocas. Essa seleção permite observar estágios bem distintos do treinamento: inicialmente, momento que a rede ainda apresenta grande imprecisão; ao decorrer do treinamento em que a região aprendida começa a se organizar; e estágios mais avançados, em que a fronteira de decisão se torna mais próxima da forma circular desejada.

Quando o número de épocas é muito pequeno, como no caso de 100, a rede ainda se

encontra em um estágio inicial de aprendizagem. Nessa situação, os pesos e vieses passaram por poucas atualizações, de modo que a região classificada pela rede tende a apresentar contornos ainda pouco adequados ao problema. Em geral, a separação entre os pontos internos e externos ao círculo ainda ocorre de maneira bastante imperfeita, o que se reflete também em valores mais elevados para o erro quadrático médio.

Ao considerar 500 e 1000 épocas, já é possível observar um progresso mais consistente. Nesses casos, a rede dispõe de mais varidades para ajustar os parâmetros das duas camadas e, com isso, a região de classificação passa a se aproximar melhor da forma esperada. Embora a fronteira ainda possa apresentar irregularidades, a separação entre as classes costuma tornar-se visualmente mais coerente, indicando que a rede está conseguindo extrair a estrutura geométrica subjacente aos dados.

Nos casos de 3000 e 5000 épocas, o treinamento já se encontra em um estágio mais avançado. Nessa fase, a rede teve tempo suficiente para refinar de modo mais cuidadoso os parâmetros, de forma que a região classificada tende a reproduzir com maior fidelidade a separação entre interior e exterior do círculo unitário. Além disso, a curva do erro quadrático médio costuma apresentar uma tendência de estabilização, sugerindo que a aprendizagem já alcançou uma aproximação satisfatória para o problema proposto.

As Figuras 4.12, 4.13, 4.14 e 4.15 ilustram a região de classificação produzida pela rede para diferentes quantidades de épocas. Em cada figura, a região sombreada representa a classe prevista pela rede, enquanto os pontos indicam os dados originais do conjunto de treinamento. A comparação entre essas imagens permite perceber como a fronteira de decisão vai sendo progressivamente refinada ao longo do treinamento.

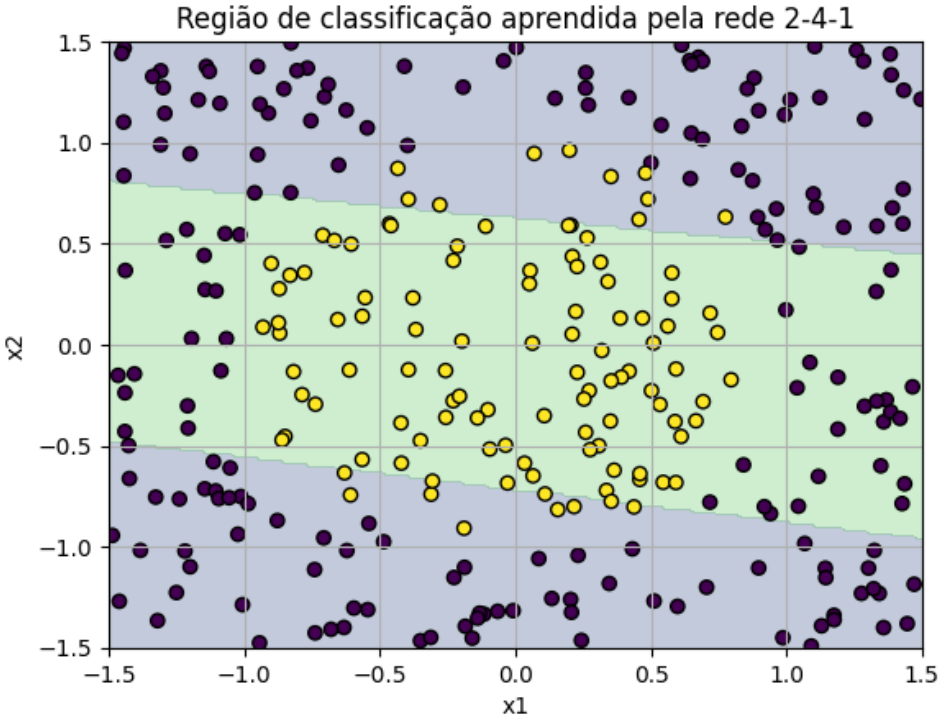


Figura 4.12: Região de classificação aprendida pela rede após 100 épocas de treinamento.
Fonte: Autoria própria.

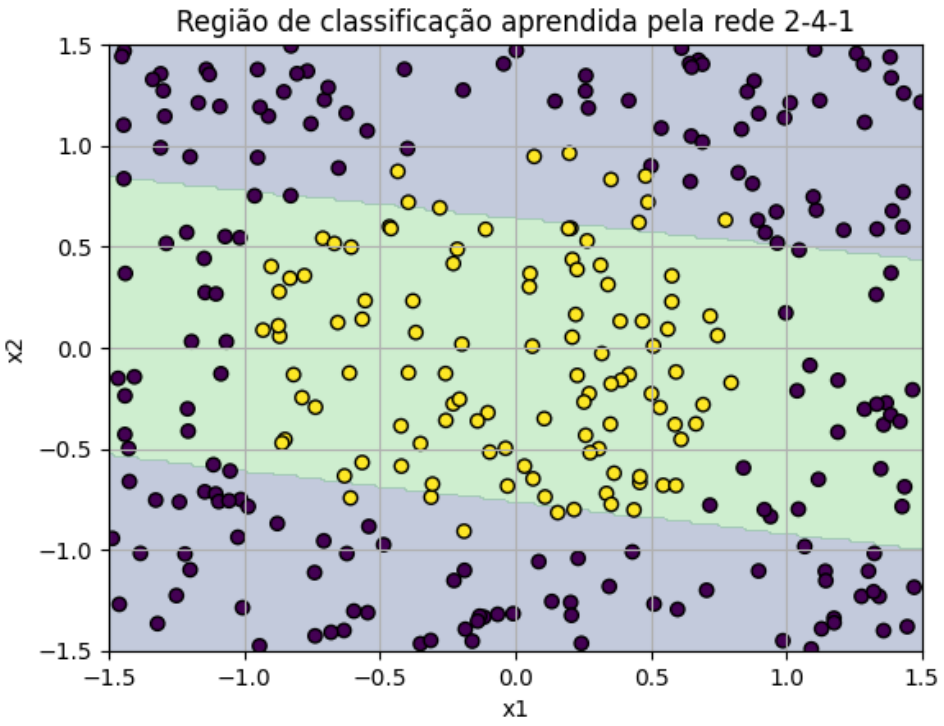


Figura 4.13: Região de classificação aprendida pela rede após 500 épocas de treinamento.
Fonte: Autoria própria.

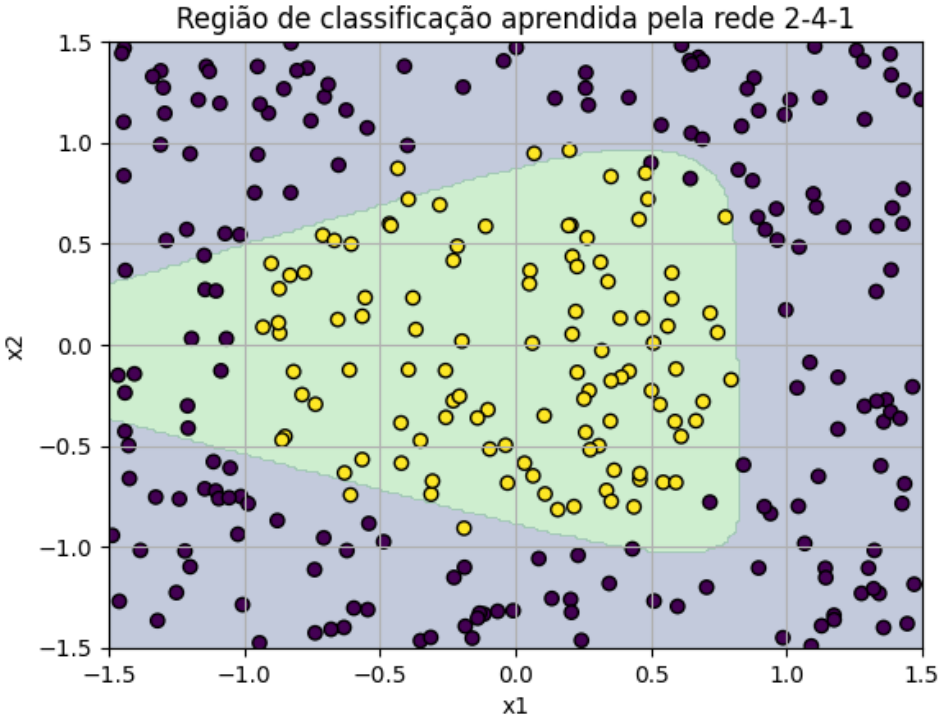


Figura 4.14: Região de classificação aprendida pela rede após 1000 épocas de treinamento.
Fonte: Autoria própria.

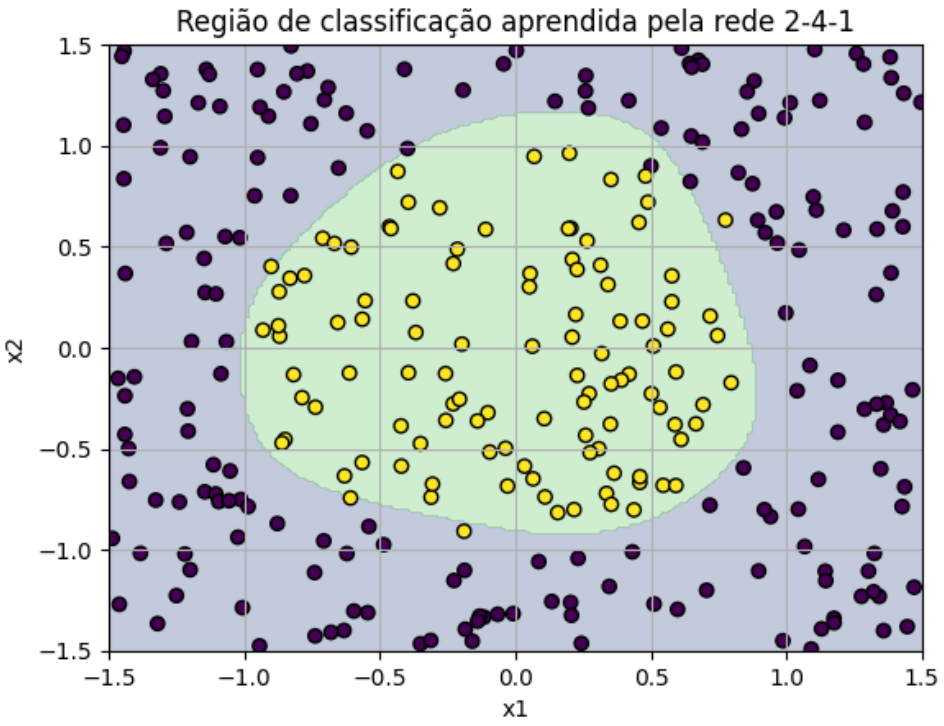


Figura 4.15: Região de classificação aprendida pela rede após 3000 épocas de treinamento.
Fonte: Autoria própria.

Em paralelo, o erro quadrático médio tende a diminuir, evidenciando a convergência do treinamento. No caso de 5000 épocas, a evolução desse erro já foi apresentada anteriormente na Figura 4.11. Como complemento, podemos comparar com a Figura 4.16, que apresenta o erro pela rede em uma quantidade menor de épocas, permitindo observar um estágio inicial do processo de aprendizagem. Ao comparar essa imagem com aquela obtida após 5000 épocas, percebe-se que, nas primeiras iterações, o erro ainda é maior, logo, com o avanço do treinamento, a rede passa a representar de forma mais adequada a separação entre os pontos internos e externos ao círculo unitário.

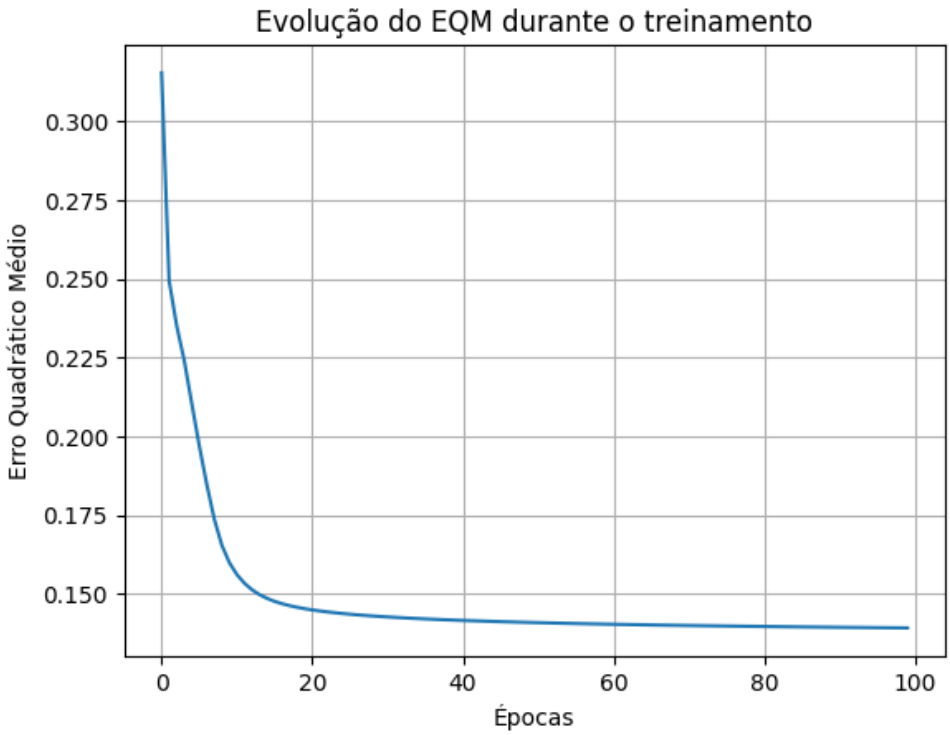


Figura 4.16: Evolução do erro quadrático médio ao longo do treinamento da rede 2-4-1 em 100 epócas.

Fonte: Autoria própria.

5. CONCLUSÕES

Ao longo deste trabalho, buscou-se desenvolver uma leitura matematicamente orientada das redes neurais artificiais, destacando que seu funcionamento não se reduz a uma implementação computacional ou ao uso de algoritmos prontos, mas se apoia em conceitos matemáticos fundamentais que permitem compreender, de modo mais rigoroso, como esses modelos recebem dados, os processam e aprendem com eles. Nesse sentido, o estudo concentrou-se especialmente em dois elementos centrais: a combinação linear e o gradiente, entendidos como partes essenciais da estrutura interna e do mecanismo de treinamento das redes neurais artificiais.

A análise desenvolvida ao longo dos capítulos permitiu evidenciar que a combinação linear está presente já na própria formulação do neurônio artificial, uma vez que é por meio dela que entradas, pesos e viés são organizados para produzir o potencial que será transformado pela função de ativação. De modo complementar, o estudo do gradiente e da regra da cadeia mostrou como o processo de aprendizagem pode ser interpretado matematicamente como um procedimento sistemático de ajuste de parâmetros voltado à redução do erro, o que se formaliza no algoritmo de retropropagação e na descida do gradiente. Assim, a investigação realizada a partir de exemplos contribui para tornar mais visível a matemática que sustenta o comportamento desses modelos.

Além da fundamentação teórica, o trabalho procurou aproximar tais conceitos de situações concretas por meio de aplicações ilustrativas. No problema de ajuste linear, observou-se que a rede foi capaz de aprender satisfatoriamente a relação entre temperaturas em graus Celsius e Fahrenheit, produzindo parâmetros próximos aos valores esperados teoricamente e apresentando redução progressiva do erro quadrático médio ao longo do treinamento. Já no problema de classificação não linear, a utilização de uma arquitetura com camada oculta permitiu representar uma fronteira de decisão mais compatível com a geometria circular do conjunto de dados, evidenciando a importância da não linearidade e da organização matricial no tratamento computacional da rede.

Desse modo, considera-se que os objetivos do trabalho foram alcançados, na medida em que se procurou mostrar que as redes neurais artificiais podem ser compreendidas não apenas como ferramentas tecnológicas amplamente difundidas, mas também como construções matemáticas cuja interpretação exige domínio de ideias de álgebra linear, cálculo diferencial e otimização. Sob essa perspectiva, o estudo reforça a relevância de abordar o tema em um curso de Licenciatura e Bacharelado em Matemática, pois ele possibilita articular conteúdos matemáticos tradicionais com um assunto atual, interdisciplinar e fortemente presente em diferentes

contextos científicos e tecnológicos.

Por fim, espera-se que este trabalho possa contribuir para uma compreensão mais clara e fundamentada das redes neurais artificiais, servindo como apoio para estudos posteriores e para novas aproximações entre a Matemática e temas contemporâneos da computação e da inteligência artificial. Também se abre, como possibilidade de continuidade, o aprofundamento em arquiteturas mais gerais, em outros tipos de funções de custo e ativação, bem como em aplicações adicionais em problemas de matemática aplicada, ampliando o diálogo entre teoria matemática, modelagem e experimentação computacional.

Referências Bibliográficas

[1] Bear, M. F., Connors, B. W. e Paradiso, M. A.: *Neurociências: Desvendando o Sistema Nervoso*. Artmed, 3ª ed., 2010.

[2] Bishop, C. M. e Bishop, H.: *Deep Learning: Foundations and Concepts*. Springer, 2024.

[3] Haykin, S.: *Neural Networks and Learning Machines*. Prentice Hall, 3ª ed., 2009.

[4] Kneusel, R. T.: *How AI Works: From Sorcery to Science*. No Starch Press, Inc., San Francisco, CA, 2024, ISBN 978-1-7185-0372-4. ISBN (ebook): 978-1-7185-0373-1. Library of Congress Control Number: 2023038565.

[5] Novikoff, A. B. J.: *On Convergence Proofs on Perceptrons*. Proceedings of the Symposium on the Mathematical Theory of Automata, pp. 615–622, 1962.

[6] Raina, R., Madhavan, A. e Ng, A. Y.: *Large-scale Deep Unsupervised Learning using Graphics Processors*. Em *Proceedings of the 26th International Conference on Machine Learning (ICML)*, 2009.

[7] Stewart, J., Clegg, D. K. e Watson, S.: *Cálculo*, vol. 1. Cengage Learning, São Paulo, 9ª ed., 2022, ISBN 9786555584011. Tradução da 9ª edição norte-americana.

A. Funções e recursos computacionais utilizados

Este apêndice apresenta, de forma sucinta, as principais funções, métodos e recursos da linguagem Python utilizados nos códigos desenvolvidos ao longo do trabalho. O objetivo não é fornecer um manual completo de programação, mas registrar brevemente o significado de cada elemento empregado na implementação computacional.

A.1 Bibliotecas utilizadas

`import numpy as np` Importa a biblioteca `NumPy`, especializada em computação numérica e manipulação eficiente de vetores, matrizes e arranjos multidimensionais. No código, ela é referenciada pela abreviação `np`.

`import matplotlib.pyplot as plt` Importa o módulo `pyplot` da biblioteca `Matplotlib`, utilizado para a construção de gráficos. No código, ele é referenciado pela abreviação `plt`.

A.2 Funções do NumPy

`np.linspace(a,b,n)` Gera n valores igualmente espaçados no intervalo de a até b. É uma função amplamente utilizada quando se deseja discretizar um intervalo numérico.

`np.random.seed(s)` Fixa uma semente para o gerador de números aleatórios. Isso faz com que a sequência gerada seja reprodutível, isto é, permite obter os mesmos resultados aleatórios em diferentes execuções do programa.

`np.random.uniform(a,b,size)` Gera números aleatórios com distribuição uniforme no intervalo $[a, b]$, organizados com o formato especificado em `size`.

`np.random.randn(m,n)` Gera números aleatórios com distribuição normal padrão, organizados em um arranjo de dimensão $m \times n$.

`np.tanh(x)` Calcula a tangente hiperbólica de x, elemento a elemento, quando x é um arranjo.

`np.exp(x)` Calcula a função exponencial e^x, também de forma elemento a elemento.

`np.meshgrid(x,y)` Constrói duas matrizes a partir de dois vetores unidimensionais, formando uma malha regular de pontos no plano. Essa função é muito útil para representar regiões bidimensionais e superfícies.

`np.vstack([...])` Empilha arranjos verticalmente, isto é, organiza vetores ou matrizes uns sobre os outros. Quando aplicada a dois vetores linha, produz uma matriz com essas linhas empilhadas.

`np.sum(x)` Soma todos os elementos do arranjo x, ou, em contextos mais gerais, realiza somas ao longo de eixos específicos.

`np.cos(x)` Calcula o cosseno de x, elemento a elemento.

`np.sin(x)` Calcula o seno de x, elemento a elemento.

A.3 Métodos e atributos de arranjos

`x.reshape(m,n)` Reorganiza os elementos do arranjo x em uma nova forma com m linhas e n colunas, sem alterar os dados armazenados. Esse recurso é importante para adequar vetores e matrizes às dimensões esperadas nas operações algébricas.

`x.reshape(-1,1)` Reorganiza o arranjo em uma matriz coluna, deixando que o próprio Python determine automaticamente o número de linhas necessário.

`x.astype(float)` Converte os elementos do arranjo x para o tipo numérico `float`, isto é, números reais em ponto flutuante.

`x.astype(int)` Converte os elementos do arranjo x para o tipo inteiro.

`x.ravel()` "Achata" o arranjo x, transformando-o em um vetor unidimensional com todos os seus elementos em sequência.

`x.T` Representa a transposta do arranjo x. Se x é uma matriz, então `x.T` troca linhas por colunas.

A.4 Funções definidas no próprio código

`tanh(x)` Função definida no código para representar explicitamente a tangente hiperbólica, utilizando internamente `np.tanh(x)`.

`tanh_prime(x)` Função definida para calcular a derivada da tangente hiperbólica, dada por

$$\tanh'(x) = 1 - \tanh^2(x).$$

`sigmoide(x)` Função definida para calcular a função sigmoide,

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

`sigmoide_prime(x)` Função definida para calcular a derivada da sigmoide, dada por

$$\sigma'(x) = \sigma(x)\big(1 - \sigma(x)\big).$$

A.5 Recursos básicos da linguagem Python

`range(n)` Gera uma sequência de inteiros de 0 até $n - 1$. É comumente utilizada em laços de repetição.

`len(x)` Retorna o número de elementos de um objeto iterável, como listas, vetores ou arranjos.

`float(x)` Converte x para um número real do tipo `float`.

`print(x)` Exibe na tela o conteúdo de x. É utilizada para apresentar resultados numéricos ao final do processamento.

`lista.append(x)` Insere o elemento x ao final de uma lista. Esse método é útil para armazenar sucessivamente valores produzidos ao longo da execução do programa.

`for` Estrutura de repetição que executa um bloco de instruções para cada elemento de uma sequência.

`if` Estrutura condicional que executa um bloco de instruções apenas quando uma determinada condição lógica é satisfeita.

`else` Parte complementar de uma estrutura condicional, executada quando a condição associada ao `if` não é satisfeita.

A.6 Operadores utilizados

`@` Operador de multiplicação matricial introduzido no Python para representar produtos entre vetores e matrizes de forma mais clara.

`+`, `-`, `*`, `/` Operadores aritméticos usuais de adição, subtração, multiplicação e divisão.

`**` Operador de potenciação. Por exemplo, `x**2` representa x^2.

`<` Operador de comparação estrita "menor que".

`>` Operador de comparação estrita "maior que".

A.7 FUNÇÕES DO Matplotlib

`plt.figure(figsize=(a,b))` Cria uma nova figura gráfica com dimensões proporcionais a a e b.

`plt.scatter(x,y,...)` Constrói um gráfico de dispersão a partir das coordenadas fornecidas.

`plt.plot(x,y,...)` Desenha linhas ou curvas no plano cartesiano.

`plt.contourf(x,y,z,...)` Produz um gráfico de regiões preenchidas a partir de uma malha e dos valores associados a cada ponto dessa malha.

`plt.xlabel(...)` Define o rótulo do eixo horizontal.

`plt.ylabel(...)` Define o rótulo do eixo vertical.

`plt.title(...)` Define o título do gráfico.

`plt.grid(True)` Ativa a grade do gráfico, facilitando a leitura visual das posições no plano.

`plt.axis("equal")` Ajusta a escala dos eixos para que uma unidade no eixo horizontal tenha o mesmo comprimento visual que uma unidade no eixo vertical.

`plt.legend()` Exibe a legenda associada aos elementos representados no gráfico.

`plt.show()` Exibe na tela a figura construída.

A.8 OBSERVAÇÃO FINAL

Em conjunto, essas funções e recursos permitem realizar operações numéricas, organizar dados em forma matricial, implementar o processo de treinamento da rede neural e construir visualizações gráficas dos resultados obtidos. A combinação entre `NumPy` e `Matplotlib` fornece, assim, uma base adequada para o desenvolvimento de experimentos computacionais de pequeno e médio porte em redes neurais artificiais.

B. Códigos-fonte utilizados no trabalho

Neste apêndice, apresentam-se os códigos-fonte desenvolvidos para os experimentos computacionais descritos no trabalho.

B.1 Código-fonte do problema de ajuste linear com redes neurais artificiais

```
# -*- coding: utf-8 -*-
"""Rede Neural 1

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1ekdupIiwGyUzuMwb7SlENHNexrscfA7Z
"""

import numpy as np
import matplotlib.pyplot as plt

C = np.linspace(0, 100, 50).reshape(-1, 1)

w = np.random.randn(1, 1)
b = np.random.randn(1, 1)

alpha = 1e-4
N_ep = 2000
n = len(C)

historico_eqm = []

for ep in range(N_ep):
```

```
    soma_erro_quadratico = 0.0

    for i in range(n):
        x = C[i].reshape(1, 1)
        t = F[i].reshape(1, 1)

        u = x @ w + b
        y = u

        erro = y - t

        soma_erro_quadratico += float(erro**2)

        delta = 2 * erro

        w = w - alpha * (x.T @ delta)
        b = b - alpha * delta

    eqm = soma_erro_quadratico / n
    historico_eqm.append(eqm)

Y = C @ w + b

print(f"w aproximado: {w[0,0]:.4f}")
print(f"b aproximado: {b[0,0]:.4f}")
print(f"EQM final: {historico_eqm[-1]:.6f}")

plt.scatter(C, F, color="red", label="Dados reais")
plt.plot(C, Y, label="Reta ajustada pela rede")
plt.xlabel("Celsius")
plt.ylabel("Fahrenheit")
plt.legend()
plt.grid(True)
plt.show()

plt.plot(historico_eqm)
plt.xlabel("Épocas")
plt.ylabel("Erro Quadrático Médio")
plt.title("Evolução do EQM durante o treinamento")
plt.grid(True)
```

```
plt.show()
```

B.2 Código-fonte do problema de classificação não linear com redes neurais artificiais

```
# -*- coding: utf-8 -*-
"""Classificacao_circunferencia

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1yRMEWWBKnPv9Wd284w2mR4W7loyTTw9D
"""

import numpy as np
import matplotlib.pyplot as plt

np.random.seed(0)
N = 300


X = np.random.uniform(-1.5, 1.5, (2, N))


T = (X[0]**2 + X[1]**2 < 1).astype(float).reshape(1, N)

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

def tanh_prime(x):
    return 1 - np.tanh(x)**2

def sigmoide(x):
    return 1 / (1 + np.exp(-x))

def sigmoide_prime(x):
    s = sigmoide(x)
    return s * (1 - s)

W1 = np.random.randn(4, 2)
```

```
b1 = np.random.randn(4, 1)


W2 = np.random.randn(1, 4)
b2 = np.random.randn(1, 1)

alpha = 0.05
N_ep = 5000
n = N


historico_eqm = []

for ep in range(N_ep):
    soma_erro_quadratico = 0.0

    for i in range(n):

        y0 = X[:, i].reshape(2, 1)
        t = T[:, i].reshape(1, 1)

        u1 = W1 @ y0 + b1
        y1 = tanh(u1)

        u2 = W2 @ y1 + b2
        y2 = sigmoide(u2)

        erro = y2 - t
        soma_erro_quadratico += float(np.sum(erro**2))


        delta2 = 2 * erro * sigmoide_prime(u2)
        delta1 = (W2.T @ delta2) * tanh_prime(u1)


        dW2 = delta2 @ y1.T
        db2 = delta2

        dW1 = delta1 @ y0.T
        db1 = delta1
```

```
        W2 = W2 - alpha * dW2
        b2 = b2 - alpha * db2

        W1 = W1 - alpha * dW1
        b1 = b1 - alpha * db1


    eqm = soma_erro_quadratico / n
    historico_eqm.append(eqm)

xx, yy = np.meshgrid(
    np.linspace(-1.5, 1.5, 200),
    np.linspace(-1.5, 1.5, 200)
)

grid = np.vstack([xx.ravel(), yy.ravel()])

u1_grid = W1 @ grid + b1
y1_grid = tanh(u1_grid)

u2_grid = W2 @ y1_grid + b2
y2_grid = sigmoide(u2_grid)

pred = y2_grid.reshape(xx.shape)

print("W1 aproximada:")
print(W1)

print("\nb1 aproximado:")
print(b1)

print("\nW2 aproximada:")
print(W2)

print("\nb2 aproximado:")
print(b2)

print(f"\nEQM final: {historico_eqm[-1]:.6f}")
```

```
plt.figure(figsize=(7,7))
plt.scatter(X[0], X[1], c=T, edgecolors="k")
plt.xlabel("x1")
plt.ylabel("x2")
plt.title("Disposição inicial dos pontos no plano")
plt.grid(True)
plt.axis("equal")
plt.show()

plt.contourf(xx, yy, pred > 0.5, alpha=0.3)
plt.scatter(X[0], X[1], c=T.ravel(), edgecolors="k")
plt.xlabel("x1")
plt.ylabel("x2")
plt.title("Região de classificação aprendida pela rede 2-4-1")
plt.grid(True)
plt.show()

plt.plot(historico_eqm)
plt.xlabel("Épocas")
plt.ylabel("Erro Quadrático Médio")
plt.title("Evolução do EQM durante o treinamento")
plt.grid(True)
plt.show()
```