Introdução ao Aprendizado de Máquina Quântico

Lucas Friedrich
Diego Samuel Starke
Jonas Maziero

Introdução ao Aprendizado de Máquina Quântico

LF EDITORIAL
2026

1ª Edição

Direção editorial: Victor Pereira Marinho e José Roberto Marinho

Capa: Fabrício Ribeiro

Edição revisada segundo o Novo Acordo Ortográfico da Língua Portuguesa

Dados Internacionais de Catalogação na publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Friedrich, Lucas
Introdução ao aprendizado de máquina quântico / Lucas Friedrich, Diego Samuel Starke, Jonas Maziero. – São Paulo: LF Editorial, 2026.

Bibliografia
ISBN 978-65-5563-723-6

1. Aprendizagem de máquina 2. Algoritmos de computadores 3. Ciência da computação 4. Teoria quântica I. Starke, Diego Samuel. II. Maziero, Jonas. III. Título.

26-345686.0 CDD-004

Índices para catálogo sistemático:
1. Ciência da computação 004

Eliane de Freitas Leite – Bibliotecária – CRB 8/8415

LF Editorial
www.livrariadafisica.com.br
www.lfeditorial.com.br
(11) 2648-6666 | Loja do Instituto de Física da USP
(11) 3936-3413 | Editora

Sumário

1 INTRODUÇÃO **4**

2 REDES NEURAIS CLÁSSICAS **7**
2.1 Introdução à arquitetura de uma rede neural 7
2.2 Treinamento 9
2.2.1 Otimização 10
2.2.2 *Backpropagation* 12
2.3 Aplicação em um problema de classificação 15
2.3.1 Gerando os dados de treinamento e teste 15
2.3.2 Função para realizar o treinamento 19
2.3.3 Criando um modelo simples 24
2.3.4 Treinando e avaliando o modelo 25
2.4 Exercícios 31

3 COMPUTAÇÃO QUÂNTICA E ALGORITMOS QUÂNTICOS VARIACIONAIS **32**
3.1 Introdução à computação quântica 32
3.1.1 Qubits 32
3.1.2 Portas lógicas quânticas 34
3.1.2.1 Porta X 35
3.1.2.2 Porta de Hadamard 35
3.1.2.3 Portas de rotação 37
3.1.2.4 Porta CNOT 37
3.1.3 Medição 38
3.1.4 Matriz densidade 40
3.1.5 Circuitos quânticos 41
3.1.6 Introdução ao PennyLane 43
3.1.6.1 Estimando probabilidades 44
3.1.6.2 Estimando valores médios 45
3.1.6.3 Circuitos quânticos parametrizados 46
3.1.6.4 Integração com o PyTorch 47
3.1.6.5 Visualização de circuitos 48
3.2 Introdução aos algoritmos quânticos variacionais 49
3.2.1 Estrutura de um circuito quântico parametrizado 49
3.2.2 Função de custo 51
3.2.3 Otimização 52
3.2.4 Exemplo prático 53

4 REDES NEURAIS QUÂNTICAS E MODELOS HÍBRIDOS **58**
4.1 Estrutura de uma rede neural quântica . . . 58
4.1.1 Camada de entrada . . . 58
4.1.2 Camadas ocultas . . . 60
4.1.3 Camada de saída . . . 61
4.1.4 Treinamento . . . 62
4.2 Modelo híbrido quântico-clássico . . . 63
4.3 Classificação com um modelo híbrido . . . 64
4.3.1 Primeiro modelo . . . 65
4.3.2 Segundo modelo . . . 68
4.3.3 Terceiro modelo . . . 71
4.4 Exercícios . . . 73

5 QUANTIFICAÇÃO DE RECURSOS QUÂNTICOS **74**
5.1 Introdução . . . 74
5.2 Fidelidade . . . 75
5.3 Teste Swap . . . 76
5.4 Quantificador de emaranhamento de Bures . . . 77
5.5 Construção do Circuito Variacional para quantificar emaranhamento . . . 80
5.6 Exemplo de aplicação . . . 82

6 ESTADO DA ARTE, LIMITAÇÕES E FUTURO **87**
6.1 Expressividade . . . 87
6.2 Barren plateaus . . . 89
6.3 Custo para otimização . . . 91
6.4 Como comparar um modelo clássico e um modelo quântico . . . 92

Referências Bibliográficas **94**

Capítulo 1

INTRODUÇÃO

Atualmente, a computação desempenha um papel central em praticamente todas as esferas da sociedade, incluindo a indústria, a ciência, o entretenimento e diversos outros setores. Apesar dos avanços significativos alcançados nas últimas décadas, evidenciados, por exemplo, pela crescente capacidade de processamento dos supercomputadores modernos, ainda existem problemas cuja solução permanece fora do alcance da computação clássica. Em muitos desses casos, a limitação está relacionada ao crescimento exponencial do tempo ou dos recursos necessários para resolver tais problemas, o que torna sua solução impraticável.

Nesse contexto, a computação quântica surge como um paradigma alternativo e promissor. Resultado da interseção entre a ciência da computação e a mecânica quântica, esse novo modelo de computação baseia-se em princípios fundamentalmente distintos daqueles que regem a computação clássica. Ao explorar fenômenos quânticos como superposição, emaranhamento e interferência, a computação quântica pode permitir a construção de algoritmos capazes de lidar de forma mais eficiente com problemas considerados intratáveis no regime clássico.

Entre as diversas propostas de aplicação da computação quântica, uma das mais promissoras é sua integração com o aprendizado de máquina, dando origem ao que se convencionou chamar de *aprendizado de máquina quântico*. Essa área de pesquisa investiga como recursos quânticos, acessíveis por meio de computadores quânticos, podem ser explorados no desenvolvimento de novos modelos de aprendizado de máquina, com o objetivo de superar limitações presentes nas abordagens clássicas. Mais do que simplesmente acelerar algoritmos existentes, o aprendizado de máquina quântico busca compreender de que forma estruturas quânticas podem levar a novos modelos, representações e estratégias de treinamento.

Este livro tem como objetivo oferecer uma introdução ao aprendizado de máquina quântico. Por se tratar de um campo relativamente recente, quando comparado ao aprendizado de máquina clássico, muitos de seus fundamentos ainda estão em desenvolvimento, e diversos desafios permanecem em aberto. Dessa forma, não é objetivo deste livro apresentar arquiteturas quânticas altamente complexas ou explorar exaustivamente resultados de ponta. Em vez disso, o foco está na introdução dos conceitos essenciais, por meio de exemplos simples, que permitam ao leitor compreender como ideias consagradas do aprendizado de máquina clássico, como o algoritmo de *backpropagation*, podem ser adaptadas e utilizadas no contexto de modelos quânticos.

A organização do livro reflete esse objetivo. No Capítulo 2, é apresentada uma revisão dos conceitos fundamentais de redes neurais artificiais clássicas. São discutidos aspectos relacionados à arquitetura em camadas, ao processamento da informação e aos métodos de treinamento, com destaque para o papel do *backpropagation*. Esse capítulo também inclui exemplos práticos de implementação utilizando a biblioteca PyTorch, com o intuito de estabelecer uma base sólida para os capítulos subsequentes.

O Capítulo 3 é dedicado a uma revisão introdutória de computação quântica. Nele, são apresentados os conceitos fundamentais relacionados à representação da informação por meio de qubits, à manipulação dessa informação utilizando portas lógicas quânticas e aos processos de medição. Além disso, são introduzidos os algoritmos quânticos variacionais, que desempenham um papel

central no aprendizado de máquina quântico, bem como a biblioteca PennyLane (BERGHOLM *et al.*, 2018), que será utilizada ao longo do livro para a implementação dos modelos quânticos e híbridos.

No Capítulo 4, exploram-se os modelos de redes neurais quânticas e os modelos híbridos quântico-clássicos. São apresentados exemplos concretos de arquiteturas híbridas, nos quais circuitos quânticos parametrizados são integrados a modelos clássicos, permitindo o treinamento conjunto desses sistemas. O foco permanece em modelos conceitualmente simples, de modo a evidenciar os princípios fundamentais que governam esse tipo de abordagem.

No Capítulo 5, demonstramos como os algoritmos quânticos variacionais podem ser utilizados na quantificação de recursos quânticos. Recursos como coerência e emaranhamento desempenham um papel central na descrição e compreensão de sistemas físicos quânticos, além de constituírem a base para diversas vantagens computacionais e informacionais. Por essa razão, a quantificação desses recursos é um problema de grande interesse para a comunidade de ciência da informação quântica. Trata-se, contudo, de uma tarefa computacionalmente desafiadora, especialmente para estados mistos e sistemas de muitos corpos. Neste capítulo, mostraremos como os algoritmos quânticos variacionais podem, em princípio, abordar esse problema de forma eficiente, aproveitando a flexibilidade de circuitos parametrizados e técnicas de otimização híbridas clássico-quânticas.

O Capítulo 6 é dedicado à discussão do estado da arte, das limitações fundamentais e das perspectivas do aprendizado de máquina quântico. Nesse capítulo, são analisados aspectos teóricos e práticos que impactam diretamente o desempenho desses modelos, incluindo questões relacionadas à expressividade dos circuitos quânticos, à ocorrência de *barren plateaus* no processo de treinamento e ao elevado custo computacional associado à otimização dos parâmetros. Além disso, discute-se a dificuldade em estabelecer critérios justos e sistemáticos de comparação entre modelos quânticos e clássicos, especialmente no que diz respeito ao desempenho, escalabilidade e consumo de recursos.

As simulações apresentadas ao longo do livro são realizadas por meio de *notebooks* desenvolvidos no Google Colab (GOOGLE, 2026), uma plataforma que permite a execução de código em Python diretamente na nuvem. O Colab oferece suporte a textos em modo Markdown com integração de expressões em LaTeX, além de blocos de código executáveis, o que o torna particularmente adequado para fins educacionais. Ainda assim, nada impede que os códigos apresentados sejam executados em outros ambientes, como Jupyter Notebooks ou editores locais, desde que as adaptações necessárias sejam realizadas.

Embora o Google Colab disponibilize algumas bibliotecas pré-instaladas, por se tratar de um ambiente baseado em máquinas virtuais temporárias, nem todas as dependências necessárias estão disponíveis por padrão, e os pacotes instalados não são preservados entre diferentes sessões. Dessa forma, determinadas bibliotecas precisam ser reinstaladas sempre que uma nova sessão é iniciada. A seguir, apresentaremos as principais bibliotecas, juntamente com as versões específicas que utilizamos na elaboração deste livro. Dessa forma, para assegurar que as execuções sejam reproduzidas corretamente, recomendamos que sempre sejam instaladas as versões aqui indicadas.

Para o desenvolvimento dos exemplos apresentados ao longo deste livro, usaremos o PyTorch, que é uma biblioteca *open-source* desenvolvida para a construção de diferentes modelos de aprendizado de máquina. Com essa biblioteca, é possível definir, treinar e avaliar modelos como redes neurais do tipo *multilayer perceptron*, redes convolucionais, redes recorrentes, entre outros. Além disso, por ter sido projetado para facilitar o desenvolvimento desses diversos modelos, o PyTorch oferece funcionalidades abrangentes, incluindo o cálculo automático de gradientes (*autograd*), o uso de otimizadores variados e a definição de funções de custo.

Outra vantagem relevante do PyTorch é a sua flexibilidade na manipulação de tensores e na integração com GPUs, permitindo acelerar de maneira significativa o treinamento de modelos complexos. A biblioteca também conta com uma comunidade ativa e extensa documentação, proporcionando fácil acesso a exemplos, tutoriais e implementações de referência. Ademais, o PyTorch é compatível com frameworks quânticos, como o Pennylane, o que possibilita a construção de modelos híbridos quântico-clássicos, a experimentação de algoritmos variacionais e o desenvolvimento de aplicações que combinam recursos clássicos e quânticos de forma prática e eficiente, mesmo em cenários experimentais limitados.

Embora o Google Colab, em geral, já venha com a versão do PyTorch pré-instalada, sugerimos a instalação da versão que utilizamos durante a preparação deste livro, conforme apresentada no Código 1.1. Esse procedimento garantirá a reprodutibilidade e compatibilidade dos códigos apresentados. Sendo assim, sempre que for iniciar um novo notebook no Google Colab, é recomendada a instalação dessa biblioteca.

Código 1.1: Instalação do `pytorch` para utilização no Google Colab com a versão `0.43.1` especificada. Com o bloco de código inferior é possível verificar a versão instalada.

```
!pip install torch==2.9.0
```

```
import torch
torch.__version__
```

Para o desenvolvimento dos modelos quânticos, utilizaremos o PennyLane, que, assim como o PyTorch, é uma biblioteca *open-source*, porém voltada para computação quântica híbrida e aprendizado de máquina quântico (BERGHOLM *et al.*, 2018). O principal objetivo do PennyLane é integrar circuitos quânticos parametrizados com ferramentas clássicas de *machine learning*, como o próprio PyTorch, permitindo assim a construção e o treinamento de modelos que combinam operações quânticas e clássicas de forma totalmente diferenciável.

Uma característica central do PennyLane é o suporte nativo a *diferenciação automática* de circuitos quânticos. Isso significa que é possível calcular gradientes de quantidades quânticas, como valores médios de observáveis, em relação a parâmetros variacionais, o que o torna especialmente adequado para algoritmos quânticos variacionais, redes neurais quânticas e modelos híbridos quântico-clássicos, como veremos mais adiante.

Como o PennyLane não faz parte da instalação padrão do Colab, sua instalação é realizada manualmente no início de cada *notebook*, por meio do código 1.2, no qual é especificada explicitamente a versão utilizada ao longo deste livro. Essa escolha garante a compatibilidade entre os códigos apresentados e a biblioteca empregada, embora outras versões possam ser utilizadas mediante as devidas adaptações. No bloco de código inferior é possível verificar a versão instalada.

Código 1.2: Instalação do `pennylane` para utilização no Google Colab com a versão `0.43.1` especificada. Com o bloco de código inferior é possível verificar a versão instalada.

```
!pip install pennylane==0.43.1
```

```
import pennylane
pennylane.__version__
```

Embora o PennyLane permita a execução de modelos híbridos quântico-clássico em computadores quânticos, é importante destacar que os computadores quânticos atualmente disponíveis apresentam limitações significativas, incluindo um número reduzido de qubits, presença de ruído, altos custos operacionais e financeiros. Essas limitações tornam inviável, na maioria dos casos, a execução eficiente de modelos de aprendizado de máquina quântico em hardware quântico real. Por esse motivo, grande parte das pesquisas atuais na área baseia-se em simulações clássicas, realizadas em ambientes de nuvem ou em máquinas locais. Em consonância com essa realidade, este livro foca especificamente no desenvolvimento e na análise de modelos quânticos simulados classicamente.

Capítulo 2

REDES NEURAIS CLÁSSICAS

Nos últimos anos, diversos estudos têm sido realizados com o objetivo de estender o aprendizado de máquina ao domínio quântico (SAGINGALIEVA *et al.*, 2023; CEREZO *et al.*, 2022; FAN *et al.*, 2023; LIANG *et al.*, 2021; LIU *et al.*, 2021). Em termos gerais, busca-se desenvolver modelos de aprendizado de máquina quântico que combinem técnicas clássicas de aprendizado com princípios fundamentais da computação quântica, tais como emaranhamento, superposição e interferência. A expectativa é que essa combinação permita a resolução de problemas que são particularmente difíceis ou custosos para modelos clássicos tradicionais.

Como os modelos atuais de aprendizado de máquina quântico ainda são fortemente inspirados em suas contrapartes clássicas, torna-se fundamental compreender o funcionamento desses modelos convencionais. Nesse contexto, este capítulo tem como objetivo apresentar uma introdução aos fundamentos do aprendizado de máquina clássico, com foco específico nas **redes neurais clássicas**, também conhecidas como *neural networks* (NN). Essas redes constituem a base conceitual de diversos modelos amplamente utilizados e servem como principal fonte de inspiração para muitas abordagens de aprendizado de máquina quântico.

Ao longo do capítulo, os conceitos essenciais para a compreensão do funcionamento dessas redes serão apresentados de forma gradual e acessível. Serão abordados aspectos como o processamento dos dados de entrada ao longo de múltiplas camadas, a maneira como o desempenho do modelo é quantificado e como essa informação é empregada para conduzir o processo de treinamento, preparando o leitor para o estudo dos modelos quânticos discutidos nos capítulos seguintes.

Além disso, será apresentado o uso da biblioteca *PyTorch* na construção de um modelo simples de classificação, com o objetivo de ilustrar, de maneira prática, os conceitos desenvolvidos. Adicionalmente, nos capítulos subsequentes, será mostrado como a mesma estrutura do *PyTorch*, utilizada no desenvolvimento de modelos clássicos, pode ser reaproveitada e estendida para a criação de modelos quânticos.

2.1 Introdução à arquitetura de uma rede neural

Antes de introduzir o conceito de redes neurais, é fundamental compreender o modelo básico que as constitui: o neurônio artificial. Um neurônio artificial é um modelo matemático simplificado inspirado no funcionamento de um neurônio biológico. Ele recebe um conjunto de sinais de entrada, cada um ponderado por um peso sináptico, realiza uma combinação linear dessas entradas e, em seguida, aplica uma função de ativação para produzir uma saída. A Fig. 2.1 ilustra esquematicamente os principais componentes desse modelo.

Matematicamente, o funcionamento de um neurônio artificial pode ser descrito da seguinte forma. Dado um vetor de entradas $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ e um vetor de pesos

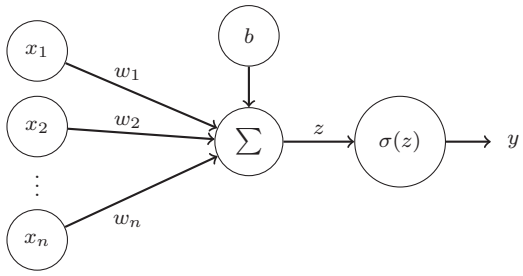

Figura 2.1: Representação esquemática de um neurônio artificial. As entradas x_i são ponderadas pelos pesos w_i, somadas ao viés b e transformadas por uma função de ativação $\sigma(z)$ para produzir a saída y.

$\boldsymbol{w} = (w_1, w_2, \ldots, w_n)$, o neurônio calcula inicialmente a soma ponderada

$$z = \sum_{i=1}^{n} w_i x_i + b, \tag{2.1}$$

onde b é o termo de viés (*bias*). Em seguida, esse valor é transformado por meio de uma função de ativação, resultando na saída

$$y = \sigma(z). \tag{2.2}$$

Por exemplo, ao utilizar a função de ativação *Rectified Linear Unit* (*ReLU*), definida como $\sigma(z) = \max(0, z)$, a saída do neurônio será nula para valores negativos de z e igual a z para valores positivos.

A partir da interconexão de múltiplos neurônios artificiais, forma-se uma *rede neural*, que pode ser entendida como um modelo matemático-computacional projetado para simular, de maneira simplificada, o funcionamento das redes de neurônios biológicos. Ela é organizada em uma sequência de *camadas*, sendo cada camada composta por um conjunto de neurônios que se conectam totalmente aos neurônios das camadas anterior e posterior. Essas camadas processam a informação de forma sequencial: a entrada inicial é transformada camada por camada até gerar a saída final, como ilustrado na Fig. 2.2.

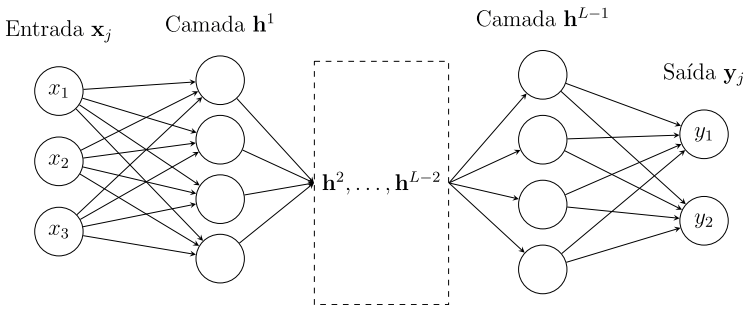

Figura 2.2: Ilustração da arquitetura de uma rede neural descrita pela Eq. (2.4).

As camadas que compõem uma rede neural podem ser classificadas em três tipos, sendo elas:

Camada de entrada: Essa camada é responsável por receber os dados que servirão como entrada para o modelo. Sua função é apenas encaminhar essas informações para as camadas seguintes, não realizando qualquer operação ou transformação sobre os dados.

Camadas ocultas: São as camadas intermediárias que realizam o processamento das informações. Para uma dada entrada $\boldsymbol{h}^{l-1}$, a saída da l-ésima camada é descrita por:

$$\boldsymbol{z}_l = \boldsymbol{W}^l \boldsymbol{h}^{l-1} + \boldsymbol{b}^l, \quad \boldsymbol{h}^l = \sigma(\boldsymbol{z}_l), \tag{2.3}$$

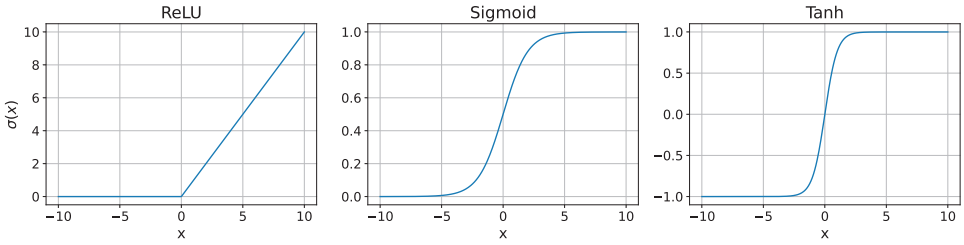

Figura 2.3: Exemplos de funções de ativação normalmente usadas no desenvolvimento de diferentes modelos de aprendizado de máquina. Cada sub-gráfico apresenta o comportamento de uma das funções de ativação dada uma entrada x.

onde $\boldsymbol{W}^l$ denota a matriz de pesos associada à l-ésima camada, $\boldsymbol{b}^l$ é o vetor de viés (*bias*) e $\sigma(\boldsymbol{z}_l)$ representa a função de ativação. Essa função introduz a não linearidade necessária para que a rede seja capaz de modelar relações complexas entre os dados. Algumas das funções de ativação mais utilizadas, além da ReLU, são a *Sigmoid* ($\sigma(x) = 1/(1 + e^{-x})$) e a *Tangente hiperbólica* ($\tanh(x) = (e^x - e^{-x})/(e^x + e^{-x})$). Na Fig. 2.3, são apresentados os comportamentos de cada uma dessas funções de ativação dado uma entrada x.

Camada de saída: É a camada final da rede, responsável por produzir o resultado do processamento. Seu cálculo segue a mesma estrutura apresentada para as camadas ocultas, conforme a Eq. (2.3). Contudo, a saída gerada por essa camada corresponde diretamente à previsão realizada pelo modelo.

Apesar de sua estrutura aparentemente simples, as redes neurais são capazes de lidar com uma ampla variedade de problemas. Além disso, diversos modelos amplamente utilizados na literatura, como redes neurais convolucionais, redes recorrentes e até mesmo os modernos modelos de linguagem de grande porte, *Large Language Models* (LLMs), seguem essa mesma lógica geral: são organizados em camadas sucessivas que transformam gradualmente uma entrada até a obtenção de uma saída.

Uma vez compreendida a estrutura de uma rede neural, podemos descrever formalmente o seu funcionamento. Definindo a entrada da rede como $\boldsymbol{h}^0 = \boldsymbol{x}_j$ e considerando que cada camada é descrita pela Eq. (2.3), a rede como um todo pode ser expressa como a composição dessas transformações:

$$\boldsymbol{y}_j = \boldsymbol{h}^L \circ \boldsymbol{h}^{L-1} \circ \cdots \circ \boldsymbol{h}^1(\boldsymbol{x}_j), \tag{2.4}$$

em que o símbolo $\circ$ indica a composição de funções. Isso significa que a saída de uma camada é utilizada como entrada da camada seguinte, de modo que o resultado de $\boldsymbol{h}^1$ alimenta $\boldsymbol{h}^2$, e assim sucessivamente, até que se obtenha a saída final $\boldsymbol{y}_j$.

2.2 Treinamento

Agora que estabelecemos a descrição matemática de uma rede neural, podemos analisar como esses modelos são treinados. Para isso, utilizaremos uma analogia simples: ensinar uma criança a identificar os dígitos de 0 a 9 (veja a Fig. 2.4). Uma forma de fazermos isso é através do seguinte processo. No início, apresentamos os dígitos escritos em um papel e explicamos à criança qual número cada símbolo representa. Também exibimos o mesmo dígito de formas variadas (com diferentes caligrafias, tamanhos e estilos) para que ela aprenda a reconhecê-lo independentemente de sua aparência.

Em seguida, mostramos novamente esses dígitos e pedimos que ela os identifique. Quando acerta, confirmamos sua resposta; quando erra, corrigimos e indicamos onde ocorreu o equívoco. Esse ciclo é essencial: apresentamos exemplos, solicitamos uma resposta e fornecemos *feedback* contínuo, reforçando acertos e corrigindo erros. Embora essa situação possa lembrar um cenário

Figura 2.4: Ilustração de uma pessoa adulta ensinando uma criança.

de aprendizado por reforço, no qual um agente aprende pela interação com o ambiente, aqui utilizaremos a analogia para ilustrar o funcionamento do aprendizado supervisionado.

Agora, se atribuirmos valores numéricos às respostas da criança, por exemplo, 1 para erro e 0 para acerto e analisarmos o comportamento médio desses valores ao longo do tempo, veremos que, à medida que a criança aprende a identificar corretamente os dígitos, essa média gradualmente se aproxima de zero. Isso indica que ela está melhorando e cometendo menos erros conforme o processo de aprendizagem avança.

Embora não saibamos exatamente o que ocorre no cérebro dessa criança que permite que ela aprenda a identificar corretamente os dígitos, sabemos que, conforme ela progride, a média entre acertos e erros tende a se aproximar de zero, utilizando os valores numéricos mencionados anteriormente. Da mesma forma, se usarmos uma rede neural para identificar os diferentes dígitos de 0 a 9, esperamos que, à medida que o modelo aprende, ele também exiba esse comportamento de melhoria contínua.

Portanto, o treinamento de uma rede neural consiste em otimizar o modelo para que, dado um conjunto de dados $\mathcal{D} = \{(\boldsymbol{x}_i, \widehat{\boldsymbol{y}}_i)\}_{i=1}^{N}$, onde $\boldsymbol{x}_i$ é uma entrada (como a imagem de um dígito) e $\widehat{\boldsymbol{y}}_i$ é o respectivo rótulo, a rede seja capaz de produzir a saída $\widehat{\boldsymbol{y}}_i$ a partir da entrada $\boldsymbol{x}_i$. Para orientar esse processo, utiliza-se uma função C, chamada função de custo, que mede quão distante a saída produzida pelo modelo está do rótulo verdadeiro. Essa função pode ser definida como segue

$$C(\boldsymbol{W}) = \frac{1}{N} \sum_{i=1}^{N} l(\widehat{\boldsymbol{y}}_i, \boldsymbol{y}_i), \tag{2.5}$$

onde $l(.,.)$ é uma função que mede a diferença entre o resultado previsto pelo modelo $\boldsymbol{y}_i$, dada a entrada $\boldsymbol{x}_i$, e o rótulo verdadeiro $\widehat{\boldsymbol{y}}_i$. Ademais, $\boldsymbol{W}$ representa todos os pesos e biases da rede, e como a saída $\boldsymbol{y}_i$ depende de $\boldsymbol{W}$ e estes são gerados aleatoriamente ao criarmos o modelo, podemos concluir que o erro do modelo depende unicamente de $\boldsymbol{W}$. Logo, o treinamento consiste em otimizar $\boldsymbol{W}$ de modo a minimizar a função C definida na Eq. (2.5).

2.2.1 Otimização

Uma vez estabelecido que o treinamento de uma rede neural está diretamente associado à minimização de uma função de custo e, consequentemente, à otimização dos parâmetros $\boldsymbol{W}$, podemos analisar como esse processo é efetivamente realizado. Em geral, a otimização dos parâmetros é realizada usando o gradiente da função de custo em relação a esses parâmetros. Por exemplo, o método do **gradiente descendente** consiste em um procedimento iterativo no qual os parâmetros do modelo são atualizados segundo a regra

$$\boldsymbol{W}_{t+1} = \boldsymbol{W}_t - \eta \nabla_{\boldsymbol{W}_t} C(\boldsymbol{W}_t), \tag{2.6}$$

onde η é a taxa de aprendizado (*learning rate*), responsável por controlar a magnitude das atualizações, e t denota a iteração (ou época) do treinamento.

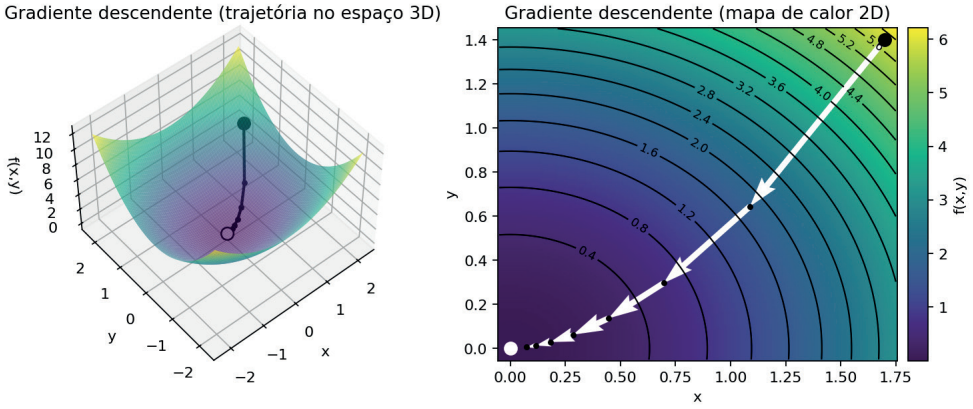

Figura 2.5: Ilustração do processo de otimização por gradiente descendente. O ponto branco indica a posição do mínimo global da função, enquanto o ponto preto representa uma posição inicial dos parâmetros. A seta indica a direção do novo ponto obtida a partir da regra de atualização descrita na Eq. (2.6).

A intuição por trás do gradiente descendente é bastante direta. Se o valor da função de custo depende exclusivamente dos parâmetros $\boldsymbol{W}$, então deslocá-los na direção oposta ao gradiente da função de custo, isto é, subtrair o termo $\nabla_{\boldsymbol{W}_t} C(\boldsymbol{W}_t)$, tende a reduzir o erro do modelo, conforme ilustrado na Fig. 2.5. Nessa figura, o ponto em branco indica a posição do mínimo global da função, o ponto preto representa uma posição inicial dos parâmetros, e a seta aponta a direção do novo ponto obtido ao aplicar a regra de atualização descrita na Eq. (2.6). Repetindo esse procedimento de forma iterativa, o modelo é gradualmente conduzido a uma configuração de parâmetros que minimiza a função de custo. Na Fig. 2.6 apresentamos uma ilustração esquemática do processo de treinamento de uma rede neural. De maneira geral, esse processo pode ser descrito pelas seguintes etapas:

- **Dados de treinamento:** Inicialmente, considera-se um conjunto de dados de treinamento composto por N pares $(\boldsymbol{x}_i, \widehat{\boldsymbol{y}}_i)$, em que $\boldsymbol{x}_i$ representa a entrada e $\widehat{\boldsymbol{y}}_i$ o rótulo correspondente que se deseja que o modelo aprenda.
- **Modelo:** As entradas $\boldsymbol{x} := \{\boldsymbol{x}_i\}_{i=1}^N$ são fornecidas ao modelo, que as processa e produz as previsões $\boldsymbol{y} := \{\boldsymbol{y}_i\}_{i=1}^N$.
- **Função de custo:** A partir das previsões $\boldsymbol{y}$ e dos rótulos verdadeiros $\widehat{\boldsymbol{y}} := \{\widehat{\boldsymbol{y}}_i\}_{i=1}^N$, calcula-se o valor da função de custo, que quantifica o erro cometido pelo modelo.
- **Cálculo dos gradientes:** Em seguida, computam-se os gradientes da função de custo em relação aos parâmetros do modelo.
- **Atualização dos parâmetros:** Por fim, os parâmetros do modelo são atualizados, por exemplo, utilizando o método do gradiente descendente. Essa atualização é então enviada de volta ao modelo, que gera novas previsões, e o processo se repete sucessivamente.

Embora o gradiente descendente padrão seja conceitualmente simples, ele pode apresentar limitações práticas, como sensibilidade excessiva à escolha da taxa de aprendizado e dificuldade em lidar com funções de custo com vales estreitos ou regiões de curvatura acentuada. Para contornar essas dificuldades, diversos métodos de otimização mais sofisticados foram propostos na literatura, entre os quais se destaca o algoritmo **Adam** (*Adaptive Moment Estimation*).

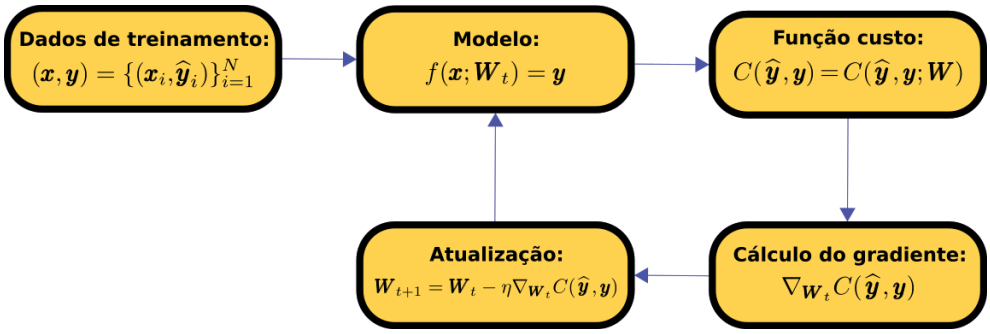

Figura 2.6: Ilustração esquemática do processo de treinamento de uma rede neural, destacando as etapas de propagação direta, cálculo do erro e atualização dos parâmetros.

O algoritmo Adam combina ideias do gradiente descendente com momento e métodos adaptativos de taxa de aprendizado. Em particular, ele mantém estimativas exponencialmente ponderadas do primeiro momento (média) e do segundo momento (variância) dos gradientes, denotadas respectivamente por $\boldsymbol{m}_t$ e $\boldsymbol{v}_t$. Essas quantidades são inicialmente nulas, ou seja, $\boldsymbol{m}_0 = \mathbf{0}$ e $\boldsymbol{v}_0 = \mathbf{0}$, e são atualizadas a cada passo t conforme

$$\boldsymbol{m}_t = \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1)\boldsymbol{g}_t, \tag{2.7}$$

$$\boldsymbol{v}_t = \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2)\boldsymbol{g}_t^2, \tag{2.8}$$

onde $\boldsymbol{g}_t = \nabla_{\boldsymbol{W}_t} C(\boldsymbol{W}_t)$ é o gradiente no instante t, e β_1 e β_2 são hiperparâmetros que controlam o decaimento exponencial dessas médias.

Para corrigir o viés introduzido pela inicialização dessas estimativas, o Adam utiliza versões corrigidas, dadas por

$$\hat{\boldsymbol{m}}_t = \frac{\boldsymbol{m}_t}{1 - \beta_1^t}, \qquad \hat{\boldsymbol{v}}_t = \frac{\boldsymbol{v}_t}{1 - \beta_2^t}. \tag{2.9}$$

A atualização final dos parâmetros é, então, realizada segundo a regra

$$\boldsymbol{W}_{t+1} = \boldsymbol{W}_t - \eta \frac{\hat{\boldsymbol{m}}_t}{\sqrt{\hat{\boldsymbol{v}}_t} + \varepsilon}, \tag{2.10}$$

em que ε é um pequeno termo positivo introduzido para evitar instabilidades numéricas.

Devido à sua capacidade de ajustar automaticamente a taxa de aprendizado para cada parâmetro e de acelerar a convergência em problemas complexos, o Adam tornou-se um dos algoritmos de otimização mais utilizados no treinamento de redes neurais profundas, sendo frequentemente adotado como escolha padrão em diversas aplicações práticas.

2.2.2 *Backpropagation*

Como vimos, o treinamento consiste em um processo interativo no qual o gradiente da função de custo é usado para otimizar os parâmetros do modelo. Porém, uma questão que fica é como esse gradiente é obtido? Uma possibilidade seria usar a derivação numérica para obter esse gradiente. Todavia, tal método possui um custo computacional elevado, o que impossibilitaria um treinamento eficiente, principalmente para modelos com centenas de milhões ou até mesmo bilhões de parâmetros.

O *backpropagation* é o método usado para a obtenção eficiente desse gradiente. Esse método se baseia na obtenção do gradiente para cada camada do modelo através da aplicação da regra da cadeia do cálculo diferencial. Para entendermos como esse método funciona da forma mais didática possível, consideremos uma rede neural extremamente simples, mostrada na Fig. 2.7. A

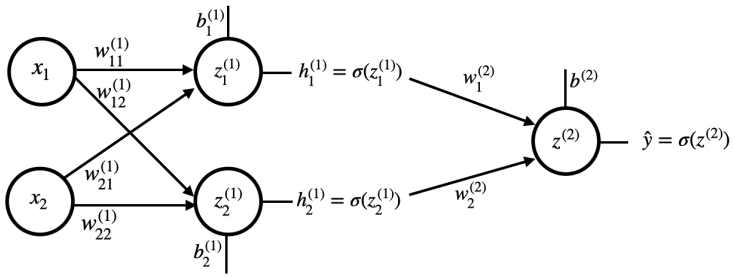

Figura 2.7: Rede neural com dois neurônios na camada de entrada, dois neurônios na camada oculta e um neurônio na camada de saída.

camada de entrada possui dois neurônios, com entradas x_1 e x_2. A camada oculta também possui dois neurônios, cujas ativações serão denotadas por $h_1^{(1)}$ e $h_2^{(1)}$. A camada de saída possui um único neurônio, cuja saída será $\hat{y}$. A dinâmica das camadas segue a Eq. (2.3). Escrevendo tudo de forma explícita, as ativações da camada oculta são dadas por:

$$z_1^{(1)} = w_{11}^{(1)} x_1 + w_{21}^{(1)} x_2 + b_1^{(1)}, \quad z_2^{(1)} = w_{12}^{(1)} x_1 + w_{22}^{(1)} x_2 + b_2^{(1)}, \tag{2.11}$$

$$h_1^{(1)} = \sigma\left(z_1^{(1)}\right), \quad h_2^{(1)} = \sigma\left(z_2^{(1)}\right). \tag{2.12}$$

A saída da rede é, então, calculada como

$$z^{(2)} = w_1^{(2)} h_1^{(1)} + w_2^{(2)} h_2^{(1)} + b^{(2)}, \quad \hat{y} = \sigma\left(z^{(2)}\right). \tag{2.13}$$

Por simplicidade, vamos considerar, neste exemplo, que nosso objetivo é minimizar a função de custo erro quadrático médio, que, para um único exemplo, pode ser escrita da seguinte forma:

$$C = \frac{1}{2}(\hat{y} - y)^2. \tag{2.14}$$

Devemos destacar que o processo descrito a seguir pode ser aplicado a qualquer função de custo e arquitetura do modelo.

Nosso objetivo é calcular as derivadas de C em relação a todos os pesos e vieses da rede. O *backpropagation* faz isso começando pela camada de saída, onde o erro é diretamente observável, e seguindo em direção à camada de entrada.

Comecemos calculando a derivada do custo em relação aos parâmetros da camada de saída. Aplicando a regra da cadeia, temos

$$\frac{\partial C}{\partial w_1^{(2)}} = \frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial w_1^{(2)}}. \tag{2.15}$$

Cada termo da expressão no lado direito da igualdade possui uma interpretação clara. O primeiro termo mede como o custo varia com a saída da rede:

$$\frac{\partial C}{\partial \hat{y}} = \hat{y} - y. \tag{2.16}$$

O segundo termo é a derivada da função de ativação:

$$\frac{\partial \hat{y}}{\partial z^{(2)}} = \sigma'\left(z^{(2)}\right). \tag{2.17}$$

O terceiro termo indica como o potencial linear depende do peso considerado:

$$\frac{\partial z^{(2)}}{\partial w_1^{(2)}} = h_1^{(1)}. \tag{2.18}$$

Juntando tudo, obtemos

$$\frac{\partial C}{\partial w_1^{(2)}} = (\hat{y} - y)\, \sigma'\left(z^{(2)}\right)\, h_1^{(1)}. \tag{2.19}$$

De forma análoga,

$$\frac{\partial C}{\partial w_2^{(2)}} = (\hat{y} - y)\, \sigma'\left(z^{(2)}\right)\, h_2^{(1)}, \quad \frac{\partial C}{\partial b^{(2)}} = (\hat{y} - y)\, \sigma'\left(z^{(2)}\right). \tag{2.20}$$

Nesse ponto, já aparece uma estrutura importante: todas as derivadas da camada de saída compartilham um mesmo fator

$$\delta^{(2)} = (\hat{y} - y)\, \sigma'\left(z^{(2)}\right), \tag{2.21}$$

chamado de *erro da camada de saída*.

Agora passamos para a camada oculta. Consideremos, por exemplo, o peso $w_{11}^{(1)}$, que conecta x_1 ao primeiro neurônio oculto. Aplicando novamente a regra da cadeia, temos

$$\frac{\partial C}{\partial w_{11}^{(1)}} = \frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial h_1^{(1)}} \frac{\partial h_1^{(1)}}{\partial z_1^{(1)}} \frac{\partial z_1^{(1)}}{\partial w_{11}^{(1)}}. \tag{2.22}$$

Os três primeiros fatores já são conhecidos e podem ser agrupados:

$$\frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial h_1^{(1)}} = \delta^{(2)} w_1^{(2)}. \tag{2.23}$$

Além disso

$$\frac{\partial h_1^{(1)}}{\partial z_1^{(1)}} = \sigma'\left(z_1^{(1)}\right), \quad \frac{\partial z_1^{(1)}}{\partial w_{11}^{(1)}} = x_1. \tag{2.24}$$

Portanto

$$\frac{\partial C}{\partial w_{11}^{(1)}} = \delta^{(2)} w_1^{(2)} \sigma'\left(z_1^{(1)}\right) x_1. \tag{2.25}$$

De forma semelhante, temos que

$$\frac{\partial C}{\partial w_{12}^{(1)}} = \delta^{(2)} w_1^{(2)} \sigma'\left(z_1^{(1)}\right) x_2, \quad \frac{\partial C}{\partial b_1^{(1)}} = \delta^{(2)} w_1^{(2)} \sigma'\left(z_1^{(1)}\right). \tag{2.26}$$

Para o segundo neurônio oculto, surge uma expressão análoga, envolvendo o peso $w_2^{(2)}$. Isso sugere definir o erro da camada oculta como

$$\delta_1^{(1)} = w_1^{(2)} \delta^{(2)} \sigma'\left(z_1^{(1)}\right), \quad \delta_2^{(1)} = w_2^{(2)} \delta^{(2)} \sigma'\left(z_2^{(1)}\right). \tag{2.27}$$

Esse exemplo simples revela a estrutura geral do *backpropagation*. O gradiente de um peso é sempre dado pelo produto entre o erro do neurônio de destino e a ativação do neurônio de origem. Em forma vetorial, para uma camada l, isso pode ser escrito como

$$\frac{\partial C}{\partial \boldsymbol{W}_l} = \boldsymbol{\delta}_l (\boldsymbol{h}^{l-1})^\top, \quad \frac{\partial C}{\partial \boldsymbol{b}_l} = \boldsymbol{\delta}_l, \tag{2.28}$$

onde o sobrescrito $\top$ denota a transposta de um vetor ou matriz, isto é, a operação que troca linhas por colunas.

Os erros das camadas obedecem à relação recursiva

$$\boldsymbol{\delta}_l = \left((\boldsymbol{W}^{l+1})^\top \boldsymbol{\delta}_{l+1}\right) \odot \sigma'\left(\boldsymbol{W}^l \boldsymbol{h}^{l-1} + \boldsymbol{b}^l\right), \tag{2.29}$$

em que o símbolo $\odot$ representa o produto de Hadamard, isto é, a multiplicação elemento a elemento entre vetores de mesma dimensão.

Assim, o que foi demonstrado passo a passo para uma rede com dois neurônios por camada se generaliza naturalmente para redes com qualquer número de neurônios e camadas. O *backpropagation* nada mais é do que a aplicação organizada dessas relações, permitindo calcular todos os gradientes necessários para otimizar a rede neural.

2.3 Aplicação em um problema de classificação

A seguir, apresentamos um exemplo simples de como aplicar uma rede neural clássica na resolução de um problema de classificação. Para isso, utilizaremos o conjunto de dados do *Modified National Institute of Standards and Technology* (MNIST) (LECUN; BOTTOU; BENGIO; HAFFNER, 1998), amplamente adotado como referência em tarefas de reconhecimento de dígitos manuscritos.

O MNIST é composto por 60.000 imagens destinadas ao treinamento e 10.000 imagens reservadas para teste, sendo cada imagem associada a um dígito inteiro entre 0 e 9. Cada amostra consiste em uma matriz de dimensões 28×28, na qual cada elemento representa a intensidade de um pixel da imagem manuscrita.

2.3.1 Gerando os dados de treinamento e teste

Inicialmente, é necessário importar os pacotes que serão utilizados ao longo do exemplo. Isso é feito no código a seguir:

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, TensorDataset
import torch.nn.functional as F
import matplotlib.pyplot as plt
import numpy as np
```

O código acima inicia com a importação do pacote `torch`, que fornece as operações fundamentais para a criação e manipulação de tensores, além de oferecer suporte à computação eficiente tanto em CPU quanto em GPU. Em seguida, o módulo `torch.nn` é importado, contendo as principais abstrações para a construção de redes neurais, como camadas, funções de ativação e a definição de modelos. O módulo `torch.optim` reúne diferentes algoritmos de otimização empregados durante o treinamento, incluindo o gradiente descendente estocástico (SGD) e o método Adam.

Na sequência, o pacote `torchvision.datasets` é utilizado para acessar conjuntos de dados amplamente empregados em tarefas de visão computacional, como o MNIST. O módulo `torchvision.transforms`, por sua vez, permite definir e aplicar transformações às imagens, como a conversão para tensores e a normalização. O `DataLoader`, proveniente de `torch.utils.data`, facilita o carregamento eficiente dos dados, organizando-os em mini-batches, possibilitando o embaralhamento das amostras e otimizando o uso dos recursos computacionais durante o treinamento.

Além disso, o módulo `TensorDataset` permite criar conjuntos de dados diretamente a partir de tensores, sendo particularmente útil quando os dados já se encontram carregados em memória. O pacote `torch.nn.functional` disponibiliza implementações funcionais de operações comuns em redes neurais, como funções de ativação, operações de interpolação e funções de perda, oferecendo maior flexibilidade na definição do modelo. Por fim, a biblioteca `matplotlib.pyplot`

é empregada para a visualização de resultados, como imagens, curvas de erro ou métricas de desempenho.

Uma vez importados os pacotes necessários, abaixo, iniciamos a construção de uma função cujo objetivo é gerar conjuntos de dados de treinamento e de teste a partir do MNIST. Essa função, denominada `create_mnist_datasets`, recebe como parâmetros de entrada:

- **ntrain**: número de amostras no conjunto de treinamento;
- **ntest**: número de amostras no conjunto de teste;
- **nclass**: número de classes consideradas em ambos os conjuntos. Cada classe corresponde a um dígito no intervalo de 0 a nclass $-$ 1;
- **new_size**: dimensão desejada para as imagens de entrada, redimensionando as imagens originais de 28×28 para new_size $\times$ new_size;
- **balance**: variável booleana que indica se o conjunto de dados deve ser balanceado, isto é, se deve conter o mesmo número de amostras por classe;
- **batch**: tamanho dos mini-batches utilizados durante o treinamento.

A implementação completa da função é apresentada a seguir:

```
def create_mnist_datasets(ntrain,ntest,nclass,new_size=28,
                          balance=True,batch=32):
    # ------------------------------------------------------------
    # 1) Transformação básica
    # ------------------------------------------------------------
    transform = transforms.ToTensor()
    # ------------------------------------------------------------
    # 2) Carregamento do MNIST
    # ------------------------------------------------------------
    train_data = datasets.MNIST(
        root="./data",
        train=True,
        download=True,
        transform=transform
    )
    test_data = datasets.MNIST(
        root="./data",
        train=False,
        download=True,
        transform=transform
    )
    # ------------------------------------------------------------
    # 3) Função auxiliar de filtragem e balanceamento
    # ------------------------------------------------------------
    def filter_and_select(dataset, n_samples):
        images, labels = [], []
        if balance:
            samples_per_class = n_samples // nclass
            class_count = {c: 0 for c in range(nclass)}

            for x, y in dataset:
                if y < nclass and class_count[y] <
                ↪  samples_per_class:
                    images.append(x)
                    labels.append(y)
                    class_count[y] += 1
                if sum(class_count.values()) >= samples_per_class *
                ↪  nclass:
                    break
        else:
```

```
            count = 0
            for x, y in dataset:
                if y < nclass:
                    images.append(x)
                    labels.append(y)
                    count += 1
                if count >= n_samples:
                    break
        return torch.stack(images), torch.tensor(labels)
    # ----------------------------------------------------------------
    # 4) Seleção dos dados
    # ----------------------------------------------------------------
    x_train, y_train = filter_and_select(train_data, ntrain)
    x_test, y_test = filter_and_select(test_data, ntest)
    # ----------------------------------------------------------------
    # 5) Downsampling espacial (PONTO-CHAVE)
    # ----------------------------------------------------------------
    # x: (N, 1, 28, 28) → (N, 1, new_size, new_size)
    x_train = F.interpolate(
        x_train,
        size=(new_size, new_size),
        mode="bilinear",
        align_corners=False
    )
    x_test = F.interpolate(
        x_test,
        size=(new_size, new_size),
        mode="bilinear",
        align_corners=False
    )
    # ----------------------------------------------------------------
    # 6) Achatamento
    # ----------------------------------------------------------------
    x_train = x_train.view(x_train.size(0), -1)
    x_test = x_test.view(x_test.size(0), -1)
    # ----------------------------------------------------------------
    # 7) TensorDatasets
    # ----------------------------------------------------------------
    train_dataset = TensorDataset(x_train, y_train)
    test_dataset = TensorDataset(x_test, y_test)
    # ----------------------------------------------------------------
    # 8) DataLoaders
    # ----------------------------------------------------------------
    train_loader = DataLoader(
        train_dataset,
        batch_size=batch,
        shuffle=True
    )
    test_loader = DataLoader(
        test_dataset,
        batch_size=batch,
        shuffle=False
    )
    return train_loader, test_loader
```

Após a apresentação do código, é importante discutir com mais detalhes o funcionamento da função `create_mnist_datasets` e destacar seus principais componentes. De maneira geral, essa função concentra todas as etapas necessárias para preparar os dados do MNIST de forma adequada ao treinamento e à avaliação de modelos de aprendizado de máquina, desde o carregamento do conjunto de dados original até sua organização final em mini-batches.

Um primeiro ponto relevante diz respeito ao processo de **seleção e filtragem das amostras**. A função interna `filter_and_select` permite escolher um número específico de exemplos e, adicionalmente, restringir o conjunto de dados a um número reduzido de classes, controlado pelo