

Introdução às redes neurais para Físicos

An introduction to Neural Networks for Physicists

Gubio G. de Lima^{*1}, Gustavo Café de Miranda¹, Tiago de S. Farias¹

¹Universidade Federal de São Carlos, Departamento de Física, São Carlos, SP, Brasil.

Recebido em 15 de maio de 2025. Revisado em 10 de outubro de 2025. Aceito em 27 de novembro de 2025.

As técnicas de aprendizado de máquina emergiram no contexto científico e se desenvolveram como ferramentas poderosas para enfrentar uma ampla gama de desafios na sociedade. A integração dessas técnicas com a física tem conduzido a abordagens inovadoras na compreensão, controle e simulação de fenômenos físicos. Este artigo visa proporcionar uma introdução prática às redes neurais e seus conceitos fundamentais, destacando perspectivas recentes dos avanços na interseção entre modelos de aprendizado de máquina e sistemas físicos. Além disso, apresentamos um material prático para orientar o leitor em seus primeiros passos na aplicação de redes neurais para resolver problemas físicos. Como exemplo ilustrativo, fornecemos quatro aplicações de complexidades crescentes para o problema de um pêndulo simples, a saber: *fit* de parâmetros da Equação Diferencial Ordinária (EDO) do pêndulo para aproximação de ângulo pequeno; *Physics Informed Neural Networks* (PINNs) para encontrar soluções da EDO do pêndulo em ângulo pequeno; *Autoencoders* em conjunto de dados de imagens do pêndulo para estimação de dimensionalidade do espaço de parâmetros do problema físico; uso de arquiteturas *Sparse Identification of Non-Linear Dynamics* (SINDy) para descoberta de modelos e expressões analíticas para o problema do pêndulo não linear (ângulos grandes).

Palavras-chave: Física clássica, Redes neurais, Tutorial.

Machine learning techniques have emerged in the scientific context and have developed into powerful tools for addressing a wide range of challenges in society. The integration of machine learning methods with physics has led to innovative approaches in understanding, controlling, and simulating physical phenomena. This article aims to provide a practical introduction to neural network and their basic concepts. It presents some perspectives on recent advances at the intersection of machine learning models with physical systems. We introduce practical material to guide the reader in taking their first steps in applying neural networks to Physics problems. As an illustrative example, we provide four applications of increasing complexity for the problem of a simple pendulum, namely: parameter fitting of the pendulum's ODE for the small-angle approximation; application of Physics-Inspired Neural Networks (PINNs) to find solutions of the pendulum's ODE in the small-angle regime; *Autoencoders* applied to an image dataset of the pendulum's oscillations for estimating the dimensionality of the parameter space in this physical system; and the use of Sparse Identification of Non-Linear Dynamics (SINDy) architectures for model discovery and analytical expressions for the nonlinear pendulum problem (large angles).

Keywords: Classical Physics, Neural network, Tutorial.

1. Introdução

O aprendizado de máquina, em inglês *Machine Learning* (ML), tem ganhado muita atenção nos últimos anos devido ao seu sucesso em tarefas comerciais, industriais e especialmente no setor de serviços [1–5]. Hoje, os algoritmos baseados em ML, comumente associados ao termo Inteligência Artificial (IA), estão profundamente integrados às tecnologias digitais. Essas técnicas são notoriamente bem-sucedidas no tratamento de grandes volumes de dados, permitindo a solução de problemas complexos mediante métodos estatísticos [6]. Esse sucesso tem atraído a atenção de pesquisadores, destacando-se como uma poderosa ferramenta estatística e uma potencial aliada na exploração científica.

Embora promissor, o uso de ML pode fornecer modelos que são, por vezes, opacos, frequentemente tratados como “caixas-pretas”, onde não é possível determinar as relações causais que conduziram o algoritmo aos seus resultados. Por esta razão, estes algoritmos podem ser recebidos com ceticismo por membros da comunidade científica, que têm preferência por modelos mais interpretáveis e com fundamentos teóricos claros [6]. Contudo, tal ceticismo não foi impeditivo para a investigação de novas aplicações e algoritmos em diferentes setores no meio científico, especialmente em áreas inclinadas às aplicações, posicionando o ML como uma ferramenta promissora para o avanço da ciência. Nos parágrafos subsequentes, apresentaremos alguns dos casos em que esses esforços demonstraram êxito no meio científico, com enfoque especial na área de física.

Embora a área de ML tenha ganhado destaque significativo nos últimos anos, como exemplo o Prêmio

*Endereço de correspondência: gubiofisika@gmail.com

Editor-Chefe: Marcello Ferreira

Nobel de Física em 2024, muitos dos modelos utilizados atualmente vêm sendo desenvolvidos e aprimorados há várias décadas [7]. A partir da segunda metade do século XX, com os avanços na capacidade computacional, equipados com técnicas estatísticas e métodos matemáticos, estabeleceram-se os fundamentos do campo do ML. Nesse contexto, diferentes abordagens começaram a emergir, incluindo a investigação do comportamento dos neurônios biológicos, que inspirou a criação de algoritmos de redes neurais artificiais (em inglês, *Artificial Neural Networks*) com o objetivo de replicar, de forma simplificada, os processos de aprendizado observados em organismos vivos [8].

As redes neurais artificiais dependem de abordagens de treinamento que orientam como os modelos interagem com os dados para resolver problemas específicos. Esses diferentes paradigmas de treinamento, como o aprendizado supervisionado, o aprendizado reforçado e o aprendizado não supervisionado, definem maneiras distintas de explorar a relação entre os dados e o modelo, permitindo sua aplicação em uma ampla variedade de contextos.

O paradigma de ML supervisionado envolve o treinamento de modelos que utilizam conjuntos de dados rotulados, nos quais cada entrada está associada a um rótulo ou valores desejados (saída). Esse processo permite que os algoritmos tentem aproximar um mapa que represente a relação entre os dados de entrada e saídas [4]. Algoritmos comuns de aprendizado supervisionado em ML são¹: K-vizinhos mais próximos [10, 11], Regressão Logística [12, 13], Máquinas de Vetores de Suporte [14, 15], Árvores de Decisão [16, 17], e redes neurais artificiais [18].

Por outro lado, o aprendizado não supervisionado é caracterizado pelo uso de algoritmos que lidam com dados não rotulados, i.e., há uma estrutura subjacente nos dados que não é explicitamente conhecida, mas cujos padrões deseja-se identificar. Nesse contexto, os modelos devem identificar relações e características exclusivamente a partir dos dados de entrada, utilizando funções apropriadas para otimizar um objetivo que visa identificar as relações inerentes entre os dados fornecidos. Otimizar estas funções leva a uma saída que deve ser consistente com restrições (às vezes adicionais) definidas pelo treinamento. Historicamente, o aprendizado não supervisionado se tornou particularmente atrativo com a publicação de artigos de redes neurais inspiradas em modelos biofísicos do córtex visual em felinos [8]. Alguns algoritmos amplamente discutidos na literatura para esse tipo de aprendizado incluem: K-Médias [19], Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [19], Principal Component Analysis (PCA) [19], *t-Distributed Stochastic Neighbor Embedding* (t-SNE) [18], e *autoencoders* [20–22].

No aprendizado por reforço, o modelo, frequentemente representado como um agente, aprende a tomar decisões através da interação contínua com um ambiente dinâmico, que pode ser total ou parcialmente observável, baseado em um processo de tentativa e erro. Nesse contexto, o agente executa uma série de ações em resposta a estados observados do ambiente e, para cada ação tomada, ele recebe uma recompensa ou penalidade, dependendo de quão benéfica ou prejudicial a ação foi para atingir um objetivo definido. O objetivo central do agente é aprender uma estratégia que define a melhor ação a ser tomada em cada estado possível do ambiente, visando maximizar a recompensa acumulada ao longo do tempo. O aprendizado por reforço demonstra eficácia em uma ampla variedade de aplicações complexas, como jogos, robótica, sistemas autônomos de controle e recomendação de produtos, e outros sistemas cuja solução pode ser escrita em termos de agentes e estratégias. Exemplos notáveis incluem o treinamento de agentes que superam humanos em jogos de tabuleiro e videogames, como o AlphaGo [23], Atari [24] e o desenvolvimento de sistemas de controle em robôs que aprendem a se movimentar e a manipular objetos em ambientes reais [25].

Historicamente, no campo da física, a pesquisa em física de partículas foi pioneira na adoção de técnicas de ML, em parte devido à necessidade de lidar com grandes volumes de dados em tempo real. As primeiras aplicações de ML nesse campo focaram em sistemas com grandes bancos de dados, que precisavam ser processados rapidamente. Na física de altas energias, por exemplo, o uso de árvores de decisão em experimentos com colisores de partículas ganhou destaque. Nesses experimentos, a rápida ativação dos detectores exige métodos computacionais eficientes para a classificação e armazenamento de dados enquanto os eventos ocorrem [26]. Posteriormente, para a mesma tarefa, o uso de redes neurais mostrou-se superior, superando os métodos tradicionais de análise de dados [27].

Outro exemplo de aplicação de ML na física de partículas envolve a investigação de modelos que dependem de expansões perturbativas da Teoria Quântica de Campos, modelos de interação partícula-detector e modelos fenomenológicos, todos focados em descrever interações em diferentes escalas de comprimento. Esses modelos geralmente apresentam parâmetros livres que precisam ser ajustados com base em dados experimentais. O uso de redes neurais para ajustar esses parâmetros, conectando modelos teóricos a resultados experimentais, tornou-se uma aplicação comum tanto em física de partículas quanto em cosmologia [6, 28].

Como mostrado pelos exemplos anteriores, as redes neurais têm demonstrado sua eficácia em tarefas computacionalmente complexas, com a etapa de otimização representando uma parte significativa do esforço computacional. No entanto, uma vez otimizadas, essas redes podem ser reutilizadas em seu domínio de aplicação com

¹ Em inglês: *K-Nearest Neighbors*, *Logistic Regression*, *Support Vector Machines* (SVM), *Decision Trees*, and *Artificial Neural Networks*.

novos dados, um processo que geralmente exige menor intensidade computacional.

Estes resultados têm se mostrado promissores na literatura em física de muitos corpos, tanto clássicos como quânticos, onde soluções para caracterização de fases em modelos de spins (i.e. *Ising* 2D) são uma questão central [29, 30]. Em matéria condensada e estudo de materiais, ML é usada, aliado a outros métodos, para cálculo de energias em *Density Functional Theory* (DFT), sendo sensivelmente mais rápida que métodos tradicionais [31].

Uma segunda abordagem, de especial interesse para a comunidade científica, por fornecer um meio-termo entre clareza e opacidade de modelos, foi pensada para agregar informação física nos modelos, integrando leis ou restrições físicas em modelos de ML, apresentando novas oportunidades para as pesquisas científicas.

Um dos principais métodos nessa abordagem é conhecido como *Physics-Informed Machine Learning* (PIML) [32–34] e *Physics-Informed Neural Network* (PINN) [35, 36], cujo uso pode ser encontrado em publicações de diversas áreas de pesquisa na Física, como: mecânica dos fluidos [37, 38]; quantificação da incerteza [39]; sistemas dinâmicos [40]; sistema quântico de muitos corpos [41], sistemas fotônicos [42]; óptica clássica [43], entre outros [44–47]. Existem outros trabalhos com paradigmas de modelagem híbrida que integram ML com conhecimento físico [48–56]. Além de adicionar informações físicas ao modelo, há na literatura aplicações que invertem o problema de pesquisa, ou seja, utilizam-se de ML para identificar características físicas. Exemplos incluem determinar equações que descrevem o sistema por meio de regressão simbólica [54] ou o uso de autodiferenciação em bibliotecas avançadas de ML [57].

A literatura científica na interseção entre física e ML tende a ser segmentada em nichos que exigem conhecimentos avançados, muitas vezes ao nível de pós-graduação. Pensando em tornar esse conhecimento mais acessível, aqui apresentamos materiais voltados para um público mais amplo, como graduandos em física que já completaram o ciclo básico, bem como estudantes de engenharia. Para isso, utilizaremos problemas de mecânica clássica combinados com técnicas de redes neurais, de modo a facilitar a compreensão e o aprendizado desses temas complexos.

Neste estudo, discutimos os conceitos fundamentais das redes neurais, começando com o Perceptron no Capítulo 2.1, onde apresentamos um passo-a-passo de como construir e aplicá-lo. Em seguida, abordamos as redes neurais profundas no Capítulo 2.2, ampliando alguns conceitos, introduzindo novos e apresentando as arquiteturas mais comuns. No Capítulo 3, apresentaremos quatro aplicações distintas de redes neurais no modelo unidimensional do pêndulo simples. Na primeira abordagem (3.1), foi realizado o aprendizado supervisionado com uma rede neural para determinar qual a constante do sistema físico (constante gravitacional). Em seguida, exploramos o uso de redes neurais profundas e

autodiferenciação para resolver a equação diferencial associada ao pêndulo (3.2). Posteriormente, investigamos a utilização de *autoencoders* (3.3) para inferir o espaço latente do sistema e estimar sua dimensão, além de introduzir o uso destes modelos para filtragem de ruídos em dados e imagens. Por fim, na seção (3.4) usaremos *autoencoders* SINDy para descobrir a equação diferencial (e não sua solução) do sistema do pêndulo não linear (para ângulos grandes).

2. Aspectos Básicos de Redes Neurais

Ao longo das últimas décadas, as redes neurais artificiais foram investigadas com o intuito de modelar o comportamento dos neurônios biológicos. A associação entre os fenômenos biológicos e a inteligência motivou a pesquisa além do campo da neurofisiologia, originando, em 1943, o modelo proposto por McCulloch e Pitts, considerado o primeiro estudo a descrever formalmente uma rede neural artificial [8]. Embora a terminologia específica tenha sido cunhada apenas mais tarde, esse trabalho estabeleceu as bases conceituais para os modelos computacionais de aprendizagem. Em 1958, Rosenblatt propôs o perceptron, um modelo teórico inspirado em princípios do funcionamento do cérebro, que buscava explicar como organismos poderiam aprender, reconhecer padrões e generalizar informações de forma probabilística [9]. Nas décadas seguintes, novas arquiteturas e métodos de aprendizagem foram desenvolvidos, consolidando as redes neurais como um dos pilares da inteligência artificial moderna [8].

A década de 2010 foi particularmente prolífica para o avanço do aprendizado de máquina e das redes neurais. Um dos fatores decisivos foi a popularização das Unidades de Processamento Gráfico (em inglês, *Graphics Processing Units*, GPUs), originalmente projetadas para executar milhares de operações matemáticas em paralelo. Diferentemente das CPUs, que possuem um número limitado de núcleos otimizados para tarefas sequenciais, as GPUs contam com milhares de núcleos mais simples, especializados no processamento simultâneo de grandes volumes de dados. Essa capacidade de paralelismo massivo é ideal para o treinamento de redes neurais, que envolve álgebra linear em larga escala, reduzindo drasticamente o tempo necessário para ajustar os parâmetros dos modelos. Como resultado, as redes neurais começaram a superar os algoritmos determinísticos tradicionais em diversas áreas de grande prestígio.

No campo do processamento de imagens, por exemplo, redes neurais convolucionais superaram os métodos clássicos no desafio ImageNet, que antes eram dominados por técnicas como SIFT (Scale-Invariant Feature Transform) combinadas com Máquinas de Vetores de Suporte (SVM) [8, 59]. No xadrez, a abordagem baseada em busca heurística e poda alfa-beta, como a do motor Stockfish *Stockfish*, foi superada pela AlphaZero, uma

rede neural que aprendeu o jogo do zero [58]. O sucesso se estendeu a outros domínios com ampla cobertura midiática, como a vitória do AlphaGo no jogo de Go² e o surgimento de modelos de linguagem avançados como o ChatGPT [60].

2.1. Perceptron

Nos algoritmos de redes neurais artificiais, o neurônio é definido como a unidade fundamental responsável pelo processamento de informação. Inspirados pelos neurônios biológicos, os neurônios artificiais recebem um conjunto de entradas, processam essas informações aplicando um conjunto de operações matemáticas que produzem uma saída. Um dos modelos mais simples de neurônios artificiais é o Perceptron [9, 20]. As redes neurais são tipicamente representadas como grafos direcionados, onde os neurônios correspondem aos vértices e as conexões entre eles, denominadas sinapses no contexto biológico e pesos no contexto algorítmico, são representadas por arestas. O Perceptron pode ser representado conforme ilustrado na Figura 1.

De maneira simplificada, no modelo biológico, interpretamos as entradas como mudanças na diferença de potencial eletrostático na vizinhança x_i do neurônio. Já os pesos w_i representam a intensidade ou sensibilidade da conexão com cada parte $i \in \{1, \dots, n\}$ da vizinhança. O *bias* b pode ser entendido como um campo médio que influencia o campo efetivo no neurônio ou como um limiar de ativação que o neurônio deve superar para ser ativado. Esses elementos atuam em conjunto sendo somados na operação Σ , cuja saída serve como argumento para a função de ativação f . Esta função f determina se o neurônio é ativado (isto é, “dispara” um sinal para outros neurônios) ou permanece inativo. As analogias com a biologia, no entanto, apresentam limitações, pois

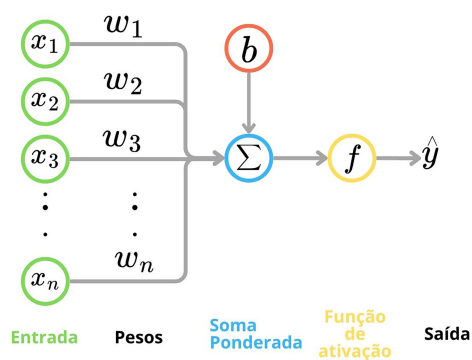


Figura 1: Ilustração das etapas de um Perceptron: x_{ij} representa os elementos do vetor de entrada $\mathbf{x}_j$ (em verde) associado à amostra j , estes elementos são multiplicados um a um por pesos w_i e somados a um *bias* b . O resultado desta soma (em azul) serve de argumento para uma função de ativação f (em amarelo), produzindo a saída $\hat{y}$.

o modelo matemático não consegue reproduzir integralmente a complexidade dos sistemas biológicos. Por exemplo, o sinal neuronal possui uma dependência temporal intrínseca, algo que não é considerado no perceptron clássico, cuja dinâmica é essencialmente estática. Para incorporar essa característica temporal, foram propostas arquiteturas mais sofisticadas, como as redes neurais recorrentes, que introduzem memória ao sistema ao levar em conta o estado anterior da rede [61]. Neste trabalho essa analogia tem um caráter meramente didático para os propósitos deste artigo e, portanto, não utilizaremos esses termos biológicos nas seções subsequentes. No contexto de redes neurais artificiais, a saída $\hat{y}$ do Perceptron indica a resposta do neurônio a um dado conjunto de entradas, refletindo se o estímulo foi suficiente para ativar o neurônio.

A linguagem *Python* se estabeleceu como o padrão para aprendizado de máquina devido à sua sintaxe simples e legível, que permite ao programador focar nos conceitos algorítmicos em vez de se prender a complexidades sintáticas. Mais importante ainda, *Python* é sustentado por um ecossistema de bibliotecas robusto e maduro, como *NumPy*, para computação numérica, e frameworks especializados como *Scikit-learn*, *TensorFlow* e *PyTorch*, que simplificam drasticamente o desenvolvimento e o treinamento de modelos de ML. Por essa razão, para facilitar a compreensão dos leitores, detalharemos a seguir cada etapa do funcionamento do Perceptron, abordando tanto os aspectos matemáticos envolvidos quanto sua implementação prática nesta linguagem de programação.

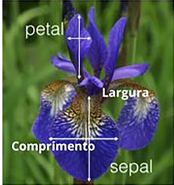
2.1.1. Exemplo didático: Classificação binária

Para compreender os conceitos representados na Figura 1, partiremos de um exemplo de aprendizado supervisionado, que será detalhado nos parágrafos a seguir. Seguindo a ordem de execução de um Perceptron, da esquerda para a direita. São eles: as entradas (dados), os pesos, a soma ponderada com a função de ativação e, por fim, a saída da rede neural.

Entradas (*inputs*): A primeira etapa envolve os dados de entrada, representados pelos círculos verdes à esquerda na Figura 1. Cada elemento x_i representa uma variável independente. Estas entradas são valores numéricos que caracterizam as variáveis do problema em análise. Por exemplo, podemos ter uma tabela ou planilha com uma lista de produtos e suas características, ou um banco de dados com informações sobre animais, plantas ou pessoas, etc. As entradas também podem ser informações de imagens [63], sons [64] ou textos [65], ou até mesmo os valores de saída de outros neurônios.

Para exemplificar, selecionamos um conjunto de dados resultante da catalogação de flores, que contém características do comprimento e largura das pétalas e sépalas. Este banco de dados é conhecido como *Iris dataset* [66], comumente utilizado para estudar problemas de classificação. Na Figura 2, temos as primeiras 5 linhas do banco

² deepmind.google/technologies/alphago/



	Coluna 1	Coluna 2	Coluna 3	Coluna 4	
	Comprimento da sépala	Largura da sépala	Comprimento da pétala	Largura da pétala	Rótulo
	5.1	3.5	1.4	0.2	0
	4.9	3.0	1.4	0.2	0
	4.7	3.2	1.3	0.2	0
	7.0	3.2	4.7	1.4	1
	6.4	3.2	4.5	1.5	1

Figura 2: Banco de dados de Iris (*Iris dataset*), onde estão explícitas as primeiras 5 linhas do *data-set*, com informações sobre flores, como comprimento e largura da sépala ou da pétala.

de dados, onde cada linha representa uma flor catalogada e cada coluna uma característica. Cada linha é um vetor $\mathbf{x}_j$ (com j representando a linha específica do conjunto de dados), dado por³:

$$\mathbf{x}_j = [x_{j1}, x_{j2}, x_{j3}, \dots, x_{jn}], \quad (1)$$

cada elemento x_{ji} do vetor, varia de $i \in \{1, \dots, N\}$ onde N é o número de colunas do banco de dados e $j \in \{0, \dots, M\}$ onde M é o número de dados (ou amostragem). A última coluna, com $i = N + 1$, é reservada aos rótulos, que neste caso representam a classificação das espécies e a coluna para $i = 0$ tem valor específico, que será mencionado no próximo passo.

Para o caso do exemplo ilustrado na Figura 2, temos 4 características (colunas) para a entrada e uma característica para o rótulo. Então, a representação matemática de uma linha específica ($j = 0$) da Figura 2 pode ser expressa por:

$$\begin{aligned} \mathbf{x}_0 &= [x_{01}, x_{02}, x_{03}, x_{04}], \\ \mathbf{x}_0 &= [5.1, 3.5, 1.4, 0.2]. \end{aligned} \quad (2)$$

Para acessar o conjunto de dados do *Iris dataset*, podemos utilizar a biblioteca *scikit-learn* [67], que já inclui esses dados de maneira acessível para uso em experimentos de ML. Para visualizar um dos elementos do conjunto de dados (por exemplo, o elemento $j = 0$), podemos utilizar o seguinte *script* em *Python*:

```
1 # Importando as biblioteca
2 from sklearn import datasets
3
4 dataset = datasets.load_iris()
5
6 j = 0 # Índice dos dados
7 x_j = dataset.data[j, :] # Entrada
8 y_j = dataset.target[j] # Alvo
9 print(x_j, y_j)
```

Saída:

```
1 [5.1, 3.5, 1.4, 0.2], [0]
```

Após o acesso ao banco de dados, o passo subsequente consiste na inicialização dos parâmetros da rede neural. Esses parâmetros armazenam, de forma implícita,

³ Adotamos o padrão americano de pontuação decimal (ponto como separador decimal), nos códigos e equações.

o conhecimento adquirido pelo algoritmo durante o processo de treinamento, uma vez que são utilizados para transformar as informações processadas por cada neurônio. Os ajustes realizados nesses parâmetros definem o mapeamento entre a entrada e a saída do modelo, permitindo que o Perceptron seja representado como uma função matemática da forma $g : (x_{j1}, \dots, x_{jN}) \rightarrow y_j$.

Pesos e *bias*: Os principais parâmetros de uma rede neural são os pesos e *bias*. Os pesos (w_i) atribuem diferentes importâncias às entradas de cada neurônio, desempenhando um papel central na definição da relevância relativa de cada entrada para a saída do modelo. O *bias* (b), por sua vez, é um parâmetro adicional que ajusta a saída do modelo independentemente das entradas, sendo essencial para garantir a flexibilidade e a adaptabilidade do algoritmo aos dados. No caso do Perceptron, o *bias* permite deslocar a linha ou o hiperplano de decisão em espaços de maior dimensão, contribuindo para a correta classificação das entradas.

A melhor forma de inicializar os parâmetros é um campo ativo de pesquisa de ML [6], pois dependerá muitas vezes da função de ativação utilizada, a função *loss*, e de características do problema tratado. Na falta de mais informação sobre o sistema, é possível gerar valores aleatórios para os pesos com distribuição normal, mas há outras opções como: inicializar todos com valores zeros, aleatoriamente com distribuição uniforme e Glorot/Xavier (para funções de ativação simétricas) [68]. Veremos adiante que o que se chama de *treinamento da rede* é uma atualização dos valores dos pesos e *bias*.

O vetor de pesos $\mathbf{w}$ e o vetor de entradas $\mathbf{x}$ podem ser escritos da seguinte forma

$$\mathbf{w} = [w_0, w_1, \dots, w_N], \quad \mathbf{x}_j = [x_{j0}, x_{j1}, \dots, x_{jN}].$$

Em algumas representações, é utilizado $w_0 = b$ e os dados de entrada são escritos de tal forma que $x_{j0} = 1$. Por simplicidade, adotaremos esta notação. Assim, podemos descrever a etapa da soma ponderada (círculo azul) ilustrada na Figura 1 utilizando a operação do produto escalar,

$$\mathbf{w}^T \mathbf{x}_j = \mathbf{x}_j^T \mathbf{w} = [x_{j0}, x_{j1}, \dots, x_{jN}] \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_N \end{bmatrix},$$

o super índice T representa o transposto, portanto, a multiplicação matricial resultante é

$$\mathbf{x}_j^T \mathbf{w} = x_{j0}w_0 + x_{j1}w_1 + \dots + x_{jN}w_N = \sum_{i=0}^N w_i x_{ji}.$$

Este produto representa uma soma ponderada das entradas de um dado j . Como exemplo para ilustrar a dinâmica, vamos considerar $\mathbf{w} = [1, 0, 0.5, -0.3]$, $b = [0.5]$ e os valores de $\mathbf{x}_0$ na equação (2), que resulta em $\mathbf{w}^T \mathbf{x}_0 = 6.24$.

Este mesmo produto pode ser numericamente calculado em *Python*⁴:

```
1 import numpy as np
2
3 w = [1, 0, 0.5, -0.3]
4 b = [0.5]
5 z = np.dot(w, x_j) + b
6 print(z)
```

Saída:

```
1 [6.24]
```

Função de ativação: Após a soma ponderada ser realizada, uma função não linear f , chamada de função de ativação, é aplicada ao resultado. As funções de ativação introduzem propriedades não lineares na rede neural, permitindo construir redes mais profundas e expressivas. Algumas comumente utilizadas incluem (veja a Tabela 1 em apêndice A para mais detalhes):

- Sigmoid ou Logística: É uma função suave, usada principalmente na camada de saída de um modelo de classificação binária, pois sua saída varia no intervalo $[0, 1]$.
- Tangente Hiperbólica (tanh): é similar à função sigmoide quanto à forma da curva, porém, varia no intervalo de $[-1, 1]$, com o centro em zero.
- Unidade Linear Retificada (ReLU): Atualmente, ReLU é uma das funções de ativação mais amplamente utilizada em redes neurais profundas devido à sua simplicidade computacional e à facilidade de cálculo de seu gradiente, sem que efeitos de não linearidade sejam perdidos. Ela reproduz a entrada se for positiva; caso contrário, produzirá zero.
- Outros tipos: LeakyRelu, Softmax, seno, cosseno entre outras [69, 70].

O poder de uma rede neural em modelar fenômenos complexos reside na combinação da não linearidade de sua função de ativação com a conexão de vários neurônios (como veremos no próximo capítulo) [71]. Do ponto de vista da álgebra linear, uma sequência de transformações lineares pode ser sempre reduzida a uma única transformação linear. Sem uma função de ativação não linear para quebrar essa sequência, uma rede neural com dezenas de camadas se comportaria, na prática, como uma única camada, tornando-se incapaz de aprender relações mais complexas que uma simples regressão linear múltipla.

A introdução dessa não linearidade permite que arquiteturas profundas funcionem como aproximadores universais. Como veremos em detalhe na Seção 2.2, uma

rede com uma composição finita de neurônios com ativações não lineares pode, em teoria, aproximar qualquer função contínua, conforme o Teorema da Aproximação Universal 2.1. Embora o papel da não linearidade não seja tão evidente no Perceptron simples, cujas tarefas são, por definição, linearmente separáveis, ela é a propriedade fundamental que possibilita o aprendizado em redes mais sofisticadas. Além disso, para que o modelo possa ser treinado por métodos de otimização baseados em gradientes, a função de ativação deve ser diferenciável, isto é, ter derivada definida e finita nos pontos de interesse.

Na saída do Perceptron, obtemos a seguinte expressão:

$$\hat{y}_j = f\left(\sum_{i=1}^n w_i x_{ji} + b\right), \quad (3)$$

que pode ser utilizada para representar certas informações. Um dos sistemas mais simples é o classificador binário, no qual o rótulo verdadeiro $y \in \{0, 1\}$ indica a classe correta, e a saída prevista $\hat{y}_j$ deve refletir uma decisão entre duas classes possíveis. No segundo exemplo, consideramos que o rótulo verdadeiro y assume valores contínuos, caracterizando um problema de regressão.

Em nosso exemplo, do banco de dados de flores, podemos classificar qual o tipo de flor a partir das características das flores. Neste caso temos três classes $\{0, 1, 2\}$ dadas pelas flores *setosa*, *versicolor*, *virginica*. Todavia, reduziremos o banco de dados para duas classes, 0 para as *setosa* e 1 para as *virginica*, para facilitar o entendimento. Escrevendo a equação (3) em *Python*:

```
1 # considerando o valor de z, obtido
   anteriormente
2 # z = [6.24]
3 def ativacao_sigmoide(z):
4     f = 1 / (1 + np.exp(-z))
5     return f
6
7
8 y_hat = ativacao_sigmoide(z)
9 print(y_hat)
```

Saída:

```
1 [0.998]
```

A escolha da função de ativação sigmoide é motivada por sua característica de limitar a saída do Perceptron ao intervalo entre 0 e 1. Esta propriedade é particularmente útil em problemas de classificação binária, pois a saída pode ser interpretada como uma probabilidade de pertencer a uma determinada classe. Por exemplo, uma saída próxima de 1 indica alta probabilidade de que a entrada pertence à classe positiva (1), enquanto uma saída próxima de 0 indica alta probabilidade de que a entrada pertence à classe negativa (0). Esta interpretação probabilística simplifica a tomada de decisões, tornando o modelo mais intuitivo e permitindo a utilização de técnicas de análise estatística. No exemplo descrito pelo código, vemos que $\hat{y}_0 = y_hat = 0.998$ indicando que

⁴ Para isto usaremos da biblioteca *Numerical Python (NumPy)* [62], que integra uma vasta gama de funções matemáticas. Estas adições possibilitam manipulações de grandes base de dado com alta performance computacional, enquanto preserva a interface concisa e legível da linguagem *Python*. Vale esclarecer aqui que podemos importar a biblioteca com uma abreviação qualquer (neste caso, por convenção de uso, “np”).

para os valores de pesos inicializados, aquele dado deve pertencer à classe (1), mas se compararmos esse resultado com o valor real da saída do neurônio, notaremos que a classe desse dado deveria ser (0). Isso evidencia um erro entre o valor predito pelo Perceptron, ainda não treinado, com o valor real. Como medir esses erros e como ajustar os pesos e *bias* para melhorar a previsão da saída, são tópicos das próximas subseções.

Função de custo (em inglês, *loss function*): também chamada de função de perda ou função de erro, é utilizada para medir o desempenho de um modelo em relação aos dados reais, quantificando o erro entre as saídas do Perceptron e os valores esperados. Essa função fornece uma métrica a ser minimizada durante o treinamento, com o objetivo de aprimorar o desempenho do algoritmo. Veremos adiante que ela também pode ser manipulada para direcionar o aprendizado do Perceptron ou de redes neurais. No nosso exemplo, utilizaremos o erro quadrático médio (em inglês, *Mean Squared Error*, MSE) que calcula a média dos erros de $m \leq M$ amostras através da fórmula:

$$\mathcal{L}(y_j, \hat{y}_j) = \frac{1}{m} \sum_j^m (y_j - \hat{y}_j)^2, \quad (4)$$

onde y_j é o rótulo verdadeiro da amostra j e $\hat{y}_j$ é a previsão feita pelo modelo. $\mathcal{L}$ representa a função custo entre y_j e $\hat{y}_j$, sobre m elementos do conjunto total de amostras de tamanho M , i.e., um conjunto de m linhas do *dataset*. No que segue iremos omitir o índice j na variável $\hat{y}_j$, exceto em casos ambíguos. Escrevendo a equação (4) em *Python* para $m = 1$:

```
1 # y_hat = [0.998], y_0 = [0]
2 # calculo do erro quadratico com j=0
3 loss = (y_j - y_hat) ** 2
4 print(loss)
```

Saída:

```
1 [0.99611167]
```

Observe que, para os valores iniciais que escolhemos, a nossa função custo está acusando um resultado maior que zero, i.e., longe do ideal. O objetivo para este dado era obter $y_j = 0$, mas o valor dado pelo Perceptron foi de $\hat{y} \approx 1$. Como a função quadrática MSE tem um mínimo global em $\mathcal{L}(y_j, \hat{y}) = 0$, queremos que todos (ou a maioria) das saídas (*outputs*) da rede, quando calculadas na função custo, sejam próximas deste mínimo global.

Aqui, é importante ressaltar que nossa escolha pelo MSE serve a um propósito didático. Por ser matematicamente simples, ele nos permite ilustrar o conceito de erro de forma clara. No entanto, para problemas de classificação como este, o MSE não é a ferramenta ideal. A abordagem mais correta envolve funções de custo projetadas especificamente para classificação, sendo a Entropia Cruzada (*Cross-Entropy*) uma das mais proeminentes. Esta função é mais eficaz porque penaliza previsões confiantes e erradas de forma mais acentuada, levando a um treinamento mais estável e rápido. Para

não complicar este primeiro exemplo, seguimos com o MSE, mas deixamos o próximo passo como um desafio ao leitor: no material complementar, disponibilizamos um exercício que guia na implementação da Entropia Cruzada.

Regra de aprendizado dos parâmetros: A otimização dos parâmetros de uma rede neural é frequentemente referida como “aprendizado” porque esse processo de ajuste permite que a rede neural tente resolver o problema para o qual foi projetada, de maneira análoga ao processo pelo qual uma pessoa aprende a resolver um problema após estudar e praticar. Existem várias técnicas para ajustar os parâmetros de redes neurais, como algoritmos genéticos [72] e *simulated annealing* [73], que buscam soluções ideais mediante processos inspirados na evolução natural e na termodinâmica, respectivamente. No entanto, os métodos baseados em gradiente destacam-se pela sua eficácia e sucesso comprovado na otimização de algoritmos de redes neurais, sendo amplamente utilizados na prática [74, 75].

Os métodos de gradiente utilizam informações derivadas da função custo, que mede o quão bem a rede está performando em relação ao problema. A função custo pode ser visualizada, de maneira análoga, como um mapa topográfico, onde a posição inicial corresponde a um ponto elevado, como o topo de uma montanha, e o objetivo é alcançar o ponto mais baixo, que representa o valor mínimo da função. Nesse contexto, o gradiente da função custo fornece a direção de maior inclinação, permitindo que o algoritmo “observe” o entorno e tome decisões sobre a direção de cada passo. Ao aplicar iterativamente um método de gradiente, os parâmetros da rede neural são ajustados gradualmente para reduzir o valor da função custo, movendo-se em direção ao mínimo.

Um dos métodos de otimização de gradiente mais simples e amplamente conhecidos é o método de gradiente descendente, ou *Gradient Descent* (GD) em inglês. Nesse método, calcula-se a derivada da função custo em relação a cada parâmetro que se deseja atualizar. O parâmetro é então ajustado subtraindo-se o produto da derivada com uma constante chamada taxa de aprendizado (*learning rate*). Essa constante controla o tamanho do passo que o algoritmo dará na direção do gradiente. O ajuste dos pesos e *bias*, através do método de gradiente descendente, é então dado pelas respectivas equações:

$$\mathbf{w}_{\text{novos}} = \mathbf{w}_{\text{antigos}} - \eta \nabla_{\mathbf{w}} \mathcal{L} \quad (5)$$

e

$$b_{\text{novos}} = b_{\text{antigos}} - \eta \nabla_b \mathcal{L}, \quad (6)$$

onde $\nabla_{\mathbf{w}, b}$ representa o gradiente da função de custo em relação aos pesos $\mathbf{w}$ ou ao *bias* b , respectivamente, enquanto η é a taxa de aprendizado. É importante destacar que essa taxa é definida manualmente antes do início do treinamento da rede neural e não é geralmente ajustada durante o processo de otimização. Por ser

um hiperparâmetro, η requer uma escolha cuidadosa, pois uma taxa de aprendizado muito alta pode fazer o algoritmo oscilar ao redor do mínimo desejado ou até divergir, enquanto uma taxa muito baixa pode resultar em um treinamento extremamente lento. O valor da taxa de aprendizado também pode ser encontrado por meio de técnicas de validação cruzada ou otimização de hiperparâmetros, como busca em grade (*grid search*) ou busca aleatória (*random search*) [76]. Escolher os valores corretos para hiperparâmetros é um passo crucial para garantir o sucesso no treinamento de redes neurais, influenciando diretamente a capacidade do modelo de convergir eficientemente e evitar problemas associados ao treinamento.

Além do método básico de gradiente descendente, existem variantes desenvolvidas para melhorar a eficiência e a qualidade do treinamento de redes neurais. Entre essas variantes estão o gradiente descendente com momento (*momentum*), o método de *Nesterov* (*Nesterov Accelerated Gradient*), o Adam (*Adaptive Moment Estimation*) [77, 78]. O gradiente com momento adiciona um termo de “memória” que ajuda a acelerar o movimento em direções que consistentemente apontam para o mínimo, enquanto o método de *Nesterov* faz uma correção adicional que antecipa a direção futura. Já o Adam combina as ideias de acumulação de momentos e adapta a taxa de aprendizado para cada parâmetro, resultando em uma convergência mais rápida e estável em muitos problemas práticos.

Os gradientes da função custo, na equação (4), em relação a $\mathbf{w}$ e b , aplicando a regra da cadeia de derivadas, são

$$\nabla_{\mathbf{w}} \mathcal{L} = -2(y_j - \hat{y})f' \mathbf{x}_j, \quad (7)$$

$$\nabla_b \mathcal{L} = -2(y_j - \hat{y})f'. \quad (8)$$

A derivação da expressão (8), como também alguns exemplos de função custo, está em maiores detalhes no Apêndice A. Nesse caso o termo f' é determinado pela função de ativação escolhida f . Podemos ver na tabela 1 no apêndice as derivadas para alguns casos. Em nosso exemplo utilizamos a função sigmoide, então,

$$\begin{aligned} \mathbf{w}_{\text{novos}} &= \mathbf{w}_{\text{antigos}} - 2\eta(y_j - \hat{y})\hat{y}(1 - \hat{y})\mathbf{x}_j, \\ b_{\text{novos}} &= b_{\text{antigos}} - 2\eta(y_j - \hat{y})\hat{y}(1 - \hat{y}). \end{aligned}$$

Implementando a atualização dos pesos e *bias* em *Python*:

```
1 eta = 0.01
2
3 w = w - 2 * eta * (y_j - y_hat) * y_hat * (1 -
4   y_hat) * x_j
5 b = b - 2 * eta * (y_j - y_hat) * y_hat * (1
6   - y_hat)
7 print(w, b)
```

Saída:

```
1 [ 1.0002, 0.000138, 0.50005, -0.29], [0.500039]
```

Após completar o processo de aprendizagem do Perceptron com o primeiro conjunto de dados da tabela de treinamento ($j = 0$), repetimos o procedimento para cada um dos dados restantes. Assim, cada linha j dos dados contribuirá para ajustar os pesos. Descrevendo as etapas para os $M = 150$ dados em um único *script*, obteremos:

```
1 w = [1, 0, 0.5, -0.3]
2 b = [0.5]
3 M = 150
4 eta = 0.01
5 loss = 0
6 for j in range(M):
7     x_j = dataset.data[j, :] # Entrada
8     y_j = dataset.target[j] # Alvo/rotulo
9     z = np.dot(w, x_j) + b
10    y_hat = ativacao_sigmoide(z)
11    w = w - 2 * eta * (y_j - y_hat) * y_hat *
12      (1 - y_hat) * x_j
13    b = b - 2 * eta * (y_j - y_hat) * y_hat *
14      (1 - y_hat)
15    loss = loss + (y_j - y_hat) ** 2 / M
```

Este ciclo deve ser iterado múltiplas vezes. Cada iteração em que exaurimos todos os dados é chamada de época, ou em inglês *epoch*, até que a função custo se aproxime do seu valor mínimo (neste caso, aproximadamente zero), indicando que o Perceptron aprendeu satisfatoriamente as informações fornecidas durante o treinamento. Para realizar várias iterações, adicionaremos mais um *loop* usando o comando “for” novamente. O código completo está disponível em uma página do *Github* [79], permitindo que o leitor reproduza o experimento numérico no seu devido tempo de aprendizado.

Na Figura 3, retirada do material disponibilizado no GitHub, apresentamos quatro gráficos bidimensionais. Os dois no topo com a inicialização arbitrária de pesos, e os dois na parte inferior depois da otimização dos parâmetros, ilustrando as fronteiras de decisão de um Perceptron aplicado ao conjunto de dados Íris. Em cada par de gráficos, o da esquerda mostra a fronteira de decisão em função do comprimento e largura da sépala, enquanto o da direita apresenta a mesma fronteira em função do comprimento e largura da pétala. Os pontos representam as amostras do conjunto de dados, onde as cores azul e vermelho indicam as diferentes classes a serem distinguidas pelo Perceptron. Para construir a fronteira de decisão do modelo de rede neural, foi configurado que todos os valores de saída da função de ativação maiores que 0.5 são classificados como pertencentes à classe (1). Consequentemente, valores de saída menores ou iguais a 0.5 são classificados como pertencentes à classe (0). Essa configuração define uma linha de corte que separa ambas as classes no espaço de características, permitindo que o modelo distinga entre elas com base nas saídas calculadas.

Nos gráficos 3a) e 3b), observamos uma considerável sobreposição das regiões coloridas, indicando uma dificuldade significativa em separar as classes, caracterizada

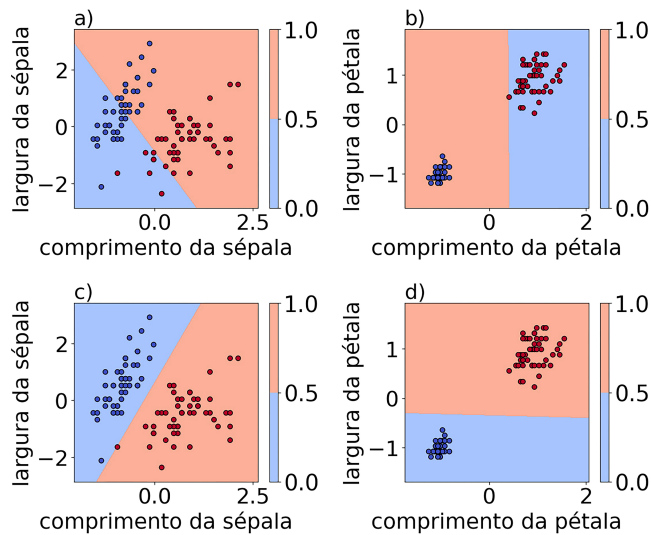


Figura 3: Evolução das Fronteiras de Decisão do Perceptron no banco de dados Íris. Os dados estão inseridos em um gráfico de dispersão, projetado em duas das quatro dimensões dos dados. A unidade de medida de ambos os eixos é dada em centímetro. Na esquerda, painéis a) e c), escolhemos as dimensões dadas por comprimento da sépala (abscissas) e largura da sépala (ordenadas), na direita, painéis b) e d), as outras duas dimensões são o comprimento da pétala (abscissas) e largura da pétala (ordenadas). Em ambos o rótulo de aprendizado está em codificado na cor: vermelho para classe 0 e azul para classe 1. Nos gráficos de cima, a) e b), temos a classificação dada pelo Perceptron com inicialização aleatória, já nos gráficos de baixo, c) e d), temos a classificação após a otimização. Veja que, após a otimização, o hiperplano (chamado de *fronteira de decisão*, onde o Perceptron acusa o valor 0.5) separa os conjuntos de dados em ambas projeções, exceto por um ponto.

por uma fronteira de decisão pouco definida. Isto reflete uma baixa acurácia do modelo antes de qualquer aprendizado ocorrer. Em contraste, os gráficos 3c) e 3d) mostram uma separação nítida entre as classes, com as regiões coloridas agora claramente divididas por uma fronteira de decisão bem ajustada após 100 épocas de treinamento.

2.1.2. Exemplo didático: regressão

Enquanto no exemplo anterior abordamos um problema de classificação, que envolve a categorização de dados em classes descritas por valores discretos, na regressão o objetivo é prever valores contínuos, como o preço de um produto, o peso ou a altura de uma pessoa. Em problemas de física, a regressão pode ser utilizada para prever grandezas físicas como posição, velocidade ou aceleração ao longo do tempo, sendo uma ferramenta essencial para estudar as relações entre quantidades físicas em experimentos. Neste exemplo, criaremos nossos próprios dados para ilustrar o processo de ajuste de um problema linear. Supomos que o conjunto de dados contenha apenas duas variáveis, uma que represente a

velocidade de um objeto sob aceleração constante e outra que represente os instantes de tempo:

$$\mathbf{t} = [0, 0.25, 0.5, 0.75, 1], \quad (9)$$

$$\mathbf{v} = [0, 0.125, 0.25, 0.375, 0.5]. \quad (10)$$

Nosso objetivo é utilizar o Perceptron como uma função que encontre a relação dos dados de entrada com as saídas, para depois prever a velocidade (saída) conhecendo o tempo (entrada). Observe que os problemas de regressão também são considerados aprendizado supervisionado, pois os rótulos de saída y_i neste caso, a velocidade, são fornecidos durante o treinamento. Com $\mathbf{t}$ sendo os dados de entrada e $\mathbf{v}$ os pontos experimentais que desejamos ajustar. Nesse problema, cada item de $\mathbf{v}$ é equivalente a y_i e $\mathbf{t}$ representa x_{ji} descrito na equação (1) com $N = 1$ e $M = 5$. Assim, tendo conhecimento dos dados de entrada e saída, o próximo passo é gerar o peso e *bias* aleatórios, no qual utilizaremos:

$$\mathbf{w} = [0.497], \mathbf{b} = [-0.138]. \quad (11)$$

A seguir, realizamos o processo de propagação em inglês *forward*, que consiste na propagação dos dados de entrada utilizando a equação (3) para cada valor de t . Antes de prosseguir, é importante esclarecer um conceito importante em redes neurais: o *batching*. Essa técnica consiste em dividir o conjunto de dados em pequenos subconjuntos chamados de *batches*, ou lotes, em português. Durante o processo de treinamento de uma rede neural, em vez de calcular a função custo para cada exemplo individualmente, o *batching* permite que o cálculo seja realizado para um grupo de exemplos de uma só vez. Em seguida, a otimização dos parâmetros do modelo é realizada com base na média ou soma dos gradientes computados para todos os exemplos do *batch*. Esta abordagem não apenas melhora a eficiência computacional do treinamento, mas também proporciona uma melhor generalização, ao reduzir o ruído estocástico associado a cada atualização de parâmetro. Assim, o *batching* é amplamente utilizado para acelerar a convergência do modelo e estabilizar o processo de otimização, especialmente em problemas com grandes conjuntos de dados. Descreveremos em detalhes alguns passos matemáticos que acontecem no Perceptron, visando auxiliar o leitor a entender todo o processo.

Iniciamos com o cálculo do produto interno, sobre o *batch* de 5 pontos em t :

$$\mathbf{w}\mathbf{t} + b = [-0.138, -0.01375, 0.1105, 0.23475, 0.359].$$

Em seguida, aplicamos a parte não linear com a função ReLU (ver apêndice A) sobre todo o *batch*:

$$f(\mathbf{w}\mathbf{t} + b) = [0, 0, 0.1105, 0.23475, 0.359].$$

O resultado de $f(\mathbf{w}\mathbf{t} + b)$ são os valores previstos para velocidade $\hat{y}$, que utilizaremos para calcular o erro em

relação a $\mathbf{v}$ com a função custo (4), que resulta em

$$\mathcal{L} = \sum_{\text{batch}} (\mathbf{v} - \hat{\mathbf{y}})^2 = 0.0746.$$

Para ajustar os novos pesos e *bias*, utilizamos a equação (8). A derivada da função ReLU segue a regra: $f'(x) = 1$ se $x > 0$, e $f'(x) = 0$ se $x \leq 0$, de modo que

$$\begin{aligned} f'(\mathbf{w}\mathbf{t} + b) &= f'([0, 0, 0.1105, 0.23475, 0.359]) \\ &= [0, 0, 1, 1, 1]. \end{aligned}$$

Assumindo que $\eta = 0.05$, temos:

$$2\eta(\mathbf{v} - \hat{\mathbf{y}})f'_j\mathbf{t} = [0, 0, 0.0069, 0.0105, 0.0141]. \quad (12)$$

Tomando a média do *batch*, obtemos

$$\eta \sum_{\text{batch}} \nabla_{\mathbf{w}} \mathcal{L} = 0.0063.$$

Utilizando os passos (10), (11), (12) na equação (8), podemos obter os novos pesos e *bias*:

$$\begin{aligned} \mathbf{w}_{\text{novos}} &= \mathbf{w}_{\text{antigos}} - 0.0063 = 0.4906 \\ b_{\text{novos}} &= b_{\text{antigos}} - 0.008 = -0.1464. \end{aligned}$$

Com isso, completamos a primeira época e esperamos obter uma função custo com valor menor que o anterior. Podemos verificar isso repetindo o processo anterior mais uma vez.

$$\begin{aligned} \hat{\mathbf{y}} &= f(\mathbf{w}\mathbf{t} + b) = [0, 0.036, 0.178, 0.32, 0.46], \\ \mathcal{L} &= \sum_{\text{batch}} (\mathbf{v} - \hat{\mathbf{y}})^2 = 0.0643. \end{aligned}$$

Comparando o valor da função custo dos novos pesos com o anterior, notamos que houve uma redução de 0.0103, isso indica que o Perceptron está começando a aprender as informações dos dados. O processo pode ser repetido várias vezes para reduzir progressivamente a função custo. Em nosso material complementar no *Github*, apresentamos a continuação deste exemplo, onde executamos o algoritmo por mais 98 épocas, totalizando 100 épocas. Os resultados deste teste são apresentados na Figura 4, onde alcançamos o valor de erro de 1.059×10^{-7} .

Na Figura 4 temos o resultado da convergência do modelo de regressão ao longo das épocas de treinamento, com a linha do modelo se ajustando progressivamente aos dados observados. Este comportamento evidencia a capacidade do Perceptron em aprender e ajustar-se a um conjunto de dados simples por meio de sucessivas atualizações de seus parâmetros.

2.2. Redes neurais profundas

Uma rede neural artificial é composta por um conjunto de neurônios artificiais interconectados, cuja organização define a estrutura do modelo. A maneira como esses

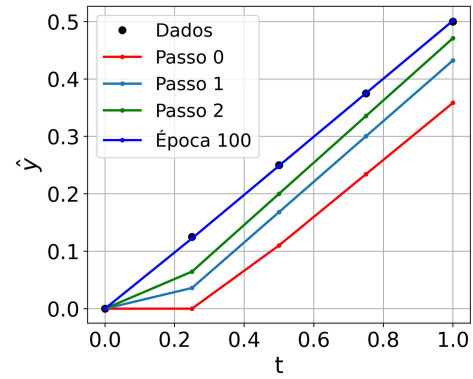


Figura 4: Comportamento do Perceptron para o problema de regressão ao longo de diferentes épocas de treinamento. No eixo horizontal temos os instantes de tempo (t) e no eixo vertical a predição da velocidade ($\hat{y}$) feita pelo Perceptron. Os pontos pretos representam os dados de treino, enquanto as linhas vermelha, verde e azul correspondem às estimativas do modelo nas épocas 0, 1, 2 e 100, respectivamente.

neurônios são interligados é conhecida como arquitetura da rede. Existem diversas formas de organizar essas conexões. Uma configuração comum envolve agrupar neurônios em unidades funcionais conhecidas como camadas. Cada camada é composta por neurônios que operam de forma independente, mas que processam informações antes de transmitir suas saídas para a camada seguinte.

Uma das arquiteturas mais conhecidas no ML é o *Perceptron Multicamadas* (MLP, do inglês *Multi-Layer Perceptron*), um tipo de rede neural *feed-forward* densa. Redes *feed-forward* distinguem-se pela ausência de conexões recorrentes, ou seja, as saídas de uma camada são utilizadas exclusivamente como entradas para a camada seguinte. Esse fluxo unidirecional de informações ocorre da camada de entrada até a camada de saída. Especificamente, os neurônios de uma camada n fornecem entradas para os neurônios da camada subsequente, $n + 1$, sem retornos ou ciclos no processo. Essa arquitetura é amplamente utilizada devido à sua simplicidade e eficácia em tarefas de classificação e regressão, onde uma relação direta entre entradas e saídas é desejada.

A representação gráfica da arquitetura MLP está ilustrada na Figura 5, lida da esquerda para a direita: as entradas são representadas pelas linhas originadas da ponta esquerda da imagem, conectadas aos nodos em verde, x_{ji} na notação do capítulo anterior. As entradas estão conectadas com a primeira camada (em vermelho), onde cada nodo é um Perceptron, cuja saída será usada de argumento na segunda camada de neurônios (em azul), os quais, por sua vez, se conectam à camada final de neurônios da direita (também em verde). Estes, finalmente, correspondem à saída da rede. Cada nodo pode ter a sua função de ativação específica, assim como as conexões neurônio a neurônio podem ser arbitrariamente determinadas, desde que seja sempre com as camadas

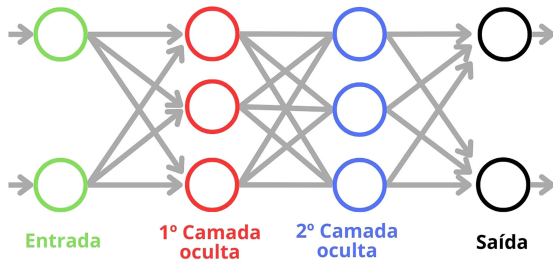


Figura 5: Representação gráfica de uma configuração específica de rede neural utilizando grafo, composta por 2 entradas, 2 camadas ocultas com 3 neurônios cada e 2 saídas. Os círculos verdes, representam as entradas x_{ji} e os pretos, as saídas $\hat{y}_i$. As camadas do meio, compostas pelos neurônios em vermelho e azul, são as camadas ocultas.

vizinhas, somente. As camadas intermediárias entre as de entrada e saída (neste exemplo: vermelha e azul) são chamadas de *camadas ocultas*.

Por simplicidade, utilizaremos o termo “redes neurais” ao nos referirmos especificamente às redes neurais artificiais com arquitetura *feed-forward*. No entanto, é importante que o leitor compreenda que o campo das redes neurais é vasto e inclui uma grande diversidade de arquiteturas, cada uma projetada com características específicas para atender a diferentes tipos de problemas. Entre essas outras arquiteturas, destacam-se as redes neurais recorrentes (RNNs), que são particularmente eficazes para lidar com dados sequenciais, como séries temporais [5, 80, 81] e processamento de linguagem natural, muitas vezes por meio do uso de LSTM (*Long Short-Term Memory*) [82]; as redes neurais convolucionais (CNNs), que exploraremos mais aprofundadamente nas seções que seguem, são amplamente utilizadas em tarefas de visão computacional, como reconhecimento de imagens e detecção de objetos, devido à sua capacidade de extrair características hierárquicas de dados espaciais [20]; e os *Transformers*, que revolucionaram o campo do processamento de linguagem natural e também têm sido aplicados com sucesso em outras áreas devido à sua habilidade de capturar dependências de longo alcance em dados sequenciais sem a necessidade de processamento recorrente [83]. Há, além dessas, outras arquiteturas como *Autoencoders*, *Deep Belief Networks*, Redes Adversárias Generativas (GANs), Máquinas de Boltzmann aliadas a métodos como Aprendizado Profundo Bayesiano, que têm contribuído significativamente para o avanço da área [84, 85]. Uma representação pictográfica das muitas possíveis arquiteturas pode ser encontrada em [5]. Recomendamos a leitura de Herberg et al. [86] para aqueles interessados em explorar essas arquiteturas detalhadamente.

Recentemente, John J. Hopfield e Geoffrey Hinton foram laureados com o Prêmio Nobel de Física de 2024, concedido por suas contribuições fundamentais ao ML com redes neurais artificiais. Hopfield desenvolveu redes capazes de armazenar e reconstruir padrões de dados

usando princípios da física conhecida como Redes de Hopfield [87], enquanto Hinton desenvolveu a máquina de Boltzmann, que reconhece padrões em dados utilizando ferramentas da física estatística. Essas inovações foram a base para avanços significativos em inteligência artificial e aprendizado profundo.

A escolha da arquitetura de uma rede neural deve ser cuidadosamente considerada e está intimamente ligada ao problema específico que se deseja resolver, às características dos dados envolvidos, e aos requisitos de desempenho e eficiência do modelo. De maneira geral, o objetivo de uma rede neural é aproximar uma função ideal $g : U \rightarrow V$, não necessariamente conhecida, onde U, V são espaços vetoriais. Assim, sendo $\mathbf{x} \in U$ um elemento do conjunto de dados e $\mathbf{W}, \mathbf{b} \in \mathbb{R}^d$ vetores dos parâmetros, então queremos que $NN : U \times \mathbb{R}^d \rightarrow V$ satisfaça

$$NN(\mathbf{x}; \mathbf{W}, \mathbf{b}) \approx g(\mathbf{x}). \quad (13)$$

Por exemplo, em regressão, podemos ter uma entrada $\mathbf{x}$ associada a um valor $g(\mathbf{x})$, onde a rede neural encontrará a função $g(\mathbf{x})$ através dos valores dos parâmetros $\mathbf{W}, \mathbf{b}$ resultando na melhor aproximação dessa função, $NN(\mathbf{x}; \mathbf{W}, \mathbf{b})$.

Vamos detalhar matematicamente o que descrevemos sobre a arquitetura *feed-forward* considerando os neurônios entre camadas totalmente conectados, pictorialmente representados na Figura 5. A rede transforma a entrada $\mathbf{x}$ em saída $NN(\mathbf{x}; \mathbf{W}, \mathbf{b})$ usando a seguinte estrutura (considerando a primeira camada como a vermelha na figura, e a entrada como os nós verdes).

Entrada = $\mathbf{x}$,

$$\begin{aligned} \mathbf{1}^{\text{a}} \text{ Camada} = & \left(f_1^{[1]} \left(\sum_{i=1}^{N_0} w_{i1}^{[1]} x_i + b_1^{[1]} \right), \dots, \right. \\ & \left. \dots, f_{N_1}^{[1]} \left(\sum_{i=1}^{N_0} w_{iN_1}^{[1]} x_i + b_{N_1}^{[1]} \right) \right), \end{aligned}$$

$$\begin{aligned} \mathbf{2}^{\text{o}} \text{ Camada} = & \left(f_1^{[2]} \left(\sum_{i'=1}^{N_1} w_{i'1}^{[2]} f_{i'}^{[1]}(\dots) + b_1^{[2]} \right), \dots, \right. \\ & \left. \dots, f_{N_2}^{[2]} \left(\sum_{i'=1}^{N_1} w_{i'N_2}^{[2]} f_{i'}^{[1]}(\dots) + b_{N_2}^{[2]} \right) \right), \end{aligned}$$

$\vdots$

$$\begin{aligned} \mathbf{Saída} = & \left(f_1^{[n]} \left(\sum_{i'=1}^{N_{(n-1)}} w_{i'1}^{[n]} f_{i'}^{[n-1]}(\dots) + b_1^{[n]} \right), \dots, \right. \\ & \left. \dots, f_{N_n}^{[n]} \left(\sum_{i'=1}^{N_{(n-1)}} w_{i'N_n}^{[n]} f_{i'}^{[n-1]}(\dots) + b_{N_n}^{[n]} \right) \right), \end{aligned}$$

Saída = $NN(\mathbf{x}; \boldsymbol{\theta})$,

onde $f_{k_n}^{[n]}$ e $b_{k_n}^{[n]}$ são a função de ativação e vetor de *bias* da n -ésima camada associado ao k -ésimo neurônio desta

camada n , respectivamente, e $w_{lk_n}^{[n]}$ é o peso que liga o l -ésimo neurônio da camada $n - 1$ ao k -ésimo neurônio da camada n .

O leitor pode estar se perguntando qual é a garantia de que esta longa composição iterada de somas pesadas de composições aproxima realmente a função desejada $g(\mathbf{x})$. A resposta a essa pergunta é afirmativa, dado o seguinte teorema [71, 88]:

Teorema 2.1 (Aproximação Universal: Kolmogorov–Arnold). Seja $g : \mathbb{R}^N \rightarrow \mathbb{R}$ uma função contínua, e sejam $\varphi_q, \phi_{q,p} : \mathbb{R} \rightarrow \mathbb{R}$, funções não lineares, com $q \in \{0, \dots, 2N\}$ e $p \in \{0, \dots, N\}$. Então g é aproximada por

$$f(\mathbf{x}) = \sum_{q=0}^{2N} \varphi_q \left(\sum_{p=0}^N \phi_{q,p}(x_p) \right) \approx g(\mathbf{x}), \quad (14)$$

onde $\mathbf{x} \in \mathbb{R}^N$.

Isto é, podemos representar qualquer função contínua com um número polinomial $\mathcal{O}(N^2)$ de funções não lineares de uma variável. O leitor deve ter percebido como a função do teorema é similar a uma rede neural com duas camadas e $b^{[n]} = 0, \forall n$. De fato, com somente duas camadas, qualquer função contínua pode ser aproximada, caso tenha neurônios suficientes. Pragmáticamente, é boa prática de ML trabalhar com um número maior de camadas menores, melhorando o desempenho computacional [20].

Visto o teorema anterior, poder-se-ia alegar que ML e Redes Neurais são *somente* métodos de aproximação de curvas e dados. Esta afirmação não está de todo errada, porém é uma generalização apressada. Pense na seguinte analogia: “Física de muitos corpos quânticos é somente a equação de *Schrödinger* em espaços de alta dimensão”, sabemos que, na prática, há muitos novos fenômenos nesta área, que requerem novas técnicas que estão ausentes em mecânica quântica de poucas partículas, o que justifica que a afirmação acima é muito reducionista. O mesmo pode ser dito de redes neurais [20].

Para aplicações práticas, construir redes neurais profundas somente com bibliotecas como *NumPy*, como fizemos para o Perceptron na Seção 2.1, é uma tarefa custosa. Devida à complexidade e tamanho das redes neurais, é de interesse que usemos bibliotecas prontas em Python, como *PyTorch* [89], *TensorFlow* [90] e *Jax* [91], com o intuito de facilitar o uso, já que muitas das operações típicas de treinos em redes neurais (as quais trabalhamos na parte anterior do texto), já vêm em funções pré-definidas nestas bibliotecas. Além disso, um recurso fundamental dessas bibliotecas é a diferenciação automática, essencial no treinamento de redes neurais. Trata-se de uma técnica eficiente para calcular gradientes com exatidão e de forma automatizada, eliminando a necessidade de derivação manual ou de aproximações numéricas, como as diferenças finitas [57]. A diferenciação

automática automatiza esses cálculos usando a regra da cadeia usual para derivadas, facilitando a aplicação de algoritmos como o gradiente descendente. Isso acelera o desenvolvimento, garante maior precisão, e preserva estruturas diferenciáveis, sendo uma das razões pelas quais essas ferramentas são amplamente adotadas no ML.

O leitor pode encontrar no link do nosso *Github* em [79] o material complementar a este artigo, onde há *jupyter notebooks* com os devidos passos a passos, explicando como criar uma rede neural usando a biblioteca *PyTorch*. Para introduzir a biblioteca, preparamos um *jupyter* específico aplicado ao mesmo problema de classificação de flores (*Iris dataset*) utilizado anteriormente.

Redes Neurais Convolucionais

As redes neurais convolucionais, do inglês *Convolutional Neural Networks* (CNNs), representadas na Figura 6, se diferenciam das redes totalmente conectadas por dois conceitos fundamentais: a localidade das conexões e o compartilhamento de parâmetros. A localidade refere-se ao fato de que um neurônio, ou um conjunto de neurônios, está conectado apenas a um subgrupo de neurônios da camada anterior, em vez de estar conectado a todos os neurônios dessa camada, como ocorre nas redes totalmente conectadas. Essa característica permite que as CNNs capturem padrões locais, como bordas e texturas em imagens, mantendo uma representação mais compacta dos dados.

O compartilhamento de parâmetros, por outro lado, significa que todos os neurônios de convolução dentro de uma mesma camada, denominados filtros, utilizam o mesmo conjunto de parâmetros, que inclui pesos e *bias*. Esses neurônios se distinguem apenas pela região da entrada à qual estão conectados, possibilitando a detecção eficiente de características similares em diferentes partes da entrada, como uma imagem. Devido a essa propriedade, as CNNs são altamente eficientes para

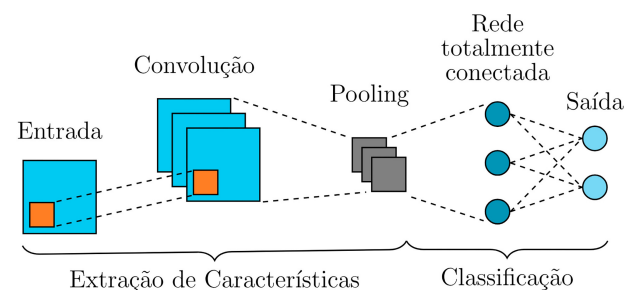


Figura 6: Representação pictórica de uma Rede Neural Convolucional. Dada uma imagem (uma matriz), submatrizes (em laranja) deste dado servem de argumento para a camada de filtros. Cada filtro serve como uma convolução desta submatriz em que cada kernel desta convolução é específico para abstrair alguma característica da imagem. O resultado desta camada é processado por uma camada de *pooling* que reduz a dimensão das matrizes resultantes. Finalmente, as matrizes reduzidas são argumentos para uma rede totalmente conectada que aprende padrões baseados nestes dados de dimensão menor.

tarefas onde a posição relativa de uma característica é mais importante do que sua localização exata, como em reconhecimento de padrões visuais e visão computacional [92, 93].

Além disso, é comum em arquiteturas convolucionais aplicar múltiplos filtros em paralelo sobre os mesmos valores de entrada. Cada filtro possui seu próprio conjunto de parâmetros que é compartilhado entre os neurônios que o compõem. Esta abordagem permite que a rede extraia diferentes tipos de características, como bordas horizontais, verticais ou texturas complexas, simultaneamente.

Quando usamos a palavra ‘convolução’ na Física, geralmente estamos expressando a ideia de que, se uma partícula em x é funcionalmente representada por $f(x)$, e a maneira como ela interage com seu ambiente (ponto a ponto) é regida por uma função (ou um campo) $G(x)$, então, pode-se representar a atuação das redondezas (digamos nos pontos x' vizinhos) sobre a partícula, pela convolução $A(x) = \int G(x - x')f(x')dx'$ [20]. Em geral, a função $G(x - x')$, chamada de *núcleo* ou *kernel*, preserva uma série de propriedades que sabemos ser importantes na física, como, por exemplo, causalidade. A mesma ideia é aplicada em CNNs: O núcleo da convolução, o *filtro*, fornece informação de como um píxel em uma imagem está relacionado com seus vizinhos. Neste caso, o uso do termo “filtro” foi herdado da área de pesquisa em visão computacional [94].

Veja na Equação (15) dois exemplos [95] de filtros usados para encontrar bordas nas imagens, nas direções x e y , e seus efeitos em imagens na Figura 7 (estes filtros são chamados em computação visual de *filtros de detecção de bordas* ou *filtros de Sobel* [96]).

$$G_x = \frac{1}{3} \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}, \quad G_y = \frac{1}{3} \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}. \quad (15)$$

Diferentemente da visão computacional clássica, onde filtros de imagem são projetados manualmente para tarefas específicas, as redes convolucionais aprendem os filtros ideais de forma autônoma. No início do treinamento, os parâmetros que definem os filtros são inicializados

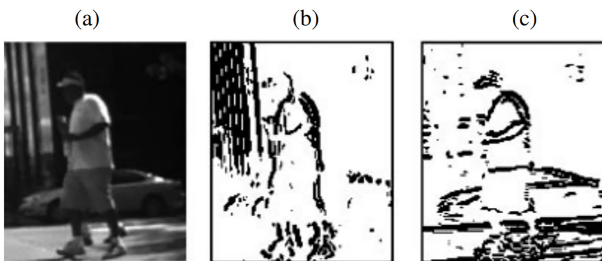


Figura 7: Exemplos de usos de convoluções para detecção de bordas. (a) Imagem original, (b) sob o filtro G_x , (c) sob o filtro G_y . Filtros estão explicitamente expressos na equação (15). Fonte [95].

aleatoriamente. À medida que o modelo é otimizado para minimizar a função de custo, esses pesos são ajustados iterativamente. Como resultado, a própria rede descobre o conjunto de filtros mais eficaz para extrair informações relevantes. Tipicamente, as camadas iniciais convergem para filtros que detectam características fundamentais, como bordas, texturas e gradientes de cor. Camadas subsequentes aprendem a combinar essas características simples para identificar padrões mais complexos, como partes de objetos, tornando o processo de extração de características mais poderoso e adaptável do que um conjunto de filtros pré-definidos.

É prática comum na literatura que as camadas convolucionais sejam seguidas de camadas de *pooling* e posteriormente camadas densas. Esta arquitetura mista está representada na Figura 6. As camadas de *pooling* desempenham um papel na redução das dimensões espaciais da entrada, ao mesmo tempo que preservam as características mais importantes da imagem ou do dado de entrada. Essa redução da dimensionalidade torna a rede neural mais eficiente em termos de custo computacional, viabilizando o uso de redes mais profundas, e torna a performance da rede menos sensível a pequenas variações na posição das características detectadas. Já as camadas densas servem para o uso final específico do que se queira deduzir das imagens analisadas, seja classificação, remoção de ruídos, etc [5]. Isso se deve ao fato de que as camadas densas reúnem e avaliam correlações entre os padrões extraídos pelas camadas convolucionais, permitindo inferências sobre características globais da imagem processadas. Há uma inspiração biológica para esta escolha, que está fora do escopo deste trabalho, mas que pode ser encontrada em [97].

O *pooling* é uma operação que considera um conjunto de elementos, como uma matriz de tamanho $K \times K$, e aplica uma regra de redução que gera uma matriz de tamanho menor, por exemplo, $\frac{K}{2} \times \frac{K}{2}$ [5]. Há várias formas de realizar essa redução, cada uma com suas próprias características e vantagens. Vamos considerar dois exemplos:

- 1) Dado um bloco de píxels de uma imagem maior, $B_{4 \times 4}$, i.e., uma submatriz da matriz de um dado de entrada, então

$$\text{pooling}(B_{4 \times 4}) = M_{2 \times 2},$$

onde

$$M_{ij} = \frac{1}{4} \sum_{k=i,j}^{i+2,j+2} B_{kl}.$$

i.e., cada elemento M_{ij} é a média dos elementos de uma submatriz 2×2 de $B_{4 \times 4}$.

- 2) Podemos fazer o mesmo, mas agora tomando o maior dos elementos $M_{ij} = \max_{k=i,j}^{i+2,j+2} (B_{kl})$. Veja que, após tal operação, o tamanho do dado de entrada está agora comprimido em uma matriz

menor. Reduzir a dimensionalidade dos dados diminui o tempo de processamento da rede, e logo acelera o processo de aprendizado também, porém sempre há perda de informação nesses processos, logo um balanço entre as duas operações deve ser encontrado.

2.3. Treinamento em redes profundas

No contexto de redes neurais profundas, o algoritmo de treinamento preserva, em essência, a mesma forma matemática apresentada nas equações (5) e (6); contudo, os pesos e *bias* passam a ser denotados por $w_{il}^{[k]}$ e $b_l^{[k]}$, respectivamente. Veja que agora os pesos e *bias* necessitam de um índice a mais que no caso do Perceptron, o qual indica a camada de que o peso advém. Queremos otimizar os pesos de qualquer camada arbitrária, mas sabemos somente o resultado da função custo, a qual é calculada pela saída da rede, e esta, por sua vez, é uma composição de todas as operações internas na rede, em uma ordem bem definida. Felizmente, há uma ferramenta muito bem conhecida para saber a derivada de funções compostas, que é a regra da cadeia. Lembremos que uma saída do l' -ésimo neurônio $y_{l'}^{[n]}$ da rede de n camadas tem derivada com relação à $w_{il}^{[k]}$ dada por $\frac{d}{dw_{il}^{[k]}} y_{l'}^{[n]}$. Escrevendo o resultado da soma com os respectivos pesos: $z_{l'}^{[n]} = \sum_{i'=1}^{N(n-1)} w_{i'l'}^{[n]} y_{i'}^{[n-1]} + b_{l'}^{[n]}$, i.e., calculada no neurônio l' da camada n , temos, pela regra da cadeia

$$\begin{aligned} \frac{d}{dw_{il}^{[k]}} y_{l'}^{[n]} &= \frac{d}{dw_{il}^{[k]}} \left(f_{l'}^{[n]}(z_{l'}^{[n]}) \right) \\ &= \frac{d}{dz_{l'}^{[n]}} f_{l'}^{[n]}(z_{l'}^{[n]}) \cdot \sum_{i'=1}^{N(n-1)} w_{i'l'}^{[n]} \frac{d}{dw_{il}^{[k]}} \left(y_{i'}^{[n-1]} \right). \end{aligned}$$

Logo, a derivada $\frac{d}{dw_{il}^{[k]}} y_{l'}^{[n]}$ depende da derivada de $\frac{d}{dw_{il}^{[k]}} y_{i'}^{[n-1]}$, o que mostra que temos uma estrutura recursiva neste cálculo. Ademais, como os índices i' se repetem na soma

$$\begin{aligned} \frac{d}{dw_{il}^{[k]}} z_{l'}^{[n-1]} &= \sum_{i'=1}^{N(n-1)} w_{i'l'}^{[n]} \frac{d}{dz_{i'}^{[n-1]}} f_{i'}^{[n-1]}(z_{i'}^{[n-1]}) \\ &\quad \cdot \frac{d}{dw_{il}^{[k]}} \left(z_{i'}^{[n-1]} \right), \end{aligned}$$

podemos definir uma matriz

$$M_{i'l'}^{[n,n-1]} = w_{i'l'}^{[n]} \frac{d}{dz_{i'}^{[n-1]}} f_{i'}^{[n-1]}(z_{i'}^{[n-1]}). \quad (16)$$

Então, podemos escrever, recursivamente

$$\frac{d}{dw_{il}^{[k]}} z_{l'}^{[n-1]} = M^{[n,n-1]} M^{[n-1,n-2]} \dots M^{[k+1,k]} \frac{d}{dw_{il}^{[k]}} z_l^{[k]}. \quad (17)$$

Veja que este produto é uma multiplicação de matrizes usual, logo o treino da rede neural é feito por um algoritmo linear, apesar da rede, em si, ser não linear. Então, para o último termo

$$\begin{aligned} \frac{d}{dw_{il}^{[k]}} z_l^{[k]} &= \sum_{i'=1}^{N_k} \frac{d}{dw_{il}^{[k]}} \left(w_{i'l}^{[k]} y_{i'}^{[k-1]} \right) \\ &= \sum_{i'=1}^{N_k} \delta_{i'i} \delta_{l'l} y_{i'}^{[k-1]} \\ &= y_i^{[k-1]}. \end{aligned}$$

Similarmente,

$$\frac{d}{db_l^{[k]}} z_l^{[k]} = 1.$$

Este processo de propagação do erro para trás é conhecido como *backpropagation*. Sua notável eficiência computacional é o que torna o treinamento de redes profundas prático. Em vez de calcular o gradiente da função de custo em relação a cada peso de forma independente, um processo que seria proibitivamente lento, o algoritmo emprega uma técnica mais inteligente. Durante a propagação direta (*forward pass*), os valores de ativação de cada neurônio são calculados e armazenados temporariamente. O *backpropagation*, então, reutiliza esses valores armazenados para aplicar a regra da cadeia de forma sistemática, calculando os gradientes da última camada em direção à primeira. Isso transforma a tarefa de otimização em uma sequência de multiplicações de matrizes, permitindo que múltiplos parâmetros sejam atualizados em paralelo. A seguir, veremos um passo a passo do algoritmo de *backpropagation*, inspirado na referência [20], de forma análoga à que usamos para atualizar os pesos do Perceptron na Seção 2.1.

1. Calcule o desvio da função custo do valor real já com respeito a uma saída l e sua derivada após aplicar a regra da cadeia uma vez:

$$\Delta_{l'} = (y_{l'}^{[n]} - \hat{y}_{l'}) \frac{d}{dz_{l'}^{[n]}} f_{l'}^{[n]}(z_{l'}^{[n]}). \quad (18)$$

2. Agora, como as derivadas $\frac{d}{dz_{i'}^{[k]}} f_{i'}^{[k]}(z_{i'}^{[k]})$ foram calculadas com a etapa propagação em que calculamos $f_{l'}^{[k]}(z_{l'}^{[k]})$, busque-as na memória e atualize $\Delta_{l'}$ com a composição matricial $\hat{M}$ da equação (17):

$$\Delta_{l'} \leftarrow \Delta_{l'} \hat{M} \frac{d}{dw_{il}^{[k]}} z_l^{[k]}. \quad (19)$$

3. Repita os passos 1 e 2 para todos os l' 's. Em seguida, tome a média dos $\Delta_{l'}$'s e chame-a de Δ . Então, atualize o peso $w_{il}^{[k]}$ como fizemos na equação (5) e (6):

$$w_{il,\text{novo}}^{[k]} \leftarrow w_{il,\text{antigo}}^{[k]} - \eta \Delta, \quad (20)$$

$$b_{l,\text{novo}}^{[k]} \leftarrow b_{l,\text{antigo}}^{[k]} - \eta \Delta. \quad (21)$$

O algoritmo de *backpropagation* apresenta limitações que podem comprometer seu desempenho e sua aplicabilidade em redes neurais profundas. Uma das principais limitações é o problema da explosão e do desaparecimento do gradiente, que ocorre quando os gradientes calculados pelo algoritmo são excessivamente grandes ou pequenos [5]. Este fenômeno pode resultar na amplificação ou atenuação cumulativa dos gradientes ao longo das camadas, dificultando a convergência adequada dos parâmetros da rede, pois o tamanho do passo da otimização depende da magnitude do gradiente, que, quando pequeno, resulta em convergência lenta. Isso exige muitos ciclos de treinamento, aumentando o tempo e os recursos computacionais necessários. Outra limitação importante é que os algoritmos de otimização que utilizam o *backpropagation* são fortemente influenciados pela inicialização dos parâmetros. Esse fator pode resultar em desempenho insatisfatório ou na incapacidade de encontrar um ótimo local.

Além disso, o algoritmo de *backpropagation* tende a ser mais eficaz quando utilizado com grandes conjuntos de dados, pois permite que a rede neural capture melhor as complexidades e variações dos padrões presentes no problema. Contudo, a exigência de grandes volumes de dados pode se tornar um obstáculo em domínios onde a coleta de dados é limitada ou envolve questões éticas e de privacidade, como em aplicações médicas ou de segurança. Nessas situações, a escassez de dados pode comprometer a capacidade da rede de aprender representações robustas [98]. Ademais, na ausência de técnicas apropriadas de regularização, como o *dropout* ou normalização, as redes neurais podem facilmente sofrer de sobreajuste (*overfitting*) [99]. Esse fenômeno ocorre quando o modelo se ajusta excessivamente aos dados de treinamento, capturando tanto padrões relevantes quanto ruídos específicos do conjunto de dados, resultando em um desempenho insatisfatório em novos dados não vistos, devido à sua capacidade limitada de generalização.

3. Aplicação de Redes Neurais na Física

Neste capítulo, apresentamos quatro abordagens distintas que utilizam redes neurais aplicadas ao estudo do pêndulo simples, um sistema físico clássico. Cada uma das abordagens explora diferentes aspectos do problema, empregando técnicas de ML para modelar, prever e interpretar o comportamento do sistema. Na primeira abordagem (Seção 3.1), utilizamos aprendizado supervisionado para estimar um parâmetro físico do sistema, especificamente a aceleração da gravidade g , a partir de dados simulados. Em seguida, na Seção 3.2, aplicamos redes neurais profundas para resolver a equação diferencial ordinária que descreve o movimento do pêndulo, explorando a capacidade dessas redes de aproximar soluções complexas de equações diferenciais. Na terceira abordagem (Seção 3.3), empregamos

autoencoders para descobrir o espaço latente do sistema, buscando reduzir a dimensionalidade dos dados e encontrar uma representação compacta e eficiente do estado do pêndulo. Por fim, na Seção 3.4, utilizamos o método SINDy (*Sparse Identification of Non-linear Dynamics*), conforme desenvolvido por [22], visando identificar a equação diferencial subjacente que governa o sistema, utilizando imagens como entrada. Cada uma dessas abordagens ilustra o potencial das redes neurais em diferentes contextos de modelagem física, destacando a versatilidade dessas ferramentas para resolver problemas complexos eficientemente.

3.1. Exemplo 1: Pêndulo

Começaremos com um exemplo fundamental: o aprendizado supervisionado de parâmetros aplicado ao pêndulo simples. Escolhemos este problema por duas razões principais. A primeira é que o modelo do pêndulo é amplamente conhecido por estudantes de graduação, tornando-o um ponto de partida acessível para introduzir conceitos de ML no contexto de sistemas físicos. A segunda razão é que o estudante já deve estar familiarizado com problemas de regressão de parâmetros, comuns em experimentos de laboratório didático, como a estimativa de constantes físicas. Neste exemplo, apresentado no Jupyter Notebook 04 [79], visamos demonstrar uma abordagem alternativa para resolver um problema que o aluno já conhece bem, utilizando redes neurais para estimar parâmetros do sistema. Embora se trate de um problema didático simplificado, com fins ilustrativos, ele possibilita estabelecer a base conceitual e metodológica, preparando o terreno para as aplicações mais inovadoras e avançadas que serão discutidas nas seções subsequentes.

A equação diferencial do pêndulo é dada por:

$$\frac{d^2\theta}{dt^2} + \frac{g}{\ell} \sin \theta = 0, \quad (22)$$

onde θ é o ângulo em relação ao eixo vertical, g é a aceleração da gravidade, e ℓ é o comprimento da haste do pêndulo. Na aproximação de ângulos pequenos, sabemos que $\sin \theta \approx \theta$, e logo a equação simplifica para

$$\frac{d^2\theta}{dt^2} + \frac{g}{\ell} \theta = 0, \quad (23)$$

com solução conhecida,

$$\theta(t) = \theta_0 \cos \left(\sqrt{\frac{g}{\ell}} t \right). \quad (24)$$

No contexto dos laboratórios didáticos, o aluno provavelmente encontrou a expressão para o período $T = \frac{2\pi}{\omega} = 2\pi \sqrt{\frac{\ell}{g}}$, isolou g , mediu experimentalmente T e ℓ , e encontrou o valor de g . Neste exemplo, suponhamos que os dados experimentais, representados por $\theta(t_i)_{data}$, foram obtidos, por exemplo, por meio da filmagem do

pêndulo em um dado instante de tempo i , ou por meio da projeção de seu movimento sobre uma fita que se desloca linearmente no tempo, como em um sismógrafo⁵. Suponhamos adicionalmente que medimos ℓ e que também sabemos $\theta_0 = \theta(0)$.

Para esta aplicação, utilizaremos uma rede neural com um único neurônio, possuindo como entrada um vetor $\mathbf{t}$ de tamanho $M + 1$ e uma saída. Essa configuração equivale a um perceptron, no qual a entrada representa os instantes de tempo e a saída corresponde aos valores previstos para a aceleração da gravidade. Denotamos essa rede como $NN(\mathbf{t}; \mathbf{w}, \mathbf{b}) = g'$, onde $\mathbf{t}$ representa os tempos, e os parâmetros $\mathbf{w}$ e $\mathbf{b}$ são os pesos e o *bias* da rede, respectivamente. O tempo foi particionado ao longo de um intervalo definido, representado como $\mathbf{t} = (t_0, t_1, \dots, t_i, \dots, t_M)$. Declarada a rede neural, precisamos construir nossa função custo. Para podermos comparar os dados, que são ângulos, com a saída da rede, que é uma constante g' .

Portanto, utilizaremos a saída da rede na equação (24) antes de passar para a função custo, o que resulta em $\theta'(t) = \theta_0 \cos \sqrt{\frac{g'}{\ell}} t$. Assim, temos duas grandezas comparáveis, os dados experimentais $\theta(t_i)_{data}$, e a função customizada $\theta'(t_i)$, para cada tempo t_i . Essa construção fornece uma medida indireta de g' , e podemos então avaliar a performance da rede por meio da função custo:

$$\mathcal{L} = \frac{1}{M} \sum_{i=0}^M (\theta(t_i)_{data} - \theta'(t_i))^2. \quad (25)$$

Quando $\mathcal{L} \approx 0$, sabemos que $g \approx g'$, e logo a rede neural aproximou bem a constante desejada, e a curva gerada por $\theta'(t)$ é próxima à original. O objetivo deste exemplo é mostrar que uma rede neural pode ser usada para regressão e recuperar os parâmetros de uma curva, i.e., esta abordagem é generalizável para exemplos mais complexos, onde se tem conhecimento explícito da expressão analítica que gera a curva, mas não se têm muitos pontos.

Na Figura 8 temos dois gráficos obtidos após o treinamento da rede neural. Para esse exemplo foi utilizada uma rede neural MLP com uma entrada (que recebe o tempo), uma camada oculta com 1 neurônio e uma saída, e função de ativação Tangente hiperbólica. O treinamento consistiu de 5 mil épocas de utilizando o otimizador *Adam*, com taxa de aprendizado inicial de 10^{-2} . Esta escolha de parâmetros foi determinada para reduzir a complexidade neste exemplo. O gráfico na Figura 8.a) representa a equação (25) para cada época. Ela tem o papel de nos informar sobre o progresso do treinamento da rede neural, pois valores decrescentes da função de custo são um indicador da convergência do modelo para um resultado melhor. Porém, cabe mencionar que isto nem sempre é verdade, podem ocorrer

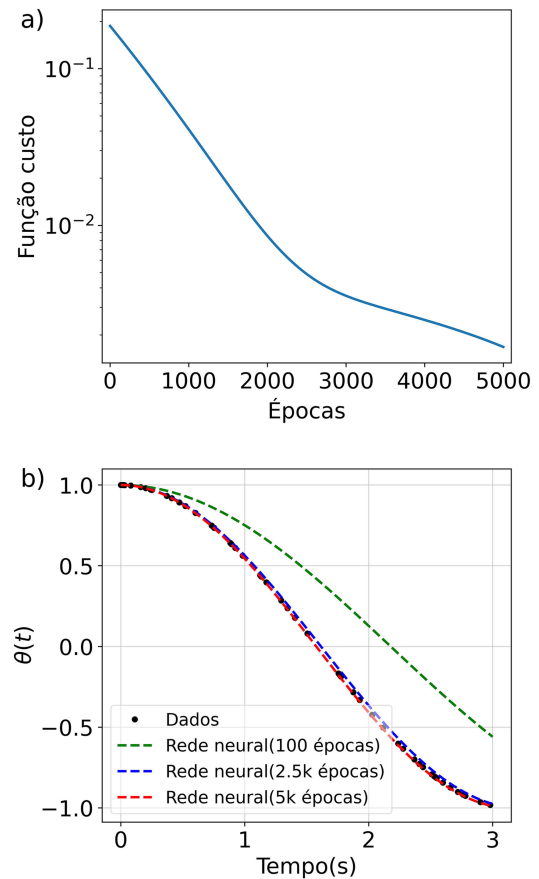


Figura 8: Resultado do treinamento da rede neural para o aprendizado da constante gravitacional do Exemplo 1. (a) Evolução da função de custo ao longo das épocas de treinamento, apresentada em escala logarítmica. (b) Comparação entre a solução dada pela equação (24) com a constante encontrada pela rede neural em três estágios de treinamento: 100 épocas (linha verde tracejada, fase inicial), 2,5 mil épocas (linha azul tracejada, fase intermediária) e 5 mil épocas (linha vermelha tracejada, fase próxima da convergência). Essas escolhas visam apenas ilustrar qualitativamente a melhoria progressiva da predição à medida que o treinamento avança; quaisquer outros valores poderiam ser usados, desde que representem diferentes estágios de convergência da função de custo.

alguns fenômenos durante a otimização, com *sobreajuste* (*overfitting*) no qual a rede neural se ajusta tanto aos dados de treinamento que perde a capacidade de generalizar e fazer previsões precisas sobre dados não vistos.

Além disso, o comportamento da função custo ao longo das épocas pode fornecer pistas sobre o desempenho e a convergência do modelo. Um declínio suave e consistente na função custo ao longo das épocas sugere que o modelo está progredindo de forma estável e aprendendo de maneira eficiente. No entanto, flutuações repentinas ou uma estagnação na redução da função custo podem indicar a necessidade de ajustes nos hiperparâmetros do modelo. Portanto, é importante analisar o gráfico da função custo ao longo das épocas,

⁵ Isto funciona aproximadamente se θ é pequeno, pois a variação de $y(t) = -\ell(1 - \cos \theta) \approx \frac{\ell}{2}\theta(t)^2$ é de segunda ordem.

sendo uma prática comum durante o ML, auxiliando na tomada de decisões para melhor otimização do modelo. Existem outros problemas que são relevantes durante o treinamento, mas que não serão tratados de maneira mais aprofundada neste texto, como incompatibilidade entre a função custo [100], platôs e/ou função custo estagnada [101], explosão e desaparecimento/esvanecimento dos gradientes [102], como também o subajuste dos modelos [99].

Na Figura 8.b) temos a comparação da saída da rede neural com a expressão teórica (24) da solução do pêndulo, mostrando um bom acordo entre as curvas. Este exemplo pode ser encontrado no *Jupyter Notebook* com título ‘03-Exemplo 1’, nele o leitor encontrará mais detalhes técnicos da implementação, como também uma versão alternativa mais eficiente para resolver esse problema usando diferenciação automática.

3.2. Exemplo 2: Resolvendo a equação diferencial

Nesta seção, apresentamos como resolver equações diferenciais ordinárias (EDOs) utilizando o método de Redes Neurais Informadas com Física, do inglês *Physics-Informed Neural Networks* (PINNs) [33]. As redes neurais são reconhecidas por sua capacidade de aproximar funções complexas, justificando seu uso para soluções aproximadas de EDOs. Esta aplicação é o cerne do método PINN, que integra conhecimento físico específico no processo de aprendizado da rede, como detalhado na introdução deste artigo. Para integrar a equação diferencial na aprendizagem da rede, incorporamos a equação (23) na função custo durante o treinamento. Esta nova função é composta não apenas pela equação diferencial, mas também pelas condições iniciais e de contorno do problema, conforme expresso na seguinte equação:

$$\mathcal{L} = \mathcal{L}_{\text{data}} + \mathcal{L}_{\text{EDO}} + \mathcal{L}_{\text{IC}} + \mathcal{L}_{\text{CC}}. \quad (26)$$

Aqui, $\mathcal{L}_{\text{data}}$ é utilizado quando dispomos de dados experimentais que representam parte da solução da equação, semelhante à equação (25); $\mathcal{L}_{\text{EDO}}$ reflete a contribuição da própria equação diferencial, $\mathcal{L}_{\text{IC}}$ corresponde à contribuição das condições iniciais (IC), e $\mathcal{L}_{\text{CC}}$ às condições de contorno (CC). Note que, apesar de adicionarmos restrições à função custo, neste exemplo ainda estamos tratando de aprendizado supervisionado, pois os ângulos são comparados com os ângulos preditos pela rede através do termo $\mathcal{L}_{\text{data}}$.

Para ilustrar este método, aplicaremos o PINN à equação do pêndulo simples com pequenos ângulos, resultando em um oscilador harmônico simples. A rede neural, denotada como $NN(\mathbf{t}; \mathbf{w}, \mathbf{b})$, será treinada para aproximar $\theta(\mathbf{t})$, assim a solução da equação diferencial do pêndulo que será inserida durante o treinamento tem

a seguinte forma:

$$\mathcal{L}_{\text{EDO}} = \sum_i \left(\frac{d^2}{dt^2} NN(\mathbf{t}_i; \mathbf{w}, \mathbf{b}) + \frac{g}{\ell} NN(\mathbf{t}_i; \mathbf{w}, \mathbf{b}) \right)^2. \quad (27)$$

Como esta é a equação diferencial, seu valor mínimo ideal é zero. A condição inicial é dada apenas por θ_0 , sendo modelada na função custo como:

$$\mathcal{L}_{\text{IC}} = (NN(t = 0; \mathbf{w}, \mathbf{b}) - \theta_0)^2. \quad (28)$$

Com este enfoque, as PINNs permitem uma integração eficaz e direta das leis físicas na arquitetura de aprendizado profundo, abrindo caminho para soluções robustas e precisas em problemas físicos complexos. Isso ocorre porque agora a rede neural contém conhecimento sobre o sistema físico, e a NN é uma aproximação da solução deste modelo, considerando que satisfaz a EDO característica, por definição. Além disso, como o aprendizado é mais estruturado (i.e., há mais informação na função custo), as fases de treino das PINNs são, em geral, mais curtas e requerem menos dados. Compare, por exemplo, os gráficos 8.a) e 9.a). É notável que, apesar da função custo em (26) ter mais termos positivos do que a definida em (25), o valor de $\mathcal{L} \approx 10^{-2}$ é alcançado com metade das épocas do que no caso do Exemplo 1. Para esse exemplo, foi utilizada uma rede neural MLP com uma entrada (que recebe o tempo), duas camadas ocultas com 10 neurônios cada e uma saída, com função de ativação seno. O treinamento consistiu de 5 mil épocas, utilizando o otimizador *Adam* com taxa de aprendizado de $\eta = 0.01$, adicionado a um *scheduler* que reduz em 10% o valor de η a cada 250 épocas.

Na Figura 9 temos dois gráficos obtidos após o treinamento da rede neural para se resolver a equação diferencial. O gráfico à esquerda (a) representa a equação (26) para cada época, e em (b) temos a comparação da saída da rede neural com a expressão teórica (24) da solução do pêndulo, mostrando um bom acordo entre as curvas. Este exemplo pode ser encontrado no *Jupyter Notebook* com título ‘04-Exemplo 2’. Nele o leitor encontrará os detalhes técnicos da implementação.

3.3. Exemplo 3: revisitando o pêndulo com *Autoencoders*

As redes neurais também podem ser aplicadas a tarefas rotineiras em computação, trazendo novas abordagens que podem revelar propriedades inesperadas ou oferecer melhorias em termos de eficiência e desempenho. Um exemplo é o uso de redes neurais para compressão de dados, utilizando a arquitetura de codificador-decodificador, conhecida como *Autoencoder*. O *Encoder* é responsável por transformar um dado de entrada (como uma imagem) em uma representação de dimensionalidade reduzida, comprimindo a informação para um formato mais compacto (como uma lista com um número

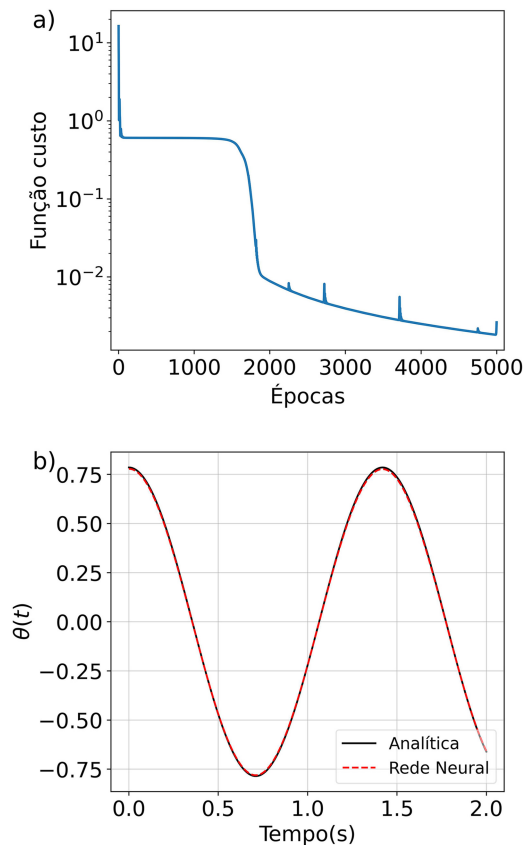


Figura 9: Resultado da resolução da equação diferencial com rede neurais via o método PINN. Em a) o gráfico da função custo em escala logarítmica em função das épocas ilustra o processo de treinamento, que evidencia como o erro evolui para cada iteração. Em b) temos a evolução do ângulo do pêndulo simples em função do tempo obtida via solução analítica, em preto, e o resultado da rede neural após o treino, em vermelho.

menor de bits). Essa técnica não apenas diminui o tamanho do dado, mas também pode capturar as principais características e abstrações do conjunto de dados original, o que pode ser útil para tarefas como a remoção de ruído, geração de novos dados, ou mesmo aprendizado não supervisionado de representações latentes [103].

Considere o seguinte exemplo: uma curva definida pela função $f(t) = \sin(\omega_1 t) + \sin(\omega_2 t)$ apresenta uma expressão funcional que pode se tornar bastante complexa, especialmente quando mais frequências são adicionadas ao modelo. No entanto, apesar da curva ser composta por infinitos pontos ao longo do tempo, toda a sua informação pode ser completamente determinada conhecendo apenas duas variáveis-chave: as frequências ω_1 e ω_2 (aqui estamos assumindo que a fase é conhecida e $\phi_i = 0$, para $i = 1, 2$). Isso implica que, para armazenar a curva inteira, basta saber que ela é uma soma de duas funções seno com suas respectivas frequências, representadas por dois números reais, sendo um exemplo de como uma descrição paramétrica pode ser extremamente compacta.

De forma semelhante, os fractais exemplificam outro caso de complexidade emergente a partir de regras

simples: figuras altamente complexas e visualmente ricas podem ser geradas por meio de equações iterativas relativamente simples. Esses exemplos ilustram como informações complexas podem ser compactamente representadas por um conjunto reduzido de parâmetros ou instruções [104].

Outro exemplo de aplicação de *encoder* é na compressão de imagens em preto e branco, cuja representação dos pixels é dada por uma matriz bidimensional (2D). Quando uma imagem é submetida à transformada discreta de Fourier, a informação espacial dos pixels é convertida em uma representação no domínio da frequência. Nesse processo, a maioria da informação significativa da imagem pode ser capturada por um conjunto relativamente pequeno de coeficientes de frequência (isto é conhecido como a aplicação de um filtro *passa baixa*, ou em inglês *low pass filter* [5]). Assim, ao invés de armazenar todos os valores dos pixels, é possível representar a imagem inteira de maneira compacta, utilizando apenas as frequências mais relevantes que compõem sua estrutura.

Um algoritmo conhecido que comprime dados por meio da Transformada Discreta do Cosseno (DCT) é o formato de imagens JPEG [5]. Fornecido um dado comprimido, para acessar a versão original, é necessário haver um algoritmo que realize a operação inversa, que, tomando o dado comprimido, recupere (mesmo que parcialmente) a imagem inicial. O algoritmo que realiza essa tarefa é chamado de *Decoder*, e, no caso ilustrado acima, seria a transformada inversa de Fourier.

Aplicando-se ambos algoritmos concatenados, *encoder* e depois *decoder*, temos um algoritmo resultante chamado de *Autoencoder*. O nome *auto* sinaliza que uma parte do algoritmo é a inversa do outro, então a entrada e saída de um *autoencoder* devem ser correspondentes. Em geral, essa correspondência não é perfeita, pois sempre há perda de informação em um processo de compressão, e o uso da palavra ‘inversa’ é, na verdade, um abuso de notação, porém, um bom *autoencoder* deve ser capaz de recuperar a imagem original com boa fidelidade.

No caso das redes neurais, podemos construir uma arquitetura para um *autoencoder*, como ilustrado na Figura 10. Temos uma entrada $\mathbf{x}(t)$ que fornece informações para um *encoder* φ , cuja saída é uma camada oculta com menos neurônios, $\mathbf{z}(t)$. Neste exemplo, e nos que seguem, assumimos que φ é composto por uma série de camadas convolucionais (CNN) seguidas de camadas totalmente conectadas. Assim, como na compressão de imagens, φ reduz a entrada $\mathbf{x}(t)$, que contém muitas informações, para alguns poucos números em $\mathbf{z}(t)$, chamados de *espaço latente* ou *camada latente* (às vezes também chamada de *code*). A tarefa do operador ψ é então decodificar a informação comprimida e recuperar a entrada inicial de forma aproximada: $\psi(\mathbf{z}(t)) = \hat{\mathbf{x}}(t) \approx \mathbf{x}(t)$. É prática comum que a rede de ψ seja a ordenação contrária de φ , i.e., camadas totalmente conectadas seguidas de camadas CNN. Então, a tarefa

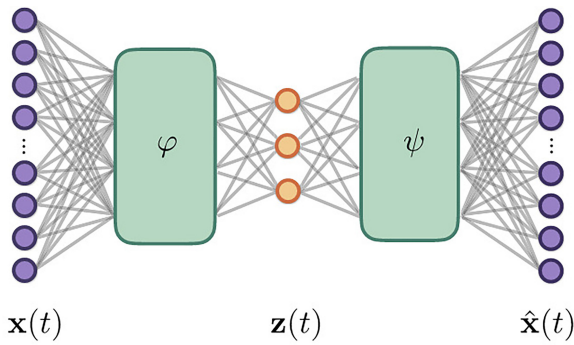


Figura 10: Representação pictórica de uma rede neural *Autoencoder*. A entrada é representada por $\mathbf{x}(t)$, passa por uma camada de neurônios (geralmente CNN), e então é comprimida para um número menor de neurônios pelo *encoder* φ , de forma que $\varphi(\mathbf{x}(t)) = \mathbf{z}(t)$. O *decoder*, a saída, ψ recupera a entrada original (aproximadamente) com a operação $\psi(\mathbf{z}(t)) = \hat{\mathbf{x}}(t)$. O objetivo do *autoencoder* após treinado é reproduzir a relação $\psi \circ \varphi = \mathbb{I}$. Aqui φ é composto por redes CNN seguidas de uma rede neural totalmente conectada, e ψ , o mesmo, porém com a ordem das camadas invertida. Fonte [22].

do *autoencoder* é aproximar a relação⁶

$$\psi \circ \varphi \approx \mathbb{I},$$

onde $\mathbb{I}$ é o operador identidade no espaço vetorial que contém $\mathbf{x}(t)$, i.e., queremos que $\psi(\varphi(\mathbf{x}(t))) \approx \mathbf{x}(t)$. Assim, quando declararmos a função de custo do tipo MSE para a otimização, teremos

$$\mathcal{L}_{\text{data}} = \|\mathbf{x}(t) - \hat{\mathbf{x}}(t)\|_2^2 = \|\mathbf{x} - \psi \circ \varphi(\mathbf{x})\|_2^2. \quad (29)$$

Este processo representa uma mudança fundamental em relação ao que fizemos anteriormente. Em vez de comparar a saída da rede com um rótulo externo, a otimização ocorre ao comparar a saída $\mathbf{z}(t)$ com a própria entrada. Por não haver uma supervisão explícita, este método é um exemplo de aprendizado não supervisionado.

Didaticamente, considere o seguinte cenário: desejamos aplicar um *autoencoder* em uma situação onde o leitor pretende enviar uma imagem para outra pessoa, mas possui a limitação de transmitir apenas 3 números reais. Se tivermos uma rede treinada, podemos “cortar” o *autoencoder* ao meio, mantendo o *encoder* e enviando o *decoder* ao destinatário da imagem. A imagem original é comprimida pelo *encoder* φ , gerando $\mathbf{z}(t)$. Assim, podemos enviar $\mathbf{z}(t)$ para a outra ponta do canal de comunicação, onde será decodificada com ψ , recuperando a mensagem original. Em física, temos outros interesses. Por exemplo, no reconhecimento de imagens, essas arquiteturas podem ser usadas para limpeza de dados originais. Em técnicas observacionais em astrofísica, *autoencoders* podem remover o ruído de

ondas eletromagnéticas, facilitando a visualização e o processamento dos dados [105].

No exemplo que segue, iremos revisitar o pêndulo com *autoencoders* e analisaremos o comportamento do espaço latente. Vamos supor que nossos dados sejam uma sequência de *frames* originados de um vídeo da dinâmica de um pêndulo com apenas uma oscilação, partindo de um ângulo inicial $\theta_0 = \theta(t=0)$ qualquer e velocidade inicial igual a zero, $\dot{\theta}(0) = 0$. Para gerar dados análogos a um vídeo da dinâmica do pêndulo, nosso conjunto de dados é composto por uma função que toma o resultado numérico da equação (22) do pêndulo, $\theta(t)$, e o converte em uma imagem de duas dimensões $\mathbf{x}(t)$.

O objetivo deste exercício é verificar se o espaço latente do *autoencoder* com as imagens do pêndulo pode ser reduzido a apenas uma variável. Isto deveria ser possível, dado que a equação do pêndulo é um sistema que pode ser descrito por uma única variável $\theta(t)$. Esperamos, então, que um *autoencoder* com espaço latente ($\mathbf{z}(t)$), de dimensão um, deva ser capaz de reproduzir as imagens originais, mesmo no regime não linear de ângulos grandes da equação do pêndulo. Veja que a não linearidade das redes neurais nos permite exigir que a compressão ocorra mesmo para a equação diferencial de grandes ângulos, o que não é possível com técnicas lineares de redução de dimensionalidade, como *Principal Component Analysis* (PCA) [22], pois estas exigem que haja uma linearização em algum momento da compressão, e isto pode levar a perdas na representação.

Para realizar o treinamento foi utilizada uma *autoencoder* com 2 camadas convolucionais, a primeira com 5 filtros, e a segunda com 10 filtros, seguidos de uma camada de *pooling* com kernel de tamanho $K = 2$, e 3 camadas densas para o *encoder*. Similarmente, o *decoder* tem as mesmas camadas, mas em ordem contrária. As camadas densas do *encoder* contêm 1.6×10^4 , 1600 e 40 neurônios, respectivamente. O tamanho da saída da última camada depende do valor do espaço latente; na primeira abordagem, usamos 1, depois 2 neurônios. Para todos neurônios a função de ativação é do tipo ReLU. O treinamento foi composto de 2 mil épocas, e o otimizador *Adam* com taxa de aprendizado de $\eta = 0.001$, sendo reduzida em 10% do seu valor a cada 500 épocas. O resultado do nosso exemplo pode ser visto na Figura 11, onde temos a comparação da imagem original (à esquerda) e a reconstruída (à direita) após o treinamento⁷.

CNNs são notórias em abstrair somente informações essenciais dos dados, no sentido de que suas representações podem ser tais que ignoram entradas descorrelacionadas nos dados [106]. Assim, podemos tomar um modelo treinado somente com imagens do pêndulo, sem ruído, e expor o modelo a dados ruidosos (onde

⁶ Note que $\psi \circ \varphi$ representa a composição de ψ com φ , e não o produto interno.

⁷ Neste exemplo, é recomendado que o leitor utilize processamento de GPU (por exemplo, disponível gratuitamente no Colab pela Google), pois o tamanho da rede utilizada torna a fase de treinamento longa.

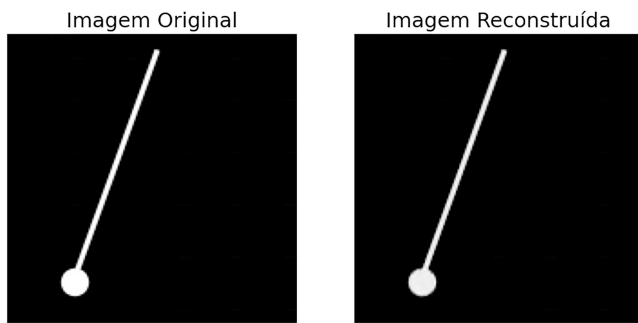


Figura 11: Comparação entre a imagem original e a imagem reconstruída utilizando um *autoencoder* com espaço latente igual a 1. A imagem à esquerda representa a imagem original e a imagem à direita é a versão reconstruída da mesma imagem.

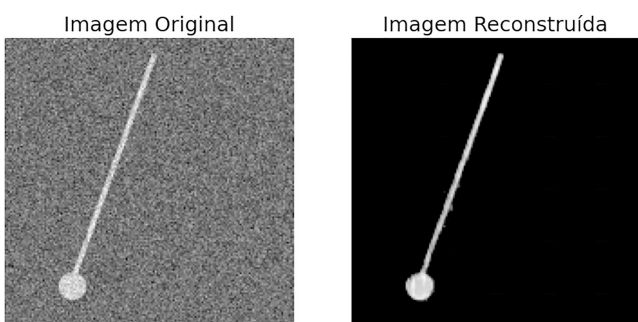


Figura 12: Comparação entre a imagem original com ruído adicionado e a imagem reconstruída utilizando um *autoencoder* com espaço latente igual a 2. A imagem à esquerda representa a imagem original com ruído adicionado e a imagem à direita é a versão reconstruída da mesma imagem. Nesta imagem o ruído é de 20%. No material suplementar, notamos que a rede ainda recupera bem o objeto e posicionamento com ruído de até 40%.

introduzimos um ruído de distribuição homogênea, logo decorrelacionada), como na Figura 12. Veja que, apesar das novas imagens serem perturbadas por um ruído de 20%, i.e., com probabilidade $P = 1/5$ de substituir um píxel por um valor aleatório, ainda assim o modelo consegue reproduzir o comportamento esperado e sem o ruído. Com base nos modelos treinados, verificou-se que, ao aumentar a dimensão do espaço latente para dimensão 2, o ruído poderia atingir até 40% antes de a rede apresentar sinais de baixa performance. Esta propriedade é bem conhecida na área, usada, por exemplo, em astrofísica observacional para limpeza de imagens [107]. Os testes discutidos acima podem ser encontrados no material complementar, i.e., no Notebook “06-Exemplo 3”.

Podemos agora analisar como cada *frame* se distribui no espaço latente, i.e., como está representado cada segundo no espaço codificado pela rede. Para ilustrar, treinamos a rede em um espaço latente de dimensão dois, $z_1(t)$ e $z_2(t)$, veja a Figura 13. Na imagem (a), temos um gráfico em 2D, com o eixo horizontal representando z_1 e o eixo vertical z_2 . Cada círculo representa o espaço

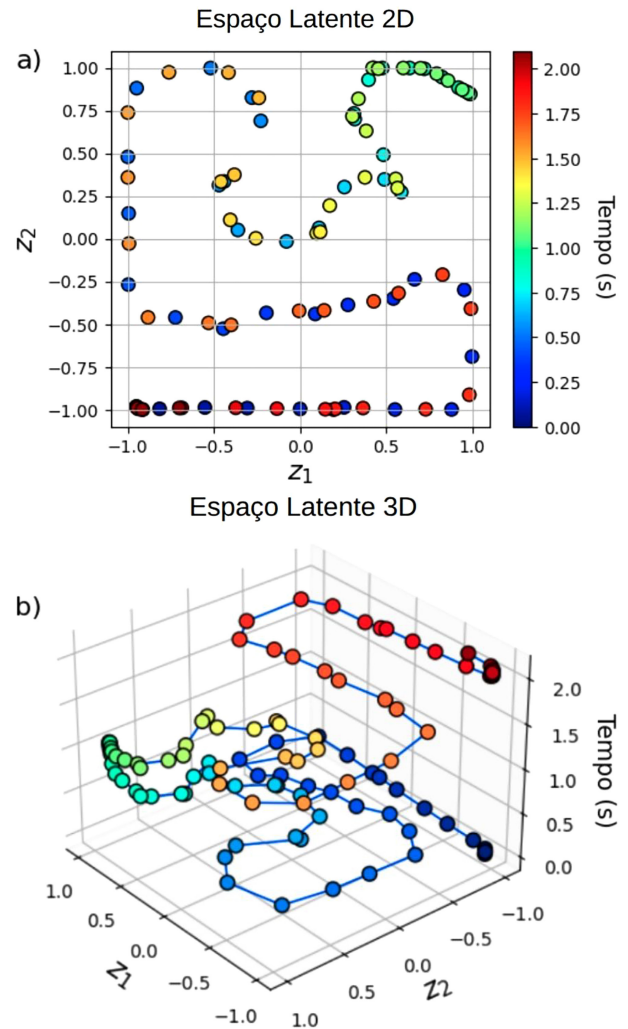


Figura 13: Visualização em 2D em a) e 3D em b) do espaço latente composto por z_1 e z_2 , em função do tempo. Em ambas as imagens, cada ponto representa a posição da imagem do pêndulo no espaço latente em um instante de tempo t_i .

latente para um *frame* do vídeo, que corresponde a cada instante de tempo do pêndulo.

Para identificar qual o instante de tempo do *frame* em cada círculo, utilizamos uma cor, variando de 0 (em azul) até 2.1 segundos (em vermelho), explicitadas no mapa de cores à direita da imagem. Note que, ao longo da curva, não há uma separação contínua nos instantes de tempo do pêndulo na figura à direita, i.e., pelo mapa de cores, temos as cores azuis e vermelhas intercaladas, enquanto estas deveriam estar idealmente em pontas opostas da curva contínua que gera estes pontos: este é o caráter não linear da codificação. Este resultado se deve ao fato de que, como a posição inicial e final do pêndulo após um período T devem ser muito próximas, a codificação deve mapear estes dados próximos, já que são quase a mesma imagem. Porém, apesar da rede ter mapeado os trechos de tempo $[0, T/2]$ e $[T/2, T]$ de maneira sobreposta, com ordem contrária, ainda é possível observar que a codificação está inteiramente contida em uma curva (i.e., uma

superfície de dimensão 1). Para garantir a continuidade da curva que contém a codificação, podemos adicionar mais um eixo, em que ordenamos os dados na mesma ordem em que os *frames* são apresentados, o resultado pode ser visto na Figura 13 (b) onde temos o mesmo gráfico em 3D. Assim, podemos visualizar a evolução do espaço latente em função do tempo, e observa-se que a rede neural foi capaz de codificar corretamente a posição do pêndulo em uma superfície de dimensão 1 parametrizada pelo tempo.

Esta abordagem mostra que podemos alavancar a capacidade de representação dos *Autoencoders* para estimar a dimensionalidade de um sistema. Neste tipo de aplicação, não nos interessa a complexidade da codificação resultante, mas somente a dimensão mínima em que conseguimos codificar o sistema, pois isso nos indica quantas coordenadas são necessárias para descrevê-lo.

3.4. Exemplo 4: *Autoencoders* e arquiteturas SINDy para descoberta de EDOs

Como discutimos na introdução, há boas justificativas para defender uma posição cética com relação a redes neurais, pois, por vezes, seu funcionamento pode ser mais opaco do que o próprio sistema que estudamos. Afinal, uma rede neural pode ter milhares (ou trilhões, em casos recentes) de parâmetros para serem ajustados que atuam de maneira não linear, sobrepujando a capacidade humana de análise [108]. Dito isto, há um esforço na comunidade para aliar conhecimentos prévios sobre sistemas físicos para reduzir a opacidade das redes neurais e podermos obter resultados que forneçam uma perspectiva mais detalhada sobre como uma rede neural opera, sem perda de generalidade.

Neste último exemplo, aplicaremos os conceitos desenvolvidos nos exemplos anteriores para explorar uma arquitetura recente e de especial interesse para a comunidade científica. O foco será a Identificação Esparsa de Dinâmica Não Linear (SINDy, *Sparse Identification of Nonlinear Dynamics*) [22]. O método SINDy busca identificar a dinâmica subjacente de um sistema não linear, restringindo o espaço latente da rede neural de modo a revelar suas equações de movimento, enquanto a rede, como um todo, mantém a funcionalidade de um *autoencoder*.

Suponha então que temos um sistema não linear, neste caso gerado pela equação (22), e que produza um conjunto de dados de imagens $\mathbf{x}(t)$. Por exemplo, uma filmagem de um pêndulo com ângulos grandes. Suponha também que o espaço latente possa ser representado por uma equação diferencial (possivelmente não linear) como

$$\frac{d}{dt}\mathbf{z}(t) = \mathbf{g}(\mathbf{z}(t)). \quad (30)$$

ou

$$\frac{d^2}{dt^2}\mathbf{z}(t) = \mathbf{g}(\mathbf{z}(t)). \quad (31)$$

A seguir, utilizamos como exemplo canônico do SINDy o caso em que $\mathbf{z}(t)$ satisfaz uma equação diferencial de primeira ordem no que segue, porém, também usaremos a segunda ordem para o exemplo particular do sistema do pêndulo não-linear. Em princípio, poderíamos ter escolhido qualquer ordem. A seguir, apresentaremos a construção de modelos para a primeira e segunda ordem em paralelo, destacando que as mesmas regras de derivação podem ser aplicadas iterativamente para alcançar ordens superiores. No material suplementar em Jupyter notebooks, exploramos explicitamente (e somente) o caso da segunda derivada a fim de recuperar a dinâmica do pêndulo.

Se *autoencoders* são capazes de recuperar as imagens originais com uma só variável em $\mathbf{z}(t)$ (no espaço latente), que podemos assumir ser uma transformação afim de $\theta(t)$ (a função geradora das imagens), que é solução da equação (22), então podemos exigir que também respeite a mesma equação diferencial, i.e., que seja solução de

$$\ddot{z}(t) + \sin(z(t)) = 0. \quad (32)$$

Por enquanto, suponha também que desconhecemos a equação do pêndulo, i.e., nosso conjunto de dados é um conjunto de imagens do tipo $\{\mathbf{x}(t), \dot{\mathbf{x}}(t), \ddot{\mathbf{x}}(t)\}$ (lembre que podemos tomar a derivada de uma imagem numericamente, sem conhecer sua expressão analítica, desde que $\mathbf{x}(t)$ seja suficientemente suave). Trataremos de responder à seguinte pergunta no que segue: como podemos garantir que a equação para z represente de fato nosso sistema?

Para isso, criamos uma biblioteca de m funções possíveis para o sistema $\{z, z^2, \dots, z^n, \sin(z), \cos(z), \sqrt{z}, 1/z, \dots\}$. Cada uma destas é uma entrada de uma matriz de funções Θ (aqui, com tamanho $1 \times m$). Também vamos definir uma matriz $(m \times 1)$ de coeficientes $\Xi = (\xi_1, \dots, \xi_m)$, de forma que a equação (30) seja reescrita como

$$\dot{z}(t) = \Theta(z(t))\Xi. \quad (33)$$

Para um espaço latente maior, veja a representação gráfica desta equação na Figura 14. No caso de segunda ordem, no lugar da equação (33), tomamos

$$\ddot{z}(t) = \Theta(z(t))\Xi. \quad (34)$$

Note que a equação (34) não é a segunda derivada da equação (33), mas sim uma equação alternativa no caso em que tratamos sistemas caracterizados por equações diferenciais de segunda ordem.

Para calcular a derivada de $\dot{z}$ em relação ao tempo, usamos a expressão $\dot{\mathbf{x}}(t)$ através da regra da cadeia, da mesma forma que fizemos para o algoritmo de *backpropagation* na Seção 2.2. Neste caso, tomando a notação da Figura 10, temos que $z = \varphi(\mathbf{x})$, logo

$$\dot{z} = \nabla_x \varphi(\mathbf{x})\dot{\mathbf{x}}. \quad (35)$$

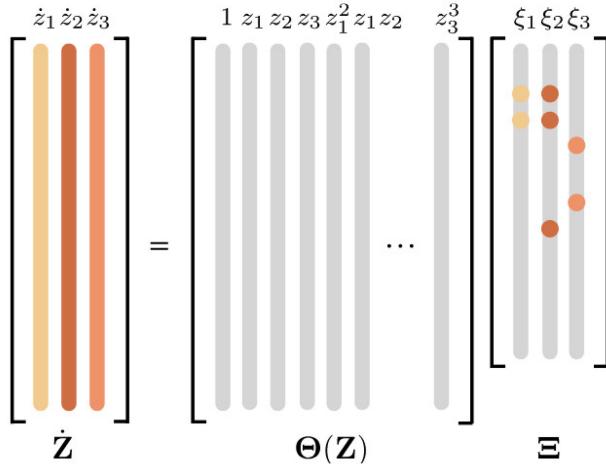


Figura 14: Generalização da equação (33) para mais variáveis. O autoencoder SINDy realiza uma busca entre diversas funções de uma biblioteca Θ , declarando a derivada do espaço latente $\mathbf{z}$ como uma combinação linear da biblioteca com coeficientes dados por Ξ . Esta equação é usada como parte da função custo para a otimização, que, iterativamente, suprime alguns coeficientes ξ_{ij} , até que Ξ esteja suficientemente esparsa, e a equação nos retorne um modelo simples. Uma otimização bem sucedida garante que o espaço latente nos dê uma expressão analítica para as equações de movimento, e, ao mesmo tempo, contendo poucos termos, graças a esparsidade do sistema encontrado. Fonte [22].

No caso paralelo segunda derivada é tomada da mesma maneira, veja o material suplementar em [22], onde obtemos

$$\ddot{\mathbf{z}} = \nabla_x^2 \varphi(\mathbf{x}) \dot{\mathbf{x}}^2 + \nabla_x \varphi(\mathbf{x}) \ddot{\mathbf{x}}. \quad (36)$$

De forma análoga, podemos impor que a derivada da nova imagem gerada, $\dot{\mathbf{x}}(t)$, reproduza fielmente as derivadas das imagens do conjunto de dados, $\dot{\mathbf{x}}(t)$. Essa restrição, aplicada por meio da minimização de uma função de custo definida a seguir, assegura que $\mathbf{z}(t)$ seja um modelo representativo do sistema, assim como o modelo original em t . No nosso caso, imporemos essa condição à segunda derivada, ou seja, $\ddot{\mathbf{x}} \approx \ddot{\mathbf{z}}$.

Ademais, para que $\mathbf{z}$ seja também um modelo das imagens geradas, iremos exigir que a relação

$$\frac{d^n}{dz^n} \tilde{\mathbf{x}}(z) \approx \frac{d^n}{dt^n} \mathbf{x}(t),$$

seja verdadeira, e então que $\mathbf{z}$ se comporte como θ , não apenas para o conjunto de dados mas também para os dados gerados. Assim, ainda seguindo a notação da Figura 10, onde $\psi(z) \approx \tilde{\mathbf{x}}$, temos que

$$\dot{\mathbf{x}} \approx \frac{d}{dz} \tilde{\mathbf{x}} = \nabla_z \psi(z) \Theta(z) \Xi. \quad (37)$$

Onde substituímos $\dot{\mathbf{z}}$ pela equação (33). Para segunda ordem temos

$$\ddot{\mathbf{x}} \approx \frac{d^2}{dz^2} \tilde{\mathbf{x}} = \nabla_z^2 \psi(z) \dot{\mathbf{z}}^2 + \nabla_z \psi(z) \Theta(z) \Xi. \quad (38)$$

Desta vez, substituímos $\ddot{\mathbf{z}}$ pela equação (34).

Dessa maneira, para o caso da derivada de primeira ordem, temos três exigências para nossa rede, o que se traduz em três termos para nossa função custo: Primeiro, o termo que tínhamos no Exemplo 3.3, de reconstrução da imagem original

$$\mathcal{L}_{\text{recon}} = \|\mathbf{x} - \psi \circ \varphi(\mathbf{x})\|_2^2, \quad (39)$$

adicionado ao segundo termo, de reconstrução da derivada da imagem em termos de $\mathbf{z}$

$$\mathcal{L}_{\dot{\mathbf{x}}} = \|\dot{\mathbf{x}} - \nabla_z \psi(z) \Theta(z) \Xi\|_2^2, \quad (40)$$

e finalmente o termo relativo à derivada do modelo gerado por $\mathbf{z}$

$$\mathcal{L}_{\ddot{\mathbf{z}}} = \|\nabla_x \varphi(\mathbf{x}) \dot{\mathbf{x}} - \Theta(z(t)) \Xi\|_2^2. \quad (41)$$

No caso do modelo para sistemas de equação com segunda derivada, reescrevemos as duas funções de custo acima para o caso de segunda ordem como

$$\mathcal{L}_{\ddot{\mathbf{x}}} = \|\ddot{\mathbf{x}} - (\nabla_z^2 \psi(z) \dot{\mathbf{z}}^2 + \nabla_z \psi(z) \Theta(z) \Xi)\|_2^2, \quad (42)$$

e finalmente o termo relativo à derivada do modelo gerado por $\mathbf{z}$

$$\mathcal{L}_{\ddot{\mathbf{z}}} = \|\nabla_x^2 \varphi(\mathbf{x}) \dot{\mathbf{x}}^2 + \nabla_x \varphi(\mathbf{x}) \ddot{\mathbf{x}} - \Theta(z) \Xi\|_2^2. \quad (43)$$

É importante notar que o cálculo destas funções é eficiente no seguinte sentido: assim como para *backpropagation*, podemos guardar os valores das derivadas $\nabla \varphi$ e $\nabla \psi$, ao calcularmos $\tilde{\mathbf{x}}$, e usá-las ambas para o cálculo de $\mathcal{L}$ e para a atualização dos pesos e *bias*.

Com os termos definidos acima, temos o que é necessário para encontrar modelos analíticos para a equação de dinâmica em $\mathbf{z}$, porém, ainda não explicamos como o modelo consegue ser esparsa, como sugere o título da arquitetura. Veja que, se otimizarmos a rede como está até agora, obteremos modelos do tipo $\ddot{\mathbf{z}} = \xi_1 \mathbf{z} + \xi_2 \mathbf{z}^2 \dots \xi_{n+1} \sin(\mathbf{z}) + \dots$. Este não é um modelo particularmente informativo sobre o sistema, já que tem tantos termos quanto forem dados. Duas novas restrições são então necessárias: primeiro, vamos exigir que Ξ tenha uma norma $\|\cdot\|_1$ pequena, forçando que os valores de ξ_i estejam próximos do intervalo $(-1, 1)$. Adicionalmente, após alguns passos da otimização, os valores de ξ_i estarão em uma distribuição tal que alguns destes serão maiores que outros (em valor absoluto), simplesmente por serem mais relevantes para a dinâmica do que outros, assim adicionemos um corte na otimização que elimine os coeficientes $|\xi_i| < 0.1$, tal que $\xi_i = 0$, após o corte. Com isso, estamos eliminando elementos da matriz Ξ e tornando-a esparsa, forçando nosso modelo a ser o mais simples possível.

Nossa função custo toma, então, a forma final

$$\mathcal{L}_{\text{SINDy}} = \mathcal{L}_{\text{recon}} + \lambda_1 \mathcal{L}_{\dot{\mathbf{x}}} + \lambda_2 \mathcal{L}_{\ddot{\mathbf{z}}} + \lambda_3 \|\Xi\|_1, \quad (44)$$

onde λ_i são hiperparâmetros a serem ajustados, e para o caso do pêndulo, foram usados os valores

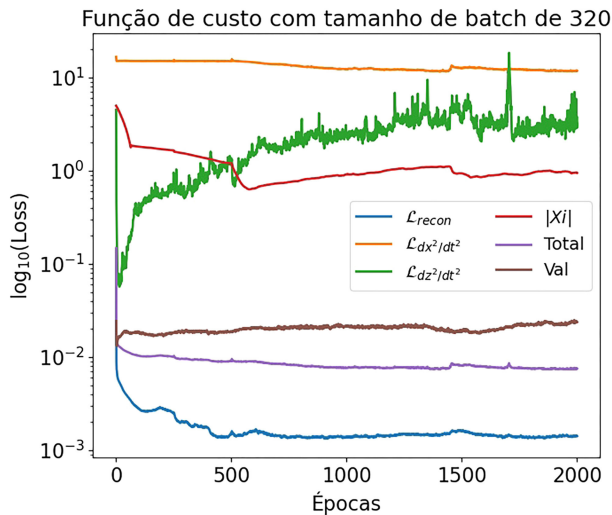


Figura 15: Múltiplas funções de custo do modelo de *Auto-encoders SINDy* em função das épocas de treinamento. No eixo y temos na escala logarítmica a métrica de erro e no eixo x temos o número de épocas. A curva azul representa os valores da função de custo parcial na equação (39), em laranja a equação (42), em verde (43), em vermelho o termo $||\Xi||_1$ da equação (44). A curva correspondente à função de custo total (Total, roxa), dada pela equação (44), apresenta uma tendência decrescente, refletindo a otimização global do modelo. Em algumas iterações do processo de treinamento, encontramos picos na função de custo, isso é representativo do estágio de remoção de coeficientes pequenos do modelo, $|\xi_i| < 0.1$, para torná-lo esparço, neste momento a performance da rede é prejudicada temporariamente. A curva marrom (Val) é dada pelo cálculo da função loss relativo aos dados de validação, dados nunca vistos pela rede, para avaliar sua performance fora do conjunto de dados.

de $\lambda_1 = 5 \times 10^{-4}$, $\lambda_2 = 5 \times 10^{-5}$, $\lambda_3 = 10^{-5}$, os mesmos usados no material suplementar em [22]. O gráfico com os valores parciais da função custo pode ser encontrado na Figura 15. O treinamento foi realizado com uma rede neural MLP contendo 5 camadas ocultas para o *encoder* e outras 5 para o *decoder*, com 256, 128, 64 e 32 neurônios nas camadas, conectados por uma camada latente com 1 neurônio. Como função de ativação foi utilizada a função ReLU, 2 mil épocas de treino, e otimizador *Adam* com taxa de aprendizado de 10^{-4} .

Note que há uma diferença fundamental entre duas possíveis abordagens de treinamento da rede. No nosso caso, a rede é treinada desde o início com uma função de custo que impõe a necessidade de encontrar e respeitar uma equação diferencial simultaneamente. Na segunda abordagem, a rede é primeiramente treinada para reconhecimento de imagens e, posteriormente, o espaço latente é ajustado de forma independente. Neste outro caso, ao otimizar a rede neural em duas etapas distintas, cada uma resulta em um espaço de parâmetros diferente. Consequentemente, a primeira otimização pode conduzir a um mínimo local distante das soluções ótimas para a segunda. No caso das arquiteturas SINDy, como ambas

as condições são otimizadas simultaneamente, a rede neural busca um caminho no espaço de parâmetros que satisfaça todas as restrições ao mesmo tempo. A prioridade atribuída a cada parâmetro é, de certa forma, determinada pelos hiperparâmetros λ_i . Assim como ocorre com outros hiperparâmetros, a escolha adequada desses valores exige familiaridade e experiência com a técnica [6]. Quanto ao caráter do aprendizado, em nenhum momento comparamos diretamente a equação encontrada (32) com a equação real do sistema (22), i.e., não informamos à rede qual é a equação que deveria ter encontrado, tornando este exemplo um caso de aprendizado não supervisionado.

A arquitetura SINDy pode ser usada de maneira mais generalizada para sistemas não lineares de mais variáveis, como atratores de Lorenz e reações de difusão [22]. Neste caso, a equação (33) se torna um sistema matricial como da Figura 14. Lembre-se que, neste caso, precisamos também tomar os polinômios cruzados como $\{z_1^{k_1} z_2^{k_2} \dots z_q^{k_q} | k_1, k_2, \dots, k_q \in \{1, 2, 3, \dots, n\}\}$, onde k_q é a ordem do monômio correspondente à dimensão q , na biblioteca Θ , para reproduzir sistemas de equações acopladas.

No material complementar deste artigo, inspirado pelo que encontramos em [109], temos o exemplo trabalhado para a equação não linear do pêndulo, onde nosso conjunto de dados é composto de imagens do tipo $\{\mathbf{x}(t), \dot{\mathbf{x}}(t), \ddot{\mathbf{x}}(t)\}$. Aqui, restringimos a energia do pêndulo para o caso em que $|\dot{\theta}^2(0)/2 - \cos(\theta(0))| \leq 0.99$, para evitar regimes de rotação completa do pêndulo (*overshoot angular*). Para a validação do treinamento, comparamos $\{\mathbf{x}, \dot{\mathbf{x}}, \ddot{\mathbf{x}}\}$ com $\{\tilde{\mathbf{x}}, \tilde{\dot{\mathbf{x}}}, \tilde{\ddot{\mathbf{x}}}\}$, pois que, por mais que calculamos a diferença dos estados e as acelerações no cálculo da função custo, queremos mostrar que a rede ainda é capaz de recuperar por vezes a velocidade, já que a informação de $\dot{\mathbf{x}}$ está implícita na equação (36). Assim, mesmo que não seja explicitamente parte do treinamento tomar a diferença com a primeira derivada, recuperamos um comportamento aproximado, o que demonstra que o aprendizado é robusto.

Para o nosso exemplo, usamos a biblioteca de polinômios até grau 4 e a função seno. Dentre 10 inicializações testadas, 8 recuperaram o modelo correto, i.e., a equação (22), as outras 2 convergiram para a solução trivial. Vale ressaltar que, em algumas inicializações, a rede recuperou o modelo da equação (23), que é a aproximação linear da equação original, enquanto outras vezes recuperou equações do tipo $\ddot{z} = \xi_0 \sin z + \xi_1 z$, eliminando sistematicamente os termos de ordem superior e atribuindo zero às funções menos relevantes, garantindo a esparsidade do modelo.

Os resultados das imagens geradas $\{\tilde{\mathbf{x}}, \tilde{\dot{\mathbf{x}}}, \tilde{\ddot{\mathbf{x}}}\}$ estão na Figura 16, comparando com os dados de validação $\{\mathbf{x}, \dot{\mathbf{x}}, \ddot{\mathbf{x}}\}$ em um modelo que encontrou a equação do pêndulo. Veja que otimizamos o modelo somente para posição e aceleração, porém, ainda assim, foi capaz de aproximar o resultado para as imagens da velocidade, o que demonstra que a rede de fato aprendeu o modelo e

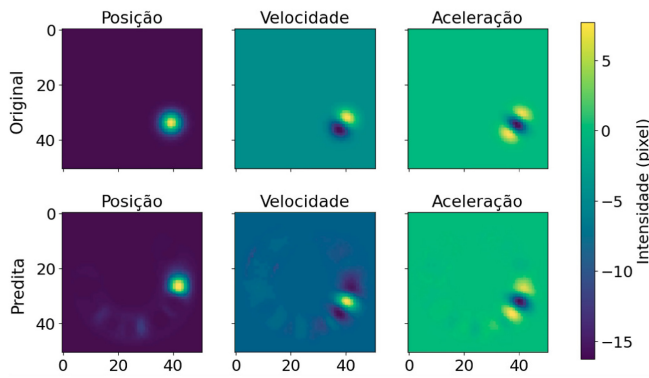


Figura 16: Imagem do Jupyter notebook comparando entrada e saída de SINDy para dados de validação de posição, velocidade e aceleração, i.e., comparando $\{x, \dot{x}, \ddot{x}\}$ com $\{\hat{x}, \hat{\dot{x}}, \hat{\ddot{x}}\}$ para 2000 épocas. Veja que para esta otimização não foram usados os dados de velocidade explicitamente para o treino (apenas implicitamente, por meio da equação (36), porém o modelo ainda é capaz de recuperá-los aproximadamente.

é capaz de reproduzir derivadas de ordem menor. Além disso, essas redes (e PINNs, em geral) requerem menos dados de treinamento do que *autoencoders* tradicionais. Vale destacar que, neste exemplo, não utilizamos redes convolucionais (CNNs), apenas uma rede totalmente conectada, e mesmo assim foram necessárias menos épocas do que em outros casos para se obter resultados satisfatórios. Observamos que, em geral, cerca de 15% dos dados necessários em outras arquiteturas são suficientes para obter resultados significativos, o que significa que, se 2000 épocas fossem exigidas em outra arquitetura, o SINDy deveria ser capaz de encontrar o modelo correto em um intervalo de até 300 a 500 épocas, como pode ser visto pela queda acentuada das curvas roxa e marrom nas primeiras épocas na Figura 15. Este caso serve como exemplo de como o conhecimento analítico reduz a necessidade de treinamento por meio de exemplos, ou seja, uma vez que a rede aprende o modelo por otimização, ela requer menos informação para reproduzir os dados do que os modelos discutidos no restante deste artigo.

4. Conclusões

Neste artigo, exploramos desde aspectos fundamentais a avançados das redes neurais, começando pela introdução do Perceptron até aplicações mais complexas na física. Revisamos conceitos básicos e demonstramos, por meio de exemplos didáticos, como as redes neurais podem ser empregadas em tarefas de classificação binária e regressão. Nos exemplos aplicados à física, mostramos a eficácia das redes neurais profundas aplicadas a um único sistema físico (o pêndulo simples), explorando o mesmo problema sob diversas abordagens: como encontrar o parâmetro físico de uma função a partir dos dados, como determinar a solução de uma equação diferencial e

como encontrar qual a equação diferencial descreve um conjunto de dados.

Dentre esses exemplos, destacamos a utilização das PINNs como uma ferramenta numérica particularmente promissora, embora aqui utilizada apenas para resolver equações diferenciais, mas que demonstra um leque de possibilidades ainda inexplorado. A abordagem das PINNs é particularmente relevante para físicos, ao permitir que as redes neurais aprendam de maneira mais eficiente ao integrar restrições físicas, reduzindo a necessidade de grandes conjuntos de dados e aumentando a precisão das previsões para sistemas regidos por leis conhecidas. Finalmente, vimos que, por meio de arquiteturas SINDy, redes neurais são capazes de aprender sistemas físicos até mesmo em sistemas não lineares e prover modelos analíticos, possivelmente auxiliando não só em estudos de sistemas que ainda carecem de modelos, como também provendo maior clareza sobre o produto da aprendizagem da rede, expressando-o de forma explícita em uma equação.

Deve-se notar, contudo, que o sucesso de todas estas abordagens depende criticamente da escolha de hiperparâmetros, como a profundidade e largura da rede ou a taxa de aprendizado do otimizador. Diferentemente de parâmetros físicos derivados de primeiros princípios, não existe uma fórmula geral para a seleção ótima desses valores. O processo assemelha-se mais à calibração de um aparato experimental complexo do que a uma derivação teórica. Essa realidade sublinha o papel insubstituível do cientista, cuja intuição e experiência guiam a experimentação necessária para ajustar o modelo, destacando a dimensão prática e, por vezes, artesanal, que acompanha a aplicação destas poderosas ferramentas.

Este estudo destaca a crescente importância das redes neurais como ferramentas poderosas na pesquisa em física, demonstrando seu potencial para revolucionar abordagens tradicionais em física teórica e computacional. Ao integrar métodos avançados de ML, como redes neurais, à modelagem física, este trabalho não apenas demonstra a precisão das simulações e a análise de dados complexos, mas também contribui para promover uma mudança na maneira como a ciência é conduzida. Essa integração interdisciplinar permite explorar novos horizontes em problemas que anteriormente eram intratáveis ou demandavam aproximações significativas. Assim, abre-se um caminho promissor para um novo paradigma científico, onde técnicas de IA e métodos físicos se complementam para avançar a compreensão dos fenômenos naturais.

Agradecimentos

Agradecemos ao prof. Dr. Celso Jorge Villas Boas pelas sugestões para a melhoria do trabalho.

Este trabalho teve o apoio do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), processos No. 465469/2014-0 e No. 311612/2021-0, e

da Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), processos No. 2022/00209-6 e No. 2023/15739-3.

Material Suplementar

O seguinte material suplementar está disponível online:
Apêndice A: Cálculo da função de custo

Disponibilidade de Dados

Todo o conjunto de dados que dá suporte aos resultados deste estudo foi disponibilizado no Github e pode ser acessado em https://github.com/Coffee4MePlz/Notebooks_NN_Physics

Referências

- [1] A.B. Ige, P.A. Adepoju, A.O. Akinade e A.I. Afolabi, *International Journal of Multidisciplinary Research and Growth Evaluation* **6**, 36 (2025).
- [2] J.M. Sahut e M. Laroche, *Computers in Human Behavior* **170**, 108684 (2025).
- [3] A.M. Khedr e S. Rani, *Journal of Open Innovation: Technology, Market, and Complexity* **10**, 100379 (2024).
- [4] A. Géron, *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow* (O'Reilly Media, Sebastopol, 2017).
- [5] S.L. Brunton and J.N. Kutz, *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control* (Cambridge University Press, Cambridge, 2019).
- [6] G. Carleo, I. Cirac, K. Cranmer, L. Daudet, M. Schuld, N. Tishby, L. Vogt-Maranto e L. Zdeborová, *Reviews of Modern Physics* **91**, 045002 (2019).
- [7] B. Li e S. Gilbert, *NPJ Digital Medicine* **7**, 336, (2024).
- [8] J. Schmidhuber, *Neural Networks* **61**, 85 (2015).
- [9] F. Rosenblatt, *Psychological Review* **65**, 386 (1958).
- [10] S. Uddin, I. Haque, H. Lu, M.A. Moni e E. Gide, *Sci Rep* **12**, 6256 (2022).
- [11] P. Cunningham e S.J. Delany, arXiv:2004.04523 (2020).
- [12] C.Y.J. Peng, K.L. Lee e G.M. Ingersoll, *The Journal of Educational Research* **96**, 3 (2002).
- [13] D. Bertsimas e A. King, *Statistical Science* **32**, 367 (2017).
- [14] W. Noble, *Nat Biotechnol* **24**, 1565 (2006).
- [15] A.C. Lorena e A.C.P.L.F. Carvalho, *Revista de Informática Teórica e Aplicada* **14**, 43 (2007).
- [16] S.B. Kotsiantis, *Artif Intell Rev* **39**, 261 (2013).
- [17] C. Kingsford e S.L. Salzberg, *Nat Biotechnol* **26**, 1011 (2008).
- [18] T. Neupert, M.H. Fischer, E. Greplova, K. Choo e M.M. Denner, arXiv:2102.04883 (2021).
- [19] G. Bonaccorso, *Machine Learning Algorithms: A reference guide to popular algorithms for data science and machine learning* (Packt Publishing, Birmingham, 2017).
- [20] F. Marquardt, *SciPost Physics Lecture Notes* **29**, 1 (2021).
- [21] I. Goodfellow, Y. Bengio e A. Courville, *Deep Learning*, disponível em <http://www.deeplearningbook.org>.
- [22] K. Champion, B. Lusch, J.N. Kutz e S.L. Brunton, *Proc. Natl. Acad. Sci. U.S.A.* **116**, 22445 (2019).
- [23] D. Silver, A. Huang, C.J. Maddison, A. Guez, L. Sifre, G. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al., *Nature* **529**, 484 (2016).
- [24] V. Mnih, K. Kavukcuoglu, D. Silver, A.A. Rusu, J. Veness, M.G. Bellemare, A. Graves, M. Riedmiller, A.K. Fidjeland, G. Ostrovski, et al., *Nature* **518**, 529 (2015).
- [25] Open AI, I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell et al., arXiv:1910.07113 (2019).
- [26] V.V. Gligorov e M. Williams, *Journal of Instrumentation* **8**, P02013 (2013).
- [27] P. Baldi, P. Sadowski e D. Whiteson, *Nat Commun* **5**, 4308 (2014).
- [28] M. Feickert e N. Benjamin, arXiv:2102.02770 (2021).
- [29] J. Carrasquilla e R. Melko, *Nature Phys* **13**, 431 (2017).
- [30] E. Nieuwenburg, Y.H. Liu e S. Huber, *Nature Phys* **13**, 435 (2017).
- [31] J.S. Smith, O. Isayev and A.E. Roitberg, *Chemical Science* **8**, 3192 (2017).
- [32] Z. Hao, S. Liu, Y. Zhang, C. Ying, Y. Feng, H. Su e J. Zhu, arXiv:2211.08064 (2022).
- [33] G.E. Karniadakis, I.G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang e L. Yang, *Nature Reviews Physics* **3**, 422 (2021).
- [34] C. Meng, S. Seo, D. Cao, S. Griesemer e Y. Liu, arXiv:2203.16797 (2022).
- [35] S. Cuomo, V.S. Cola, F. Giampaolo, G. Rozza, M. Raissi e F. Piccialli, *Journal of Scientific Computing* **92**, 88 (2022).
- [36] S.A. Faroughi, N. Pawar, C. Fernandes, M. Raissi, S. Das, N.K. Kalantari e S.K. Mahjour, arXiv:2211.07377 (2022).
- [37] S. Cai, Z. Mao, Z. Wang, M. Yin e G.E. Karniadakis, arXiv:2105.09506 (2021).
- [38] P. Sharma, W.T. Chung, B. Akoush e M. Ihme, *Energies* **16**, 2343 (2023).
- [39] A.F. Psaros, X. Meng, Z. Zou, L. Guo e G.E. Karniadakis, arXiv:2201.07766 (2022).
- [40] R. Wang e R. Yu, arXiv:2107.01272 (2021).
- [41] J. Carrasquilla e G. Torlai, arXiv:2101.11099 (2021).
- [42] P. Freire, E. Manuylovich, J.E. Prilepsky e S.K. Turitsyn, *Adv. Opt. Photon* **15**, 739 (2023).
- [43] B. Hadad, S. Froim, E. Yosef, R. Giryas e A. Bahabad, *Journal of Optics* **25**, 123501 (2023).
- [44] D. Zhang e Z. Tan, *Appl. Sci.* **12**, 5338 (2022).
- [45] H. Zhu, H. Lin, S. Wu, W. Luo, H. Zhang, Y. Zhan, X. Wang, A. Liu e L.C. Kwek, *Information* **15**, 95 (2024).
- [46] A.M. Palmieri, E. Kovlakov, F. Bianchi, D. Yudin, S. Straupe, J.D. Biamonte e S. Kulik, *npj Quantum Inf* **6**, 1 (2020).

- [47] Y. Quek, S. Fort e H.K. Ng, npj Quantum Inf **7**, 105 (2021).
- [48] K. Zubov, Z. McCarthy, Y. Ma, F. Calisto, V. Pagliarino, S. Azeglio, L. Bottero, E. Luján, V. Sulzer, A. Bharambe et al., arXiv:2107.09443 (2021).
- [49] J. Blechschmidt e O.G. Ernst, GAMM-Mitteilungen **44**, e202100006 (2021).
- [50] J. Willard, X. Jia, S. Xu, M. Steinbach e V. Kumar, arXiv:2003.04919 (2020).
- [51] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart e A. Anandkumar, arXiv:2010.08895 (2021).
- [52] S. Jiao, J. Feng, Y. Gao, T. Lei, Z. Xie e X. Yuan, Optics letters **44**, 5186 (2019).
- [53] K. Yadav, S. Bidnyk e A. Balakrishnan, J. Opt. Soc. Am. B **41**, 1739 (2024).
- [54] S.M. Udrescu e M. Tegmark, Science Advances **6**, eaay2631 (2020).
- [55] Y. Zhou, C. MacPhee, M. Suthar e B. Jalali, arXiv:2301.12531 (2023).
- [56] C. Banerjee, K. Nguyen, C. Fookes e K. George, ACM Comput. Surv. **57**, 1 (2024).
- [57] A.G. Baydin, B.A. Pearlmutter, A.A. Radul e J.M. Siskind, Journal of Machine Learning Research **18**, 1 (2018).
- [58] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al., arXiv:1712.01815 (2017).
- [59] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al., arXiv:1409.0575 (2014).
- [60] P.P. Ray, Internet of Things and Cyber-Physical Systems **3**, 121 (2023).
- [61] Z.C. Lipton, J. Berkowitz e C. Elkan, arXiv:1506.00019 (2015).
- [62] C.R. Harris, K.J. Millman, S.J. Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N.J. Smith et al., Nature **585**, 357 (2020).
- [63] N. Jmour, S. Zayen e A. Abdelkrim, em: *International Conference on Advanced Systems and Electric Technologies (IC ASET)* (Hammamet, 2018).
- [64] Y.K. Lin, M.C. Su e Y.Z. Hsieh, Appl. Sci. **10**, 5965 (2020).
- [65] Y.T. Zhou, Scientific Programming **2022**, 6028693 (2022).
- [66] R.A. Fisher, *UCI Machine Learning Repository*, disponível em: <https://archive.ics.uci.edu/dataset/53/iris>, acessado em: 09/04/2024.
- [67] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al., The Journal of Machine Learning Research **12**, 2825 (2011).
- [68] X. Glorot e Y. Bengio, em *Thirteenth International Conference on Artificial Intelligence and Statistics* (Sardinha, 2010).
- [69] T. Szandala, em: *Bio-inspired Neurocomputing*, editor por A.K. Bhoi, P.K. Mallick, C.M. Liu e V.E. Balas (Springer, Singapore, 2021).
- [70] V. Sitzmann, J.N.P. Martel, A.W. Bergman, D.B. Lindell e G. Wetzstein, arXiv:2006.09661 (2020).
- [71] A. Dawid, J. Arnold, B. Requena, A. Gresch, M. Płodzień, K. Donatella, K.A. Nicoli, P. Stornati, R. Koch, M. Büttner et al., arXiv:2204.04198 (2022).
- [72] M.A.S. Tobi, G. Bevan, P. Wallace, D. Harrison e K.P. Ramachandran, International Journal of Advance Computational Engineering and Networking **4**, 50 (2016).
- [73] C.L. Kuo, E.E. Kuruoglu e W.K.V. Chan, Entropy **24**, 348 (2022).
- [74] M. Rosario, *An Overview of Stochastic Gradient Descent in Machine Learning*, disponível em: http://mdelrosario.com/papers/eec263_SGDinML_delRosario.pdf, acessado em: 09/04/2024.
- [75] R. Abdulkadirov, P. Lyakhov e N. Nagornov, Mathematics **11**, 2466 (2023).
- [76] P. Liashchynskyi e P. Liashchynskyi, arXiv:1912.06059 (2019).
- [77] S. Ruder, arXiv:1609.04747 (2017).
- [78] D.P. Kingma e J. Ba, arXiv:1412.6980 (2014).
- [79] G.G. Lima e G.C. Miranda, *Notebooks acompanhando o artigo*, disponível em: https://github.com/Coffee4MePlz/Notebooks_NN_Physics/tree/Ver%C3%A3o-PT-br, acessado em: 24/12/2025.
- [80] D.E. Rumelhart, G.E. Hinton e R.J. Williams, *Learning internal representations by error propagation* (Universidade da Califórnia, San Diego, 1985).
- [81] Y. LeCun, Y. Bengio e G. Hinton, Nature **521**, 436 (2015).
- [82] S. Hochreiter e J. Schmidhuber, Neural computation **9**, 1735 (1997).
- [83] T. Lin, Y. Wang, X. Liu e X. Qiu, arXiv:2106.04554 (2022).
- [84] U. Michelucci, arXiv:2201.03898 (2022).
- [85] Z. Alom, T.M. Taha, C. Yakopcic, S. Westberg, P. Sidike, S. Nasrin, M. Hasan, B.C. Essen, A.A.S. Awwa e V.K. Asari, Electronics **8**, 1 (2019).
- [86] E. Herberg, arXiv:2304.05133 (2023).
- [87] H. Ramsauer, B. Schäfl, J. Lehner, P. Seidl, M. Widrich, T. Adler, L. Gruber, M. Holzleitner, M. Pavlović, G.K. Sandve et al., arXiv:2008.02217 (2020).
- [88] Z. Liu, Y. Wang, S. Vaidya, F. Ruehle, J. Halverson, M. Soljačić, T.Y. Hou e M. Tegmark, arXiv:2404.19756 (2024).
- [89] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga et al., Advances in neural information processing systems **32**, 8024-8035 (2019).
- [90] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G.S. Corrado, A. Davis, J. Dean, M. Devin et al., arXiv:1603.04467 (2015).
- [91] J. Bradbury, R. Frostig, P. Hawkins, M.J. Johnson, C. Leary, D. Maclaurin, G. Necoala, A. Paszke, J. VanderPlas, S. Wanderman-Milne et al., *JAX: composable transformations of Python+NumPy programs*, disponível em: <http://github.com/google/jax>, acessado em: 09/04/2024.
- [92] A. Krizhevsky, I. Sutskever e G.E. Hinton, Communications of the ACM **60**, 84 (2017).
- [93] M. Bojarski, A. Choromanska, K. Choromanski, B. Firner, L. Jackel, U. Muller e K. Zieba, arXiv:1611.05418 (2017).

- [94] T. Lindeberg, *Int. J. Comput. Vis.* **30**, 79 (1998).
- [95] J.M. Park e Y.L. Murphey, em: *Wiley Encyclopedia of Computer Science and Engineering*, editado por B.W. Wah (Wiley, Hoboken, 2008).
- [96] T. Lindeberg, *Int. J. Comput. Vis.* **11**, 283 (1993).
- [97] K. Fukushima, *Biol. Cybernetics* **36**, 193 (1980).
- [98] S. Basodi, C. Ji, H. Zhang e Y. Pan, *Big Data Mining and Analytics* **3**, 196 (2020).
- [99] D. Bashir, G.D. Montañez, S. Sehra, P.S. Segura and J. Lauw, em: *AI 2020: Advances in Artificial Intelligence* (Springer, Cham, 2020).
- [100] C. Huang, S. Zhai, W. Talbott, M.A. Bautista, S.Y. Sun, C. Guestrin e J. Susskind, em: *International Conference on Machine Learning* (Long Beach, 2019).
- [101] M. Ainsworth e Y. Shin, *SIAM Journal on Scientific Computing* **43**, A3438 (2021).
- [102] B. Hanin, em: *32nd Conference on Neural Information Processing Systems* (Montreal, 2018).
- [103] B. Lusch, J. Nathan Kutz e S.L. Brunton, *Nat Commun* **9**, 4950 (2018).
- [104] T.W. Gamelin, *Complex Analysis* (Springer, New York, 2001).
- [105] C. Gheller e F. Vazza, *Monthly Notices of the Royal Astronomical Society* **509**, 990 (2021).
- [106] P. Vincent, H. Larochelle, Y. Bengio e P.A. Manzagol, em: *25th International Conference on Machine Learning* (Nova Iorque, 2008).
- [107] C. Gheller e F. Vazza, *Monthly Notices of the Royal Astronomical Society* **509**, 990 (2022).
- [108] M. Schuld e F. Petruccione, *Machine Learning with Quantum Computers* (Sprenger, Cham, 2021).
- [109] P. Sillano, *SINDy Pendulum*, disponível em: <https://github.com/pietro-sillano/SindyPendulum>, acessado em: 09/04/2024.